CMSC630-A1

Bobby Best

March 2019

1 Implementation

1.1 Image Loading

Images can be loaded from files individually or in bulk using the ‘Image.fromFile()‘ and ‘Image.fromDir()‘ methods respectively, which use OpenCV to read the images into memory as NumPy matrices of shape (height, width, 3), where the 3 represents the red, green, and blue color channels of the image. Each matrix is then used to construct an Image object that can be used in all of the operations detailed below.

1.2 Grayscale Calculation

When an image is loaded, it initially only contains the red, green, and blue color channels. When the ‘Image.getGrayscale()‘ method is called, the three are averaged into a single grayscale channel, which is then stored for later to avoid unnecessary recomputation. In addition, if the ‘luminosity‘ flag is set when the method is called, it will use a special set of weights ([0.30, 0.59, 0.11]) in the computation instead of a normal average; this is supposed to be better for the human eye, see https://www.tutorialspoint.com/dip/grayscale_to_rgb_conversion.htm.

Note that if you manually modify the red, green, or blue color channels outside of using the provided class methods, then the stored grayscale image must be removed using the ‘Image.invalidateLazies()‘ method or else the outdated version will continue to be returned.

1.3 Histogram Calculation

Histograms are calculated simply by counting the number of pixels of each value in the desired color channel of the image. Like the grayscale matrices, they are cached for later use to avoid recalculation until ‘Image.invalidateLazies()‘ is called.

1.4 Equalization

Image equalization is implemented using an algorithm from https://www.tutorialspoint.com/dip/histogram_equalization.htm. It first calculates the *cumulative distribution function* (CDF) of the image, which has the same domain as the image's histogram and is made by setting each value to be the sum of the values before it, plus the corresponding histogram value divided by the total number of pixels, i.e.

$$cdf_0 = 0$$
$$cdf_i = cdf_{i-1} + \frac{hist_i}{\text{total number of pixels}}$$

This results in a monotonically increasing set of values ranging from [0-1], and then the method simply multiplies this by 255 and uses the result as a lookup table to replace the pixel values of the image.

1.5 Quantization

There are several available techniques for quantization implemented, but they all follow the same general structure; the domain of the histogram ([0-255]) is divided up into several *buckets*, and then all pixels that fall into a bucket are replaced with some uniform value, effectively reducing the total number of values. First the method takes in a *quantization delta* as a parameter, which represents the number of values that should be in each bucket, i.e. at $delta = 1$ there will be no change, and at 255 there will be a single bucket containing all values. The differences in the techniques then come in determining what value each bucket takes, as detailed below:

- **Uniform**: The bucket value is decided by the position of the bucket; the first bucket starts at zero, and each subsequent one is *delta* higher than the one before it, forming a staircase shape.
- **Mean**: The bucket value becomes the mean of the original set of values within it.
- **Median**: The bucket value becomes the median of the original set of values within it.

1.6 Filtering

Filters are implemented by taking in a two dimensional matrix (a *filter*) with odd height and width, meaning it has a definite center. The filter is passed over the image pixel by pixel, and for each one the pixel resting under the center takes on a new value that is a function of the filter items and the pixels currently underneath them. This function is decided by the filtering strategy, as follows:

- **Linear**: Each filter value is multiplied by the pixel value underneath it, and the center pixel becomes the sum of all of these. The values are normalized into the [0-255] range at the end if necessary.
- **Mean**: This is the same as the linear filter but with the added step of dividing by the size of the filter, allowing for each pixel to become a weighted average of the pixels around it.
- **Median**: In this method the filter values become 'votes', allowing each pixel value to have a weighted say in what the median of the set is, which becomes the center value.

Another important point of filtering is how we treat the outer border of the image; since the filter can't leave the bounds of the image, it can never modify the outermost edge(s). There are four implemented ways to deal with this:

- **Ignore**: Simply do nothing, leave the borders unchanged and hope nobody notices (they probably won't)
- **Crop**: Strip off the borders and return a smaller image
- **Pad**: Set the unreachable pixels to zero and leave the image with a black border
- **Extend**: Take the outermost *reachable* layer of pixels and copy them into the unavailable spaces, creating a fun stretching effect

1.7 Noise Addition

There are two methods implemented to add noise to an image: salt and pepper, and gaussian. For both, you provide a rate value between 0 and 1, and each pixel has that percentage of a chance to become noisy. For salt and pepper, each pixel simply becomes either 0 or 255 at random, but for gaussian they become random samples from a normal distribution generated from the mean and standard deviation of all the pixels in the image. If you wish, you can also supply a mean and standard deviation of your own instead.

2 Configuration File

The *run.py* script included in the project allows you to run a set of operations on a large batch of images in parallel. It reads configuration details from $config.yml$, which allows you to define an input directory of images, an output directory to save the resulting images to, and a list of operations (steps) to perform on each image. There is an example configuration file, EXAMPLE_CONFIG.yml that defines all the possible steps and shows the properties that can be defined for each. For example, the $filter$ operation is defined by creating a step with the name "filter" that has the following properties:

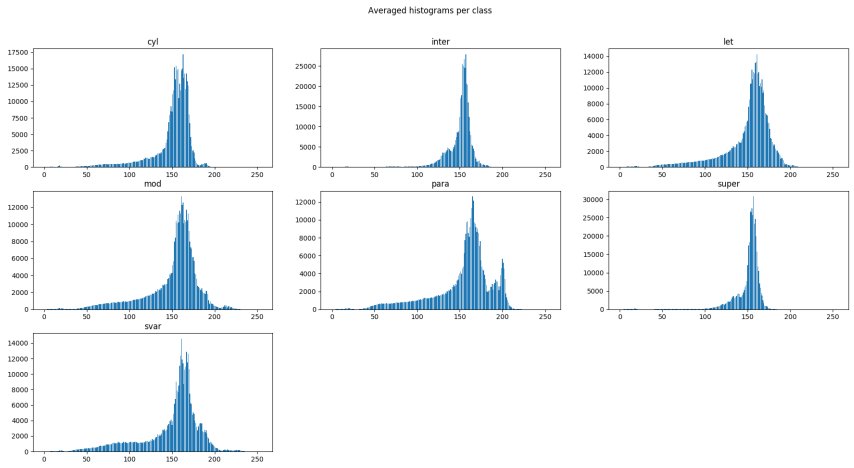

Figure 1: Average Histograms per Cell Class

- color: The color channel to operate on, one of "red", "green", "blue", "rgb", or "gray"
- stategy: The filtering strategy to use, one of "linear", "mean", or "median"
- border: The border strategy to use, one of "ignore", "crop", "pad", or "extend"

You can create as many steps as you wish in whatever order you want, when you execute *run.py* they will all be validated to ensure there are no errors and then executed in order, parallelized across the batch.

3 Results

The product of this project is a fairly robust image processing system that can apply a range of operations to large sets of images in batch, and it can operate on any color channel desired, or all three at once. The runtime leaves something to be desired, but is not far off from what you'd expect; some of the more basic operations (equalization, noise addition, etc.) run in a few seconds, but even with as much parallelization as possible I killed the only attempt to apply a filter to the whole batch around the 30 minute mark. This is hard to avoid in plain Python, if I rewrote the filter code in C it could be a bit better, or if I really wanted to be fancy I could use CUDA like a certain classmate of mine and bring it down to 30 seconds, but I value my sanity.