ECEN 4532 - Lab 2: Introduction to Image Processing

Andrew Teta

February 9, 2019

Contents

Introduction **3**

1 Introduction **3**
Background 3
1.1 Background 3

2 Grayscale Conversion **4**
2.1 Introduction 4
2.2 Method 4
2.3 Results 4

3 Contrast Enhancement **5**
3.1 Introduction 5
3.2 Method 5
3.3 Results 6

4 Appendix **9**
4.1 Appendix A: Main 9
4.2 Appendix B: Grayscale Conversion 9

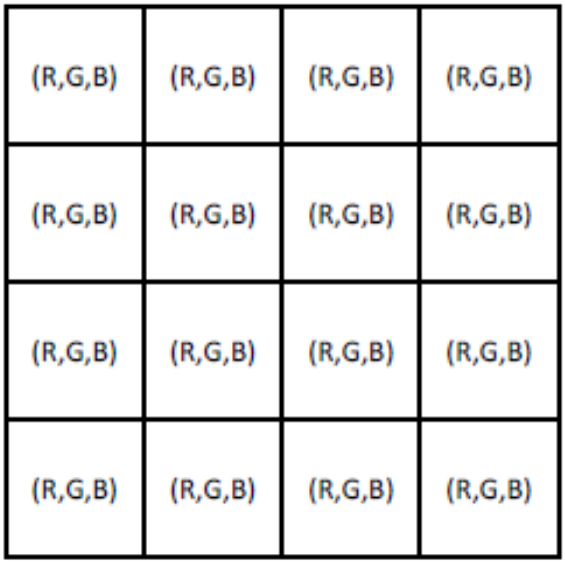


Figure 2: Example of a 4x4 color BMP image file

1 Introduction

In this discussion, we will be exploring some common manipulations applied to images as an introduction to image processing. The analysis will be performed in Python, using a few libraries such as `Pillow` for basic image file manipulations (opening, saving), `numpy` for efficient array operations, `scipy.signal` for some Fourier functionality, `scipy.interpolate` for interpolation, `collections.Counter` to count occurrences of multiple values in a list, and `matplotlib.pyplot` for basic plotting. We will use functions from these libraries to help perform grayscale conversion, contrast enhancement, edge detection, and image resizing, although most of the code for these will be developed from scratch.

1.1 Background

We will be working with a common uncompressed image format, BMP (short for bitmap). This file type begins with a short header, followed by a 2-dimensional array of pixel information. Color BMP files store three 8-bit numbers for each pixel, corresponding to an R, G, and B intensity value.

Different combinations of the three chroma values produce unique colors. Given 8-bit color depth, it is possible to represent 2^{24} colors.

In this lab, we will only be processing monochromatic (grayscale) images, with only 8 bits per pixel (0=black, 255=white). This will simplify analysis. The monochrome value associated with a pixel can be found from its RGB values using a dot product

$$Y = (0.299, 0.587, 0.114) \cdot (R, G, B) \tag{1}$$

The BMP header encodes image size and some other metadata. We will use `Pillow` to import and export images in Python. `Pillow` basically converts the input image to a BMP format when `open()` is called.

2 Grayscale Conversion

2.1 Introduction

In many image processing tasks, color is unnecessary. Converting between color spaces (there are more than just grayscale and RGB) can be done using a matrix multiplication. YUV is a common matrix to use, found in TV applications (Y represents luminosity).

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.14713 & -0.28886 & 0.436 \\ 0.615 & -0.51499 & -0.10001 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2}$$

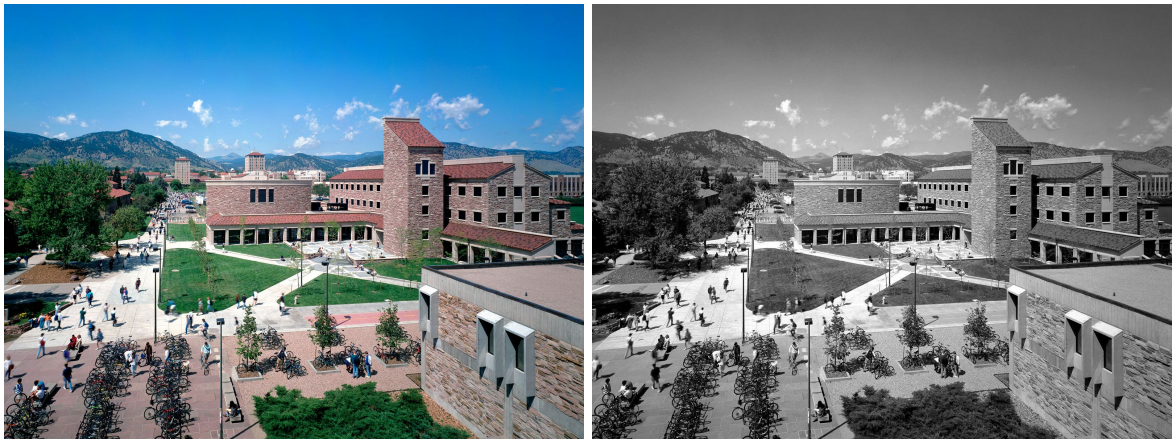
U and V are color difference signals and can be positive or negative, while Y ranges from 0 to 255.

