Other Project

<https://towardsdatascience.com/computer-vision-auto-grading-handwritten-mathematical-answersheets-8974744f72dd>

find contours in eqution <https://blog.ayoungprogrammer.com/2013/01/equation-ocr-part-1-using-contours-to.html/>

# Line Detection

cv.findContours

# Thresholding

<https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_thresholding/py_thresholding.html>

* binary image

Here, the matter is straight forward. If pixel value is greater than a threshold value, it is assigned one value (may be white), else it is assigned another value (may be black). The function used is cv2.threshold. **First argument is the source image, which should be a grayscale image**. Second argument is the threshold value which is used to classify the pixel values. Third argument is the maxVal which represents the value to be given if pixel value is more than (sometimes less than) the threshold value. OpenCV provides different styles of thresholding and it is decided by the fourth parameter of the function. Different types are:

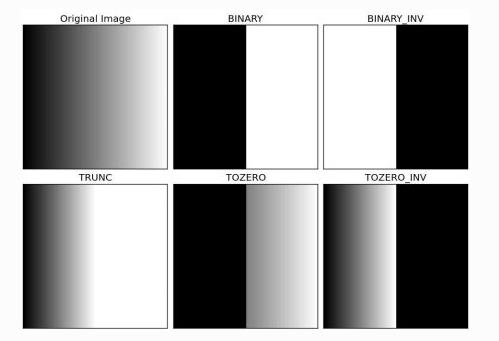
cv2.THRESH\_BINARY

cv2.THRESH\_BINARY\_INV

cv2.THRESH\_TRUNC

cv2.THRESH\_TOZERO

cv2.THRESH\_TOZERO\_INV



Documentation clearly explain what each type is meant for. Please check out the documentation.

Two outputs are obtained. First one is a retval which will be explained later. Second output is our thresholded image.

import cv2

import numpy as np

from matplotlib import pyplot as plt

img = cv2.imread('gradient.png',0)

ret,**thresh1** = cv2.threshold(img,**127,255**,**cv2.THRESH\_BINARY**)

ret,thresh2 = cv2.threshold(img,127,255,cv2.THRESH\_BINARY\_INV)

ret,thresh3 = cv2.threshold(img,127,255,cv2.THRESH\_TRUNC)

ret,thresh4 = cv2.threshold(img,127,255,cv2.THRESH\_TOZERO)

ret,thresh5 = cv2.threshold(img,127,255,cv2.THRESH\_TOZERO\_INV)

titles = ['Original Image','BINARY','BINARY\_INV','TRUNC','TOZERO','TOZERO\_INV']

images = [img, thresh1, thresh2, thresh3, thresh4, thresh5]

for i in xrange(6):

plt.subplot(2,3,i+1),plt.imshow(images[i],'gray')

plt.title(titles[i])

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

plt.show()

**Adaptive Thresholding**

In the previous section, **we used a global value as threshold value**. But it may **not be good in all the conditions where image has different lighting conditions in different areas**. In that case, we go for adaptive thresholding. In this, the **algorithm calculate the threshold for a small regions of the image**. So we get different thresholds for different regions of the same image and it gives us better results for images with varying illumination.

It has three ‘special’ input params and only one output argument.

**Adaptive Method** - It decides how thresholding value is calculated.

cv2.ADAPTIVE\_THRESH\_MEAN\_C : threshold value is the mean of neighbourhood area.

cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C : threshold value is the weighted sum of neighbourhood values where weights are a gaussian window.

**Block Size** - It decides the size of neighbourhood area.

**C** - It is just a constant which is subtracted from the mean or weighted mean calculated.

Below piece of code compares global thresholding and adaptive thresholding for an image with varying illumination:

import cv2

import numpy as np

from matplotlib import pyplot as plt

img = cv2.imread('dave.jpg',0)

img = cv2.medianBlur(img,5)

ret,th1 = cv2.threshold(img,127,255,cv2.THRESH\_BINARY)

th2 = cv2.**adaptiveThreshold**(img,255,cv2.ADAPTIVE\_THRESH\_MEAN\_C,\

cv2.THRESH\_BINARY,11,2)

th3 = cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C,\

cv2.THRESH\_BINARY,11,2)

titles = ['Original Image', 'Global Thresholding (v = 127)',

'Adaptive Mean Thresholding', 'Adaptive Gaussian Thresholding']

images = [img, th1, th2, th3]

for i in xrange(4):

plt.subplot(2,2,i+1),plt.imshow(images[i],'gray')

plt.title(titles[i])

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

plt.show()



Otsu’s Binarization

In the first section, I told you there is a second parameter retVal. Its use comes when we go for Otsu’s Binarization. So what is it?

