Digital Image Processing: Project 1

Jacob Taylor Cassady
CECS 625:Digital Image Processing
9/18/19

Contents

1. Introduction

Image transformations can be used to improve the perceptual quality of an image. These transformations are functions that take in an image as an input and return a new image as an output. The transformations come in a large variety, many with different benefits and approaches. An image, Fig0326.tif, was used to test different image transformations in an attempt to reveal hidden information. These transformations were designed as algorithms using the Python 3.7 and accompanying 3rd party libraries: NumPy, scipy, opencv, and matplotlib.

2. Approach

Two transfer algorithms were developed in an attempt to reveal some underlying information in the given image. These algorithms are: Adaptive Histogram Equalization and Adaptive Statistical Equalization. Both algorithms utilize an adaptive approach that looks at a subset, or "neighborhood", of pixels within an image when applying each transform. For an input pixel(i,j) the function returns a neighborhood centered at (i,j) and of size (m, n). This approach used a neighborhood of size (5,5) and utilized a padding of 0 pixel values for edge cases. The pertinent chunk of code for calculating the neighbors is shown in **Figure 1** below. Please refer to the appendices for a copy of the full source code.

```
# Retrieve relevant frame from image.  Initailize array of zeros to store result.
frame = deepcopy(image[y_start:y_stop, x_start:x_stop])
result = np.zeros(shape=neighborhood_shape)

# Transfer image frame to resultant array taking into account the amount of padding.
for row_index, row in enumerate(frame):
    for column_index, pixel in enumerate(row):
        if column_index+left_pad < (neighborhood_shape[1] - right_pad) and row_index + top_pad < (neighborhood_shape[0] - bottom_pad):
            result[row_index+top_pad, column_index+left_pad] = frame[row_index, column_index]

return result
```

Figure 1 : get_neighborhood() Pertinent Code Snippet

2.1. Adaptive Histogram Equalization

The Adaptive Histogram Equalization algorithm applies histogram equalization by iterating over neighborhoods and calculating the cumulative density function for each neighborhood. The center pixel value of the neighborhood is then used as an input in the neighborhood's cumulative density function. The output of this function replaces the center pixel value from the input image within output image, g. For reference to the exact implementation of this algorithm, please refer to **Figure 2** below.

```
def adaptive_histogram_equalization(image, neighborhood_shape = (5, 5)):
    g = np.zeros(image.shape)

    for row_index, row in enumerate(image):
        for column_index, pixel in enumerate(row):
            relevant_neighborhood = get_neighborhood(image, (row_index, column_index), neighborhood_shape)
            image_cdf, bins = exposure.cumulative_distribution(relevant_neighborhood, 256)
            image_cdf *= 255
            g[row_index, column_index] = image_cdf[pixel]

    return g
```

Figure 2 : Adaptive Histogram Equalization Algorithm

2.2. Adaptive Statistical Equalization

The Adaptive Statistical Equalization algorithm references the statistics of a neighborhood surrounding pixel(i, j) of the input image to decide rather or not the pixel value should be enhanced by a factor of E in the output image g at the same pixel location (i, j).

The enhancement decision hinges on two comparisons. First off, the neighborhood mean must be less than or equal to a constant k0 * the global mean of the image. Since k0 must be less than 1 and greater than 0, this means the neighborhood mean must be less than some percentage of the global mean to be enhanced.

The second comparison focuses on standard deviation. The pixel's neighborhood's standard deviation must be larger than or equal to a constant k1* the global standard deviation while still being less than or equal to a constant k2 * the global standard deviation. Again, since k1 and k2 must be less than 1 but greater than 0 these values create a range of pixel values to enhance. Variable creates a lower threshold while k2 creates the ceiling.

