ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 453E

COMPUTER VISION
HOMEWORK REPORT

HOMEWORK NO : 1

DUE DATE : 24.10.2023

GROUP MEMBERS:

150210921 : Duc Quang Nguyen

1 Introduction

Homework 1 requires the implementation of a number of algorithms in Python code without using Numpy, OpenCV, or any other computer vision or data processing libraries. Questions 1 and 2 involve converting RGB images into grayscale ones, while question 3 involves performing two local operations on an image.

2 Question 1

The goal of question 1 is to convert an RGB image into grayscale, using the NTSC formula.

First, a colored image I is read with the OpenCV's imread() function. Without specifying any parameters, this function returns a 3-D Numpy array of shape (number of rows, number of columns, number of channels) representing I. The channels, however, are arranged in the BGR order. To change I into the RGB (reverse) order, the "[::-1]" syntax is used.

The conversion function is named NTSC() and defined in a separated file called "NTSC.py". The function takes one parameter I, which is an RGB image to be converted (in Numpy array form), and returns J, the grayscale version of this image (also in Numpy array form). It begins by setting J as a zero Python list of the same shape as I. Then, two for-loops are used to iterate through each pixel of I, computing pixel values for J according to the given NTSC formula. After that, Python round() function is used to round non-integer results to integers. Finally, J is converted from being a list to being a Numpy array to allow OpenCV to process J as an image.

Figure 1 shows the result of grayscaling a colored image with this method.

Figure 1: (left) Original image - (right) Image after grayscaling

3

Figure 2: Different original RGB colors and their corresponding gray tones

Figure 2 compares several RGB colors in the original image and their corresponding gray tones in the grayscale image. The grayscaling process manages to preserve different tones of blue color: the darker blue color of the sea scales to a darker gray tone while the lighter blue color of the sky scales a lighter gray tone. The dark green color of the trees and bushes are also represented by a dark gray tone quite similar to the dark tone of the sea. On the other hand, yellow scales to a much lighter gray tone than either blue or green. This is expected since yellow, a combination of red and green in the RGB scheme, receives a higher gray-level value as the NTSC formula takes into account all red, green, and blue values in the original image.

Overall, the darkest blue and green colors in the original image have the darkest gray tones in the grayscale image, while the lighest yellow, pink, and white colors have the lightest gray tones.

3 Question 2

The goal of question 2 is to perform conditional scaling, which is a method to scale the intensity range of a grayscale image J according to a reference image I in order to obtain a new grayscale image J'.

In step 1, similar to question 1, a colored image is read and converted to grayscale with the NTSC formula. The only difference is that the NTSC() function here is defined in a separate file named "functions.py" along with other necessary functions. The output of step 1 is J, the grayscale version of the original input image.

Step 2 is similar to step 1: another colored image is read and converted to grayscale with the NTSC formula. This grayscale image is the reference image I, which will act as the framework to scale J later on.

Because the conditional scaling formula requires finding mean and variance of I and J, step 3 is dedicated to finding these values. Two functions, compute_sums() and mean_variance(), are defined in "functions.py" for this purpose. compute_sums() takes one parameter, an image I(x, y), and returns two values, $\sum I(x, y)$ and $\sum I^2(x, y)$ as x ranges from 1 to number of rows and y ranges from 1 to number of columns. mean_variance() uses these two sums to compute the mean and variance of I. Figure 3 shows the code of compute_sums() and mean_variance() functions. The formula for computing mean and variance are also included below.

```python
# For all pixels p in a grayscale image,
# compute sum of p and sum of p^2
def compute_sums(image):
    s = 0
    s2 = 0
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            p = image[i][j]
            s += p              # sum over all pixels
            s2 += p * p         # sum over all pixels squared
    return (s, s2)

# Compute the mean and variance of all pixel values in a grayscale image
def mean_variance(image):
    # no_of_pixels = no_of_rows * no_of_cols
    no_of_pixels = image.shape[0] * image.shape[1]
    s, s2 = compute_sums(image)
    mean = s / no_of_pixels
    var = s2 / no_of_pixels - mean * mean
    return (mean, var)
```

Figure 3: Code of compute_sums() and mean_variance() functions

The formula for computing mean and variance are

$$\text{no\_of\_pixels} = \text{no\_of\_rows} \times \text{no\_of\_cols}$$

$$\mu_I = \frac{\sum I(x,y)}{\text{no\_of\_pixels}}$$

$$\sigma_I = \frac{\sum I^2(x,y)}{\text{no\_of\_pixels}} - \mu_I^2$$

Step 4 conditionally scales each pixel of J according to the given formula to obtain a new pixel value of J'. The process is defined in the conditional_scaling() function, which takes 5 parameters, image J, mean of J, variance of J, mean of I, variance of I. Two for-loops are used to index each pixel value of J and set the correct pixel value of J'.

