EN3160 – Assignment 1 on Intensity Transformations and Neighborhood Filtering

Name – De Silva A.L.U.P. **Index – 200105F**

GitHub Link – https://github.com/UlinduP/EN3160_Image_Processing_and_Machine_Vision.git

Question 01

```
c = np.array([(50,50),(50,100),(150,255),(150,150)])
t1 = np.linspace(0, c[0,1], c[0,0]+1-0).astype('uint8')
t2 = np.linspace(c[1,1], c[2,1], c[2,0] - c[1,0]).astype('uint8')
t3 = np.linspace(c[3,1], 255, 255 - c[3,0]).astype('uint8')

transform = np.concatenate((t1, t2), axis=0).astype('uint8')
transform = np.concatenate((transform, t3), axis=0).astype('uint8')
assert len(transform) == 256
```

Figure 1 - Generating the Transform

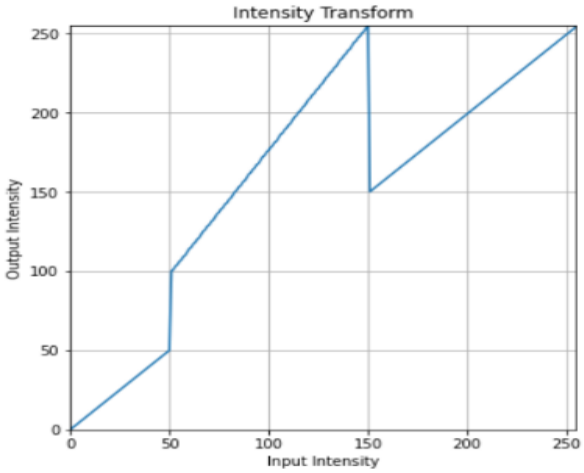

Figure 2 - Intensity Transformation

Figure 3(a) - Image before applying transformation

Figure 3(b) - Image after applying transformation

Discussion – The pixel values in [0,50] and [150,255] are mapped to the same values. Hence, we do not see much difference in her hair color, which would mostly fall in the [0,50] range. Mid-values between 50 and 150 are exaggerated to higher values, so we see white patches on the right side of the face. The left side of the face is lighter, falling in the range of 150+, so no significant change is observed.

Question 02

```
lower_thresh = 180
upper_thresh = 255
c = np.array([(0,lower_thresh),(lower_thresh,upper_thresh)])
t1 = np.linspace(0, c[0,0], c[0,1]+1-0).astype('uint8')
t2 = np.linspace(c[1,1], 255, 255 - c[1,0]).astype('uint8')

transform = np.concatenate((t1, t2), axis=0).astype('uint8')
assert len(transform) == 256
```

Figure 4(a) - Generating transformation for white matter

```
lower_thresh = 100
upper_thresh = 180
c = np.array([(0,lower_thresh),(lower_thresh,255),(upper_thresh,255)])
t1 = np.linspace(0, c[0,0], c[0,1]+1-0).astype('uint8')
t2 = np.linspace(c[1,1],c[2,1],c[2,0] - c[1,0] ).astype('uint8')
t3 = np.linspace(0, 0, 255 - c[2,0]).astype('uint8')

transform = np.concatenate((t1, t2), axis=0).astype('uint8')
transform = np.concatenate((transform, t3), axis=0).astype('uint8')
assert len(transform) == 256
```

