IPPR Lab 2: Image Enhancement in the Spatial Domain

Nischal Regmi
Everest Engineering College

1 Introduction

In the spatial domain, a grayscale image is an array of intensity values. Similarly, a color image is an array of three dimensional vectors whose components correspond to the intensity of red, green or blue components. This lab will focus entirely on grayscale image processing. In most of the problems, the algorithms for grayscale processing can be easily converted to color image processing.[1] I will first mention in brief the required theoretical constructs along with some notes about their implementation in Java.

2 Basic Image-Enhancement Methods

Basic grayscale image-enhancement methods use simple transformations of the form $s = T(r)$ where are r and s are the pixel intensities of the original and transformed images. From the implementation point of view, such transformation are much simple as they require for-loops that iterate over all the pixels in the image. However, in some transformations, we should be cautious to assure that the resulting intensities are within the system defined range of $[0, L-1]$. Examples of such transformations are log and gamma, which we have not studied in the theory class.

Another transformation of importance is histogram equalization. Assume that we have a $M \times N$ image having L intensity levels $[0, 1, \ldots, L-1]$. Let $r_0, r_1, \ldots, r_k$ denote the intensity levels and denote the number of occurrences of a particular intensity as n_k. Then, the estimated probability for the occurrence of the r_k is

$$p(r_k) = \frac{n_k}{MN}$$

Histogram equalization is achieved by the following transformation function.

$$s_k = (L-1)\sum_{j=0}^{k} p_j$$

To implement histogram equalization, we need to store the counts n_k in an array of size L. In Java, you can create an array dynamically as

```
int[] h = new int[L]; //L is the number of intensity levels
```

In this representation, the array location $h[k]$ corresponds to the count n_k. You should therefore loop-over the image pixels and store the counts appropriately. Note that for 8-bit grayscale images, $L = 256$ so the the array $h[]$ will be of size 256.

After computing the histogram, it will be easier if you calculate and store the probabilities in a separate array $p[]$. Then, calculate the mapping and store it in another array $T[]$. Finally, you can use the histogram equalization transformation based on the above given equation.

[1] While implementing color image processing algorithms, we can alternately think an image as a combination of three different arrays corresponding to the red, green and blue components. In such a representation, we could apply the many of the grayscale algorithms to the individual arrays and combine the result as a single approach.

3 Correlation and Convolution

Convolution in the spatial domain requires the concept of kernel, which in the context of image processing, is simply a matrix. We require that the number of rows and columns in the convolution matrix are odd. Consider a kernel matrix w of size $(2a+1) \times (2b+1)$. Consider a particular pixel at location (x, y). Then we define correlation and convolution convolution at this point as the following transformations.

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x+s, y+t) \qquad \text{(Correlation)}$$

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x-s, y-t) \qquad \text{(Convolution)}$$

You can notice that computation of convolution of an entire image requires for-loops that are nested over four levels. The higher two levels of for-loop corresponds to the image pixels and the lower two to the kernel elements. Thus, convolution is computationally costly if the kernel is large. Computation of convolution is computationally less costly in the frequency domain, which we will deal in a later lab work. In this lab, you will be asked to implement convolution when the kernels are small as 3×3 or 5×5. We will deal with square kernels.

Before implementing the algorithm for correlation or convolution, you should first implement the code for *zero padding*. That is, if your kernel size is $(2a+1) \times (2b+1)$, then you should add a rows of zeros to the above and below of the input matrix, and b columns of zeroes to the left and right. Suppose that the grayscale image is stored in the integer array $f[][]$ and kernel is given in the array $w[][]$. Then you can create a zero-padded matrix using the following Java code.

```
int a = (w.length-1)/2;
int b = (w[0].length-1)/2;

/*first create an array f_padded to store the padded version
of the input image f */
int[][] f_padded = new int[2*a+f.length][2*b+f[0].length];

/*f_padded is initilized to 0s by default. Now copy the values
appropriately from f to f_padded */
for(int x=0;x<f.length;x++)
        for(int y=0;y<f[0].length;y++)
                f_padded[a+x][b+y] = f[x][y];
```

In this Java code, *k.length* and *k*[0].*length* give the number of rows and columns in the array k. Similarly, we have used *f.length* and *f*[0].*length* for rows and columns in f. Remaining portion of the code is obvious. You can now implement convolution operations in this padded array. You should be careful in using the indices properly. In addition, we need to define a kernel as an array. For example, the kernel for the Laplacian can be specified as

```
int[][] w = {{0,1,0},
             {1,-4,1},
             {0,1,0}};
```

For your convenience, I am proving the piece of Java code for calculating the convolution of a kernel matrix $w[][]$ with the padded image $f_{\text{padded}[][]}$. The result is stored in the array $F[][]$.

```
//compute the convolution
for(int x=0;x<F.length;x++)
        for(int y=0;y<F[0].length;y++) {
                for(int s = -a; s<=a; s++)
                        for(int t = -b; t<=b; t++) {
                                int v = w[s+a][t+b];
                                F[x][y]=F[x][y] + v*f_padded[(a+x)-s][(b+y)-t];
        }
}
```

4 Spatial Filtering

The basic idea of implementing spatial filter is to find the correlation of a kernel (sometimes called as mask) w with the image f. Alternatively, you can first rotate the kernel and find its convolution with the image. Refer the textbook for appropriate kernels/masks required in the following problems.

5 Problems

1. Log-transform is defined as $s = \log r$. It highlights pixels that have low intensities without disturbing the original order of intensity values. In other words, logarithm is a monotonically increasing function. Implement log-transform. Since logarithms are usual small numbers, be sure to re-scale the pixel intensities in the maximum possible range $[0, 255]$. You need to use the function *Math.log()* to calculate logarithm in Java. Demonstrate the utility of log transform by it to any appropriate image.

2. Implement the algorithm for histogram equalization. Apply histogram equalization for the image you used in the previous problem. Explain the difference in the output and the reasons behind it.

3. Write a program that finds the correlation (or convolution if you want) a given kernel matrix with a given image. Based on your code for finding correlation(or convolution), write programs to

   (a) Find the Laplacian of the image. Then enhance the image using the Laplacian and display it.

   (b) Blur the image using averaging. Display the blurred image.

   (c) Find the image gradient using Sobel's masks and display it.

   Beware that image processing algorithms could result in pixel intensities that are beyond the range $[0, 255]$. We cannot create a BufferedImage object from intensities that are beyond $[0, 255]$.You should therefore reassign any pixel with negative intensity to the value 0 and any pixel with intensity greater than 255 to the value 255 before creating a BufferedImage object.

Note: Writing Java programs directly dealing with the BufferedImage objects could lead to cumbersome program code. As mentioned in the previous lab sheet, students are advised to adopt the following strategy, (1) Extract the gray scale values of the image into an array (2) Perform image processing operations in the array(s), and (3) Convert the final array back to a BufferedImage object.