In global thresholding, we used an arbitrary value for threshold value, right? So, how can we know a value we selected is good or not? Answer is, trial and error method. But consider a bimodal image (In simple words, bimodal image is an image whose histogram has two peaks). For that image, we can approximately take a value in the middle of those peaks as threshold value, right ? That is what Otsu binarization does. So in simple words, it automatically calculates a threshold value from image histogram for a bimodal image. (For images which are not bimodal, binarization won’t be accurate.)

For this, our cv2.threshold() function is used, but pass an extra flag, cv2.THRESH\_OTSU. For threshold value, simply pass zero. Then the algorithm finds the optimal threshold value and returns you as the second output, retVal. If Otsu thresholding is not used, retVal is same as the threshold value you used.

Check out below example. Input image is a noisy image. In first case, I applied global thresholding for a value of 127. In second case, I applied Otsu’s thresholding directly. In third case, I filtered image with a 5x5 gaussian kernel to remove the noise, then applied Otsu thresholding. See how noise filtering improves the result.

import cv2

import numpy as np

from matplotlib import pyplot as plt

img = cv2.imread('noisy2.png',0)

# global thresholding

ret1,th1 = cv2.threshold(img,127,255,cv2.THRESH\_BINARY)

# Otsu's thresholding

ret2,th2 = cv2.threshold(img,0,255,cv2.THRESH\_BINARY+cv2.THRESH\_OTSU)

# Otsu's thresholding after Gaussian filtering

blur = cv2.GaussianBlur(img,(5,5),0)

ret3,th3 = cv2.threshold(blur,0,255,cv2.THRESH\_BINARY+cv2.THRESH\_OTSU)

# plot all the images and their histograms

images = [img, 0, th1,

img, 0, th2,

blur, 0, th3]

titles = ['Original Noisy Image','Histogram','Global Thresholding (v=127)',

'Original Noisy Image','Histogram',"Otsu's Thresholding",

'Gaussian filtered Image','Histogram',"Otsu's Thresholding"]

for i in xrange(3):

plt.subplot(3,3,i\*3+1),plt.imshow(images[i\*3],'gray')

plt.title(titles[i\*3]), plt.xticks([]), plt.yticks([])

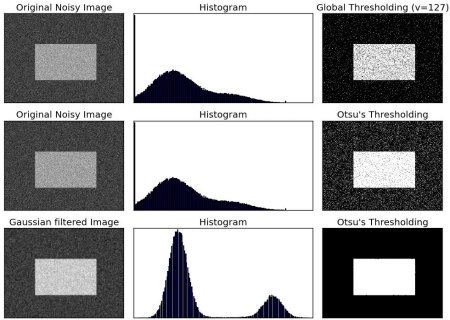
plt.subplot(3,3,i\*3+2),plt.hist(images[i\*3].ravel(),256)

plt.title(titles[i\*3+1]), plt.xticks([]), plt.yticks([])

plt.subplot(3,3,i\*3+3),plt.imshow(images[i\*3+2],'gray')

plt.title(titles[i\*3+2]), plt.xticks([]), plt.yticks([])

plt.show()



# Dilation

<https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_morphological_ops/py_morphological_ops.html>

It is just opposite of erosion. Here, a pixel element is ‘1’ if atleast one pixel under the kernel is ‘1’. So it increases the white region in the image or size of foreground object increases. Normally, in cases like noise removal, erosion is followed by dilation. Because, erosion removes white noises, but it also shrinks our object. So we dilate it. Since noise is gone, they won’t come back, but our object area increases. It is also useful in joining broken parts of an object.

dilation = cv2.dilate(img,kernel,iterations = 1)

# Contours

<https://docs.opencv.org/master/d4/d73/tutorial_py_contours_begin.html>

**Contours can be explained simply as a curve joining all the continuous points (along the boundary), having same color or intensity.** The contours are a useful tool for shape analysis and object detection and recognition.

For better accuracy, **use binary images.** So before finding contours, **apply threshold or canny edge detection**.