2.2 Method

We will use `Pillow` for image input/output in Python. Importing an image will also involve converting it into a `numpy` array for processing. The basic implementation is to open the file, convert it to a `numpy` array, take the dot product of the Y vector and the image (as in eq. 1), and clip out-of-range values. See 4.2 for the full implementation.

2.3 Results

Two images before and after grayscale conversion are shown below.



(a) Before (b) After

Figure 3: A picture of the Math building on CU Boulder campus before and after grayscale conversion.

(a) Before (b) After

Figure 4: A picture of a bear before and after grayscale conversion.

3 Contrast Enhancement

3.1 Introduction

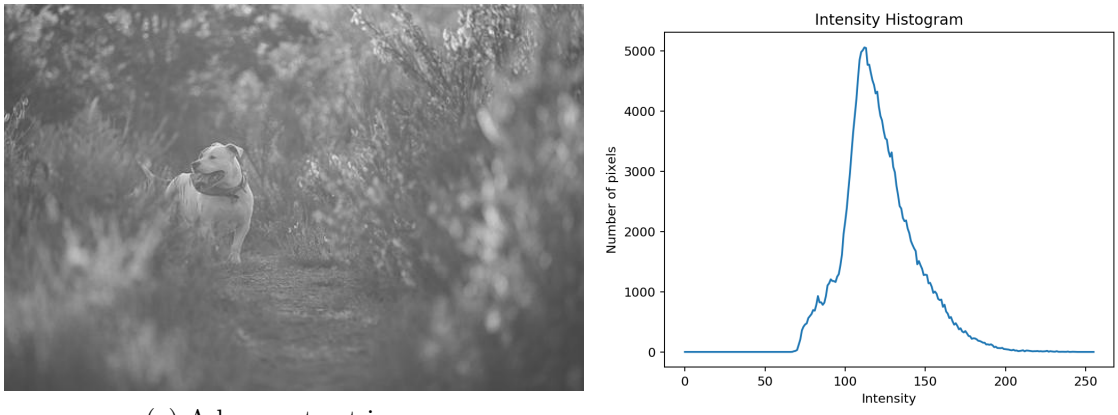
Sometimes a picture will have low dynamic range and appears low contrast. A simple contrast enhancement process based on *histogram equalization* can make a huge improvement. In this part of the lab, we will be implementing histogram equalization as a contrast enhancement algorithm. This means that we will take an image with tightly grouped intensities and try to 'equalize' it so that the intensities are evenly distributed.

3.2 Method

We begin by opening an image and converting it to grayscale, using the `Pillow` function, `convert('L')`. Then, we can use `collections.Counter` to find the number of occurrences of each intensity value. `Counter` returns an array of the values, but is structured like a dictionary and is not sorted. Thus, looping over intensity (0 to 255) we can extract the frequencies in a sorted `numpy` array.

```
image = image.convert('L')
image = np.asarray(image, np.float)
hist_before = np.zeros(256, dtype=int)
freq = Counter(np.reshape(image, image.shape[0] * image.shape[1]))
for p in range(256):
    hist_before[p] = freq[p]
```

At this point, `hist_before` holds a histogram of intensity.



(a) A low-contrast image.

