[Session 3] OpenCV Supplement

Interesting functions

We will briefly introduce some of interesting image processing functions. Not only the function use, we will briefly look at the background.

(Binarization)

As the name suggests, it bifurcates a given color. Note that each channel can have the integer value 0 to 255. We can specify a certain "threshold" and apply this:

$$Bin(x) = \begin{cases} MAX & if\ x > Threshold \\ 0 & if\ x \le Threshold \end{cases}$$

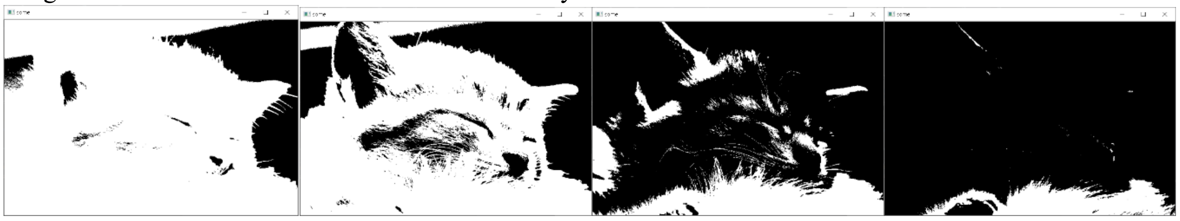
The function can be used like this:

- **`cv2.threshold(image, threshold, max, type, dst)`**
  - **`image`**: image to apply binarization.
  - **`threshold`**: the threshold value (T)
  - **`max`**: the max value
  - **`type`**: Binarization type flag
    - **`cv2.THRESH_BINARY`**: Usual binarization, maxval if x > T, and 0 otherwise
    - **`cv2.THRESH_INVERSE`**: 0 if x > T, and maxval otherwise
    - **`cv2.THRESH_TRUNC`**: T if x > T, x otherwise
    - **`cv2.THRESH_TOZERO`**: x if x > T, 0 otherwise
    - **`cv2.THRESH_TOZERO_INV`**: 0 if x > T, x otherwise
  - **`dst:`** the destination

And it returns 2-tuple (`thresh, image`)

- `thresh`: the threshold that was used
- `image`: modified image

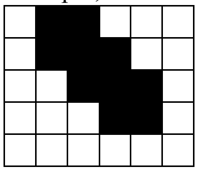
This is useful because it can be used for distinguishing something useful / or not clearly and simplifying the given image. Note that the threshold should be set carefully:

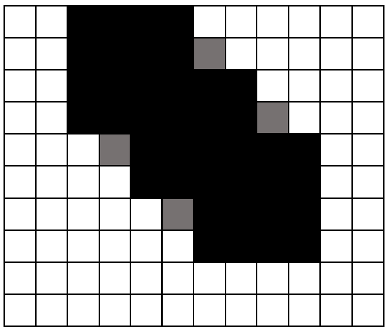


(Resize)

Resizing seems straightforward, but it needs **"interpolation"** mechanism.

When you zoom the above image, then you can see an aliasing effect, so the edge seems is too rough. The remedy for this is called anti-aliasing. We can interpolate the aliasing, by inserting some color between two space (in this example, between white and black). Suppose that we want to make some image larger.





In the above example, the shaded area may need some interpolation. Thus, we specify the interpolation method when we call resize function.

- **`cv2.resize(src, dsize, dst, fx, fy, interpolation)`**
  - `dst, fx, fy, interpolation` is keyword-argument (with a default value(`None`))
  - For interpolation, We usually use `cv2.INTER_AREA` when zoom out scaling, and `cv2.INTER_CUBIC` / `cv2.INTER_LINEAR` when zoom in scaling

(Affine Transformation)
Affine transformation is a kind of transformation mechanism. This is useful because some "nice" property is conserved even after the transformation is applied. We can do (parallel) transition, scaling, rotation, shearing, etc. Because the image is a kind of matrix, you can apply affine transformation, if you are aware of some backgrounds of linear algebra.

- **`cv2.getAffineTransform(src, M)`**
  - **`src`**: the source image
  - `M:` affine transformation matrix

To get some affine transformation matrix, we can use some backgrounds for matrix and linear algebra. Also, we can get an affine matrix for 2D rotation by using the function.