In OpenCV, finding contours is like **finding white object from black background. So remember, object to be found should be white and background should be black.**

A **binary image** is one that consists of pixels that can **have one of exactly two colors, usually black and white**. Binary images are also called bi-level or two-level. This means that each pixel is stored as a single bit—i.e., a 0 or 1. The names black-and-white, B&W, monochrome or monochromatic are often used for this concept, but may also designate any images that have only one sample per pixel, such as grayscale images.

#Binary thresholding and inverting at 127

th, **threshed** = cv2.threshold(gray, 127, 255, cv2.THRESH\_BINARY\_INV|cv2.THRESH\_OTSU)

Binary images often arise in [digital image processing](https://en.wikipedia.org/wiki/Digital_image_processing) as [masks](https://en.wikipedia.org/wiki/Mask_(computing)#Image_masks) or [thresholding](https://en.wikipedia.org/wiki/Thresholding_(image_processing)), and [dithering](https://en.wikipedia.org/wiki/Dither). Some input/output devices, such as [laser printers](https://en.wikipedia.org/wiki/Laser_printer), [fax machines](https://en.wikipedia.org/wiki/Fax), and bilevel [computer displays](https://en.wikipedia.org/wiki/Visual_display_unit), can only handle bilevel images.

Let's see how to find **contours of a binary image**:

import numpy as np

import cv2 as cv

im = cv.imread('test.jpg')

imgray = cv.cvtColor(im, cv.COLOR\_BGR2GRAY)

ret, thresh = cv.**threshold**(imgray, 127, 255, 0)

contours, hierarchy = cv.**findContours**(thresh, cv.RETR\_TREE, cv.CHAIN\_APPROX\_SIMPLE)

See, there are three arguments in cv.**findContours**() function, **first one is source image, second is contour retrieval mode, third is contour approximation method**. And it **outputs the contours and hierarchy**. Contours is a Python list of all the contours in the image. **Each individual contour is a Numpy array of (x,y) coordinates of boundary points of the object**.

To **draw the contours**, cv.drawContours function is used. It can also be used to draw any shape provided you have its boundary points. Its first argument is source image, second argument is the contours which should be passed as a Python list, third argument is index of contours (useful when drawing individual contour. To draw all contours, pass -1) and remaining arguments are color, thickness etc.

**cv.DrawContours(img, contours, contourIdx, colour, thickness)**

RGB red (255,0,0)

To draw all the contours in an image:

cv.drawContours(img, contours, -1, (0,255,0), 3)

To draw an individual contour, say 4th contour:

cv.drawContours(img, contours, 3, (0,255,0), 3)

But most of the time, below method will be useful:

cnt = contours[4]

cv.drawContours(img, [cnt], 0, (0,255,0), 3)

Note

Last two methods are same, but when you go forward, you will see last one is more useful.

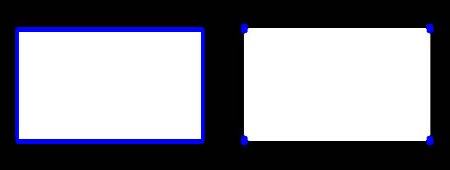
**Contour Approximation Method**

This is the third argument in cv.findContours function. What does it denote actually?

Above, we told that contours are the boundaries of a shape with same intensity. It stores the (x,y) coordinates of the boundary of a shape. **But does it store all the coordinates** ? That is specified by this contour approximation method.

If you pass cv.CHAIN\_APPROX\_NONE, all the boundary points are stored. But actually do we need all the points? For eg, you found the contour of a straight line. Do you need all the points on the line to represent that line? No, **we need just two end points of that line. This is what cv.CHAIN\_APPROX\_SIMPLE does. It removes all redundant points and compresses the contour, thereby saving memory.**

Below image of a rectangle demonstrate this technique. Just draw a circle on all the coordinates in the contour array (drawn in blue color). First image shows points I got with cv.CHAIN\_APPROX\_NONE (734 points) and second image shows the one with cv.CHAIN\_APPROX\_SIMPLE (only 4 points). See, how much memory it saves!!!

