Winter Workshop

Computer Vision

-BY Nikhil Giri

Digital Representation of Image:

Images are usually reprented as multidimensional Arrays. Different repersentaion leads to different types of image.

Binary Image: Just a 2D array made of 0(black) and 1(white)

Black and White Image: made up of only black and white

8 Bit Color format: 2D array made up of values between 0-255 where 0 is black and 255 is white all other values intermediate between Black and White

16 Bit Color format: Also known as high format color. Distribution different than GrayScale Images

Color images can be either RGB or CMYK type

The Programming aspect of CV-

Coding language: **Python**

Python notes: https://colab.research.google.com/drive/1KXwSfr_ejAe-Ez3EJsBcOQm1z0DIbHtF

OpenCV is the package used for Computer vision. It is used along with **numpy** which is a package for advanced multi-dimensional array which in turn is used to represent images digitally.

How we import them:

```
import numpy as np
import cv2
import math
```

Basic functions of OpenCv:

Reading Image: Returns as numpy array digitally representing the image.

SYNTAX: **cv2.imread(path,flag)**

//path is the location of the saved image. Eg-> “cube.jpg”

//flag represents how the image will be read ->

cv2.IMREAD_COLOR: (1) it reads the image as a colored image. It is the default value

cv2.IMREAD_GRAYSCALE: (0) it reads the image as black and white image

cv2.IMREAD_UNCHANGED:(-1) it reads the image as it is, even the alpha (transparency)

Named Window: Opens a Window object which can be used to show any image.

SYNTAX: **cv2.namedWindow(name,flag)**

//name will be the name of the named window

//flag represents the behavior of the window

WINDOW_NORMAL: enables us to resize the window

WINDOW_AUTOSIZE: adjust according to image and does not allow us to resize

Showing Image: Opens a window with the image.

SYNTAX: **cv2.imshow(win,img)**

//win is a window object or name of a window object

//img is the image matrix-> digital image of any sort

Wait Key: Stops the code for a given amount of time.

SYNTAX: **cv2.waitKey(value)**

// value is an integer in millisecond which is the amount of time the code will stop for

EXAMPLE->

Code to make a circle 200px radius in an 800x800 pixel image

```
import numpy as np
import cv2
import math

mat = np.full((800,800),255)
for i in range(800):
    for j in range(800):
        if math.sqrt((400-i)**2 + (400-j)**2)<=200:
            mat[i][j] =0                                #becomes black
cv2.namedWindow('image',cv2.WINDOW_NORMAL)
cv2.imshow('image',mat.astype(np.uint8))
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Creating Trackbar: Creates a trackbar to change certain value in the window and see the changes.

SYNTAX: **cv2.createTrackbar(name,win,initial_value,max_value,callable_func)**

//name is the name written to the left of trackbar in window

//win is the name of the window on which the trackbar will be created

//initial_value is the value of trackbar when the window starts

//max_value is the maximum value of the trackbar

//callable_funct is the value returned by trackbar when user changes the trackbar value also

The funct should just take int as its parameter.

Example->

Here, I am saving the value of trackbar in a global variable opacity and now I can change the value of opacity by changing the value at trackbar.