- **`cv2.getRotationMatrix2D(center, angle, scale)`**

Use the return value as a transformation matrix.

Encode / Decode

We know that computer can understand only 0 and 1: we call this **bit**. And modern computer can process several bits at once. Have you heard of "32-bit computer", or "64-bit computer"? This is a unit that computer can process at once. Also, we know that, a **byte** consists of 8 bits. So, we can express 32 / 64 bit like this:

- **Example**: 11000001 11010101 10010000 00111011 (32 bit)

And then, we can express each byte as a hexadecimal. (each byte can be 0x00 - 0xFF)

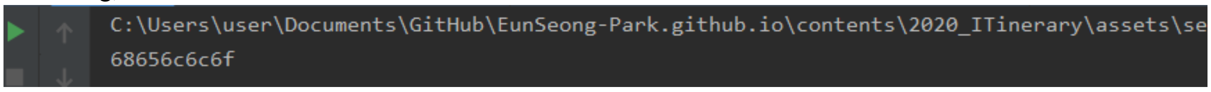
- **Example**: 0xC1 0xD5 0x90 0x3B

Instead of 0 and 1, we usually use this bytecode / byte-array.

Anyway, encoding is a procedure to convert something to binary, and decoding is a procedure to convert binary to something we can understand. Let's see an example of string.

```python
corpus = "hello"
encoded = bytearray(corpus.encode()).hex()
print(encoded)
```

The string, "`hello`" is encoded like this:



```
C:\Users\user\Documents\GitHub\EunSeong-Park.github.io\contents\2020_ITinerary\assets\se
68656c6c6f
```

Why?
It can be divided into 5 parts: 68 / 65 / 6c / 6c / 6f. And this is **hexadecimal ASCII** code, so each one is corresponding to "h", "e", "l", "l", and "o", respectively.

Also, we can encode / decode in many ways, such as UTF-8, ANSI, etc. The procedures of encoding / decoding are different in many data types: string, image, audio, or video.

Codec / Compression

Encoding and decoding is applied to many types of media or data, but encoding/decoding of video is more interesting. Because this procedure involves many things:

- It determines **quality** and **speed** to encode. Good quality is not always the answer, why? Suppose that you are famous YouTube creator. You want to show some streaming, but your encoding is too slow, so your subscribers suffer with buffering. So, a proper "trade-off" is needed
- It can **compress** the size of file. If we have a movie video with 1024*768 size, 60 fps, and 60 min, then this may be about 2.8 GB, although the sound is excluded. Today's codec reduces its size with many ways.

So, how can we compress a video?

(Example method 1)
Suppose that, the picture is showed for 10 mins.

... (10 mins after) ...

In this case, **should we store each frame?** Perhaps we do not want to this. So some codec does not store the frame between these. Even if some part is changed, it only saves the changed part of frame. This assumes that the very next frame will look similar in most case, do you agree? It is called **temporal compression.**

(Example method 2)

Suppose that the frame. It is monotonic. Then, **should we store all the pixel?** We can use like...

- **Original**: first pixel is black, second pixel is black, ..., 105736th pixel is black ....
- **Compressed**: from first to 105736th pixel, it's all black!

Roughly, first one needs 105736 of data. but second one needs just 2 of data: start and end.

In addition, we **can replace some "very similar color" into one**. Then we can use the above method. Can I do that? Yes, because you cannot notice if these following colors are different or not!

R: 255 vs R: 253

If you are interested in this, you can find some other compression method, not only video but string, image, audio, etc. I will briefly introduce about the compression of string: using **Huffman code**.

(**Huffman code**)
Let's take an example:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAABBBBC
```

As we could see in the previous topic, each character uses 1 byte = 8 bits. The above has 142 characters, so it uses 142*8 = 1136 bits. But, we can compress this, like this.

This is a rule:

- We will use "A" as "1" (1 bit)
- We will use "B" as "01" (2 bit)
- We will use "C as "00" (2 bit)

Then, we can rewrite this:

```
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111101010100
```

Because each one is a bit. so we only use 137+8+2 = 147 bits.

This method is based on the idea: **Make the character we use frequently be short**. So we make A shortest. But this needs some algorithm to use. You can find the background here:
https://en.wikipedia.org/wiki/Huffman_coding