Artificial Intelligence and Computer Vision

Laboratory 5

Report Sheet

Jakub Bujak 249120

Task description:

1. **Perform filtering of Gaussian noise:**
    a. For one low frequency and one high frequency image add a Gaussian noise to picture using OpenCV built-in functions. Use two different setting for standard deviation
    b. Perform filtering for both pictures
    c. Perform sharpening of pictures
    d. Compare to original and observe association between frequency, noise deviation and quality of result image
2. **Perform filtering of salt and pepper noise:**
    a. For one low frequency and one high frequency image and a S&P noise to picture using an OpenCV built-in functions. Use two different setting for noise.
    b. Perform filtering for both pictures
    c. Perform Sharpening of pictures
    d. Compare to original and observe association between frequency, noise deviation and quality of result image
3. **Perform edge detection using a Sobel filters:**
    a. For one low frequency and one high frequency image perform edge detection using OpenCV built-in functions.
    b. Describe observations, compare results of high and low frequency image filtering
4. **Perform edge detection using a Sobel filters for images with Gaussian noise:**
    a. Take high frequency and add a Gaussian noise
    b. Perform Sobel edge detection
    c. Describe observations
5. **Perform edge detection using a custom written Laplace filter**
    a. Take high frequency image
    b. Implement Laplace filter using a given kernel
    c. Perform filtering
    d. Compare to filtering done with built-in functions
    e. Describe observations

Report:

First we import the libraries: Numpy, OpenCV, and Random. This part is common for all tasks.

```
import cv2 as cv
import numpy as np
from random import randrange
```

Task 1:

Defining the function for creating the gaussian noise:

```
def noise(img):
    sr = 0
    dv = 4
    gaussian_noise = np.zeros(img.shape)
    x = cv.randn(gaussian_noise, sr, dv)
    gaussian_noise = np.uint8(gaussian_noise)
    output = cv.addWeighted(img, 1, gaussian_noise, 1, 0)

    return output
```

First we define the mean level and standard deviation level. Gaussian noise is represented as a zeros matrix filled by image dimensions. Then we fill the array with normally distributed random numbers, and represent the result as an unsigned integer. The output is calculated as the weighted sum of two arrays.

Defining the function for filtering the gaussian noise:

```
def gaussian_filter(img):
    blur = cv.GaussianBlur(img, (3,3), cv.BORDER_DEFAULT)

    return blur
```

Here, instead of a box filter, a Gaussian kernel is used. Gaussian filtering is done by convolving each point in the input array with a Gaussian kernel and then summing them all to produce the output array. We should specify the width and height of the kernel which should be positive and odd. We also should specify the standard deviation in the X and Y directions, sigmaX and sigmaY respectively. If only sigmaX is specified, sigmaY is taken as the same as sigmaX. If both are given as zeros, they are calculated from the kernel size.

Defining the function for sharpening:

```
def kernel_sharp(img):
    kernel_sharpening = np.array([[-1, -1, -1],
                                  [-1, 9, -1],
                                  [-1, -1, -1]])
    sharpened = cv.filter2D(img, -1, kernel_sharpening)

    return sharpened
```

In the kernel sharpening method, we specify the 9 neighbouring pixels and apply the convolution process on them. Different inputs in the array would result in different sharpening. Convolution is the process of adding each element of the image to its local neighbours, weighted by the kernel. This is related to a form of mathematical convolution. The matrix operation being performed is not traditional matrix multiplication, despite being similar. Then we apply the filter2D function from OpenCV that convolves an image with the kernel, meaning that the function applies an arbitrary linear filter to an image. When the aperture is partially outside the image, the function interpolates outlier pixel values according to the specified border mode.

Calling the used functions:

```
# Load images
img1 = cv.imread('lena.png', 0)
img2 = cv.imread('pic.jpg', 0)

# Copy one of the images (select one)
#img = img1.copy()
img = img2.copy()

cv.imshow('Original image', img)

# Adding Gaussian noise to image
img_with_noise = noise(img)
cv.imshow('Image noise', img_with_noise)

#Removing noise
img_filter = gaussian_filter(img)
cv.imshow('Gaussian filtering', img_filter)

#Sharpening the picture
img_sharp = kernel_sharp(img_filter)
cv.imshow('Sharp', img_sharp)

cv.waitKey()
cv.destroyAllWindows()
```

In this part, we start calling up our functions to see the results.

