Exercise Set 4
More Python

University of Oslo - IN3110/IN4110

Fall 2020

Note: These exercises are not mandatory. You don't have to put your work into your Github repository.

Exercise 1: Blurring Images

In this exercise, you will make a program for blurring an image. We will represent an image in Python as a 3-dimensional array (H, W, C), where H and W are the height and width of the image respectively, and C is the channels.

`image` is then a 3-dimensional numpy array with channels $C = 3$, in the order blue, green and red (BGR). Blur can be applied to a (H, W, C) image by applying convolution with an averaging kernel. That means, for each pixel in the image, we set the corresponding pixel in the blurred image to be an average of the pixel values in the neighboring pixels. If `src` is the original image, and `dst` is the new blurred image, the application of a 3×3 averaging kernel at pixel h and w in channel c is:

```
dst[h,w,c] = (src[h,w,c] + src[h-1,w,c] + src[h+1,w,c]
                         + src[h,w-1,c] + src[h,w+1,c]
                         + src[h-1,w-1,c] + src[h-1,w+1,c]
                         + src[h+1,w-1,c] + src[h+1,w+1,c])/9
```

Note that we treat each channel independently.

Important: There are libraries that can apply convolution kernels for you, but you are not supposed to use these. Please implement the operation manually for this exercise.

On the edges of the image, not all immediate neighboring pixels exist. The solution is to define the "pixels" outside the image to have the same value as the pixels on the edge of the image. That is, if on the left side of the image such that $h = 0$, then we define the pixel value at $[h - 1, w, c]$ to be the same as the value at $[h, w, c]$ and the pixel value at $[h - 1, w + 1, c]$ to be equal to the value at $[h, w + 1, c]$. In essence, we add a single pixel outside the image by repeating the values of the pixels at the border of the image. This process can easily be done with numpy using `pad` with *edge* mode. After padding `src`, the indices between `dst` and `src` no longer match. Thus, when using a padded

image `src`, the formula above must be modified such that the indices of `src` are shifted one pixel in each of the two spatial dimensions. We recommend using `OpenCV` for the implementation. OpenCV reads images as (unsigned) integers. However, after applying the averaging, the new blurred image likely contains floating point numbers. Thus, for OpenCV to understand your blurred image, you must convert the values back to integers. This can be done as follows

```
dst = dst.astype("uint8")
```

Note: Because the input image is read as an array of unsigned 8-bit integers (uint8), adding such values will cause an overflow when the sum exceeds 255. To combat such overflows, one can either divide all terms individually in the previously shown sum, or convert `src` to a type with a higher maximum value, such as unsigned 32-bit integers (uint32):

```
src = src.astype("uint32")
```

Once the blurred image is converted back to the correct type, you can save it to a file using `imwrite`.

```
cv2.imwrite("blurred_image.jpg", dst)
```

Exercise 2: Decorators

Assume that `f` is a Python function. Because functions are objects in Python, we can pass functions to other functions. It turns out that writing

```
f = some_func(f)
```

Is quite common. There is a shorthand for this. When writing the definition of `f`, we can say

```
@some_func
def f():
    # Whatever f does
```

We have applied a decorator to `f`. An example of a decorator is one that prints the output of f before it is returned. It can be implemented in the following way:

```
def print_output(f):
    def wrapper(x):
        output = f(x)
        print(output)
        return output
    return wrapper
```

Using this decorator does not change the behaviour of `f`, except that the output is always printed when f is called.

Exercise: Make a decorator that tracks the time a function uses to execute

Decorators can be (and usually are) implemented as classes, where the class' `__call__` method is used as the wrapper. This can be useful if we want to keep some information between calls. We are going to try this next.

Exercise: Create a decorator that caches output from f. i.e., before calling `f`, check if we already know what the return value is, and return this if this is the case. This is useful if `f` takes a long time to execute.