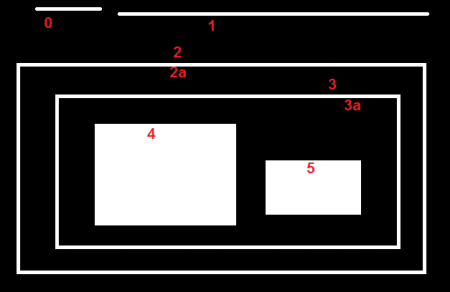


## Hierarchy

<https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_contours/py_contours_hierarchy/py_contours_hierarchy.html#contours-hierarchy>

using cv2.findContours() function, we have passed an argument, **Contour Retrieval Mode**. We usually passed cv2.RETR\_LIST or cv2.RETR\_TREE and it worked nice. But what does it actually mean ?

But in some cases, some shapes are inside other shapes. Just like nested figures. In this case, we call outer one as **parent** and inner one as **child**. This way, contours in an image has some relationship to each other.



In this image, there are a few shapes which I have numbered from **0-5**. **2 and 2a denotes the external and internal contours of the outermost box.**

Here, contours **0,1,2 are external or outermost**. We can say, they are in hierarchy-0 or simply they are in **same hierarchy level.**

Next comes contour-**2a. It can be considered as a child of contour-**2 (or in opposite way, contour-2 is parent of contour-2a). **So let it be in hierarchy-1**. Similarly contour-3 is child of contour-2 and it comes in next hierarchy. Finally contours 4,5 are the children of contour-3a, and they come in the last hierarchy level. From the way I numbered the boxes, I would say contour-4 is the first child of contour-3a (It can be contour-5 also).

So each contour has its own information regarding what hierarchy it is, who is its child, who is its parent etc. OpenCV represents it as an array of four values : **[Next, Previous, First\_Child, Parent]**

“**Next denotes next contour at the same hierarchical level**.”

For eg, take contour-0 in our picture. Who is next contour in its same level ? **It is contour-1.** So simply **put Next = 1**. Similarly for Contour-1, next is contour-2. So Next = 2.

What about contour-2? There is **no next contour in the same level. So simply, put Next = -1.** What about contour-4? It is in same level with contour-5. So its next contour is contour-5, so Next = 5.

“**Previous denotes previous contour at the same hierarchical level.”**

It is **same as above.** Previous contour of contour-1 is contour-0 in the same level. Similarly for contour-2, it is contour-1. And for contour-0, there is no previous, so put it as -1.

“**First\_Child denotes its first child contour.”**

There is no need of any explanation. For **contour-2, child is contour-2a**. So it gets the corresponding index value of contour-2a. What about contour-3a? It has two children. But we take only first child. And it is contour-4. So First\_Child = 4 for contour-3a.

**“Parent denotes index of its parent contour.”**

It is just **opposite of First\_Child.** Both for contour-4 and contour-5, parent contour is contour-3a. For contour-3a, it is contour-3 and so on.

Note: **If there is no child or parent, that field is taken as -1**

**Contour Retrieval Mode**

**1. RETR\_LIST**

This is the simplest of the four flags (from explanation point of view). It simply retrieves all the contours, but **doesn’t create any parent-child relationship.**

So here, 3rd and 4th term in hierarchy array is always -1. But obviously**, Next and Previous** terms will have their corresponding values. Just check it yourself and verify it.

>>> hierarchy

array([[[ 1, -1, -1, -1],

[ 2, 0, -1, -1],

[ 3, 1, -1, -1],

[ 4, 2, -1, -1],

[ 5, 3, -1, -1],

[ 6, 4, -1, -1],

[ 7, 5, -1, -1],

[-1, 6, -1, -1]]])

**2. RETR\_EXTERNAL**

If you use this flag, it **returns only extreme outer flags. All child contours are left behind**.

So, in our image, how many extreme outer contours are there? ie at hierarchy-0 level?. Only 3, ie contours 0,1,2, right? Now try to find the contours using this flag. Here also, values given to each element is same as above. Compare it with above result. Below is what I got :

>>> hierarchy

array([[[ 1, -1, -1, -1],

[ 2, 0, -1, -1],

[-1, 1, -1, -1]]])

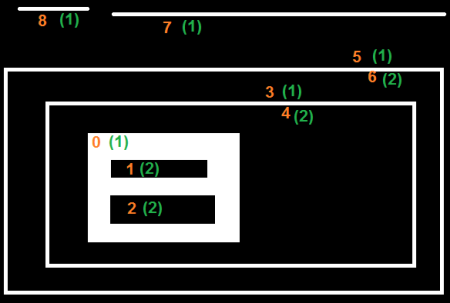
You can use this flag **if you want to extract only the outer contours**. It might be useful in some cases.

