1.3.1 Basic Operations on Images

Goal

Learn to:

- Access pixel values and modify them
- Access image properties
- Setting Region of Image (ROI)
- Splitting and Merging images

Almost all the operations in this section is mainly related to Numpy rather than OpenCV. A good knowledge of Numpy is required to write better optimized code with OpenCV.

(Examples will be shown in Python terminal since most of them are just single line codes)

Accessing and Modifying pixel values

Let's load a color image first:

```
>>> import cv2
>>> import numpy as np

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

You can access a pixel value by its row and column coordinates. For BGR image, it returns an array of Blue, Green, Red values. For grayscale image, just corresponding intensity is returned.

```
>>> px = img[100,100]
>>> print px
[157 166 200]

# accessing only blue pixel
>>> blue = img[100,100,0]
>>> print blue
157
```

You can modify the pixel values the same way.

```
>>> img[100,100] = [255,255,255]
>>> print img[100,100]
[255 255 255]
```

Warning: Numpy is a optimized library for fast array calculations. So simply accessing each and every pixel values and modifying it will be very slow and it is discouraged.

Note: Above mentioned method is normally used for selecting a region of array, say first 5 rows and last 3 columns like that. For individual pixel access, Numpy array methods, `array.item()` and `array.itemset()` is considered to be better. But it always returns a scalar. So if you want to access all B,G,R values, you need to call `array.item()` separately for all.

Better pixel accessing and editing method :

```
# accessing RED value
>>> img.item(10,10,2)
59

# modifying RED value
>>> img.itemset((10,10,2),100)
>>> img.item(10,10,2)
100
```

Accessing Image Properties

Image properties include number of rows, columns and channels, type of image data, number of pixels etc.

Shape of image is accessed by `img.shape`. It returns a tuple of number of rows, columns and channels (if image is color):

```
>>> print img.shape
(342, 548, 3)
```

Note: If image is grayscale, tuple returned contains only number of rows and columns. So it is a good method to check if loaded image is grayscale or color image.

Total number of pixels is accessed by `img.size`:

```
>>> print img.size
562248
```

Image datatype is obtained by `img.dtype`:

```
>>> print img.dtype
uint8
```

Note: `img.dtype` is very important while debugging because a large number of errors in OpenCV-Python code is caused by invalid datatype.

Image ROI

Sometimes, you will have to play with certain region of images. For eye detection in images, first perform face detection over the image until the face is found, then search within the face region for eyes. This approach improves accuracy (because eyes are always on faces :D) and performance (because we search for a small area).

ROI is again obtained using Numpy indexing. Here I am selecting the ball and copying it to another region in the image:

```
>>> ball = img[280:340, 330:390]
>>> img[273:333, 100:160] = ball
```

Check the results below:

Splitting and Merging Image Channels

The B,G,R channels of an image can be split into their individual planes when needed. Then, the individual channels can be merged back together to form a BGR image again. This can be performed by:

```
>>> b,g,r = cv2.split(img)
>>> img = cv2.merge((b,g,r))
```

Or

```
>>> b = img[:,:,0]
```

Suppose, you want to make all the red pixels to zero, you need not split like this and put it equal to zero. You can simply use Numpy indexing which is faster.

