Visual Computing

Lab Exercises

Samuel Silva

November 21, 2024

Contents

6 Image Processing Pixels

In this hands-on we will be performing a first exploration of image processing using OpenCV in Pytho covering:

- First OpenCV examples.
- Image operations; reading and displaying images with different formats, direct pixel manipulation.
- Image subtraction
- Histograms, contrast stretching, and histogram equalization
- Mean, median, and Gaussian filtering
- Image resizing

6.1 Support Documentation and Tutorials

While doing this hands-on and if you want to further explore the different methods, you may find the following links useful:

The OpenCV documentation is available at:

`http://docs.opencv.org`

OpenCV-Python Tutorials are available everywhere, but a good source of examples can be:

`https://www.geeksforgeeks.org/opencv-python-tutorial/`

6.2 Setting the Mood, Honey!

For this hands-on, you need to start a new Python project in your editor of choice and you need to install package `opencv-python`.

To be able to use OpenCV in your script, you just need to have the following import:

```
import cv2
```

6.3 Image Basics

Loading the Gu.. Image

OpenCV supports a wide range of features around images starting by being able to read them. To read an image from disk, just add this line to your script:

```
img = cv2.imread("lena.jpg")
```

It is possible to obtain details about the image properties such as its dimension. For instance, to obtain height and width, image size, and data type:

```
height, width = img.shape[:2]
imgSize = img.size
imgDType = img.dtype
```

The `[:2]` index is to ensure we are only getting the two first dimensions. If we are loading a greylevel image, it would be expected that it would have 2 dimensions, but, by default, OpenCV always loads three color channels (by replicating the greylevel channel), unless if specifically configured to not do so. If you remove the [:2] part, the script will not run. One good practise to avoid confusion is to ask for it not to mess around with our stuff.

```
img = cv2.imread("lena.jpg", cv2.IMREAD_UNCHANGED)
```

You can print the values to the console to be able to examine them.

Out of sigh, Out of Mind

Loading images and processing them is all very nice, but we often need to visualize those images to inspect its contents or the outcomes of our sophisticated processing techniques. To view a loaded image, just add the following code:

```
cv2.imshow('My Precious', img)
cv2.waitKey(0)
```

The first line provides a title for the window and the variable where the image is. The second line is just for the script to halt waiting for a key so that you can see the image window. When the script ends, the window is destroyed. So, we need this line to pause execution.

You can modify the code to read any image you want.

Direct pixel manipulation

Now, let us keep with Lena. Read it from the image file and make a copy that we will use for some experiments. To this effect you can use the following line:

```
img2 = img.copy()
```

Why not make just `img2 = img`? Just try and see. If you cannot understand why, just ask.

Access the pixel intensity values of the image copy; set to 0 every pixel of the copy image whose intensity value is less than 128 in the original image.

You can access a given pixel as an element in an array [i,j]. To go along the image array, you can just use a `for` cycle, e.g.:

```
for i in range(width):
    for j in range(height):
            ...
```

Check the results.

Image subtraction

Now, imagine that you wanted to check the difference between the two images. When the changes are strong, it is easy to detect, as in what you just did in the previous task, but when they are subtle, it is harder to notice it. Modify the previous example to make just a slight change to the values between 100 and 180 by subtracting 30 to each pixel.

One possible approach can be to subtract one from the other and see where they differ. So, subtract the original image to the image you just modified. To do so:

```
imgDiff = cv2.subtract(img2, img)
```

Visualize the result. Now invert the order of the image, in the subtraction and visualize it, again. Did it give you the same result? What happened?

We need to be careful, since the OpenCV `subtract` operation is saturated. It means that when it reaches the pixel value boundaries, it will clamp the result. For instance, if the different is negative, it will be clamped to zero, hence, the disappearance. This is very useful when we are subtracting images to remove parts, e.g., the background, because the resulting image is well behaved. But when actually wanting to check for differences, it may be misleading. But there is a method to check the absolute difference:

```
imgDiff = cv2.absdiff(img, img2)
```

Drawing over the image

Considering the image you modified, can you draw a grid over the image with a spacing of 20 pixels? Use the same approach you used for the previous task. Now visualize the result of the subtractions, again.

Another way of drawing lines over an image is by using the method `line`. Draw the diagonals in white.

```
cv2.line(img2, (startX, startY), (endX, endY), (color))
# example:
cv2.line(img2, (0,0), (width//2, height//2), (128))
```

Circles are also quite easy to draw with the method circle, for instance:

```
cv2.circle(img2, (100, 100), 25, (255), cv2.FILLED)
# you can use cv2.LINE_4, or cv2.LINE_8, also
```

Thresholding

Details on thresholding here

And before we end this part regarding some basics, we will experiment a little bit with thresholding. There is a specific method for it, on OpenCV:

```
ret,imgThresh = cv2.threshold(img2,127,255,cv.THRESH_BINARY)
```

Experiment with the different parameters and with the different possible operation types: `THRESH_BINARY`, `THRESH_BINARY_INV`, `THRESH_TRUNC`, `THRESH_TOZERO` and `THRESH_TOZERO_INV`.

