Computer Vision

Assignment 2

Alireza Moradi

September 30, 2020

1 Histogram Equalization

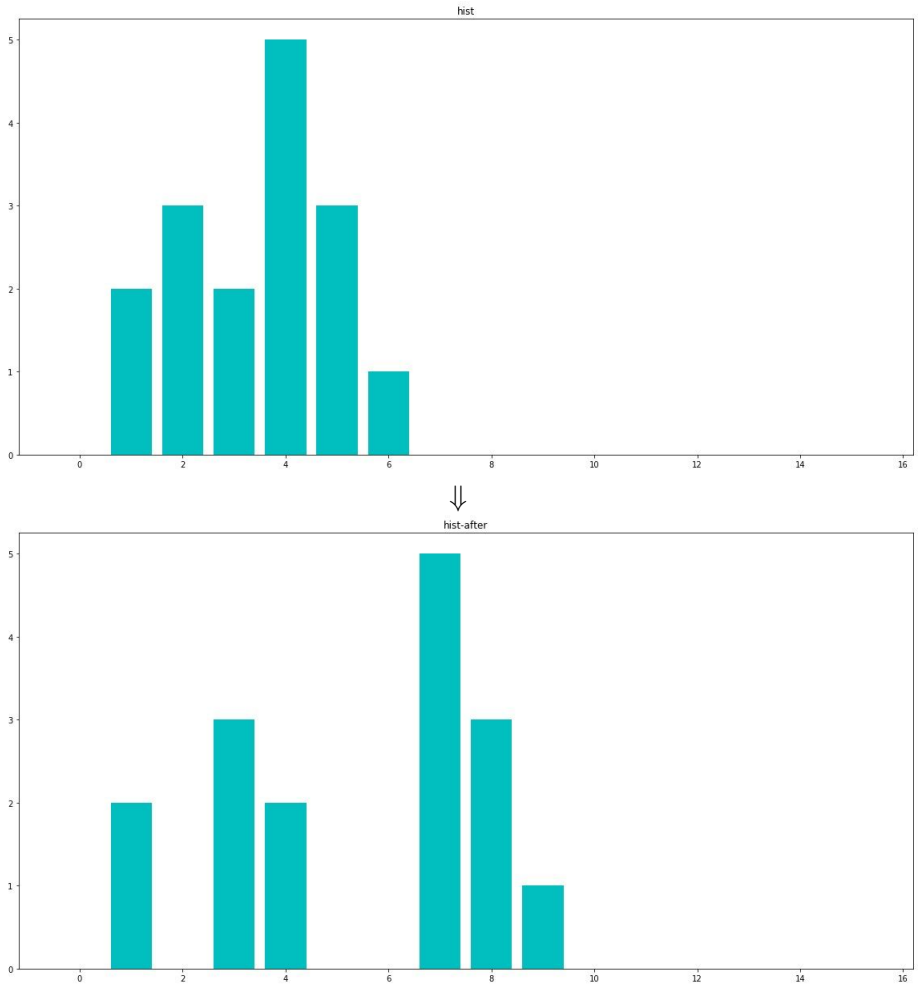
1	2	4	4
2	1	4	5
5	3	4	3
5	4	6	2

$\Downarrow$

k	0	1	2	3	4	5	6	7	8	9
n_k	0	2	3	2	5	3	1	0	0	0
$\sum_{j=0}^{k} n_j$	0	2	5	7	12	15	16	16	16	16
$\sum_{j=0}^{k} \frac{n_j}{n}$	0	$\frac{2}{16}$	$\frac{5}{16}$	$\frac{7}{16}$	$\frac{12}{16}$	$\frac{15}{16}$	1	1	1	1
$(L-1)\sum_{j=0}^{k} \frac{n_j}{n}$	0	1.13	2.81	3.94	6.75	8.44	9	9	9	9
$round$	0	1	3	4	7	8	9	9	9	9

