05_DVC4IL: Digital Imaging / Visual Computing
Lab Report 1

Eren Yeşiltepe

May 8, 2024

Abstract

In this lab unit, we deal with 2 exercises: Dynamic contrast adjustment and histogram matching. In both exercises, we used histograms to manipulate the images.

1.1 Task A

The contrast adjustment is a linear point operation ($f(a) = ka + d$) that scales and offsets pixels (a) in the input image such that the contrast is increased. The scaling factor k is computed as the ratio between the new maximum and minimum pixel value, a_hi, a_lo , in the input image:

$$k = \frac{255}{a\_hi - a\_lo}.$$

. The offset is the difference between the minimum pixel value and 0 (the new minimum):

$$d = -a\_lo.$$

The result of this operation is that the minimum pixel value (alo) in the input image is mapped to 0 and the maximum pixel value (ahi) is mapped to 255. Typically, the minimum and maximum pixel values (ahi,alo) are chosen such that a certain percentage of pixels is darker and brighter. Darker and brighter pixels are clipped to 0 and 255, respectively. A typical choice is to clip the 1% darkest and 1% brightest pixels.

1.1.1 What I did

I used

$$f_{ac}(a) = a_{min} + (a - a_{lo}) * \frac{a_{max} - a_{min}}{a_{hi} - a_{lo}}$$

equation to remap the points.

1.1.2 Including code

```
def remap(x,min,max):
    res=(255/(max-min))*(x-min)
    return res

def calculate_threshold_indices(cumulative, n_pixels, hi_lo):
    a_lo = next((i for i, perc in enumerate(cumulative) if perc / n_pixels >= hi_lo)
    , -1)
    a_hi = next((i for i in range(len(cumulative) - 1, -1, -1) if cumulative[i] /
    n_pixels <= 1.0 - hi_lo), -1)
    return a_lo, a_hi

def auto_contrast(img, hi_lo = 0.1):
    """ Function to perform auto contrast on an image

    Args:
        img: The image to perform auto contrast on
        hi_lo: The percentage of pixels to clip from the top and bottom

    Returns:
        The image with auto contrast applied
    """
    img_copy = img.copy()

    histogram, bins = np.histogram(img_copy.flatten(), 256, [0, 256])

    cumulative = np.cumsum(histogram)

    # Calculate threshold indices
    n_pixels = img_copy.shape[0] * img_copy.shape[1]
    a_lo, a_hi = calculate_threshold_indices(cumulative, n_pixels, hi_lo)

    for i in range(img_copy.shape[0]):
        for j in range(img_copy.shape[1]):
            if a_lo < img_copy[i, j] and img_copy[i, j] < a_hi:
                img_copy[i, j] = remap(img_copy[i, j], a_lo, a_hi)
    return img_copy

grayscale = cv2.imread("gogh.jpg", cv2.IMREAD_GRAYSCALE)
# test the auto contrast function
auto_contrast_img = auto_contrast(grayscale, hi_lo = 0.01)
show_image_and_hist(auto_contrast_img, use_cumulative=False)
```