```
opacity  = 50

def set_opacity(val):
    global opacity
    opacity = val

cv2.namedWindow("window",cv2.WINDOW_NORMAL)
cv2.createTrackbar("Opacity","window",50,100,set_opacity)
```

Application #1 :<u>RGB TO GRAYSCALE IMAGES</u>

RGB images are 3D arrays of size (h,w,3). To convert it into a GRAYSCALE image we need to transform it into a 2D array of size (h,w).

We achieve That via various methods:

Method #1: Take average of all three value.

That is new value = (R+B+G)/3

i.e. gray[i][j] = ((img[i][j][0])+(img[i][j][1])+(img[i][j][2]))/3

If we use numpy as np package we can do it easily by doing : **gray[i][j]=int(np.sum(img[i][j])/3)**

Method #2: Instead of equal weightage for all color this method gives more preference to the color human

eyes can see better

Since Human eye can see Green Color more clearly and Blue color least

New value can be calculated as:

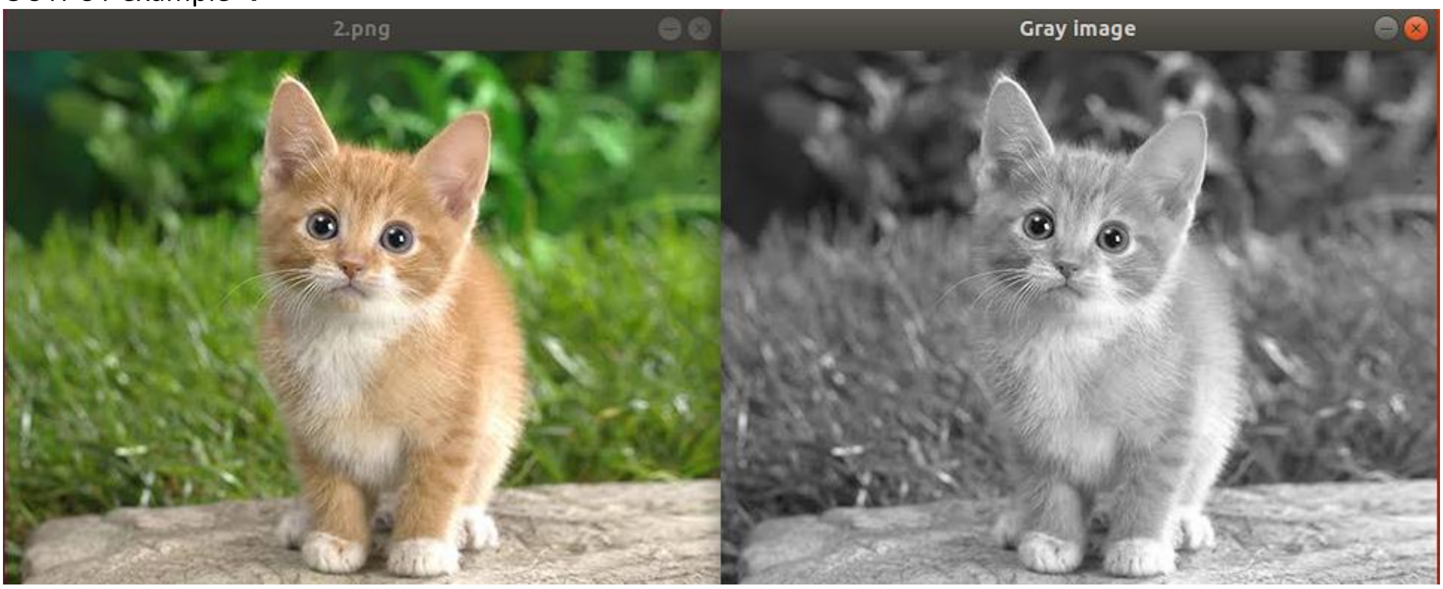
Grey = 0.21R+0.72G+0.07B

Grey = 0.299R + 0.587G +0.114B

Method #3: Take mean of greatest and lowest value

Grey = (max(r,b,g)+min(r,b,g))/2

OUTPUT example →



Application #2: GEOMETRIC OPERATIONS

#1: **FLIP IMAGE :** returns an image matrix which is the flipped version of src img

SYNTAX: **cv2.flip(img, flipCode)**

//img is the image matrix-> digital image of any sort

//flipCode represents nature of the flip

Possible inputs : 0(Vertical), 1(Horizontal), -1(Both)

#2: **IMAGE PADDING** : Adds border to the image thus changing its size and returns the new image

SYNTAX: **cv2.copyMakeBorder(img, top, bottom, left, right, borderType, value)**

//img is the image matrix-> digital image of any sort

//Top, bottom, left, right are size of padding on the respective sides

//borderType represents the kind of border eg->

BORDER_CONSTANT Python: cv.BORDER_CONSTANT	`iiiiii|abcdefgh|iiiiiii` with some specified `i`
BORDER_REPLICATE Python: cv.BORDER_REPLICATE	`aaaaaa|abcdefgh|hhhhhhh`
BORDER_REFLECT Python: cv.BORDER_REFLECT	`fedcba|abcdefgh|hgfedcb`
BORDER_WRAP Python: cv.BORDER_WRAP	`cdefgh|abcdefgh|abcdefg`

//value is an optional parameter if border type is cv2.BORDER_CONSTANT

#3: **IMAZE RESIZING**

->DOWNSIZING IMAGE: Reduces size of image by a factor of n

```
def downsize(orig_img: np.ndarray , scale_factor :int ) -> np.ndarray:
    global h
    global w
    dw_img  =np.zeros((int(h/scale_factor),int(w/scale_factor),1), dtype = "uint8")
    for i in range(int(h/scale_factor)):
        for j in range(int(w/scale_factor)):
            add = int(np.mean(orig_img[i*scale_factor:(i+1)*scale_factor, j*scale_factor:(j+1)*scale_factor], axis=(0, 1)))
            dw_img[i, j] = add

    return dw_img
```

→UPSIZING IMAGE: Increases size of a image by a factor of n

```
def upsize(orig_img: np.ndarray , scale_factor :int ) -> np.ndarray:
    global h
    global w
    up_img  =np.zeros((h*scale_factor,w*scale_factor,1), dtype = "uint8")
    for  i  in range (h):
        for j in range (w):
            up_img[i*scale_factor:i*scale_factor + scale_factor,j*scale_factor:j*scale_factor + scale_factor ]  = orig_img[i,j]
    return up_img
```

#4: **ROTATION :** When we rotate a image by alpha in the given code we find the maximum size of new image (maxL) and then create new image and then find the new position of each respective(i1,j1) and also find new position of the centre pixel (m1,n1) and using m1, n1 as m,n we rotate the image