Results of Task 1:

In the top left corner, we have the original image. In the top right corner we can see the randomized Gaussian Noise. In the bottom left we can see the noise filtered, and finally in the bottom right, the sharpened image.

Task 2:

Defining the functions for noise generation:

First we define our noise function, and noise components (“salt & pepper”).

```
def noise(img):
    s = salt(img)
    p = pepper(img)
    noise = cv.addWeighted(s, 0.5, p, 0.5, 0)
    output = cv.addWeighted(img, 0.5, noise, 0.5, 0)

    return output
```

We call the defined salt and pepper, together into the noise, and output them together via the OpenCv addWeighted function – it calculates the weighted sum of two arrays, and blends them together on our picture.

```
def salt(img):
    rows, cols = img.shape[:2]
    dens = (rows*cols*0.5)//2
    dens = int(dens)

    print(dens)

    for d in range(0, dens):
        x = randrange(rows)
        y = randrange(cols)
        img[x, y] = 255

    return img

def pepper(img):
    rows, cols = img.shape[:2]
    dens = (rows * cols * 0.2)//2
    dens = int(dens)

    print(dens)

    for d in range(0, dens):
        x = randrange(rows)
        y = randrange(cols)
        img[x, y] = 0

    return img
```

Equally distributed salt and pepper noises over the image dimensions (here denoted as x,y).

Defining the filtering function:

```
def filter(img):
    blur = cv.medianBlur(img, 3)

    return blur
```

The whole filter is defined as a simple OpenCV function. It Blurs an image using the median filter. The function smoothes an image using the median filter with the *ksize×ksize* aperture. Each channel of a multi-channel image is processed independently. In-place operation is supported.

Defining the function for sharpening:

```
def kernel_sharp(img):
    kernel_sharpening = np.array([[-1, -1, -1],
                                  [-1, 9, -1],
                                  [-1, -1, -1]])
    sharpened = cv.filter2D(img, -1, kernel_sharpening)

    return sharpened
```

Sharpening function is identical to the one from the first task.

Calling the used functions:

```
# Load images
img1 = cv.imread('lena.png', 0)
img2 = cv.imread('pic.png', 0)

# Copy one of the images (select one)
img = img1.copy()
#img = img2.copy()


cv.imshow('Original image', img)

# Adding S&P noise to image
img_with_noise = noise(img)
cv.imshow('Image noise', img_with_noise)

#Removing noise
img_filter = filter(img)
cv.imshow('Image median filtering', img_filter)

#Sharpening the picture
img_sharp = kernel_sharp(img_filter)
cv.imshow('Sharp', img_sharp)

cv.waitKey()
cv.destroyAllWindows()
```

In this part, we start calling up our functions to see the results.

Results of Task 1:

Here we can see the Lena picture in four different versions. The left top corner is the original image. On the right top corner, the salt & pepper noise is put over the picture. A really dense spread over the original picture can be observed. In the left bottom corner the salt & pepper picture has been filtered through the median blur filter, very much reducing the noise of the picture. And lastly, in the bottom right corner we can see the sharpened image of the filtered picture. The sharpening, apart from outlining the edges on the original picture, has also shown itself on the noise leftovers.

Task 3:

Defining the functions for edge detection:

```
def edging(img):
    scale = 1
    delta = 0
    depth = cv.CV_16S
    grad_x = cv.Sobel(img, depth, 1, 0, ksize=5, scale=scale, delta=delta, borderType=cv.BORDER_DEFAULT)
    grad_y = cv.Sobel(img, depth, 0, 1, ksize=5, scale=scale, delta=delta, borderType=cv.BORDER_DEFAULT)
    abs_grad_x = cv.convertScaleAbs(grad_x)
    abs_grad_y = cv.convertScaleAbs(grad_y)
    grad = cv.addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0)
    return grad
```