### 3. RETR\_CCOMP

This flag retrieves all the contours and arranges them to a 2-level hierarchy. ie external contours of the object (ie its boundary) are placed in hierarchy-1. And the contours of holes inside object (if any) is placed in hierarchy-2. If any object inside it, its contour is placed again in hierarchy-1 only. And its hole in hierarchy-2 and so on.

Just consider the image of a “big white zero” on a black background. Outer circle of zero belongs to first hierarchy, and inner circle of zero belongs to second hierarchy.

We can explain it with a simple image. Here I have labelled the order of contours in red color and the hierarchy they belongs to, in green color (either 1 or 2). The order is same as the order OpenCV detects contours.



So consider first contour, ie contour-0. It is hierarchy-1. It has two holes, contours 1&2, and they belong to hierarchy-2. So for contour-0, Next contour in same hierarchy level is contour-3. And there is no previous one. And its first is child is contour-1 in hierarchy-2. It has no parent, because it is in hierarchy-1. So its hierarchy array is [3,-1,1,-1]

Now take contour-1. It is in hierarchy-2. Next one in same hierarchy (under the parenthood of contour-1) is contour-2. No previous one. No child, but parent is contour-0. So array is [2,-1,-1,0].

Similarly contour-2 : It is in hierarchy-2. There is not next contour in same hierarchy under contour-0. So no Next. Previous is contour-1. No child, parent is contour-0. So array is [-1,1,-1,0].

Contour - 3 : Next in hierarchy-1 is contour-5. Previous is contour-0. Child is contour-4 and no parent. So array is [5,0,4,-1].

Contour - 4 : It is in hierarchy 2 under contour-3 and it has no sibling. So no next, no previous, no child, parent is contour-3. So array is [-1,-1,-1,3].

Remaining you can fill up. This is the final answer I got:

>>> hierarchy

array([[[ 3, -1, 1, -1],

[ 2, -1, -1, 0],

[-1, 1, -1, 0],

[ 5, 0, 4, -1],

[-1, -1, -1, 3],

[ 7, 3, 6, -1],

[-1, -1, -1, 5],

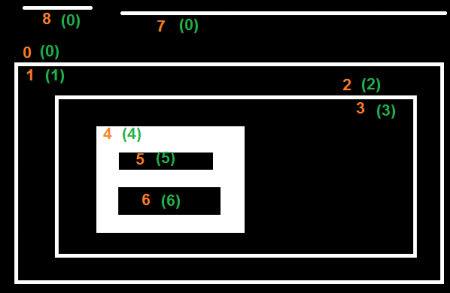
[ 8, 5, -1, -1],

[-1, 7, -1, -1]]])

### 4. RETR\_TREE

And this is the final guy, Mr.Perfect. It retrieves all the contours and creates a full family hierarchy list. **It even tells, who is the grandpa, father, son, grandson and even beyond... :)**.

For examle, I took above image, rewrite the code for cv2.RETR\_TREE, reorder the contours as per the result given by OpenCV and analyze it. Again, red letters give the contour number and green letters give the hierarchy order.



Take contour-0 : It is in hierarchy-0. Next contour in same hierarchy is contour-7. No previous contours. Child is contour-1. And no parent. So array is [7,-1,1,-1].

Take contour-2 : It is in hierarchy-1. No contour in same level. No previous one. Child is contour-2. Parent is contour-0. So array is [-1,-1,2,0].

And remaining, try yourself. Below is the full answer:

>>> hierarchy

array([[[ 7, -1, 1, -1],

[-1, -1, 2, 0],

[-1, -1, 3, 1],

[-1, -1, 4, 2],

[-1, -1, 5, 3],

[ 6, -1, -1, 4],

[-1, 5, -1, 4],

[ 8, 0, -1, -1],

[-1, 7, -1, -1]]])

# Find lines via hist

via histogram of line pixel values (averaged) -> only >> o if there is content

##find and draw the upper and lower boundary of each lines

hist = cv2.reduce(dil\_cleaned\_img,1, cv2.REDUCE\_AVG).reshape(-1)

mtx Source 2D matrix.

vec Destination vector. Its size and type is defined by dim and dtype parameters.

dim Dimension index along which the matrix is reduced. 0 means that the matrix is reduced to a single row. **1 means that the matrix is reduced to a single column.**

reduceOp Reduction operation that could be one of the following:

CV\_REDUCE\_SUM The output is the sum of all rows/columns of the matrix.