```python
def rotate(orig_img: np.ndarray =None, alpha: float = None ) -> np.ndarray:
    alpha = (alpha*np.pi)/180.0
    global h
    global w
    maxL = int(np.sqrt((h**2)+(w**2)))+1
    rota_img = np.zeros((maxL,maxL,1), dtype=np.uint8)

    def dist(y,x) : return (np.sqrt((x**2)+(y**2)))

    def theta(y,x) : return np.arctan2(y,x)

    (m,n) = (h-1)//2,(w-1)//2
    (m1,n1) = (maxL-1)//2,(maxL-1)//2

    for i in range(h):
        for j in range(w):
            a= dist(i-m,j-n)
            ang =theta(i-m,j-n)
            j1 = int(a*np.cos(ang +alpha))
            i1 = int(a*np.sin(ang + alpha))
            j1 += n1
            i1 += m1
            rota_img[i1,j1] = orig_img[i,j]
    return rota_img
```

Application #3: MORPHOLOGICAL OPERATIONS

#1 : **EROSION :** removes white noises from a black background image

#2 : **DILATION :** increases white noises from a black background image

```python
def erode(img : np.ndarray = None, kernel : np.ndarray =None) -> np.ndarray:
    h,w = img.shape[:2]
    kh ,kw = kernel.shape
    eroded = np.zeros((h,w),dtype="uint8")
    x,y= (kh-1)//2, (kw-1)//2
    for i in range(x,h-x):
        for j in range(y,w-y):
            addit = cv2.add(img[i-x:i+x+1,j-y:j+y+1],kernel)
            eroded[i,j] = np.min(addit,axis=(0,1))

    return eroded

def dilate(img : np.ndarray = None, kernel : np.ndarray =None) -> np.ndarray:
    h,w = img.shape[:2]
    kh ,kw = kernel.shape
    dilated = np.zeros((h,w),dtype="uint8")
    x,y= (kh-1)//2, (kw-1)//2
    for i in range(x,h-x):
        for j in range(y,w-y):
            addit = cv2.add(img[i-x:i+x+1,j-y:j+y+1],kernel)
            dilated[i,j] = np.max(addit,axis=(0,1))

    return dilated
```

(If the background is white and noises are black, erosion becomes dilation and vice versa)

#3: **HISTOGRAMS:** Graphical representation of intensity of a image vai showing a bar diagram comparing values of each pixel

```
def histogram(img : np.ndarray =None)-> np.ndarray:
    h,w = img.shape
    freq = np.zeros((256))
    for i in range (h):
        for j in range(w):
            freq[img[i,j]] +=1
    freq /= (h*w)
    freq *= 640
    res = np.full((int(np.max(freq)), 256*5),fill_value=255 ,dtype=np.uint8)
    for i in range(256):
        res[-1 - int(freq[i]):,i*5:(i+1)*5] = 0
    return res
```

Application #4: SPACIAL FILTERS

→BLURING FILTERS:

→https://opencv24-python-tutorials.readthedocs.io/en/stable/py_tutorials/py_imgproc/py_filtering/py_filtering.html

→https://setosa.io/ev/image-kernels/

#AVERAGE BLUR (Using Average Kernels)

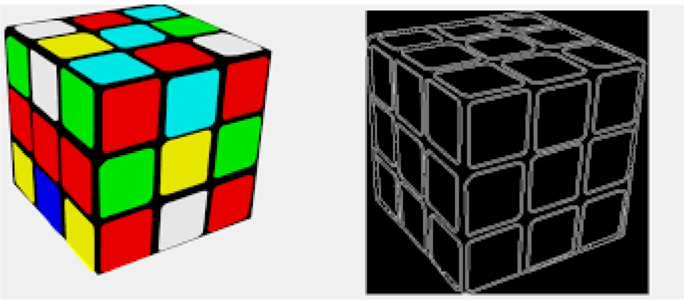
#GAUSSIAN BLUR (Using Gaussian Kernels)

#MEDIAN FILTERING (Median of all values covered by kernel is kernel's output)

→Used to remove Salt & Pepper Noise

