Kathryn Young

CMSC 630 Project Part I

To give an overall description of how I implemented this project, I read the images locally from a folder where they were downloaded to my computer using *imread*. The 500 images were read into an array, where each index represented a single image of three dimensions (length, width, RGB values). These images were then converted into grayscale, which was used as the basis for carrying out the rest of the operations in the project: salt and pepper noise, gaussian noise, histogram calculation, averaged histograms, histogram equalization, selected image quantization, linear filter, and median filter. Each function was originally written to handle one image, but then another function was written to handle all of the images. The start time taken right before each preprocessing function, as well as right after. The start time was subtracted from the end time to obtain the total time. The total time was divided by the number of images to obtain the average time per image.

Grayscale

First, each image was converted into grayscale. To do this, the average of the RGB values was taken and assigned as a singular value for the pixel (replacing the array of three values).

Batch processing time: 591.449735894
Per image processing time: 1.1852700118116233

Salt and Pepper

To add salt and pepper noise, I took a user specified floating point to determine the strength of the salt and pepper noise. A random number generator then generated a floating point between 0 and 1, and if the value generated was less than the user specified floating point, then the pixel was converted to salt and pepper noise. To determine whether the point was salt or pepper, an integer variable was used. The variable was assigned to 0 at the beginning. If the pixel was determined to be noise, if the variable was 0 it would be white (salt) and if the variable was 1 it would be black (pepper). Once this pixel was handled, the value of the variable was set to the opposite of what it was.

Batch processing time: 78.48412410500009
Per image processing time: 0.15728281383767553

Gaussian Noise

To add gaussian noise, a 2D array with random values based on the specified standard deviation was created. That array was added to the original image. Since adding the gaussian distributed values could lead to values over 255 and below 0, the values were normalized through the following equation to get the resulting image with gaussian noise.

$$Normalized\ value = \frac{(value - min\ value) * 255}{max\ value - min\ value}$$

Batch processing time: 29.65631754700007
Per image processing time: 0.0594314980901805

Histogram Calculation

To calculate the histogram for each image, an array of size 256 was initialized as the “buckets”, starting at values of 0. The image was looped through and as every pixel was reached, that index of the array was incremented by a value of 1. Once the image has been looped through, the histogram is complete.

Batch processing time: 37.35645800700013
Per image processing time: 0.07486264129659344

Average Histograms by Class

To calculate the average histograms by class, the histogram calculations from the first histogram calculation function were used. Since we know that in the set of (now) 499, there were:

- 50 columnar epithelial cells
- 50 parabasal squamous epithelial cells
- 50 intermediate squamous epithelial cells
- 50 superficial squamous epithelial cells
- 99 mild nonkeratinizing dysplastic cells
- 100 moderate nonkeratinizing dysplastic cells
- 100 severe nonkeratinizing dysplastic cells

Arrays of size 256 were initialized with values of 0. In the loop through the number of images, where i represents the current index we are looking at, if *i* is under 50, then we add its histogram (divided by a factor of the number of included images… *50*) to the current histogram. After those 50 iterations, we are left with the average histogram for the first class. The same operation is applied for the other six classes. The function returns six histograms.

Batch processing time: 0.026476739999907295
Per image processing time: 5.305959919821101e-05

Histogram Equalization for Each Image

To equalize the histograms, the cumulative sums for each index within a single image were calculated and normalized to within values 0 - 255. Then, for each pixel in the image, the current intensity value was used to index the normalized values, which resulted in a new and higher contrast image.

Batch processing time: 55.397855833999984
Per image processing time: 0.11101774716232461

Image Quantization

To implement image quantization, based on the *x* levels specified by the user, the histogram was divided so that pixels could take *x* number of values from 0 to 255. The values are equidistant from each other based on the distance which was based on the number of levels.

Then, an array of thresholds was created based on the what values are closest to the decided pixel values. We go through the array of thresholds, and when the intensity value of the pixel we are looking at is less than the threshold, we know that it should take that index's value within the first array containing new pixel values. Once the pixel was found to be less than the threshold value, an integer variable (originally set to zero) was assigned to -1 so that the value wouldn't be reassigned as it continues through the array.

Batch processing time: 383.5494567500002
Per image processing time: 0.7686361858717439
MSE: [250.07964715375587, 129.19436344630282, 116.15774235255282, 96.43246588908453, 218.61931713981804, … , 83.01629667326877]

Linear Filter

The function written to apply a linear filter was implemented to take user input of the image, the radius of the filter (which included the center pixel), and the weights. All of the border pixels where the filter would be out of bounds were eliminated, reducing the size of the image. The starting position of the filter was calculated (in terms of x and y relative the the pixel being calculated), and then from there the filter was applied through use of a loop, adding the value of the necessary pixel from the original image, multiplied by the corresponding value in that position on the filter. These values were added to this cumulative value. The cumulative value was then divided by the number of pixels in the filter (i.e. to get the average). After these operations were carried out on the whole image, the image intensities were normalized to output the resulting image.

Batch processing time: 1296.5471304120001
Per image processing time: 2.598290842509018

Median Filter

The median filter was implemented similarly to the linear filter in structure. The function was written to take an input image, radius (which included the center pixel), and weights. All of the border pixels outside the bounds of the filter were eliminated, which reduced the size of the image. The starting position of the filter was calculated (in terms of x and y relative the the pixel being calculated), and then from there the filter was applied through use of a loop. Since this median filter is a weighted median filter, that means that the integer values input to the filter describe the weights of the intensity values to be used when calculating the median. As we go through the filter, the intensity values for each pixel are added to a list a number of times corresponding to the filter value. After the filter has been looped through, the median of the list is taken as the new pixel value and applied to that pixel position in the image. The filter proved to be pretty time costly. This was the most time consuming operation performed in this phase of the project.

Batch processing time: 2819.1760857590007
Per image processing time: 5.6496514744669355