Figure 4(b) - Generating transformation for gray matter

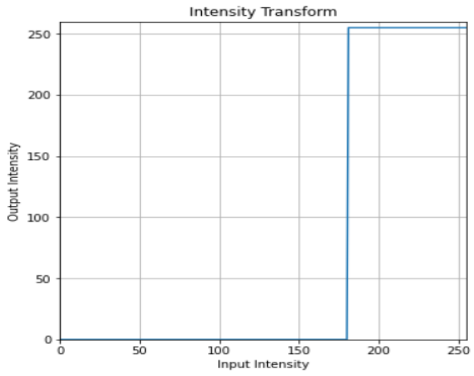

Figure 5(a) - Intensity transformation for white matter

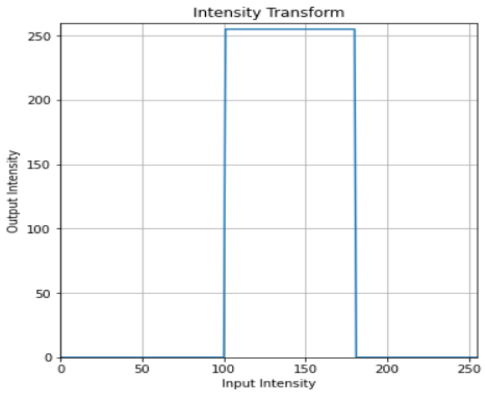

Figure 5(b) - Intensity transformation for gray matter

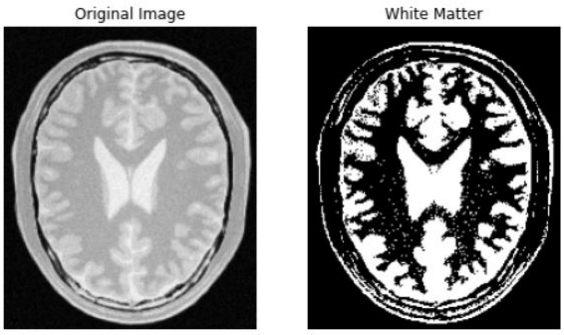

Figure 6(a) - Accentuated white matter

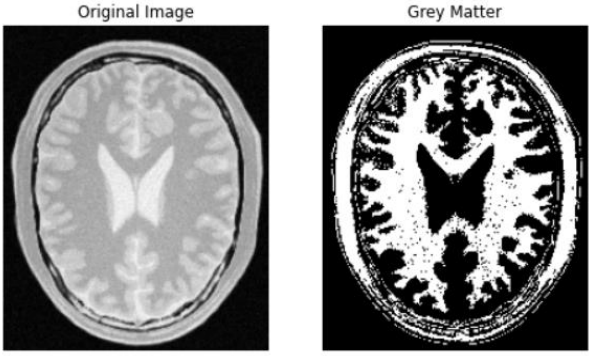

Figure 6(b) - Accentuated gray matter

Discussion – Compared to the original with Figure 6(a), white parts are enhanced since all the values past 180 are transformed to purely white. In Figure 6(b), gray parts are taken as values from 100 to 180 in the middle, which are enhanced to white and others to black.

Question 03

```
L, a, b = cv.split(cv.cvtColor(img, cv.COLOR_BGR2LAB))
gamma = [0.2, 0.8, 1.2, 2]

for i in gamma:
    t = np.array([(p/255)**i*255 for p in range(0,256)]).astype(np.uint8)
    g = cv.LUT(L, t)

    corrected_img = cv.merge([g, a, b])

    hist1 = cv.calcHist([img], [0], None, [256], [0, 256])
    hist2 = cv.calcHist([corrected_img], [0], None, [256], [0, 256])
```

Figure 7 - Gamma correction and generating histograms

Figure 8(a) - Gamma = 0.2

Figure 8(b) - Gamma = 0.8

Figure 8 (c) - Gamma = 2

Discussion - Gamma Value = 1: The gamma correction curve is linear, and the image remains unchanged. The histogram retains its original shape.

Gamma Value < 1 (e.g., 0.2): Gamma compression occurs. This darkens the image and reduces contrast. The histogram shifts to the left, concentrating more values towards the darker end.

Gamma Value > 1 (e.g., 2.0): Gamma expansion occurs. This brightens the image and increases contrast. The histogram shifts to the right, concentrating more values towards the lighter end.

Question 04

(a) image_hsv = cv.cvtColor(image_bgr, cv.COLOR_BGR2HSV)
h, saturation_plane, v = cv.split(image_hsv)

(b) def intensity_transform(x, a, sigma=70):
f_x = np.clip(x+a*128*np.exp(-(x-128)**2/(2*sigma**2)), 0, 255)
return f_x
transforemed_saturation_plane = intensity_transform(saturation_plane, 0.4)

- Visually Pleasing results were obtained for values in the [0.3, 0.4] range.