```
>>> img[:,:,2] = 0
```

Warning: `cv2.split()` is a costly operation (in terms of time), so only use it if necessary. Numpy indexing is much more efficient and should be used if possible.

Making Borders for Images (Padding)

If you want to create a border around the image, something like a photo frame, you can use **cv2.copyMakeBorder()** function. But it has more applications for convolution operation, zero padding etc. This function takes following arguments:

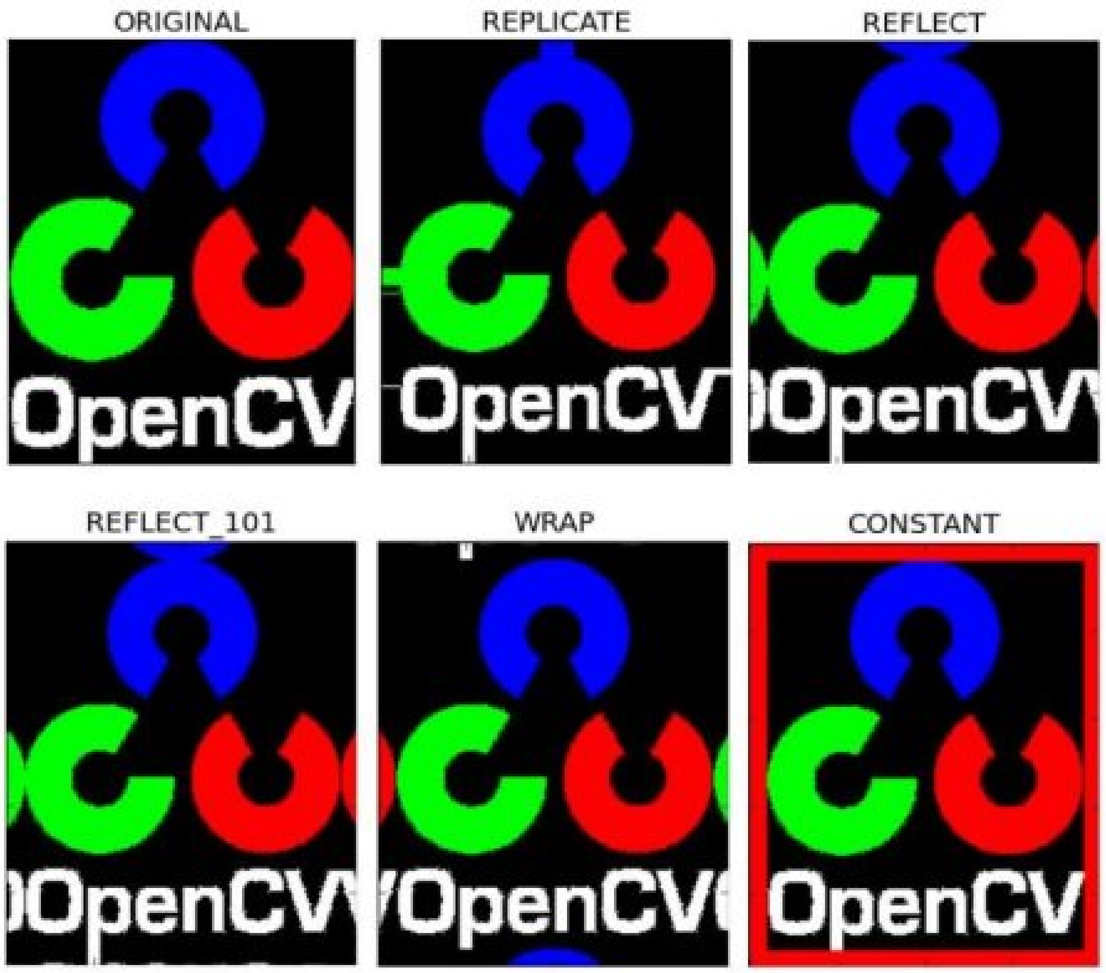
- **src** - input image
- **top**, **bottom**, **left**, **right** - border width in number of pixels in corresponding directions
- **borderType - Flag defining what kind of border to be added. It can be following types:**

  - **cv2.BORDER_CONSTANT** - Adds a constant colored border. The value should be given as next argument.
  - **cv2.BORDER_REFLECT** - Border will be mirror reflection of the border elements, like this : *fedcba|abcdefgh|hgfedcb*
  - **cv2.BORDER_REFLECT_101** or **cv2.BORDER_DEFAULT** - Same as above, but with a slight change, like this : *gfedcb|abcdefgh|gfedcba*
  - **cv2.BORDER_REPLICATE** - Last element is replicated throughout, like this: *aaaaaa|abcdefgh|hhhhhhh*
  - **cv2.BORDER_WRAP** - Can't explain, it will look like this : *cdefgh|abcdefgh|abcdefg*
- **value** - Color of border if border type is `cv2.BORDER_CONSTANT`

Below is a sample code demonstrating all these border types for better understanding:

```python
import cv2
import numpy as np
from matplotlib import pyplot as plt

BLUE = [255,0,0]

img1 = cv2.imread('opencv_logo.png')

replicate = cv2.copyMakeBorder(img1,10,10,10,10,cv2.BORDER_REPLICATE)
reflect = cv2.copyMakeBorder(img1,10,10,10,10,cv2.BORDER_REFLECT)
reflect101 = cv2.copyMakeBorder(img1,10,10,10,10,cv2.BORDER_REFLECT_101)
wrap = cv2.copyMakeBorder(img1,10,10,10,10,cv2.BORDER_WRAP)
constant= cv2.copyMakeBorder(img1,10,10,10,10,cv2.BORDER_CONSTANT,value=BLUE)

plt.subplot(231),plt.imshow(img1,'gray'),plt.title('ORIGINAL')
plt.subplot(232),plt.imshow(replicate,'gray'),plt.title('REPLICATE')
plt.subplot(233),plt.imshow(reflect,'gray'),plt.title('REFLECT')
plt.subplot(234),plt.imshow(reflect101,'gray'),plt.title('REFLECT_101')
plt.subplot(235),plt.imshow(wrap,'gray'),plt.title('WRAP')
plt.subplot(236),plt.imshow(constant,'gray'),plt.title('CONSTANT')