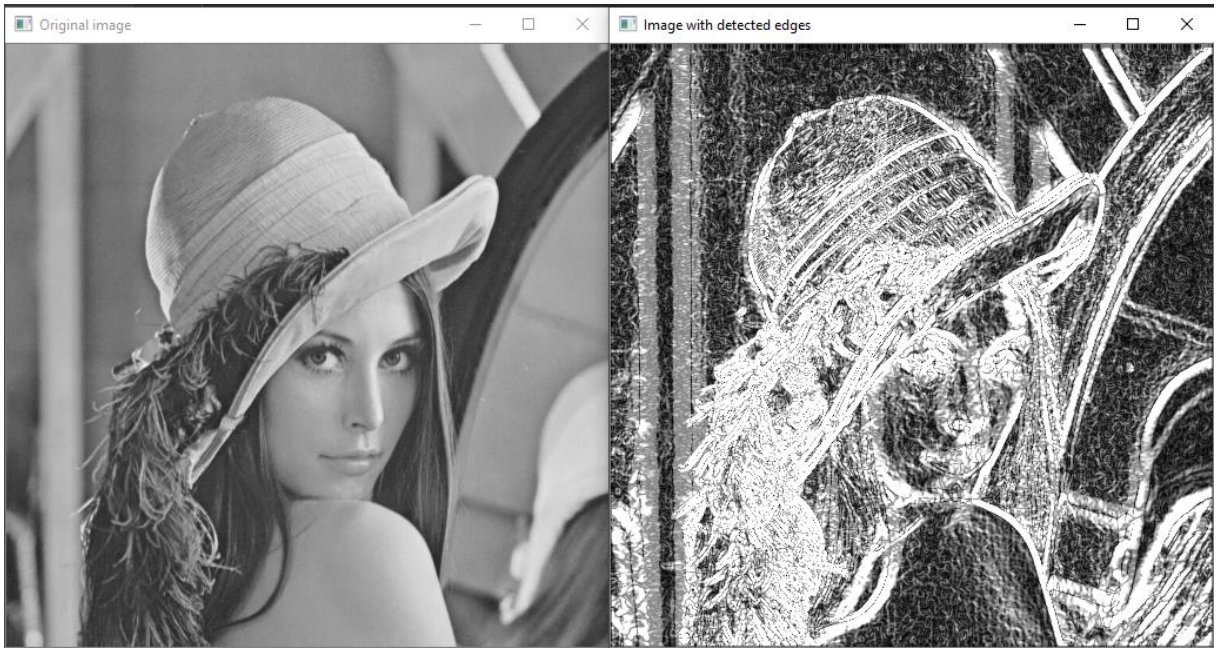
This defined function is using Sobel Operator to look for the edges of pictures. Sobel Operator is a specific type of 2D derivative mask which is efficient in detecting the edges in an image.

Calling the used functions:

```
# Load images
img1 = cv.imread('lena.png', 0)
img2 = cv.imread('pic.png', 0)

# Copy one of the images (select one)
img = img1.copy()
#img = img2.copy()


cv.imshow('Original image', img)

#Detecting the edge
edge_img = edging(img)
cv.imshow('Image with detected edges', edge_img)

cv.waitKey()
cv.destroyAllWindows()
```

In this part, we start calling up our functions to see the results.

Results of Task 3:



We can see that the Sobel edge detecting has found edges on the picture and highlighted them.

The limitations of this technique are connected with poor localization of the edges, meaning that the filter may actually display us many edges, where we should actually see only one edge, and the filter may miss the edges which are neither horizontal, nor vertical, as is searches for them in these two directions.

Task 4:

Defining the function for creating the gaussian noise:

```
def noise(img):
    sr = 0
    dv = 4
    gaussian_noise = np.zeros(img.shape)
    x = cv.randn(gaussian_noise, sr, dv)
    gaussian_noise = np.uint8(gaussian_noise)
    output = cv.addWeighted(img, 0.5, gaussian_noise, 0.5, 0)

    return output
```

Defining the function for edge detection:

```
def edging(img):
    scale = 1
    delta = 0
    depth = cv.CV_16S
    grad_x = cv.Sobel(img, depth, 1, 0, ksize=5, scale=scale, delta=delta, borderType=cv.BORDER_DEFAULT)
    grad_y = cv.Sobel(img, depth, 0, 1, ksize=5, scale=scale, delta=delta, borderType=cv.BORDER_DEFAULT)
    abs_grad_x = cv.convertScaleAbs(grad_x)
    abs_grad_y = cv.convertScaleAbs(grad_y)
    grad = cv.addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0)

    return grad
```