$\Downarrow$

1	3	7	7
3	1	7	8
8	4	7	4
8	7	9	3



2 IP Cameras

Analog cameras record footage and send the signal via a coaxial cable directly to a DVR (Digital Video Recorder). The DVR then converts the video from analog to digital signals, compresses the file and stores it on a hard drive. The traditional CCTV cameras are usually of this type.
IP Cameras are digital video cameras that can send and receive data via network, unlike an analog camera that should send the feed to a DVR. IP cameras have much better picture quality and can do more stuff like motion detection. They are also very scalable unlike analog cameras where you're stuck with a fixed number of cameras.
Link 1
Link 2

3 Implementations

3.1 Compute Histogram

The only thing we have to do for computing the histogram is to calculate how many times each color bucket has been repeated.

```
def compute_histogram(image):
    '''
    Computes histogram of the input image.

    Parameters:
        image (numpy.ndarray): The input image.

    Returns:
        numpy.ndarray: The numpy array of numbers in histogram.
    '''

    histogram = np.zeros((256), np.int)

    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            histogram[image[i][j]] += 1

    return histogram
```

3.2 Histogram Equalization

To do `Histogram Equalization` we have to calculate the frequency of each color bucket. Then we have to normalize it (divide by number of pixels). This will map the colors to numbers between 0 and 1. So we have to multiply it by *color buckets* − 1 to map the color numbers to 0 and *color buckets* − 1. Finally everything is rounded to the nearest integer.

```
def histogram_equalization(image):
    '''
    Equalizes the histogram of the input image.

    Parameters:
        image (numpy.ndarray): The input image.

    Returns:
        numpy.ndarray: The result image that it's histogram be eqaulized.
    '''

    h = compute_histogram(image)
    out_image = image.copy()
    levels = 256
    nk = np.zeros(levels)
    cdf = np.zeros(levels)

    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            nk[image[i][j]] += 1
    cdf[0] = nk[0]
    for i in range(1,len(nk)):
        cdf[i] = cdf[i-1]+nk[i]

    normalized_cdf = np.zeros(levels)

    for i in range(len(normalized_cdf)):
        normalized_cdf[i] = round((cdf[i]/pixel_count)*(levels-1))

    for i in range(out_image.shape[0]):
        for j in range(out_image.shape[1]):
            out_image[i][j] = normalized_cdf[out_image[i][j]]

    return out_image
```

3.3 Histogram Stretching

To do `Histogram Stretching` we use the following formula:

$$g(x,y) = (\frac{f(x,y) - f_{min}}{f_{max} - f_{min}})(MAX - MIN) + MIN$$

Where f_{max} and f_{min} are the maximum and minimum color buckets available in the image respectively, and MAX and MIN are the maximum and minimum color buckets possible respectively. This formula is applied to every pixel in the image.

```python
def histogram_stretching(image):
    '''
    Streches the histogram of the input image.

    Parameters:
        image (numpy.ndarray): The input image.

    Returns:
        numpy.ndarray: The result image that it's histogram be streched.
    '''

    h = compute_histogram(image)
    out_image = image.copy()
    f_min = np.min(out_image)
    f_max = np.max(out_image)
    MAX = 255
    MIN = 0

    for i in range(out_image.shape[0]):
        for j in range(out_image.shape[1]):
            out_image[i][j] = ((out_image[i][j] - f_min)/(f_max - f_min)) * (MAX - MIN) + MIN

    return out_image
```