(c) image_copy = image_hsv.copy()
image_copy[:, :, 1] = transforemed_saturation_plane

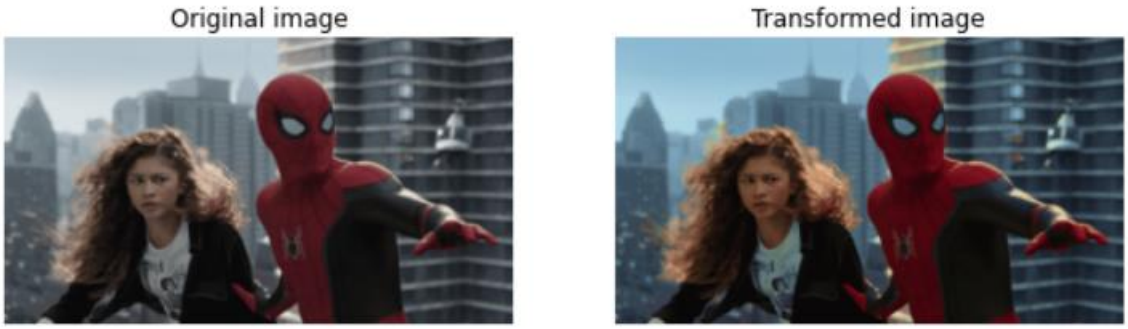

Figure 9 - Original and Enhanced images

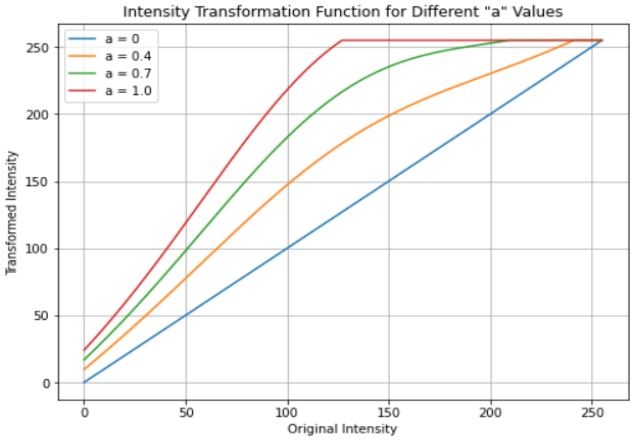

Figure 7 - Intensity Transformations

intensity_values = np.arange(256)

a_values = [0, 0.4, 0.7, 1.0]

for a in a_values:

transformation_values = intensity_transform(intensity_values, a)

Discussion - The code modifies the saturation plane by applying this intensity transformation, effectively altering the image's color saturation. Areas with lower saturation become more pronounced, increasing color vibrancy and contrast. This improves color saturation but lacks clarity in highlighting image details. Parameter 'a' requires fine-tuning for optimal results.

Question 05

```
def histogram_equalization(im):
    cv.imread(im,cv.IMREAD_GRAYSCALE)
    histogram = np.zeros(256, dtype=int)
    for pixel_value in img.flat:
        histogram[pixel_value] += 1
    cdf = np.zeros(256, dtype=int)
    cdf[0] = histogram[0]
    for i in range(1, 256):
        cdf[i] = cdf[i - 1] + histogram[i]
    num_pixels = img.size
    equalized_image = np.zeros_like(img)
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
```

```
            pixel_value = img[i, j]
            equalized_pixel = int((cdf[pixel_value] / num_pixels) * 255)
            equalized_image[i, j] = equalized_pixel
    histogram_equalized = np.zeros(256, dtype=int)
    for pixel_value in equalized_image.flat:
        histogram_equalized[pixel_value] += 1
```