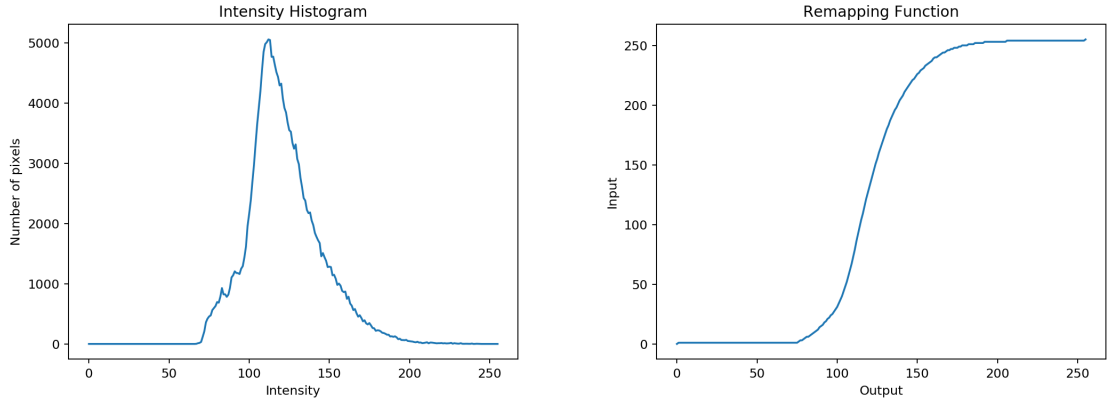
(b) Intensity histogram.

Figure 5: Intensity histogram of a low contrast image. Note the lack of any values on the low or high end of the spectrum. This is what makes it low contrast, as there is low dynamic range.

Next, we want to find a function to map intensities of one value to another in a way that will 'equalize' the histogram in figure 5b. We wish to remap the intensities of the original image, so that each intensity occurs with the same frequency. Then the ideal pixel count for any given intensity would be $P = (total\#ofpixels)/(numberofintensityvalues)$. We will ignore values of 0 or 255 as they cannot be remapped to anything except themselves. Looping over intensity (1 to 254), we hold a running sum, counting the number of pixels that have been remapped in a variable `curr_sum`. Initially, we find pixels to be remapped to a value of 1. We do this until we've remapped at least as many pixels as we had aimed for. Then, we start mapping to a new value, determined by `round(curr_sum/P)`, which accounts for some overshoot.

```
remap = np.zeros(256, dtype=int)
remap[-1] = 255
histSum = sum(hist_before[1:-2])
```

```
P = histSum / 254
T = P
outval = 1
curr_sum = 0
# build remap table
for inval in range(1, 255, 1):
    curr_sum += hist_before[inval]
    remap[inval] = outval
    if (curr_sum > T):
        outval = round(curr_sum/P)
        T = outval*P
```



(a) histogram of low-contrast image

(b) remapping function

Figure 6: Histogram and remapping function for a low contrast image.

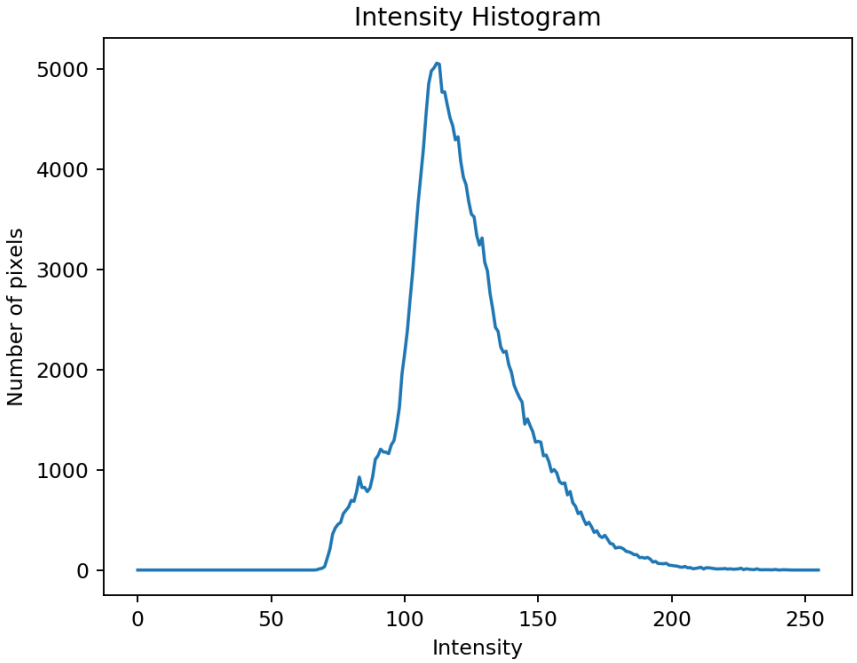
Figure 6b shows how intensity values are remapped for the image in 5a. Notice that many of the pixel intensities centered around the middle will be spread out to occupy most of the spectrum. Next, we need to apply the remapping function to the image and produce the contrast-enhanced picture. To do this, we will use `numpy.where`.

```
image_equalized = np.zeros_like(image)
for intensity in range(256):
    image_equalized = np.where(image == intensity, remap[intensity], image_equalized)
```

3.3 Results

The full results of contrast enhancement for two images can be seen in figures 7 and **??**.