Figure 4 includes an example of grayscaling a colored input image with the NTSC formula, then conditionally scaling it by referencing on a much darker grayscale image.

Figure 4: (top left) Original image - (top right) NTSC grayscale image (bottom left) Reference grayscale image - (bottom right) Final result

As seen above, due to the dark reference image, the final image was scaled to a much darker gray tones than the NTSC image. In fact, the mean of pixel values in the NTSC image is 139.87, while the mean of pixel values in the image after conditional scaling is 63.92, the same as that in the reference image. The conditional scaling process has scaled the mean of the NTSC image to be similar to that of the reference image, resulting in a darker-tone image.

4 Question 3

The goal of this question is to perform minimum and maximum local operations on a grayscale image, using two filter sizes of 7x7 and 21x21.

One limitation of the code written for this question is that it only applies to square images of size N. However, the code can be modified to process a rectangular image of size MxN.

Step 1 is the same as in the other two questions. A colored image was chosen, read by OpenCV, then turned into a grayscale image with the NTSC formula. Figure 5 shows the original image and the grayscale image after performing this step.

Figure 5: (left) Original image - (right) Image after grayscaling

Step 2 is to pad the grayscale image with 0. The number of pads is set to be half of the filter size, rounded down to the nearest integer (since the filter size is always an odd number). 2 functions are written for this step. The first one is no_of_paddings(), which takes the filter size as an argument and returns the number of pads needed. The second one is padding(), which takes two arguments: the grayscale image and the number of pads. It does the padding by first setting up a Python list as the padded image, whose shape is the image's original size plus two times the number of pads. The function then iterates over the original grayscale image and copies each pixel into the padded image, avoiding the zero pads on the borders.

Figure 6 shows the code of these two functions.

```
# Compute the number of paddings needed according to filter size
def no_of_paddings(filtersize):
    return filtersize // 2

# Pad a square image with 0
def padding(image, no_of_paddings):
    padded_img = np.zeros((image.shape[0] + no_of_paddings * 2,
                           image.shape[1] + no_of_paddings * 2))

    # Iterate over image and copy each pixel value of image into the appropriate position of padded_img
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            padded_img[i+no_of_paddings][j+no_of_paddings] = image[i][j]

    return padded_img
```

Figure 6: Code of no_of_paddings() and padding() functions

Two filters of different sizes - 7x7 and 21x21 - are used for filtering, which means each filter requires a different number of pads - 3 and 10, respectively. Figure 7 demonstrates the result of padding on the grayscale image of Figure 6. These two images are now bordered by black lines, which are the zero pads added to the grayscale image. The 10-padded image, as expected, has thicker black border lines than the 3-padded one.

Figure 7: (left) 3-padded image - (right) 10-padded image

Step 3 is to define the filtering window according to the given filter sizes. This process is done in function define_window(), which takes the padded image, the reference pixel position (column and row), and the filter size as arguments. The reference pixel is defined as the center of the window. define_window() then collects all the pixel values within the window in a list, stores in a list and returns that list for later processing.

Step 4 is to perform the filtering process, which is done in 2 functions: minimum_filtering() and maximum_filtering(). The functions slide through the pixels of the padded image with 2 for-loops, define the filtering window using the define_window() function, use Python's min() and max() functions to find the minimum and maximum values within the window, then stores the result in a linear list defined beforehand. The "step" parameter of the Python for-loop is used to perform strides, and the "start" and "stop" parameters are set so that the windows will not go over the image border. This process returns a 1-D list of the minimum / maximum value of each filtering window.

To let OpenCV process the filtered image, the 1-D list needs to be converted to a 2-D Numpy array of known size. However, a side-effect of striding in local filtering is that the size of the image after filtering will be smaller than before filtering. Fortunately, the output size can be computed with the following formula:

$$\text{Output size} = \lfloor \frac{\text{input size} - \text{filter size}}{\text{stride}} \rfloor + 1$$

Input size refers to the size of the image before filtering, and $\lfloor x \rfloor$ refers to floor function of x. This formula is defined in the filter_output_size() function.

Figure 8 shows the two images after minimum filtering and maximum filtering size 7x7. The minimum-filtered image is darker than before filtering and the padded border is still present. The maximum-filtered image, on the other hand, is lighter than before filtering and the padded border is no longer present due to the filtering windows maxing out these padded values. These filtered images are also smaller than the original image, though it is not visible here since all images have been resized to be included in this report. The details of the filtered images blur compared to the original image due to the filtering effect.

