DEPARTAMENTO CIÊNCIA
E TECNOLOGIA

Computer Vision
Degree in Information Technology
2º Semester 2020/2021

Worksheet 4

Goals:

- Image Enhancement Techniques
- Low and High pass filters
- Image Segmentation

Exercises

Part I – Low pass Filtering

1- Write and run the following code in a python file:

```
import numpy as np
import cv2
image = cv2.imread("DCT_img.jpg")
cv2.imshow("Original", image)
cv2.waitKey(0)

# averaging blurring
blurred = np.hstack([cv2.blur(image, (3, 3)),
cv2.blur(image, (5, 5)),
cv2.blur(image, (7, 7))])
cv2.imshow("Averaged", blurred)
cv2.waitKey(0)
```

- Import the packages.
- Load the image.
- Then use the cv2.blur function on your image. This function requires two arguments: the image you want to blur and the size of the kernel:
  - define a kxk sliding window (or kernel) on top of the image, where k is always an odd number.
  - This window (or kernel) is going to slide from left-to-right and from top-to-bottom.
  - The pixel at the centre of this matrix (or kernel) is then set to be the average of all other pixels surrounding it.
- What do you conclude?
- You use of the np.hstack function just to stack your output images together in order to avoid to create three separate windows using cv2.imshow function.
- Apply the same code to another image

2- **Average filter**. Write and run the following code in a python file:

```
import numpy as np
import cv2
image = cv2.imread("../DCT_img.jpg")

mask = 1/(3*3)*np.ones([3,3])
local_mean1 = cv2.filter2D(image, -1, mask)

mask = 1/(5*5)*np.ones([5,5])
local_mean2 = cv2.filter2D(image, -1, mask)

mask = 1/(7*7)*np.ones([7,7])
local_mean3 = cv2.filter2D(image, -1, mask)

blurred = np.hstack([local_mean1,local_mean2,local_mean3])
cv2.imshow("filter2d", blurred)
cv2.waitKey(0)
```

- Let us see the code in more detail. We perform the convolution method with cv2.f ilter2D function. It does the same thing as the cv2.Blur function however we need to create the mask first.
- The first argument is the image, the second argument is the output depth. -1 implies the same depth as the input image. The last argument is the kernel.
- The blurring that you had applied to the image is a Low-Pass Filter and it is more often called as averaging filter.
- Low-Pass Filters (LPFs) are spatial filters whose effect on the output image is equivalent to attenuating high-frequency components (i.e., fine details in the image) and preserving low-frequency components (i.e., coarser details and homogeneous areas in the image). Linear LPFs are typically used to either blur an image or reduce the amount of noise present in the image. Examples: Averaging filter, Gaussian, rank filters.

3- **Gaussian Filter**. Write and run the following code in a python file:

```
import numpy as np
import cv2
image = cv2.imread("../DCT_img.jpg")
cv2.imshow("Original", image)
cv2.waitKey(0)

# gaussian blurring
blurred = np.hstack([cv2.GaussianBlur(image, (3, 3),0),
cv2.GaussianBlur(image, (5, 5),0),
cv2.GaussianBlur(image, (7, 7),0)])
cv2.imshow("Gaussian", blurred)
cv2.waitKey(0)
```

- Let us examine the code:
- cv2.GaussianBlur function performs our filter.
- The first argument to the function is the image we want to blur. The second argument is a tuple representing our kernel size, the last parameter is the s. By setting this value to 0, we are instructing OpenCV to automatically compute them based on our kernel size.
- We have less of a blur effect than when using the averaging method. However, the blur itself is more natural due to the computation of the weighted mean, rather than allowing all pixels in the kernel neighbourhood to have equal weight.
- Apply another images

4- **Nonlinear Filtering**. Write and run the following code in a python file:

```
import numpy as np
import cv2
image = cv2.imread("../DCT_img.jpg ")
cv2.imshow("Original", image)
cv2.waitKey(0)

# median blur
blurred = np.hstack([cv2.medianBlur(image, 3),
cv2.medianBlur(image, 5),
cv2.medianBlur(image, 7)])
cv2.imshow("Median", blurred)
cv2.waitKey(0)
```