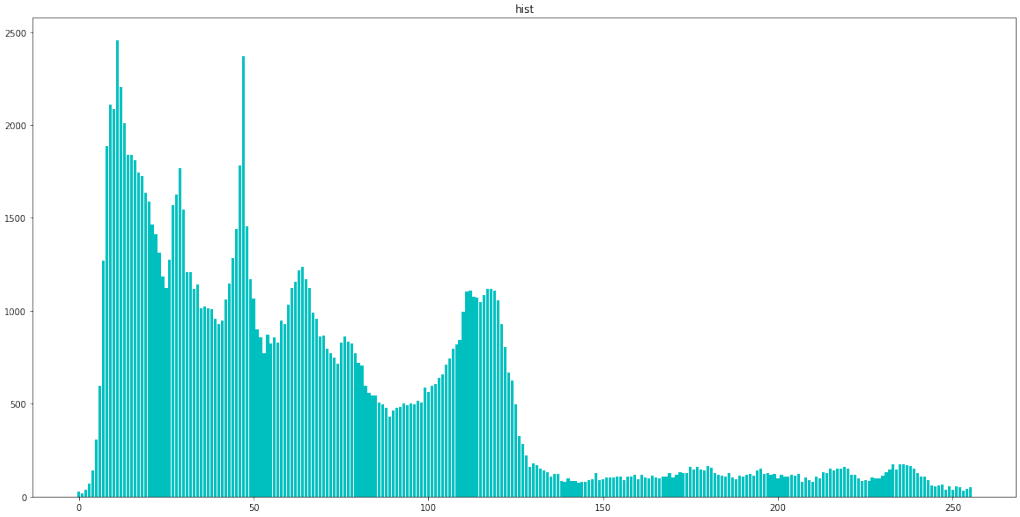
3.4 Comparison

As we see in the histogram of image 2 there are some noises(outliers) in the picture that would mess f_{max} and f_{min} and this will cause the formula used in the previous question to not work good because f_{max} and f_{min} are close to MAX and MIN respectively, so the formula is doing almost nothing. One way to fix this is to use `histogram clipping`, where we will clip some of the top and bottom bins in the histogram and then compute the histogram. In image 3 because the colors are 2 chunks and each of the chunks is near the minimum or the maximum bin in the histogram, the only thing histogram stretching is going to do is put these chunks farther from each other and near the border bins, So there's not much difference in the resulting image. The solution to this is to use `histogram equalization`.

```
img2 = cv2.imread(os.path.join('images', 'img2.jpg'), cv2.IMREAD_GRAYSCALE)
img3 = cv2.imread(os.path.join('images', 'img3.jpg'), cv2.IMREAD_GRAYSCALE)
h2 = compute_histogram(img2)
show_histogram(h2, 'img2_hist', 'q3d_img2_hist')
h3 = compute_histogram(img3)
show_histogram(h3, 'img3_hist', 'q3d_img3_hist')
out2 = histogram_stretching(img2)
out3 = histogram_stretching(img3)
image_list = []
image_list.append([img2, 'input_image2'])
image_list.append([out2, 'histogram_streched_image2'])
image_list.append([img3, 'input_image3'])
image_list.append([out3, 'histogram_streched_image3'])
plotter(image_list, 2, 2, True, 20, 10, 'q3d')
h2_res = compute_histogram(out2)
show_histogram(h2_res, 'img2_streched_hist', 'q3d_img2_streched_hist')
h3_res = compute_histogram(out3)
show_histogram(h3_res, 'img3_streched_hist', 'q3d_img3_streched_hist')
```

3.5 CLAHE

As shown below histogram equalization improved the contrast of the background but the details of the statue's face are lost due to high brightness. If we look at the histogram of the original image we see that it's not confined to a specific region so histogram equalization won't do well.



On the other hand, `adaptive histogram equalization` divides the image to small blocks called `tiles` and then applies histogram equalization. `CLAHE` also applies `contrast limiting` which means that if any histogram bin is above the specified contrast limit, those pixels are clipped and distributed uniformly to other bins before applying histogram equalization.


```python
def clahe(image):
    '''
    Applys the OpenCV's CLAHE on the input image.

    Parameters:
        image (numpy.ndarray): The input image.

    Returns:
        numpy.ndarray: The result image.
    '''
    out_image = image.copy()
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
    out_image = clahe.apply(image)

    return out_image
```