In other words, pixel (i,j) of input image f(i, j) is enhanced by E in output image g(i,j) if the neighborhood's mean is less than (k0*100)% of the global mean and the neighborhood's standard deviation is greater than (k1*100)% of the global standard deviation and less than (k2*100)% of the global standard deviation. Consequently, areas of higher intensity are enhanced as well as those within a range of standard deviation. For reference to the exact implementation of this algorithm please see **Figure 3** below.

```python
def adaptive_statistical_enhancement(image, neighborhood_shape = (5,5), k0=0.4, k1=0.02, k2=0.4, E=4):
    assert 0 <= k0 <= 1, "k0 must be a constant < 1"
    assert 0 <= k1 <= 1, "k1 must be a constant < 1"
    assert 0 <= k2 <= 1, "k2 must be a constant < 1"
    assert 1 <= E, "E must be a constant > 1"
    g = np.zeros(image.shape)

    global_mean = np.mean(image)
    global_std = np.std(image)

    for row_index, row in enumerate(image):
        for column_index, pixel in enumerate(row):
            relevant_neighborhood = get_neighborhood(image, (row_index, column_index), neighborhood_shape)
            neighborhood_mean = np.mean(relevant_neighborhood)
            neighborhood_std = np.std(relevant_neighborhood)

            if neighborhood_mean <= k0 * global_mean and k1*global_std <= neighborhood_std <= k2*global_std:
                g[row_index, column_index] = E * pixel
            else:
                g[row_index, column_index] = pixel

    return g
```

Figure 3 : Adaptive Stastical Equalization Algorithm

3. Results

The algorithms described above were both implemented successfully. They produce different results with different tradeoffs between the two. Images in this section were generated using matplotlib with Python 3.7.

3.1. Data

These algorithms were tested on one image. The input image tested was Fig0326.tif shown in **Figure 4** below. The image contains 5 black squares geometrically spaced. The squares look as though they have maximum intensity values while the surrounding space has minimum intensity values. There is little to no perceived noise in the image.

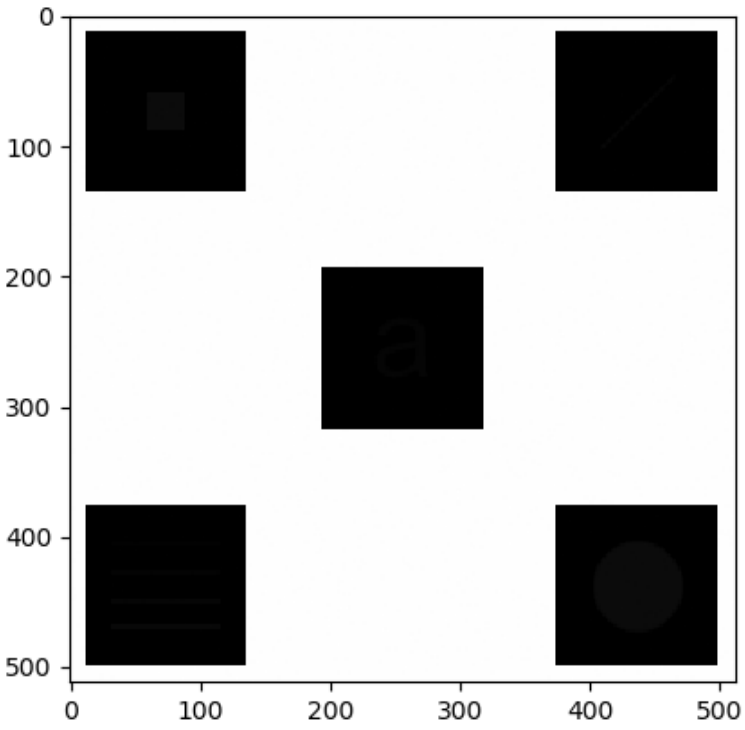

Figure 4 : Fig0326.tif Input

3.2. Results

Four different representations of the input image described in section 3.1 of this document are shown in **Figure 5** below. On the far left, is the input image Fig0326.tif without any transformations. Next is the same image with histogram equalization applied without any adaptive implementation. Following is Fig0326.tif after Adaptive Histogram Equalization. Lastly, the far right shows the output image after Adaptive Statistical Equalization. The two latter examples reveal some hidden information in the images not exposed in the input image or from standard histogram equalization.

