When life gives you lemons...

Implementing filtering functions

Daniel Zagoury, Matanel Shachamorov

Department of Computer Science, Holon Institute of Technology - Image Processing course

Keywords: filter, smoothing, sharpening, median, averaging, cv2, padding

Abstract

During an Image Processing course, we were given a task to implement by ourselves a few of the most common filters in classic image processing. This assignment was given to us in order to better understand the complexity, method of operation and thought process behind those filters, which are now applied by a simple line of code.

In this project, we worked on a picture of a lemon stand in the supermarket we took on our phones, and viewed the different effects of the filters on our image. We also compared the results and runtime of our implementations to the library function in OpenCV, also known as cv2.

Methods and libraries

We started by importing the numpy library so we can create matrixes (filters and images), cv2 so we can pre-process the image, and then later show and save the results and time library to measure the runtime of our functions.

In the first part of the project, we were required to implement a function that has an input of a square image, a square (linear) filter, and a type of padding. We broke down the problem unto smaller sub sections and worked on each one separately.

We began working on the padding types with the given input: 0 for zero-padding, 1 for replicate-padding and 2 for mirror-padding, and with the padding, we made sure the output image will have the same dimensions as the input image. After we implemented the paddings, we created an array of zeros with the same size as the input image and started the convolution process on our padded image and keeping the results in our output image, which we returned as the output of our filtering function implementation.

In the second part of the project, we were required to implement a function that has an input of a square image, a non-linear filter, it's dimensions and a type of padding. We worked similarly to the first part, with a few mild differences - or filter was not square, so we needed to pay attention to the different rows and columns dimensions of our current filter. We also did not use our convolution function, but instead, used numpy functions - median and average - as those were the non-linear filters that we implemented.

Results and discussion

After we implemented all the functions, we loaded our image and applied all the filters that we were required to apply -

1. Original vs. Grayscale image:

Original Image

Grayscale Image

2. Smooth:

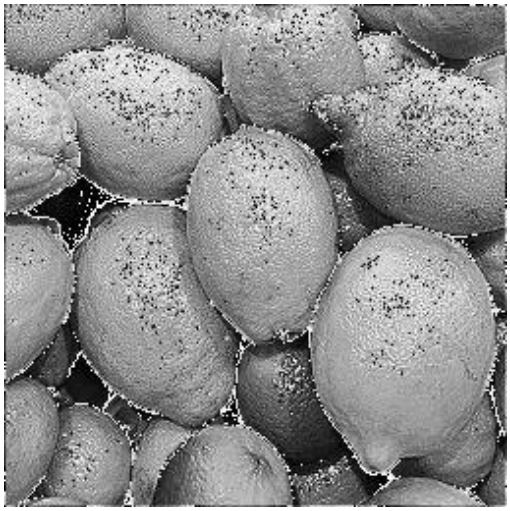
Smooth - Our implementation

Smooth - Library function

Differences, if existing, are non-visible to the naked eye

3. Sharp:

Sharp - Our implementation



Sharp - Library function

Our image appears to be overly sharped compared to the library function sharpening, even though same filter was applied (one-pass Laplace filter)

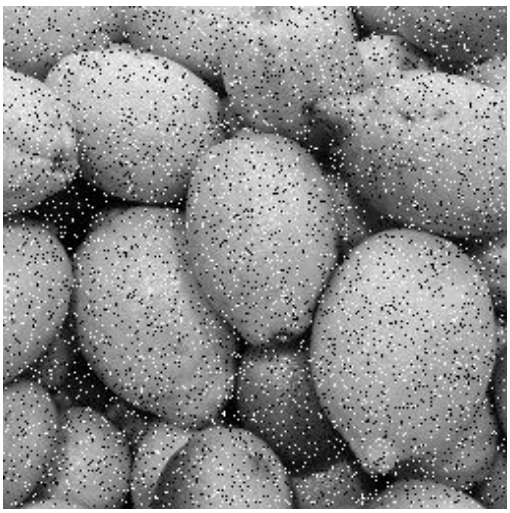
4. Median:

Median filter

We can see that the picture looks a little smudged, but still maintains most of it's quality

5. Salt and Pepper noise:

Salt and Pepper noise



After adding the noise, the picture looks a little grainy

6. Smoothing using the 1st part function:

Salt and Pepper noise - Smoothed

After applying the smoothing operation we implemented, we can see that the noise is still visible and the whole image appears to be blurry.

As part of the process, we used the time library to calculate the runtime of our implementation compared to the library functions. Our implementations of the functions took about 1.4 secs to execute, while the library functions were much faster (less than 0.001 secs). We assume that it is due to certain optimizations done

in the library functions that we were not aware of. Alternatively, this could be due to usage of more complex and efficient calculation methods compared to the methods that we used that were more straight-forward and naïve.