```
def Average_Blur(img: np.ndarray =None ,k : int =None) -> np.ndarray:
    '''
    res = np.ones_like(img,dtype = "float32")
    kernel = np.ones((k,k),dtype = np.float32)/(k*k)
    h,w = img.shape[:2]
    f = int((k-1)/2)
    for n in range(3):
        for i in range(f,h-f):
            for j in range(f,w-f):
                res[i,j,n] = np.sum(np.sum(np.multiply(img[i-f:i+f+1,j-f:j+f+1,n],kernel)))
    res =res.astype(np.uint8)
    '''
    #or we can use inbuilt function
    res =cv2.blur(img,(k,k))
    return res


def Gaussian(img : np.ndarray =None ,k : int =None) ->np.ndarray:

    res =cv2.GaussianBlur(img,(k,k),0)
    return res

def Median(img : np.ndarray =None ,k : int =None) ->np.ndarray:

    res =cv2.medianBlur(img,k)
    return res
```

<u>Application #5:</u> <u>EDGE DETECTION</u>



#1: **<u>SOBEL FILTER</u>**

SYNTAX: **<u>dst =cv2.Sobel(img,ddepth,dx,dy,ksize)</u>**

//Dx and dy: specify which direction you want to take the gradient in

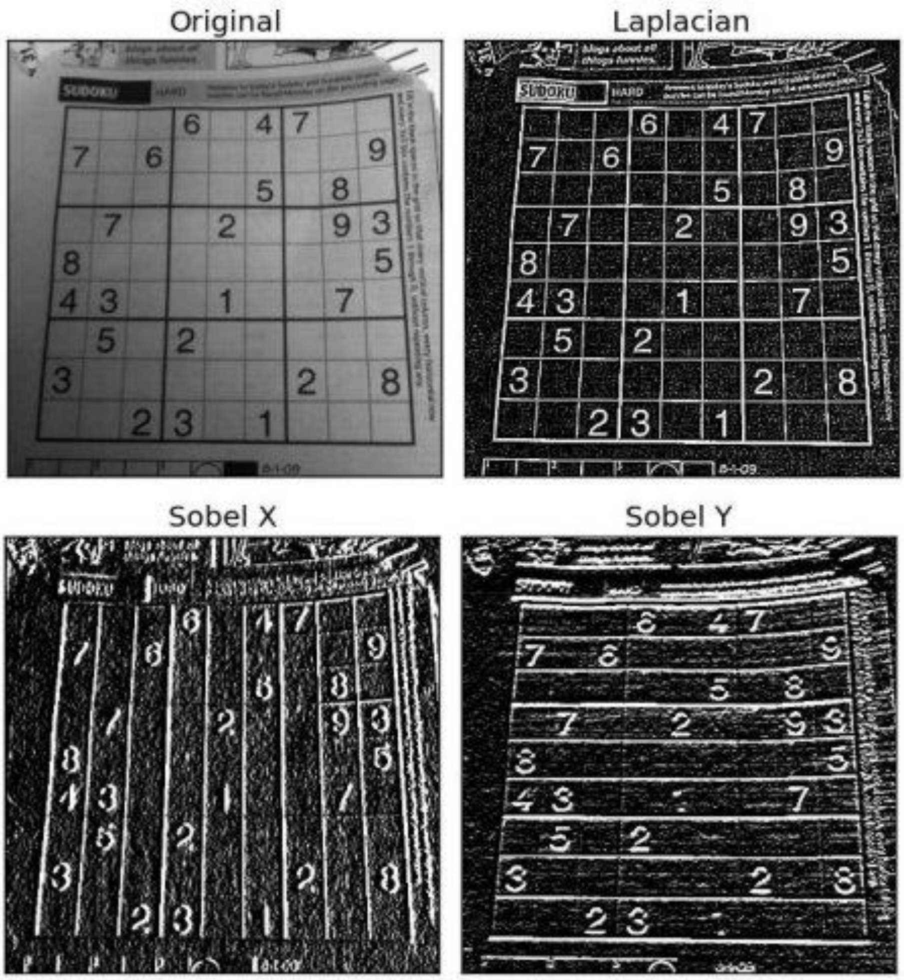
//Ksize: filter size which can be 3,5,7 (if filter size is 3 then Scharr filter is used)

//Ddepth: taken as CV_64F

#2: **<u>CANNY FILTER</u>**

SYNTAX: **<u>result=cv2.Canny(img,maxVal,minVal)</u>**

//maxVal, minVal are the values used in hysteresis thresholding. Generally taken in the ratio 2:1 or 3:1.



Important Data Structures and refences

LINKED LIST

→https://www.geeksforgeeks.org/data-structures/linked-list/

STACKS → (LIFO)

→https://www.geeksforgeeks.org/stack-data-structure/

QUEUE →(FIFO)

BFS (BREADTH FIRST SEARCH):

→https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

DFS (DEPTH FIRST SEARCH):

→https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

More links for additional info:

https://medium.com/@rinu.gour123/ai-python-computer-vision-tutorial-with-opencv-b7f86c3c6a1a

TODO: To upload all codes in google drive and add their links here