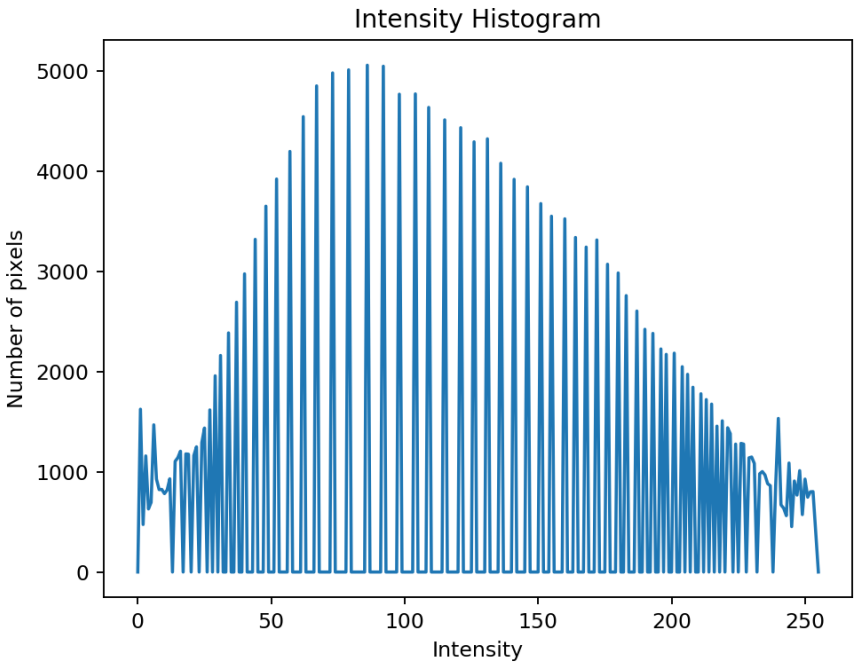
(a) original



(b) before

Figure 7: Contrast enhancement.