Summary

During this project, we were asked to implement common filtering functions in classic Image Processing. We broke down each part of the assignment and delt with them one at a time to achieve the required goal. We also learned about the complexity of seemingly simple concepts and aspects of Image Processing, while also getting the chance to "get our hands dirty", implement those concepts by ourselves and view the results and effects of those filters on our own images, while also deepening our understanding of those concepts.

Acknowledgments

We used the help of ChatGPT throughout this project to help us better understand and give ideas about implementations. Selective prompts:

- "how to make image sharper using cv2"
- "how to make image sharper without cv2"
- "common non-linear filters in image processing"
- "explain median filter"

Appendix

Code:

Part 1 functions

Convolution Function

```python
def sum_matrix_conv(matrix1, matrix2):
    # Number of rows in the first matrix
    dim = len(matrix1)
    result = 0

    # Perform matrix convolution
    for i in range(dim):
        for j in range(dim):
            result += matrix1[i][j] * matrix2[i][j]

    return result
```

Linear Filter Function (inc. padding)

```python
# pad variable is an int indicating the padding kind to add - 0: zero padding, 1:replicate padding, 2: mirror padding
def my_imfilter(s, fil, pad):
    in_dim = s.shape[0]
    fil_dim = len(fil[0])
    pad_num = int(fil_dim/2)

    # creating an array that will be used as the padded image
    pad_im = np.zeros((in_dim + 2 * pad_num, in_dim + 2 * pad_num), dtype = np.uint8)

    if (pad == 0): # zero padding
        pad_im[pad_num : in_dim + pad_num, pad_num : in_dim + pad_num] = s

    if (pad == 1): # replicate padding
        pad_im[pad_num : in_dim + pad_num, pad_num : in_dim + pad_num] = s
        # corners
        pad_im[ : pad_num,  : pad_num] = s[0][0]
        pad_im[ : pad_num, in_dim + pad_num : ] = s[0][in_dim-1]
        pad_im[in_dim + pad_num : , : pad_num] = s[in_dim-1][0]
        pad_im[in_dim + pad_num : , in_dim + pad_num : ] = s[in_dim-1][in_dim-1]
        # columns
        pad_im[pad_num : in_dim + pad_num, : pad_num] = s[:, 0][:, np.newaxis]
        pad_im[pad_num : in_dim + pad_num, in_dim + pad_num : ] = s[:, -1][:, np.newaxis]
        # rows
        pad_im[ : pad_num, pad_num : in_dim + pad_num] = s[0, :]
        pad_im[in_dim + pad_num : , pad_num : in_dim + pad_num] = s[-1, :]

    if (pad == 2): # mirror padding
        pad_im[pad_num : in_dim + pad_num, pad_num : in_dim + pad_num] = s
        # sides
        pad_im[pad_num : in_dim + pad_num, : pad_num] = s[:, pad_num:0 :-1]
        pad_im[pad_num : in_dim + pad_num, in_dim + pad_num : ] = s[:, in_dim - 2 : in_dim - pad_num - 2 :-1]
        # top & bottom
        pad_im[ : pad_num , : ] = pad_im[2 * pad_num : pad_num :-1, : ]
        pad_im[pad_num + in_dim :  , : ] = pad_im[pad_num + in_dim - 2 : in_dim-2 :-1, : ]

    out_image = np.zeros((in_dim, in_dim), dtype = np.uint8)

    for i in range (in_dim):
        for j in range (in_dim):
            out_image[i][j] = sum_matrix_conv(pad_im[i : i + fil_dim, j : j + fil_dim], fil)
            if (out_image[i][j] > 256):
                out_image[i][j] = 256
            if (out_image[i][j] < 0):
                out_image[i][j] = 0

    return out_image
```

