DEPARTMENT OF ELECTRONIC AND TELECOMMUNICATION ENGINEERING
UNIVERSITY OF MORATUWA

EN2550 FUNDAMENTALS OF IMAGE PROCESSING AND MACHINE VISION

ASSIGNMENT 01

Thanushan K. 190621M

This is submitted as a partial fulfilment for the module EN2550.

March 5, 2022

1 Performing a given intensity transformation for a given image

Intensity transformations are transformations that can be implemented on an image in order to further analyse the image. The output pixel only depends on the particular input pixel of the image. Image negative transformations, Gamma transformations, log transformations and piecewise transformations are the most commonly used transformations. Piecewise transformations include contrast stretching, intensity level slicing, bit plane slicing and many other.
The given transformation is a piecewise intensity transformation that intensifies the input pixel values in the range from 50 to 150 while keeping the remaining output pixel values same as the input pixel values. The result is shown in the following figure.

Figure 1: Performing an intensity transformation to a given image

2 Performing Intensity Transformations to accentuate white matter and gray matter

We can use intensity transformations to accentuate different colour regions in the grayscale images. For example, we can separately accentuate white matter and gray matter in a grayscale image by performing intensity transformations. These kind of transformations will be helpful in many applications such as medical image processing.

(a) In this part we make the white colour regions of the brain proton density image more noticeable. This can be done by transforming the region where the pixel values are greater than 180 to 255 and the pixel values less than 180 to 0.

(b) In this part we make the gray colour regions of the brain proton density image more noticeable. This can be done by transforming the region where the pixel values between 50 and 180 to 255 and other pixel values that are less than 50 and greater than 180 to 0.

Figure 2: White matter accentuated images

Figure 3: Gray matter accentuated images

3 Performing Gamma Correction($\gamma = 0.5$) and Calculating Histograms

Gamma correction is one of the basic intensity transformation that can be used to enhance contrast in an image. In this question we have to do gamma correction to the L plane of the image in the Lab color space. The transformation was done by considering $\gamma = 0.5$. The output pixel gets a value by going through a transformation as shown in equation 1. The results are shown below.

$$Outputpixel = 255 * \left(\frac{Inputpixel}{255}\right)^{\gamma} \tag{1}$$

Histograms were plot for the initial image and the gamma corrected image. We can observe that the peak of the histogram has moved slightly towards the right side. This shows that certain pixel values of the gamma corrected image is higher than the original image. This can be observed in the images of figure 5. The darkness of the original has reduced. This phenomenon can be obesreved when $\gamma < 1$. When $\gamma > 1$, the image gets darker than the original image and the peak of the histogram will shift towards left further to a certain extent and then pixels in the right will shift towards left. The Histograms are shown below.

4 Histogram Equalization

Histogram equalization is also an intensity transformation that can be performed in an image such that the pixels are distributed all over the region that the pixels could take values. The result will be an improved version of the image. In this question we have to use a manual function for equalizing the image. Initially, the histogram of the input image is obtained. Then the cdf value is calculated at each point. The

Figure 4: Results of doing gamma correction to a single plane ($\gamma = 0.5$)

Figure 5: Histograms of the original image and the gamma corrected image

calculated cdf value is multiplied by the maximum possible pixel value (L-1) and divided by the product of the dimensions of the input image. The transformation is given by equation 2.

$$s_k = \frac{L-1}{MN} \sum_{j=0}^{k} n_j \tag{2}$$

Here M and N are the dimensions of the input image. The value of k ranges from 0 to L-1. Finally, the histogram of the output image is obtained. The histograms of the Equalized images created from the custom function and the in-built opencv function are plotted together and compared. We can obtain almost similar histogram plots in both cases. The results are shown in the following figure.

Figure 6: Histogram plots before and after equalization

```python
#Equalization function
def Equalize(Original_image, Original_image_histogram, S):
    N = len(Original_image_histogram)
    Image = np.zeros((S[0], S[1]))
    Equalize_List = []
    for i in range(N):
        Sumlist = Original_image_histogram[:i+1]
        Value = ((N-1)/(S[0]*S[1]))*(sum(Sumlist))
        Equalize_List.append(Value)
    Equalize_List = np.round(Equalize_List)
    for i in range(S[0]):
        for j in range(S[1]):
            Value = Original_image[i,j]
            Image[i,j] = Equalize_List[Value]
    Image = Image.astype(np.uint8)
    return Image
```

Figure 7: Code for equalizing image

5 Zooming a given image

We need to zoom images to do an analysis of these images. There are various techniques in zooming. Nearest neighbor interpolation, bilinear interpolation, bicubic interpolation are some of those techniques. In this question, we consider nearest neighbor interpolation and bilinear interpolation to calculate the normalized sum of squared difference (SSD).

(a) In this part, we zoom the image using nearest neighbor interpolation method. The image pixels values are determined by finding the pixel value at the nearest pixel position to the required pixel position.

```python
#Zoom functon
def NearestNeighborZoom(Image, scale):
    shape = np.shape(Image)
    Rows = int(shape[0]*scale)
    Columns = int(shape[1]*scale)
    Zoomed_image = np.zeros((Rows, Columns, shape[2]), dtype = np.uint8)
    for i in range(0, Rows):
        for j in range(0, Columns):
            X = min(shape[0]-1, round(i/scale))
            Y = min(shape[1]-1, round(j/scale))
            Zoomed_image[i, j] = Image[int(X), int(Y)]
    return Zoomed_image
```

Figure 8: Code for nearest neighbor interpolation Zooming

(b) Bilinear interpolation technique is a technique where the nearest 4 neighbors of the required point are taken and 2 values in the x direction and y direction are calculated by multiplying the input pixel values with the ratio on how the point is located and summing them up. Then the final output pixel is obtained by doing a similar step for the immediate results obtained earlier.