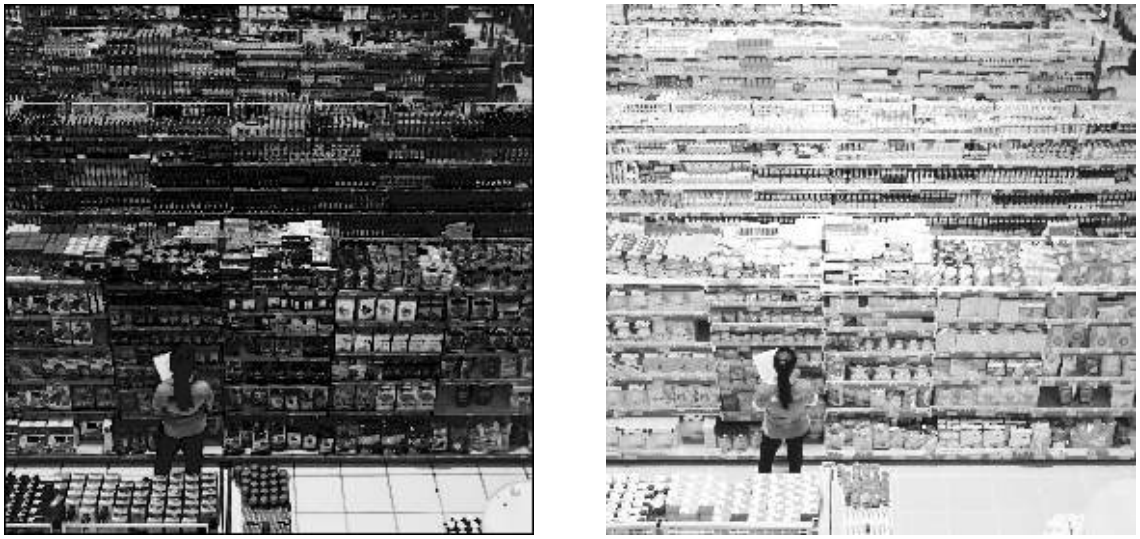

Figure 8: Image after minimum filtering (left) and maximum filtering (right) of size 7x7

Figure 9 shows the two images after minimum filtering and maximum filtering size 21x21. The minimum-filtered image is darker and the maximum-filtered image is brighter than before. The details also become much more blurred, possibly due to the bigger filtering window minning and maxing out the reference pixel values. The larger stride in this case also results in a much smaller image compared to the smaller stride in the case of a 7x7 filter.

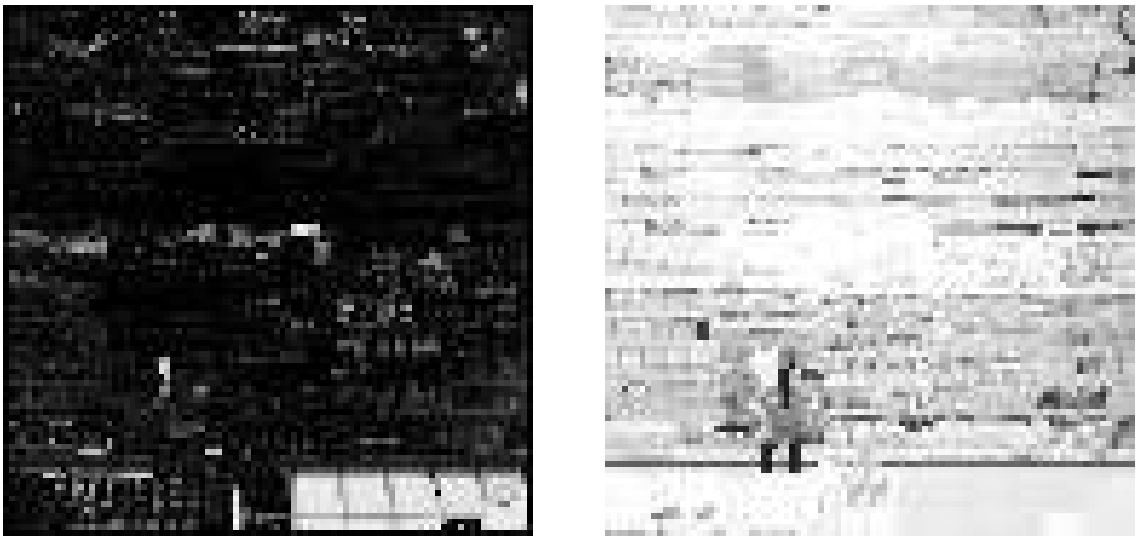

Figure 9: (left) Image after minimum filtering (left) and maximum filtering (right) of size 21x21