- In linear filtering each output pixel is a weighted summation of some number of input pixels; easier to compose and are amenable to frequency response analysis.
- In nonlinear filtering a combination of neighbouring pixels is performed.
- They are also known as rank-order filters.
- Median Filter consists in finding the median value on a neighbourhood with a given size. It is more effective in not blurring edges.
- When applying a median blur, we first define our kernel size k.
- Then, as in the averaging blurring method, we consider all pixels in the neighbourhood of size kxk.
- Unlike the averaging method, instead of replacing the central pixel with the average of the neighbourhood, we instead replace the central pixel with the median of the neighbourhood.
- Median blurring is more effective at removing salt-and pepper style noise from an image because each central pixel is always replaced with a pixel intensity that exists in the image.
- Averaging and Gaussian methods can compute means or weighted means for the neighbourhood – this average pixel intensity may or may not be present in the neighbourhood. But, the median pixel must exist in our neighbourhood. By replacing

our central pixel with a median rather than an average, we can substantially reduce noise.

- Let us examine the code:
- Applying a median blur is accomplished by making a call to the cv2.medianBlur function.
- This method takes two parameters: the image we want to blur and the size of our kernel. Notice that we removed detail and noise.

Part II– High Pass Filtering

1- Write and run the following code in a python file

```
import numpy as np
import cv2
image = cv2.imread("../DCT_img.jpg")
cv2.imshow("Original", image)
cv2.waitKey(0)

mask = np.array([[0,-1,0],[-1,4,-1],[0,-1,0]])
laplacian_filter = cv2.filter2D(image, -1, mask)

imgadd = cv2.add(image, laplacian_filter)
cv2.imshow("laplacian", imgadd)
cv2.waitKey(0)
```

- Laplacian filter is a High-Pass Filter (HPF) whose effect on the output image is equivalent to preserving or emphasizing its high-frequency components (e.g., fine details, points, lines, and edges), i.e. to highlight transitions in intensity within the image.
- Linear HPFs can be implemented using 2D convolution masks with positive and negative coefficients, which correspond to a digital approximation of the Laplacian, a simple, isotropic (i.e., rotation invariant) second-order derivative that can respond to intensity transitions in any direction.
- Let us examine the code:
- The convolution was performed by the cv2.f ilter2d function.
- The arithmetic calculation was carried out by cv2.add function1

Part III– Image Segmentation

Extract symbolic information: image and data preprocessing serve the purpose of extracting regions of interest, enhancing and cleaning up the images.

They can be directly and efficiently processed by the feature extraction component. Image segmentation is the task of finding groups of pixels that ”go together” .

Image segmentation techniques can vary widely according to:

- type of image (e.g., binary, gray, color).
- choice of mathematical framework (e.g., morphology, image statistics, graph theory).
- type of features (e.g., intensity, color, texture, motion) and
- approach (e.g., top-down, bottom-up, graph-based).

We can classify the techniques into three categories:

- Intensity-based (non-contextual) methods: work based on pixel distributions (i.e., histograms).
- Region-based (contextual) methods: rely on adjacency and connectivity criteria between a pixel and its neighbors.
- Other methods: these include segmentation based on texture, edges, and motion, among others.

In the Intensity-based methods we have the Thresholding. In general, we seek to convert a grayscale image to a binary image, where the pixels are either 0 or 255.

1- **Simple Thresholding**. Write and run the following code in a python file

```
import cv2
image = cv2.imread("../coins.jpg")
image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
blurred = cv2.GaussianBlur(image, (5, 5), 0)
cv2.imshow("Image", image)

(T, thresh) = cv2.threshold(blurred, 155, 255, cv2.THRESH_BINARY)
cv2.imshow("Threshold Binary", thresh)

(T, threshInv) = cv2.threshold(blurred, 155, 255, cv2.THRESH_BINARY_INV)
cv2.imshow("Threshold Binary Inverse", threshInv)
```

- A simple thresholding example would be selecting a pixel value T, and then setting all pixel intensities less than T to zero, and all pixel values greater than T to 255. In this way, we are able to create a binary representation of the image.
- Let us examine the code:
- Import the packages, load the image, convert to grayscale.