The SSD values calculated for 3 images under each zooming technique are shown in the table below. From this table we can conclude that bilinear interpolation zooming is better than nearest neighbor interpolation zooming, since the SSD values in bilinear interpolation method are less than that in nearest neighbor interpolation method.

```python
def BilinearZoom(Image, scale):
    shape = np.shape(Image)
    Rows = int(shape[0]*scale)
    Columns = int(shape[1]*scale)
    Zoomed_image = np.zeros((Rows, Columns, shape[2]), dtype = Image.dtype)
    for m in range(0, Rows):
        for n in range(0, Columns):
            h = m/scale
            w = n/scale
            h_prev = int(max(0, np.floor(h)))
            w_prev = int(max(0, np.floor(w)))
            h_next = int(min(shape[0]-1, np.ceil(h)))
            w_next = int(min(shape[1]-1, np.ceil(w)))
            if (h_next==h_prev) and (w_next==w_prev):
                Pixel = Image[int(h), int(w), :]
            elif(h_next == h_prev):
                Immediate1 = Image[int(h), int(w_prev), :]
                Immediate2 = Image[int(h), int(w_next), :]
                Pixel = Immediate1*(w_next - w) + Immediate2 * (w - w_prev)
            elif (w_next == w_prev):
                Immediate1 = Image[int(h_prev), int(w), :]
                Immediate2 = Image[int(h_next), int(w), :]
                Pixel = Immediate1*(h_next - h) + Immediate2*(h - h_prev)
            else:
                pixel1 = Image[h_prev, w_prev, :]
                pixel2 = Image[h_next, w_prev, :]
                pixel3 = Image[h_prev, w_next, :]
                pixel4 = Image[h_next, w_next, :]
                Immediate1 = pixel1*(h_next - h) + pixel2*(h - h_prev)
                Immediate2 = pixel3*(h_next - h) + pixel4*(h - h_prev)
                Pixel = Immediate1*(w_next - w) + Immediate2*(w - w_prev)
            Zoomed_image[m, n, :] = Pixel
    return Zoomed_image
```

Figure 9: Code for nearest bilinear interpolation Zooming

Image	**SSD-Nearest Neighbor Interpolation**	**SSD-bilinear Interpolation**
Image 1	641.7879	628.1125
Image 2	268.6875	259.3884
Image 3	391.3376	369.592

Table 1: SSD Values.

6 Sobel Filtering

Sobel Filtering is a filtering technique that helps to filter out the edges of a given image. Sobel vertical filter is used to find the horizontal edges of the image, while the sobel horizontal filter is used to identify the vertical edges.

(a) In this part, the image is sobel filtered using the opencv in-built function filter2D.

Figure 10: Sobel filtering using filter2D function

(b) In this part, we construct a filtering function code so that the image can be filtered using any given kernel. We initially determine the padding values required to pad the image. Then the kernel is run

through the whole image and the pixel values located in the locations, covered by the kernel are multiplied by the values in the kernel and they are summed up to get the output pixel value. We get similar results as using the in built function.

Figure 11: Sobel filtering using custom filter function

```
#Manual Sobel Filtering
def ManualFilter(Image, Kernel):
    shape = np.shape(Kernel)
    k_hh, k_hw = int(np.floor(shape[0]/2)), int(np.floor(shape[1]/2))
    h, w = Image.shape
    Image_changedtype = Image.  (parameter) Image: Any
    filtered_image = np.zeros(Image.shape, 'float')
    for i in range(k_hh, h - k_hh):
        for j in range(k_hw , w - k_hw):
            filtered_image[i, j] = np.dot(Image_changedtype[i - k_hh : i + k_hh + 1 , j - k_hw : j + k_hw + 1].flatten(), Kernel.flatten())
    return filtered_image
```

Figure 12: Code for manual filtering

(c) In this part, 2 arrays [1 ,2, 1] and $[-1, 0, 1]^T$ are defined and the input image is filtered using the arrays one by one for vertical filtering. The image is filtered with the transposes of the arrays for horizontal filtering. The results obtained are similar to the results obtained using the filter2D function. This shows the associative property of the filters.

Figure 13: Sobel filtering using associative property

7 Enhancing a given Image

(a) A mask of the flower in the input image is created binary thresholding function in opencv. The mask image is read, foreground and background models are created accordingly. Finally Grabcut function was executed to separate the foreground flower and the background image. A value for binary

Figure 14: Separating foreground and background

thresholding was set such that the flower region is only shown as white. A custom mask was used instead of rectangular mask to prevent errors of selecting the exact position.

(b) The separated foreground image is enhanced by increasing the saturation while the background is blurred using the Gaussian blur function. the foregroundand background sections are merged together later.

Figure 15: Enhancing the image

(c) When the kernel size increases, the edges of the foreground image region which is black in the background image gains a significant value after gaussian blur so that when the foreground image is added to the background image, the edges of the foreground image do not gain the expected pixel value. Therefore, the edges are quite dark.

Code Files : All relevant codes and files can be found in **GitHub**.