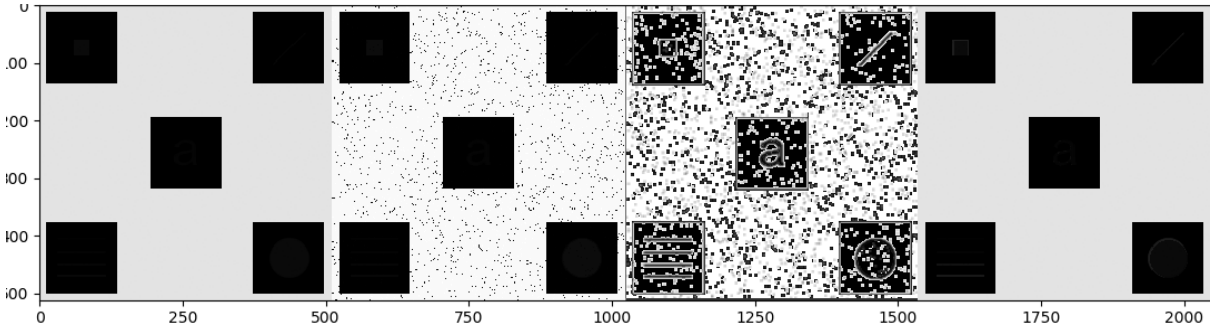

Figure 5 : Equalization Algorithm Outputs

3.2.1. Adaptive Histogram Equalization

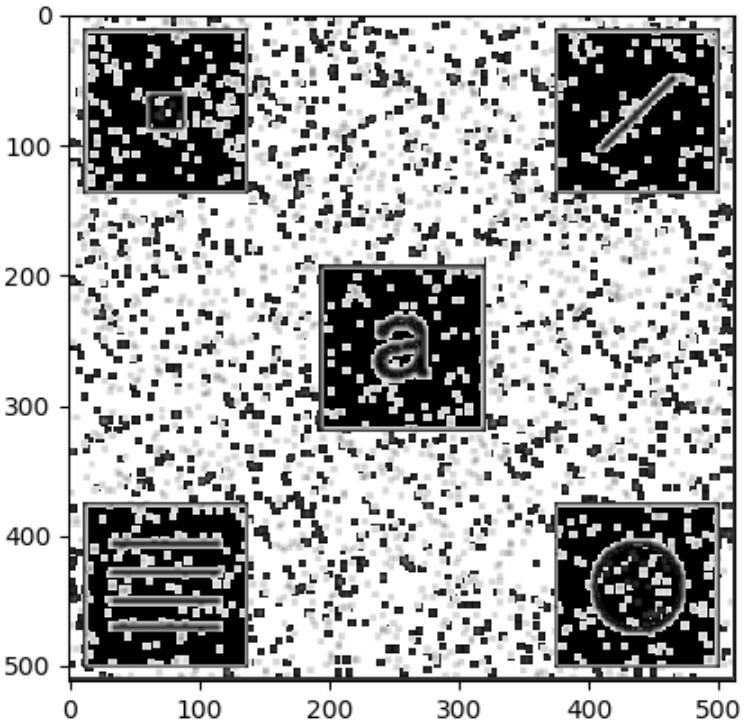

Figure 6 : Adaptive Histogram Equalization Fig0326.tif Output

3.2.2. Adaptive Statistical Equalization

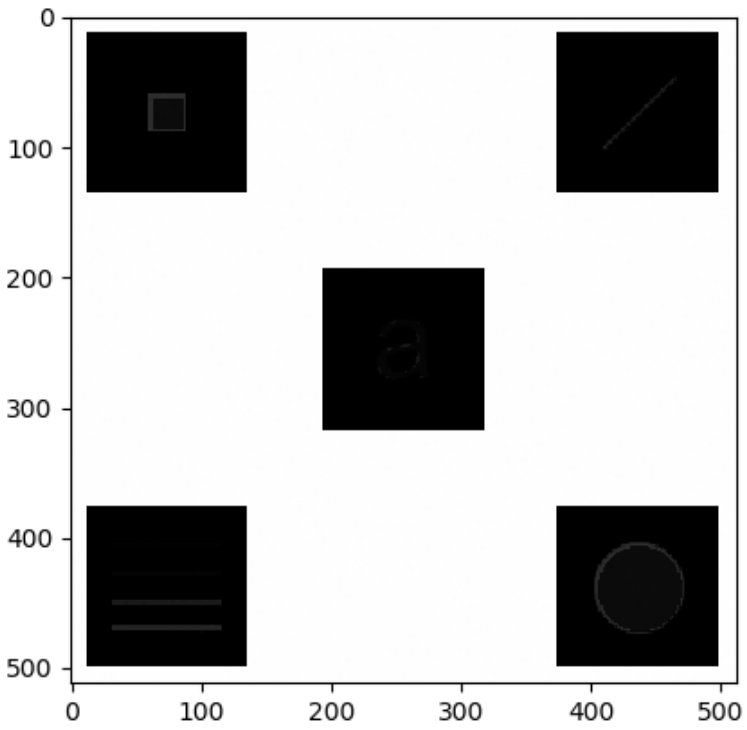

Figure 7 : Adaptive Statistical Equalization Fig0326.tif Output

4. Discussion

The output of Adaptive Histogram Equalization reveals symbols inside of each of the black boxes. Neighborhood size for the output shown in **Figure 6** from section **3.2.1** was (5, 5). As you can see, some areas of the image gained perceptual clarity while adding a lot of increased noise in all areas of the image. This is a consequence of the algorithm's failure to focus only on areas of interest like other algorithms such as Adaptive Statistical Equalization.

The output of Adaptive Statistical Equalization also reveals symbols inside of each of the black boxes but without adding the extra noise from the previously mentioned algorithm. This can be found in **Figure 7** from section **3.2.2**. There is no longer noise in the white areas of the image, as they do not meet the neighborhood mean cutoff with respect to the global mean. Although this algorithm produces less noise, certain symbols are tougher to perceive such as the 'a' found in the center box of **Figure 6**. This could be mitigated though as k1 and k2 can be tuned to create better perception with respect to the edges of the symbols.

It is important to note none of these symbols were exposed when applying standard Histogram Equalization. The hidden symbols might be masked when the image isn't analyzed adaptively.

5. Conclusion

In conclusion, both algorithms reveal underlying information in the input image that was not found using standard histogram equalization. The output image from Adaptive Histogram Equalization included much more noise than Adaptive Statistical Equalization but also did a better job of improving perception of the hidden symbol's edges. These algorithms could be used together to gather information using Adaptive Histogram Equalization about the image's underlying edges while using Adaptive Statistical Equalization to best tune to the found symbols.

6. References

NumPy Documentation - https://docs.scipy.org/doc/
Matplotlib Documentation - https://matplotlib.org/3.1.1/contents.html
Scipy Documentation - https://www.scipy.org/
OpenCV Documentation - https://docs.opencv.org/2.4/

7. Appendices

7.1. FileManager.py

```
import cv2

class FileManager():
    @staticmethod
    def read_grayscale_image(image_path):
        return cv2.imread(image_path, 0)
```