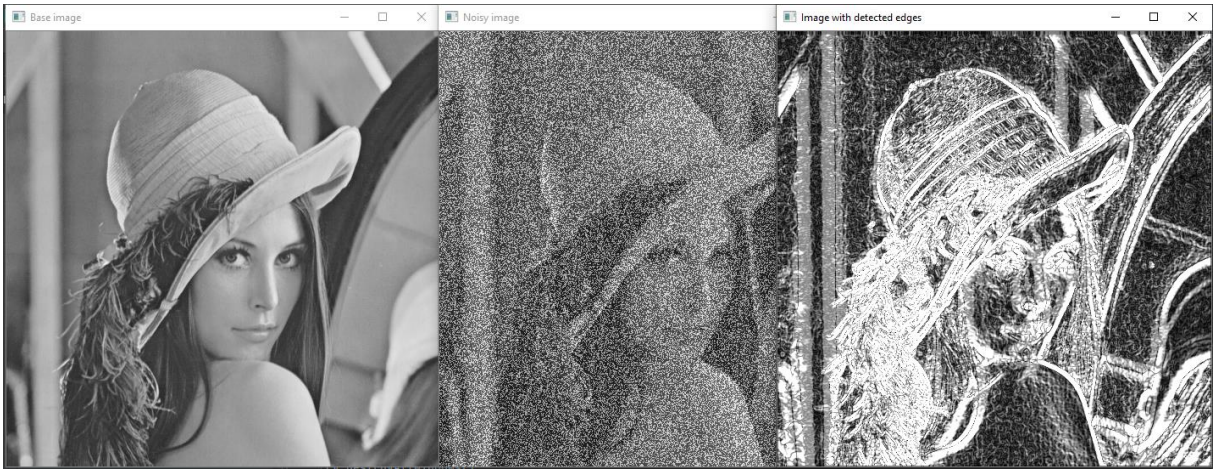
The task combines the Gaussian noise application (from Task 1) with edge detecting (from Task 3). The code is almost identical as in the previous tasks.

Calling the used functions:

```
# Load image
img = cv.imread('lena.png', 0)
#img = cv.imread('pic.jpg', 0)


cv.imshow('Base image', img)

#Adding Gaussian noise
img_noisy = noise(img)
cv.imshow('Noisy image', img_noisy)

#Detecting the edge
edge_img = edging(img)
cv.imshow('Image with detected edges', edge_img)

cv.waitKey()
cv.destroyAllWindows()
```

In this part, we start calling up our functions to see the results.

Results of Task 4:



Here we can see from the left, the original picture, picture with Gaussian noise, and picture with noise after detecting the edges. One could notice, that the edges has also been detected over the noise on the picture, resulting in a bigger number of edges over the picture (in comparison with Task 3).

Task 5:

Defining the function for edge detection:

```
def edging(img):
    scale = 1
    delta = 0
    depth = cv.CV_16S
    grad_x = cv.Sobel(img, depth, 1, 0, ksize=5, scale=scale, delta=delta, borderType=cv.BORDER_DEFAULT)
    grad_y = cv.Sobel(img, depth, 0, 1, ksize=5, scale=scale, delta=delta, borderType=cv.BORDER_DEFAULT)
    abs_grad_x = cv.convertScaleAbs(grad_x)
    abs_grad_y = cv.convertScaleAbs(grad_y)
    grad = cv.addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0)

    return grad
```

Again reused function from Task3.