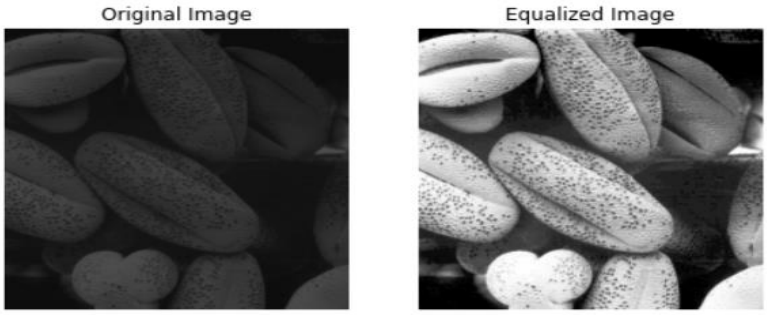

Figure 11 - Images before and after equalization

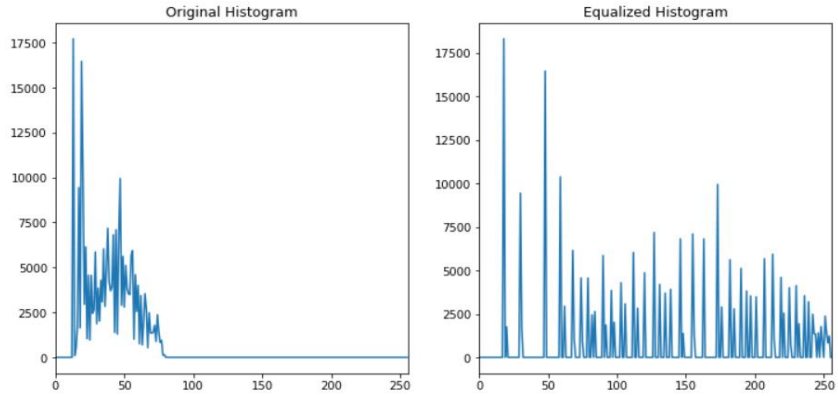

Figure 12 - Histogram before and after equalization

Discussion – This calculates the histograms and performs histogram equalization using fundamental principles. Histogram values of the original are mapped to the entire range. The transformed intensities are used to create an equalized image. This image has a more balanced distribution of intensities, effectively spreading out the data across the entire dynamic range. Hence, the darker image has brightened up.

Question 06

(a) hsv_image = cv.cvtColor(image, cv.COLOR_BGR2HSV)
hue, saturation, value = cv.split(hsv_image)

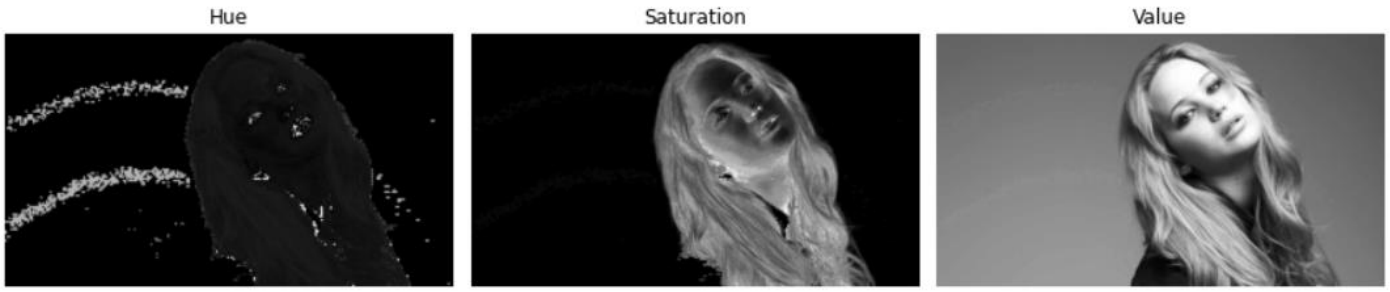

Figure 13

(b) The foreground object is highly saturated compared to the background. Therefore, the Saturation plane is the appropriate plane for the threshold in extracting the foreground mask.

(c) saturation_min = 15, saturation_max = 255
```
foreground_mask = cv.inRange(saturation, saturation_min, saturation_max)
foreground_mask = cv.morphologyEx(foreground_mask, cv.MORPH_CLOSE,
cv.getStructuringElement(cv.MORPH_ELLIPSE,(80, 80)))
foreground = cv.bitwise_and(image, image, mask=foreground_mask)
histogram = cv.calcHist([foreground], [0], foreground_mask, [256], [0, 256])
```
(d) cumulative_histogram = np.cumsum(histogram)