**CV\_REDUCE\_AVG The output is the mean vector of all rows/columns of the matrix**.

CV\_REDUCE\_MAX The output is the maximum (column/row-wise) of all rows/columns of the matrix.

CV\_REDUCE\_MIN The output is the minimum (column/row-wise) of all rows/columns of the matrix.

dtype When it is negative, the destination vector will have the same type as the source matrix. Otherwise, its type will be CV\_MAKE\_TYPE(CV\_MAT\_DEPTH(dtype), mtx.channels()) .

stream Stream for the asynchronous version.

# MNIST and other datasets

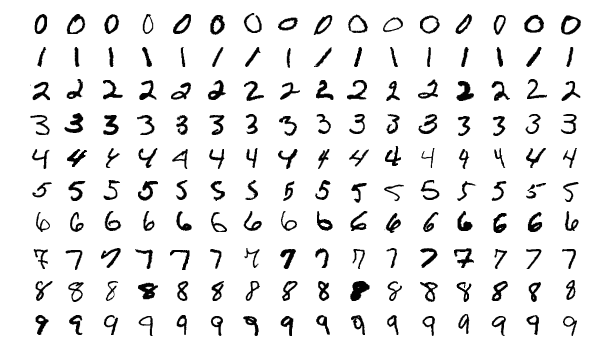
## MNIST

<http://yann.lecun.com/exdb/mnist/>

The original black and white (bilevel) images from NIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. **the images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field**.

<https://towardsdatascience.com/handwritten-digit-mnist-pytorch-977b5338e627>

It is a collection of 70000 handwritten digits split **into training and test set of 60000 and 10000 images respectively**.



Source: Wikimedia

## Kaggle

<https://www.kaggle.com/xainano/handwrittenmathsymbols>

Dataset consists of jpg files(45x45)

mnist has 70000 pics for 10 digits -> 7000 images per digit

* aspect ratio 1 with 28x28 pixels for all images

# cv transformations

scaling (20x20 center)

Scaling

Scaling is just resizing of the image. OpenCV comes with a function cv2.resize() for this purpose. The size of the image can be specified manually, or you can specify the scaling factor. Different interpolation methods are used. Preferable interpolation methods are cv2.INTER\_AREA for shrinking and cv2.INTER\_CUBIC (slow) & cv2.INTER\_LINEAR for zooming. By default, interpolation method used is cv2.INTER\_LINEAR for all resizing purposes. You can resize an input image either of following methods:

import cv2

import numpy as np

img = cv2.imread('messi5.jpg')

res = cv2.resize(img,None,fx=2, fy=2, interpolation = cv2.INTER\_CUBIC)

#OR

height, width = img.shape[:2]

res = cv2.resize(img,(2\*width, 2\*height), interpolation = cv2.INTER\_CUBIC)

# pytorch

## dataset loading

Compose transforms

Now, we apply the transforms on a sample.

Let’s say we want to rescale the shorter side of the image to 256 and then randomly crop a square of size 224 from it. i.e, we want to compose Rescale and RandomCrop transforms. torchvision.transforms.Compose is a simple callable class which allows us to do this.

all transformations : <https://pytorch.org/docs/stable/torchvision/transforms.html>

transform = **transforms.Compose**([transforms.ToTensor(),

transforms.Normalize((0.5,), (0.5,)),

])

The ToTensor operation in PyTorch convert all tensors to lie between (0, 1).

# Plot image

# path

path = r'C:\Users\Rajnish\Desktop\geeksforgeeks.png'

# Reading an image in default mode

image = cv2.imread(path)

# Window name in which image is displayed

window\_name = 'image'

# Using cv2.imshow() method

# Displaying the image

cv2.imshow(window\_name, image)

#waits for user to press any key

#(**this is necessary to avoid Python kernel form crashing)**

cv2.waitKey(0)

#closing all open windows

cv2.destroyAllWindows()

waitKey(0) method is waiting for an input infinitely. When you see a frame of the corresponding image, do not try to close the image using close in top right corner.

Instead press some key. waitkey method will take that as an input and it will return back a value. Further you can also check which key was pressed to close the frame.

Additionally waitKey(33) will keep the frame active for 33 ms and then close it automatically.

destroyWindow() will destroy the current frame if there. destroyAllWindows() will destroy all the frames currently present.