Saving your work for posterity

Finally, you can take the abstract masterpiece you just created and write to a file using:

```
cv2.imwrite(filename, img2)
```

6.4 History and Equalization

Build and visualize the image histogram

Compile and test the file **Part_02_exe_04.py**

Analyze the code, in particular the following steps:

1. Defining the features and computing the image histogram.
2. Computing image features from the histogram.
3. Creating and displaying an image representing the histogram.

The image displaying the histogram can be created using `matplotlib`. As an alternative, OpenCV drawing functions can also be used. Observe what happens when some histogram features are changed: for instance, size (look for **histSize**) and range of values (**histRange**).

Analyzing the histograms of different images

For some of the example images given, analyze their histograms. In particular, analyze the different features of the image histograms for the image set ireland-06-* and classify each one of those images.

Contrast-Stretching

Expand the previous code to allow applying the Contrast-Stretching operation to a given gray-level image.

The original image and the resulting image should be visualized, as well as the respective histograms. To accomplish that create a function that receives an image and returns a new one with the contrast adjusted.

```
def contrastStretch(img):
    ...
    ...
    return newImg
```

To do that:

1. Use the `minMaxLoc` function to determine the smallest and largest image intensity values. It returns 4 values: `min, max, indexMin, indexMax = cv2.minMaxLoc(img)`
2. Create a new image that uses the entire range of intensity values (from 0 to 255).

For each image pixel, the intensity of the corresponding pixel in the resulting image is given by:

$final[x, y] = \frac{original[x,y]-min}{max-min} \times 255$

Apply the Contrast-Stretching operation to the *DETI.bmp* image and the input.png image. Visualize the histograms of the different images. What differences do you notice?

Histogram Equalization

And, finally, expand the previous example to apply Histogram-Equalization to a given gray-level image, using the `cv2.equalizeHist` function. The original image and the resulting image should be visualized, as well as the respective histograms.

Apply the Histogram-Equalization operation to the *TAC_PULMAO.bmp* image.

What is the difference between the histograms of the original image and the equalized image?

What does the Histogram-Equalization operation allow?

6.5 Image Filters

Averaging Filters

Compile and test the file `Part_03_exe_01.py`. Analyze the code and verify how an averaging filter is applied using the function:

```
cv2.blur(src, ksize[, dst[, anchor[, borderType]]])     dst
```

Write additional code allowing to:

- Apply (5×5) and (7×7) averaging filters to a given image.
- Apply successively (e.g., 3 times) the same filter to the resulting image.
- Visualize the result of the successive operations.

Test the developed operations using the *Lena_Ruido.png* and *DETI_-Ruido.png* images.

Analyze the effects of applying different averaging filters to various images and compare the resulting images among themselves and with the original image. Use the following test images: *fce5noi3.bmp*, *fce5noi4.bmp*, *fce5noi6.bmp*, *sta2.bmp*, *sta2noi1.bmp*.

Median Filters

Similarly to the previous example, apply median filters to a given image. Use the function:

```
cv2.medianBlur(src, ksize[, dst])
```

Test the developed operations using the *Lena_Ruido.png* and *DETI_-Ruido.png* images.

Use the developed code to analyze the effects of applying different median filters to various images, and to compare the resulting images among themselves and with the original image, as well as with the results of applying averaging filters. Use the same test images as before.

Gaussian Filters

Finally, experiment with applying Gaussian filters to a given image.

Use the function:

```
cv2.GaussianBlur(src, ksize, sigmaX[, dst[, sigmaY[, borderType
    ]]])
```

Test the developed operations using the *Lena_Ruido.png* and *DETI_-Ruido.png* images.

Compare the results with those obtained using average and median filters. Use the same test images as before.

6.6 Pixelaaaaaate!

There is a very simple technique to pixelate an image which consists in resizing it to a small size and, then, getting it back to its original size. Can you explain why these steps leave the image with a pixelated look?

Resizing an image

Open file `Part_04_ex_01.py`. It already has some auxiliary functions that we will use later. Do not worry with them. You will not need to understand them in detail. They are just there to provide the final touch to the task. You will probably need to install some extra python packages: check for `scikit-learn` and `scikit-image`.

First, start by opening image *paddington.png* and obtaining its dimensions (remember the first task in this hands-on).

Now, we will establish a size to which to downsize our image, the first step towards pixelation. Let us define variables w and h:

```
w,h = (32, 32)
```

These values are a first guess and can be changed later to check if there are best choices. And, here comes resizing:

```
tmpImg = cv2.resize(img,(w,h), interpolation=cv2.INTER_LINEAR)
```

Now, let's restore the image to its original size:

```
dst = cv2.resize(tmpImg, (width, height), interpolation=cv2.
    INTER_NEAREST)
```

if you visualize the image, you can see how it went. Also visualize the original. Can you understand why we do not use `cv2.INTER_LINEAR` or `v2.INTER_NEAREST` for both resizings? Exchange them and see what happens.

One aspect that pops out is that the pixelated image looks kind of "rough and edgy". Maybe some smoothing might help? What about a little average filtering? Try it. At what stage should it be done?

The Final Cut

Just for an extra touch, the functions already provided in the file enable determining a fixed number of colors for the pixelated image. To that effect to force 5 colors, you can write:

```
final = kMeansImage(dst, 5)
```

Now, you can just experiment with the different parameters and with other images.