plt.show()
```

See the result below. (Image is displayed with matplotlib. So RED and BLUE planes will be interchanged):



Additional Resources

Exercises

1.3.2 Arithmetic Operations on Images

Goal

- Learn several arithmetic operations on images like addition, subtraction, bitwise operations etc.
- You will learn these functions : **cv2.add()**, **cv2.addWeighted()** etc.

Image Addition

You can add two images by OpenCV function, `cv2.add()` or simply by numpy operation, `res = img1 + img2`. Both images should be of same depth and type, or second image can just be a scalar value.

Note: There is a difference between OpenCV addition and Numpy addition. OpenCV addition is a saturated operation while Numpy addition is a modulo operation.

For example, consider below sample:

```
>>> x = np.uint8([250])
>>> y = np.uint8([10])

>>> print cv2.add(x,y) # 250+10 = 260 => 255
[[255]]

>>> print x+y          # 250+10 = 260 % 256 = 4
[4]
```

It will be more visible when you add two images. OpenCV function will provide a better result. So always better stick to OpenCV functions.

Image Blending

This is also image addition, but different weights are given to images so that it gives a feeling of blending or transparency. Images are added as per the equation below:

$$g(x) = (1 - \alpha)f_0(x) + \alpha f_1(x)$$

By varying α from $0 \rightarrow 1$, you can perform a cool transition between one image to another.

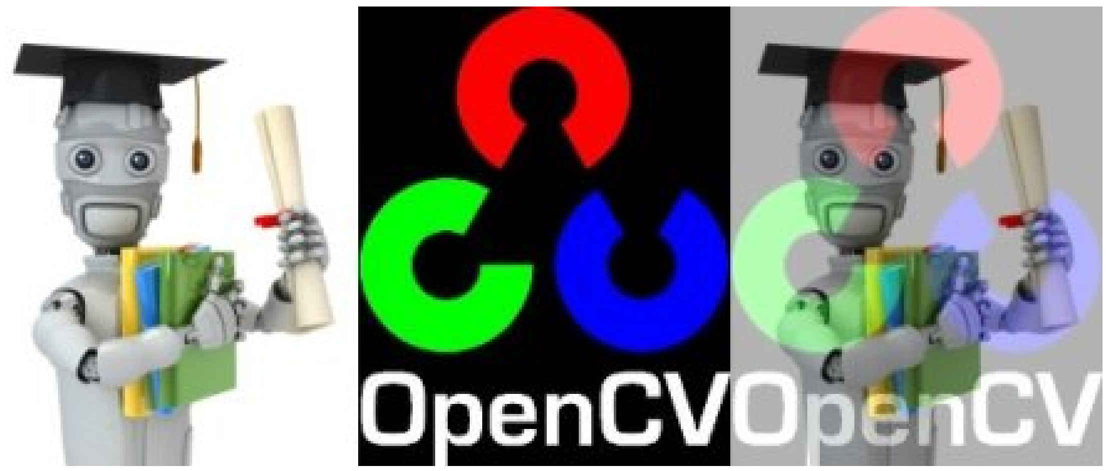
Here I took two images to blend them together. First image is given a weight of 0.7 and second image is given 0.3. `cv2.addWeighted()` applies following equation on the image.

$$dst = \alpha \cdot img1 + \beta \cdot img2 + \gamma$$

Here γ is taken as zero.

```
img1 = cv2.imread('ml.png')
img2 = cv2.imread('opencv_logo.jpg')

dst = cv2.addWeighted(img1,0.7,img2,0.3,0)

cv2.imshow('dst',dst)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Check the result below:



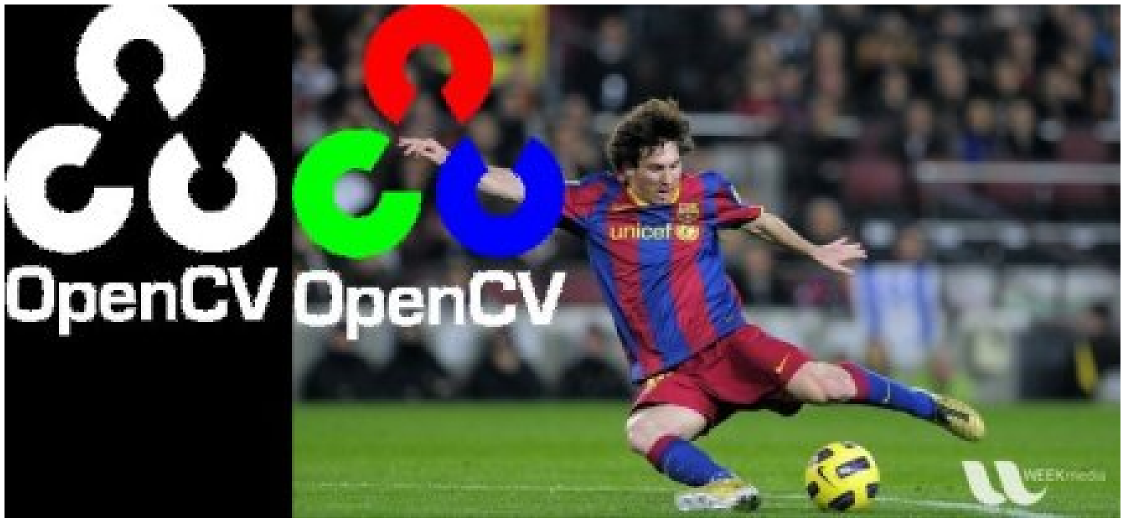
Bitwise Operations

This includes bitwise AND, OR, NOT and XOR operations. They will be highly useful while extracting any part of the image (as we will see in coming chapters), defining and working with non-rectangular ROI etc. Below we will see an example on how to change a particular region of an image.

I want to put OpenCV logo above an image. If I add two images, it will change color. If I blend it, I get an transparent effect. But I want it to be opaque. If it was a rectangular region, I could use ROI as we did in last chapter. But OpenCV logo is a not a rectangular shape. So you can do it with bitwise operations as below:

```
# Load two images
img1 = cv2.imread('messi5.jpg')
img2 = cv2.imread('opencv_logo.png')

# I want to put logo on top-left corner, So I create a ROI
rows,cols,channels = img2.shape
roi = img1[0:rows, 0:cols ]

# Now create a mask of logo and create its inverse mask also
img2gray = cv2.cvtColor(img2,cv2.COLOR_BGR2GRAY)
ret, mask = cv2.threshold(img2gray, 10, 255, cv2.THRESH_BINARY)
mask_inv = cv2.bitwise_not(mask)

# Now black-out the area of logo in ROI
img1_bg = cv2.bitwise_and(roi,roi,mask = mask_inv)

# Take only region of logo from logo image.
img2_fg = cv2.bitwise_and(img2,img2,mask = mask)

# Put logo in ROI and modify the main image
dst = cv2.add(img1_bg,img2_fg)
img1[0:rows, 0:cols ] = dst

cv2.imshow('res',img1)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

See the result below. Left image shows the mask we created. Right image shows the final result. For more understanding, display all the intermediate images in the above code, especially `img1_bg` and `img2_fg`.



Additional Resources

Exercises

1. Create a slide show of images in a folder with smooth transition between images using `cv2.addWeighted` function

1.3.3 Performance Measurement and Improvement Techniques

Goal

In image processing, since you are dealing with large number of operations per second, it is mandatory that your code is not only providing the correct solution, but also in the fastest manner. So in this chapter, you will learn

- To measure the performance of your code.
- Some tips to improve the performance of your code.
- You will see these functions : **cv2.getTickCount**, **cv2.getTickFrequency** etc.

Apart from OpenCV, Python also provides a module **time** which is helpful in measuring the time of execution. Another module **profile** helps to get detailed report on the code, like how much time each function in the code took, how many times the function was called etc. But, if you are using IPython, all these features are integrated in an user-friendly manner. We will see some important ones, and for more details, check links in **Additional Resouces** section.

Measuring Performance with OpenCV

cv2.getTickCount function returns the number of clock-cycles after a reference event (like the moment machine was switched ON) to the moment this function is called. So if you call it before and after the function execution, you get number of clock-cycles used to execute a function.

cv2.getTickFrequency function returns the frequency of clock-cycles, or the number of clock-cycles per second. So to find the time of execution in seconds, you can do following:

```
e1 = cv2.getTickCount()
# your code execution
e2 = cv2.getTickCount()
time = (e2 - e1)/ cv2.getTickFrequency()
```

We will demonstrate with following example. Following example apply median filtering with a kernel of odd size ranging from 5 to 49. (Don't worry about what will the result look like, that is not our goal):

```
img1 = cv2.imread('messi5.jpg')

e1 = cv2.getTickCount()
for i in xrange(5,49,2):
    img1 = cv2.medianBlur(img1,i)
e2 = cv2.getTickCount()
t = (e2 - e1)/cv2.getTickFrequency()
print t

# Result I got is 0.521107655 seconds
```

Note: You can do the same with `time` module. Instead of `cv2.getTickCount`, use `time.time()` function. Then take the difference of two times.