- Apply Gaussian blurring with a s = 5 radius to remove some of the high frequency edges in the image.
- Compute the thresholded image using the cv2.threshold function:
    - The grayscale image to threshold.
    - The T value.
    - Maximum value applied during thresholding: any pixel value that is greater than 155 is set to 255.
    - Thresholding method which indicates that pixel values p greater than T are set to the maximum value (the previous argument).
- The cv2.threshold function returns two values:
    - The value we manually specified for thresholding.
    - The thresholded image.
- Perform masking by using the cv2.bitwise_and function, giving the original image and the inverted thresholded image.

2- **Adaptative Thresholding**. Write and run the following code in a python file

```
import cv2
image = cv2.imread("../coins.jpg")
image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
blurred = cv2.GaussianBlur(image, (5, 5), 0)
cv2.imshow("Image", image)
cv2.waitKey(0)

thresh = cv2.adaptiveThreshold(blurred, 255, cv2.ADAPTIVE_THRESH_MEAN_C,
cv2.THRESH_BINARY_INV, 11, 4)
cv2.imshow("Mean Thresh", thresh)
cv2.waitKey(0)

thresh =
cv2.adaptiveThreshold(blurred,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
cv2.THRESH_BINARY_INV, 15, 3)
cv2.imshow("Gaussian Thresh", thresh)
cv2.waitKey(0)
```

This method allows us to handle cases where there may be dramatic ranges of pixel intensities and the optimal value of T may change for different parts of the image.

In opposite we have global thresholding methods where a threshold value is used for the entire image to separate foreground objects from background. We obtain better results in images with an obviously bimodal histogram. They are not good for images with low contrast or non-uniform illumination.

Let us examine the code:

- The cv2.adaptiveThreshold function applied to the image an adaptative method.
- The first parameter is the image we want to threshold.
- The second parameter is the maximum value of 255.

- The third parameter is the method. By supplying cv2.ADAPTIVE_THRESH_MEAN_C, we indicate that we want to compute the mean of the neighbourhood of pixels and treat it as our T value. In here we also tested Gaussian (weighted mean) thresholding, cv2.ADAPTIVE_THRESH_GAUSSIAN_C.
- The fourth parameter indicate that any pixel intensity greater than T in the neighbourhood should be set to 255, otherwise it should be set to 0.
- The fifth parameter is the neighbourhood size. This integer value must be odd and indicates how large our neighbourhood of pixels is going to be. We supply a value of 11, indicating that we are going to examine 11x11 pixel regions of the image, instead of trying to threshold the image globally, as in simple thresholding methods.
- The sixth parameter is an integer that is subtracted from the mean, allowing us to fine-tune our thresholding.

In general, choosing between mean adaptive thresholding and Gaussian adaptive thresholding requires a few experiments on your end. The most important parameters to vary are the neighbourhood size and C, the value you subtract from the mean.

3- **Otsu**. Write and run the following code in a python file

```
import numpy as np
import mahotas
import cv2
image = cv2.imread("../coins.jpg")
image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
blurred = cv2.GaussianBlur(image, (5, 5), 0)
cv2.imshow("Image", image)

#Otsu
T = mahotas.thresholding.otsu(blurred)
print("Otsu’ s threshold: {}". format(T))
thresh = image.copy()
thresh[thresh > T] = 255
thresh[thresh < 255] = 0
thresh = cv2.bitwise_not(thresh)
cv2.imshow("Otsu", thresh)
cv2.waitKey(0)
```

Otsu’s method assumes there are two peaks in the grayscale histogram of the image and then tries to find an optimal value to separate those.

Let us examine the code:

- As in previous thresholding examples, we convert the image to grayscale and then blur it slightly.
- To compute our optimal value of T, we use the otsu function in the mahotas.thresholdingpackage.
- We make a copy of our grayscale image so that we have an image to threshold.
- Makes any values greater than T white.
- Makes all remaining pixels that are not white into black pixels.

- Then invert our threshold by using cv2.bitwise_not.

Exercise for practical work:

Using this knowledge try to segment an image. It will help in the future to detect where the objects are.