Defining the function for Laplace filtering:

```
def edging_1(img):
        kernel_sharpening = np.array([[-1, -1, -1],
                                      [-1, 9, -1],
                                      [-1, -1, -1]])
        sharpened = cv.filter2D(img, -1, kernel_sharpening)

        return sharpened


def edging_2(img):
        kernel_sharpening = np.array([[-1, -1, -1],
                                      [-1, 12, -1],
                                      [-1, -1, -1]])
        sharpened = cv.filter2D(img, -1, kernel_sharpening)

        return sharpened
```

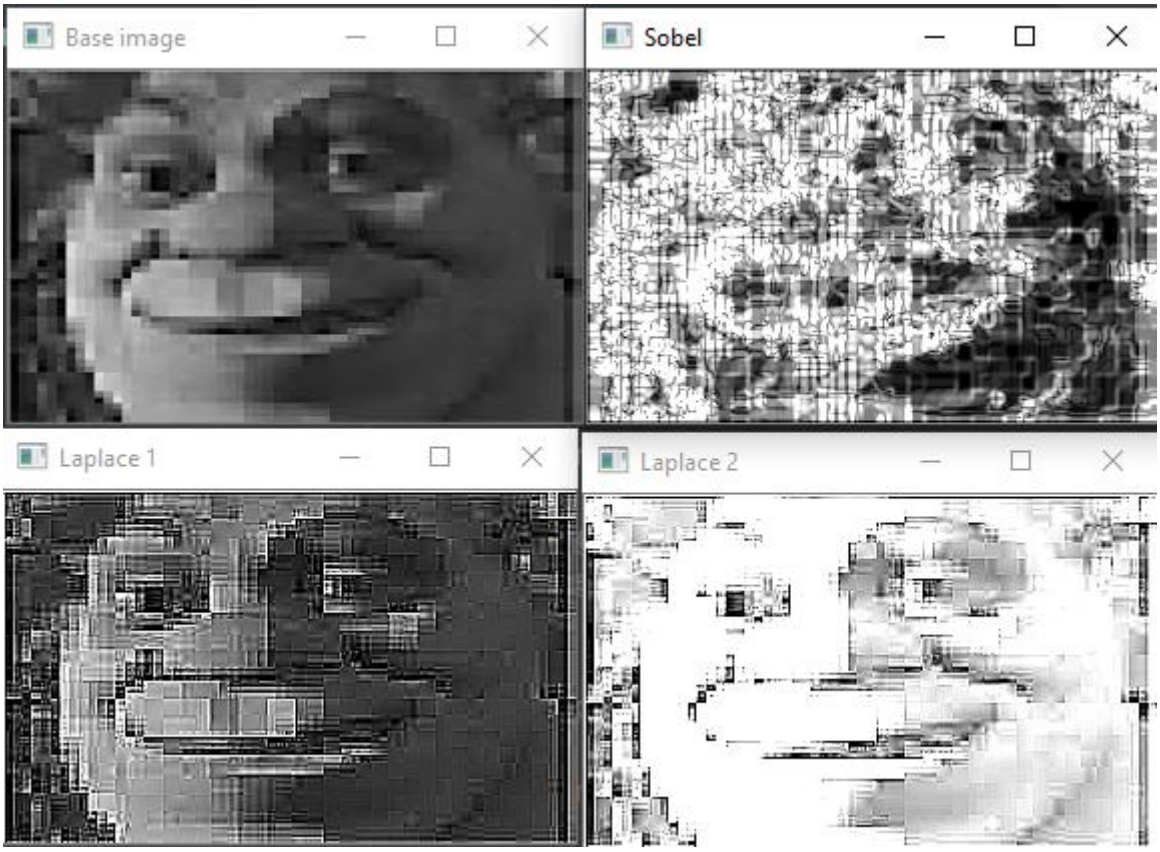
Laplace filters based on the given kernel. The functions behave similarly to the sharpening function. Based on numpy arrays, and OpenCV 2D filtering.

Calling the used functions:

```
# Load image
#img = cv.imread('lena.png', 0)
img = cv.imread('pic.jpg', 0)


cv.imshow('Base image', img)

#Detecting the edge

e1 = edging_1(img)
cv.imshow('Laplace 1', e1)

e2 = edging_2(img)
cv.imshow('Laplace 2', e2)

e_s = edging(img)
cv.imshow('Sobel', e_s)

cv.waitKey()
cv.destroyAllWindows()
```

In this part, we start calling up our functions to see the results.

Results of Task 5:



Here are the results of the edge detecting, by Sobel Operator and the LaPlace filters over the given kernel. We can see that different level of LaPlace pixel filtering, results in much different color scale in output pictures.