Part 2 functions

Non-Linear Filter Function (inc. padding)

```python
# nlf variable is a string that indicates what non-linear filter will be applied.
# we implemented median filter and average filter.
def my_im_nl_filter(s, m, n, nlf, pad):
    in_dim = s.shape[0]
    pad_rows = int(m/2)
    pad_cols = int(n/2)

    # creating an array that will be used as the padded image
    pad_im = np.zeros((in_dim + 2 * pad_rows, in_dim + 2 * pad_cols), dtype = np.uint8)

    if (pad == 0): # zero padding
        pad_im[pad_rows : in_dim + pad_rows, pad_cols : in_dim + pad_cols] = s

    if (pad == 1): # replicate padding
        pad_im[pad_rows : in_dim + pad_rows, pad_cols : in_dim + pad_cols] = s
        # corners
        pad_im[ : pad_rows,  : pad_cols] = s[0][0]
        pad_im[ : pad_rows, in_dim + pad_cols : ] = s[0][in_dim-1]
        pad_im[in_dim + pad_rows : , : pad_cols] = s[in_dim-1][0]
        pad_im[in_dim + pad_rows : , in_dim + pad_cols : ] = s[in_dim-1][in_dim-1]
        # columns
        pad_im[pad_rows : in_dim + pad_rows, : pad_cols] = s[:, 0][:, np.newaxis]
        pad_im[pad_rows : in_dim + pad_rows, in_dim + pad_cols : ] = s[:, -1][:, np.newaxis]
        # rows
        pad_im[ : pad_rows, pad_cols : in_dim + pad_cols] = s[0, :]
        pad_im[in_dim + pad_rows : , pad_cols : in_dim + pad_cols] = s[-1, :]

    if (pad == 2): # mirror padding
        pad_im[pad_rows : in_dim + pad_rows, pad_cols : in_dim + pad_cols] = s
        # sides
        pad_im[pad_rows : in_dim + pad_rows, : pad_cols] = s[:, pad_cols:0 :-1]
        pad_im[pad_rows : in_dim + pad_rows, in_dim + pad_cols : ] = s[:, in_dim - 2 : in_dim - pad_cols - 2 :-1]
        # top & bottom
        pad_im[ : pad_rows , : ] = pad_im[2 * pad_rows : pad_rows :-1, : ]
        pad_im[pad_rows + in_dim :  , : ] = pad_im[pad_rows + in_dim - 2 : in_dim-2 :-1, : ]

    out_image = np.zeros((in_dim, in_dim), dtype = np.uint8)

    if (nlf == 'median'):
        for i in range (in_dim):
            for j in range (in_dim):
                out_image[i][j] = np.median(pad_im[i : i + m, j : j + n])

    if (nlf == 'average'):
        for i in range (in_dim):
            for j in range (in_dim):
                out_image[i][j] = int(np.average(pad_im[i : i + m, j : j + n]))

    return out_image
```

Salt and Pepper Noise Function

```python
def add_salt_and_pepper_noise(image, salt_prob, pepper_prob):
    row, col = image.shape
    s_vs_p = 0.5

    # Salt noise
    num_salt = np.ceil(salt_prob * image.size * s_vs_p)
    coords = [np.random.randint(0, i - 1, int(num_salt)) for i in image.shape]
    image[coords[0], coords[1]] = 255

    # Pepper noise
    num_pepper = np.ceil(pepper_prob * image.size * (1.0 - s_vs_p))
    coords = [np.random.randint(0, i - 1, int(num_pepper)) for i in image.shape]
    image[coords[0], coords[1]] = 0

    return image
```

"Main" code (applying the filters)

Loading & Preprocessing

```
img=cv2.imread("lemons.jpg")
img = cv2.resize(img, (256, 256))
gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Application of Filters

```
sharpen = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]]) # One Pass Laplace filter 3x3
smoothen = np.array([[1/9, 1/9, 1/9], [1/9, 1/9, 1/9], [1/9, 1/9, 1/9]]) # Smooth filter 3x3

# getting the time before and after each filter so we can measure the runtime of our functions
time1 = time.time()
sharped_image = my_imfilter(gray_img, sharpen, 0)
time2 = time.time()
smoothed_image = my_imfilter(gray_img, smoothen, 0)
time3 = time.time()
lib_smooth = cv2.filter2D(gray_img, -1, smoothen)
time4 = time.time()
lib_sharp = cv2.filter2D(gray_img, -1, sharpen)
time5 = time.time()

print("Our sharpening time is: {} seconds, the library filter2D sharpening time is: {} seconds"
      .format(time2-time1, time5-time4))
print("Our smoothing time is: {} seconds, the library filter2D smoothing time is: {} seconds"
      .format(time3-time2, time4-time3))

# showing and saving the results of each filter we applied
cv2.imshow("Grayscale", gray_img)
cv2.imwrite('gray_img.jpg', gray_img)

cv2.imshow("Sharpened", sharped_image)
cv2.imwrite('sharped_image.jpg', sharped_image)

cv2.imshow("Smoothed", smoothed_image)
cv2.imwrite('smoothed_image.jpg', smoothed_image)

cv2.imshow("Lib Smoothen", lib_smooth)
cv2.imwrite('lib_smooth.jpg', lib_smooth)

cv2.imshow("Lib Sharpen", lib_sharp)
cv2.imwrite('lib_sharp.jpg', lib_sharp)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Application of Salt and Pepper Noise

```
# Add salt and pepper noise
noisy_image = add_salt_and_pepper_noise(gray_img, salt_prob=0.1, pepper_prob=0.1)

# Save or display the result
cv2.imwrite('noisy_image.jpg', noisy_image)
cv2.imshow('Noisy Image', noisy_image)

smoothed_snp = my_imfilter(noisy_image, smoothen, 0)

cv2.imwrite('smoothed_snp.jpg', smoothed_snp)
cv2.imshow('Smoothed SnP', smoothed_snp)

median_im = my_im_nl_filter(gray_img, 3, 5, 'median', 0)

cv2.imwrite('median_im.jpg', median_im)
cv2.imshow('Median_im', median_im)

cv2.waitKey(0)
cv2.destroyAllWindows()
```