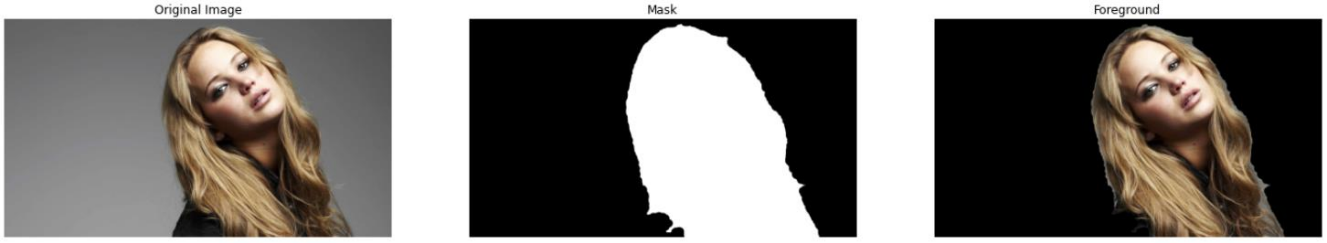

Figure 14

(e)
```
hsv_foreground = cv.cvtColor(foreground, cv.COLOR_BGR2HSV)
value_foreground = hsv_foreground[:, :, 2]
equalized_value_foreground = cv.equalizeHist(value_foreground)
hsv_foreground[:,:,2] = equalized_value_foreground
```

(f)
```
background_mask = cv.bitwise_not(foreground_mask)
extracted_background = cv.bitwise_and(image, image, mask=background_mask)
result = cv.add(extracted_background, cv.cvtColor(hsv_foreground, cv.COLOR_HSV2BGR))
```

Figure 15

Discussion – The foreground object is highly saturated in comparison to the background. Therefore, the saturation plane is selected for thresholding. Exact values need to be fine-tuned. In the saturation plane, the lighter pixels are selected using cv.inRange. But darker pixels are in the foreground like the eyes, which are ignored in the mask. To get them, and to remove noise, morphologyEx operation is used.

Question 07

(a)
```
sobel_v = np.array([(-1, -2, -1), (0, 0, 0), (1, 2, 1)], dtype='float32')
sobel_h = np.array([(-1, 0, 1), (-2, 0, 2), (-1, 0, 1)], dtype='float32')
imv = cv.filter2D(im, -1, sobel_v)
imh = cv.filter2D(im, -1, sobel_h)
grad_mag = np.sqrt(imv**2 + imh**2)
```

(b)
```
rows, columns = im.shape
kernal_size = 3
imv = np.zeros((rows-kernal_size+1, columns-kernal_size+1), dtype='float32')
imh = np.zeros((rows-kernal_size+1, columns-kernal_size+1), dtype='float32')
for row in range(rows-kernal_size+1):
    for column in range(columns-kernal_size+1):
        imv[row, column] = np.sum(im[row:row+kernal_size, column:column+kernal_size] * sobel_v)
        imh[row, column] = np.sum(im[row:row+kernal_size, column:column+kernal_size] * sobel_h)
```

```
grad_mag = np.sqrt(imv**2 + imh**2)
```

(c)
```
sobel_h_kernel = np.array([1, 2, 1], dtype=np.float32)
sobel_v_kernel = np.array([1, 0, -1], dtype=np.float32)
im1 = cv.sepFilter2D(im, -1, sobel_h_kernel, sobel_v_kernel)
im2 = cv.sepFilter2D(im, -1, sobel_v_kernel, sobel_h_kernel)
grad_mag = np.sqrt(imv**2 + imh**2)
```

Discussion - In all the methods, similar results were obtained. After applying the convolution operation, the resultant dimension of the image is (rows - kernal size + 1, columns – kernel size +1). Sobel vertical computes the gradient of the image in the vertical direction, highlighting changes in intensity from top to bottom. Vertical edges, such as the edges of tall objects or structures in an image, are enhanced. Sobel horizontal computes the gradient of the image in the horizontal direction, highlighting changes in intensity from left to right. Horizontal edges, such as the edges of wide objects or structures in an image, are enhanced. The vertical and horizontal gradients are combined to calculate the overall gradient at each pixel.

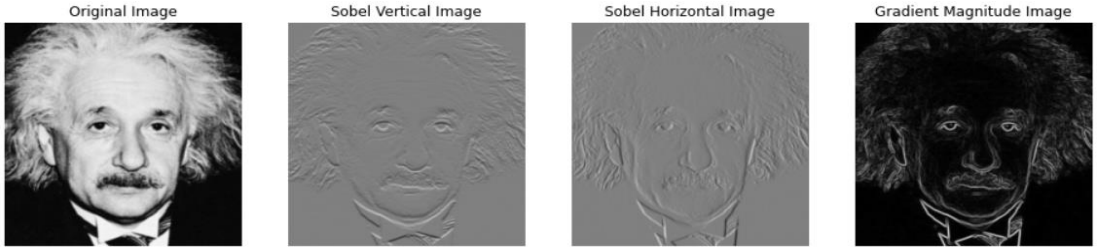

Figure 16

Question 08

(a)
```
if (method == 'nearest-neighbour'):
    for i in range(rows):
        for j in range(cols):
            original_i = min(int(round(i / scale)), image.shape[0] - 1)
            original_j = min(int(round(j / scale)), image.shape[1] - 1)
            zoomed[i][j] = image[original_i][original_j]
```