1.1.3 Results

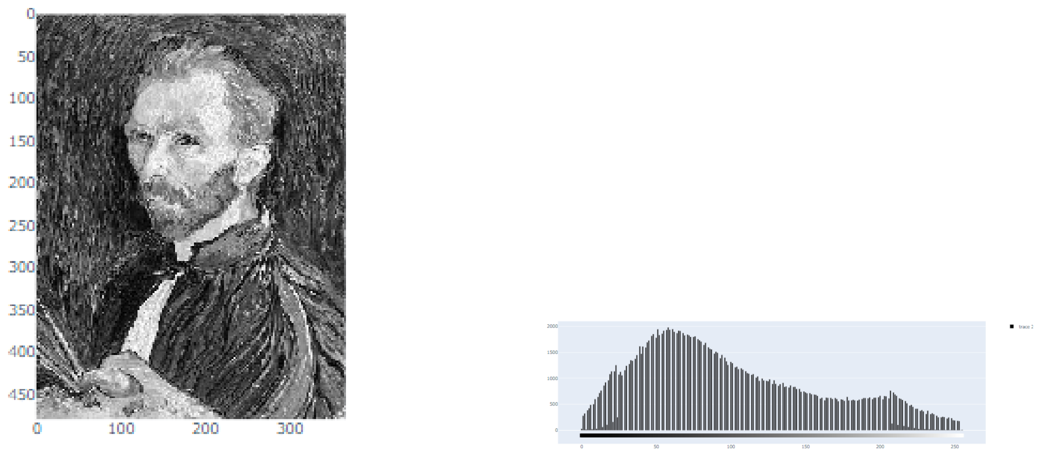

Figure 1.1: Contrast applied image

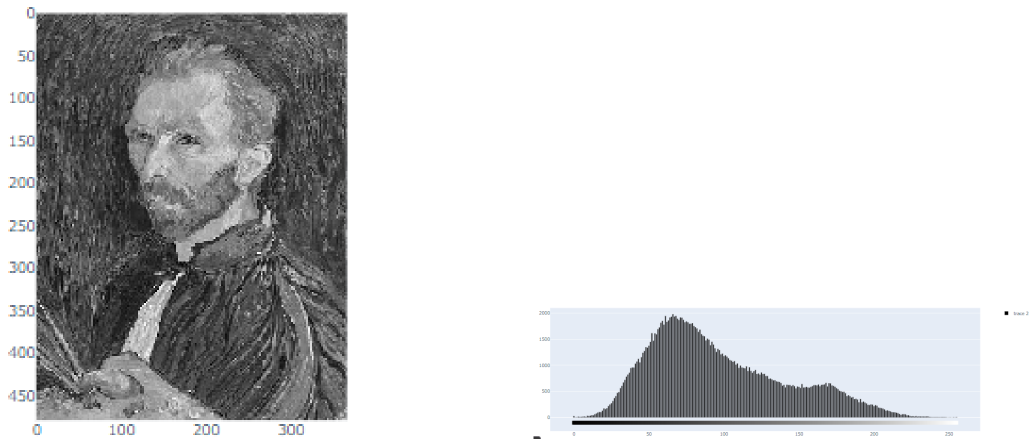

Figure 1.2: Unprocessed Image

1.2 Task B

- The aim of the task is to write a function that takes two images as parameters and then alters the second image so that the two images' histograms will be the same. It is used to transfer the tonal characteristics of one image to another.

1.2.1 What I did

First, we compute the histogram and cumulative distribution function (CDF) for the reference image. Then, we do the same for the input image. Next, we align the CDFs of the input image with that of the reference image, utilizing linear interpolation to determine corresponding intensity values. Lastly, we apply this alignment to the input image, resulting in the matched image.

1.2.2 Including code

```
# Solution Task B

import numpy as np

def match\_histograms(img, ref):
    # Ensure images have the same resolution
    assert img.shape == ref.shape, "Images must have the same resolution"

    # Ensure pixel values are in the range [0, 255]
    img = np.clip(img, 0, 255).astype(np.uint8)
    ref = np.clip(ref, 0, 255).astype(np.uint8)

    # Calculate histogram of reference image
    hist_ref, _ = np.histogram(ref.flatten(), bins=256, range=(0, 256))

    # Compute cumulative distribution function (CDF) of reference image
    cdf_ref = hist_ref.cumsum()
    cdf_ref_normalized = cdf_ref / float(cdf_ref.max())

    # Compute mapping between CDFs
    hist_img, bins = np.histogram(img.flatten(), bins=256, range=(0, 256))
    cdf_img = hist_img.cumsum()
    cdf_img_normalized = cdf_img / float(cdf_img.max())

    mapping = np.interp(cdf_img_normalized, cdf_ref_normalized, range(256))

    # Apply mapping to input image
    matched_img = mapping[img]

    return matched_img

reference = cv2.imread("gogh.jpg", cv2.IMREAD_GRAYSCALE)
image = cv2.imread("cat.jpg", cv2.IMREAD_GRAYSCALE)[:480, :367]

# using your match_histograms implementation
matched = match_histograms(image, reference).astype(np.uint8)

# display images
imgs = [reference, image, matched]
titles = ['Reference', 'Image', 'Matched']
fig = make_subplots(2, len(imgs), subplot_titles=titles,
    horizontal_spacing = 0.05, vertical_spacing = 0.1)
for i, (img, title) in enumerate(zip(imgs, titles)):
    fig.add_trace(go.Image(z=cv2.cvtColor(img, cv2.COLOR_GRAY2BGR), name="Image"),
     1, i+1)
    traces = show_hist_stats(img, use_cumulative=True).data
    for trace in traces:
        fig.add_trace(trace, 2, i+1)
fig.show()
```

1.2.3 Results

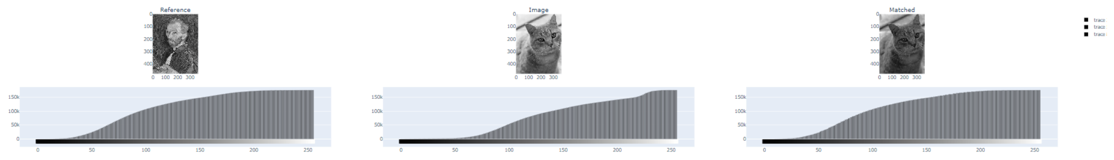

Figure 1.3: Final result