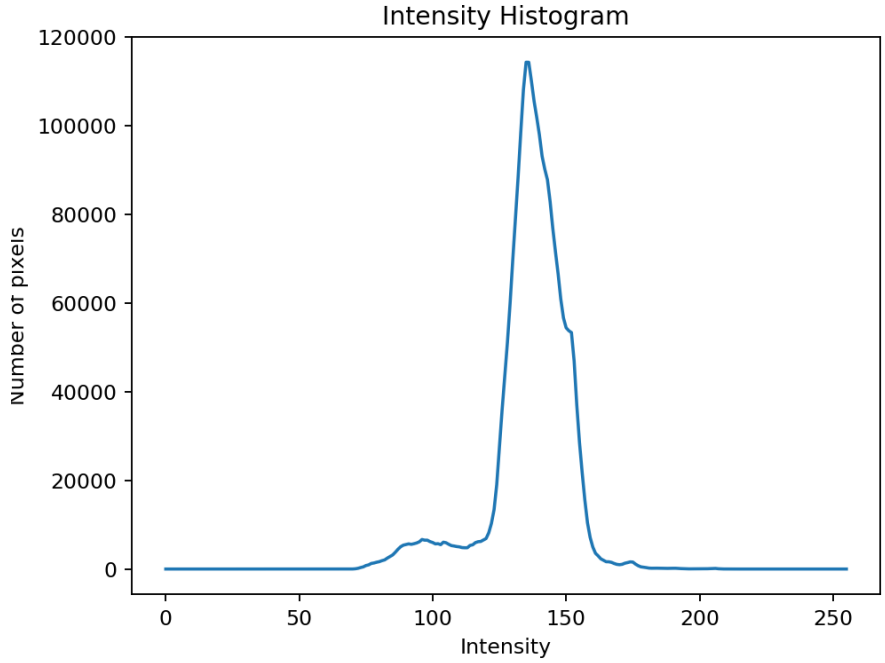
(c) contrast enhanced



(d) after

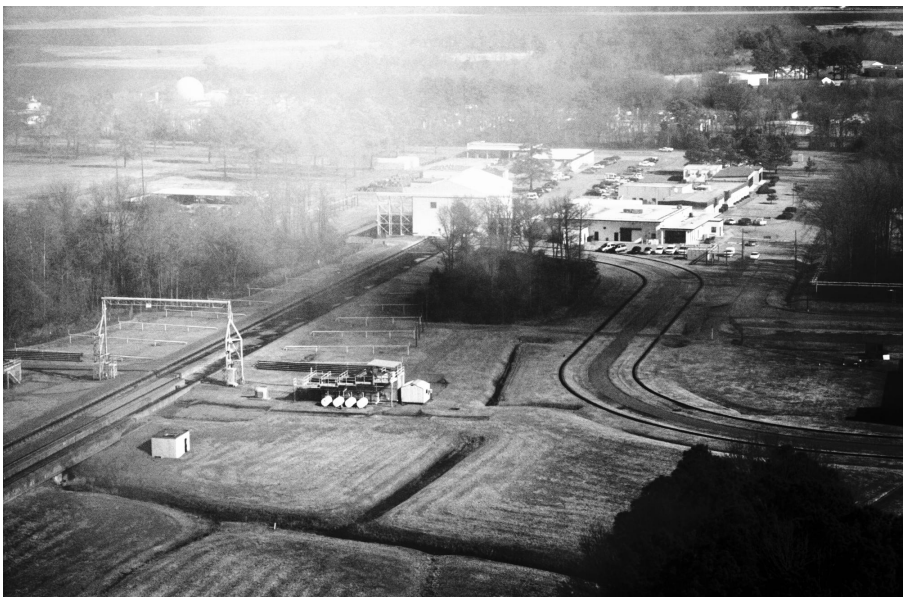
Figure 7: Contrast enhancement.

(a) original

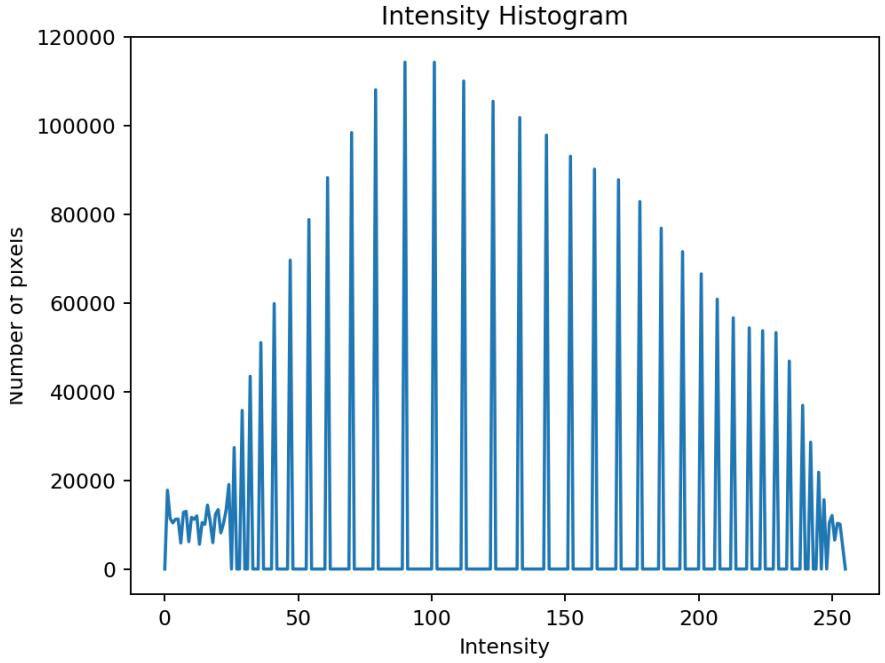


(b) before

Figure 8: Contrast enhancement.



(c) contrast enhanced



(d) after

Figure 8: Contrast enhancement.

4 Appendix

4.1 Appendix A: Main

```

```

4.2 Appendix B: Grayscale Conversion

```
def grayscale(image):
    imageIn = np.asarray(image,np.uint8)
    rows = imageIn.shape[0]
    cols = imageIn.shape[1]

    # convert image to grayscale
    Y = [0.299,0.587,0.114]
    imGray = np.dot(imageIn,Y)
    imGray = imGray.astype(np.uint8)

    # convert ndarray into linear array
    linIm = np.reshape(imGray,rows * cols)

    # clip out of bounds values
    linIm = np.where(linIm < 0,0,linIm)
    linIm = np.where(linIm > 255,255,linIm)

    # reshape back into 2D array
    imGray = np.reshape(linIm,[rows,cols])

    return imGray

```