(b)
```
if (method == 'bilinear interpolation'):
    for i in range(rows):
        for j in range(cols):
            x,y = i/scale, j/scale
            x1, y1 = int(np.floor(x)), int(np.floor(y)
            x2, y2 = int(np.ceil(x)), int(np.ceil(y))
            if (x2 >= image.shape[0]): x2 = x1
            if (y2 >= image.shape[1]): y2 = y1
            dx, dy = x - x1, y - y1
            zoomed[i][j] = image[x1][y1]*(1-dx)*(1-dy) +
image[x1][y2]*(1-dx)*dy + image[x2][y1]*dx*(1-dy) + image[x2][y2]*dx*dy
```

Figure 17 - Zoomed using bilinear interpolation

Discussion – The results from bilinear interpolation are better than those from the nearest neighbor method in all the images. Nearest neighbor interpolation can create blocky artifacts in the zoomed image because it does not consider the values of neighboring pixels smoothly. Bilinear interpolation, while smoother, can still introduce blurring and other artifacts because it assumes a linear transition between pixel values. Before calculating the normalized SSD value, error handling is done to check whether the generated image and the large original have the same size. If not, the same size is attained using cv.resize for the generated image. (Code below) The results are still significantly noisy and below par compared with the original.

```
try:
    assert large_originals[i].shape == zoomed_image_nearest.shape
```

```
    except AssertionError:
        zoomed_image_nearest = cv.resize(zoomed_image_nearest,
(large_originals[i].shape[1],large_originals[i].shape[0]), interpolation=cv.INTER_NEAREST)
        zoomed_image_bilinear = cv.resize(zoomed_image_bilinear,
(large_originals[i].shape[1],large_originals[i].shape[0]), interpolation=cv.INTER_LINEAR)
    finally:
        ssd_nearest = normalized_SSD(zoomed_image_nearest, large_originals[i])
        ssd_bilinear = normalized_SSD(zoomed_image_bilinear, large_originals[i])
```

Image Name (As given in the folder)	Normalized SSD value (Nearest Neighbor)	Normalized SSD value (Bilinear Interpolation)
Im01small	0.49838	0.49266
Im02small	0.20384	0.20019
Im03small	0.23394	0.23826
Im04small	0.69888	0.69763

SSD values are preferred to be as close to 0 as possible, but the calculated SSD values for all the 11 images are considerably high, further showing the below-par performance of the models.

Question 09

```
mask = np.zeros(img.shape[:2], np.uint8)
background_model = np.zeros((1, 65), np.float64)
foreground_model = np.zeros((1, 65), np.float64)
rectangle = (40, 10, 505, 505)
cv.grabCut(img, mask, rectangle, background_model, foreground_model, 5,
cv.GC_INIT_WITH_RECT)
mask1 = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')
img1 = img * mask1[:, :, np.newaxis]
cv.grabCut(img, mask, rectangle, background_model, foreground_model, 5,
cv.GC_INIT_WITH_RECT)
mask2 = np.where((mask == 3) | (mask == 1), 0, 1).astype('uint8')
img2 = img * mask2[:, :, np.newaxis]
blurred_img = img1 + cv.GaussianBlur(img2, (15, 15), 0)
```

Figure 18

Discussion – Dark pixels just beyond the edge of the flower in the enhanced image can be due to many reasons. After GrabCut segmentation, two masks are created: one for the foreground (the flower) and one for the background. Pixels in the background mask, especially those close to the boundary, might not be accurate. Some pixels just beyond the edge of the flower may still be assigned to the background because of the limitations of the initial segmentation. Blurring is a local operation and affects neighboring pixels. The blurring operation can extend into the region where some of the background pixels were inaccurately classified as being part of the flower. This can make them appear darker because they are mixed with the true background. The pixels just beyond the flower's edge, which were inaccurately classified as background but got blurred, contribute to the darker appearance in that region.