7.2. main.py

```
import sys
import cv2
from copy import deepcopy
import numpy as np
import matplotlib.pyplot as plt
from skimage import exposure
from scipy.stats import norm

from FileManager import FileManager

def display_image(image):
    # Display image:
```

```
  plt.imshow(image, cmap='gray')
  plt.show()

def adaptive_histogram_equalization(image, neighborhood_shape = (5, 5)):
  g = np.zeros(image.shape)

  for row_index, row in enumerate(image):
    for column_index, pixel in enumerate(row):
      relevant_neighborhood = get_neighborhood(image, (row_index,
column_index), neighborhood_shape)
      image_cdf, bins =
exposure.cumulative_distribution(relevant_neighborhood, 256)
      image_cdf *= 255
      g[row_index, column_index] = image_cdf[pixel]

  return g

def adaptive_statistical_enhancement(image, neighborhood_shape = (5,5),
k0=0.4, k1=0.02, k2=0.4, E=4):
  assert 0 <= k0 <= 1, "k0 must be a constant < 1"
  assert 0 <= k1 <= 1, "k1 must be a constant < 1"
  assert 0 <= k2 <= 1, "k2 must be a constant < 1"
  assert 1 <= E, "E must be a constant > 1"
  g = np.zeros(image.shape)

  global_mean = np.mean(image)
  global_std = np.std(image)

  for row_index, row in enumerate(image):
    for column_index, pixel in enumerate(row):
      relevant_neighborhood = get_neighborhood(image, (row_index,
column_index), neighborhood_shape)
      neighborhood_mean = np.mean(relevant_neighborhood)
      neighborhood_std = np.std(relevant_neighborhood)

      if neighborhood_mean <= k0 * global_mean and k1*global_std <=
neighborhood_std <= k2*global_std:
        g[row_index, column_index] = E * pixel
      else:
        g[row_index, column_index] = pixel

  return g

def get_neighborhood(image, pixel_location, neighborhood_shape):
  # Retrieve data from inputs
  rows = len(image[:, 0])
```

```
    columns = len(image[0, :])
    pixel_row, pixel_column = pixel_location

    # Initialize variables dependent on if statements
    left_pad = 0
    right_pad = 0
    top_pad = 0
    bottom_pad = 0

    # Compute image_frame WRT neighborhood_shape and pixel location
    mid_to_right = neighborhood_shape[0] // 2
    mid_to_top = neighborhood_shape[1] // 2
    y_start = int(pixel_row-mid_to_right-1)
    if y_start < 0:
        top_pad = y_start * -1 -1
        y_start = 0

    y_stop = int(pixel_row+mid_to_right+1)
    if y_stop > columns - 1:
        bottom_pad = -(y_stop - columns - 1)
        y_stop = columns

    x_start = int(pixel_column-mid_to_top-1)
    if x_start < 0:
        left_pad = x_start * -1 -1
        x_start = 0

    x_stop = int(pixel_column+mid_to_top+1)
    if x_stop > rows:
        right_pad = x_stop - rows - 1
        x_stop = rows

    # Retrieve relevant frame from image.  Initailize array of zeros to store result.
    frame = deepcopy(image[y_start:y_stop, x_start:x_stop])
    result = np.zeros(shape=neighborhood_shape)

    # Transfer image frame to resultant array taking into account the amount of
padding.
    for row_index, row in enumerate(frame):
        for column_index, pixel in enumerate(row):
            if column_index+left_pad < (neighborhood_shape[1] - right_pad) and
row_index + top_pad < (neighborhood_shape[0] - bottom_pad):
                result[row_index+top_pad, column_index+left_pad] = frame[row_index,
column_index]

    return result
```

```
if __name__ == "__main__":
    args = sys.argv

    if(len(args) != 2):
        print("Command Line Arguments should follow the format:")
        print("python main.py [relative_image_path]")
    else:
        image_path = args[1]

        image = FileManager.read_grayscale_image(image_path)

        equalized_image = cv2.equalizeHist(image)

        ahe_image = adaptive_histogram_equalization(image)

        ase_image = adaptive_statistical_enhancement(image)

        # stack images side by side
        res = np.hstack((image, equalized_image, ahe_image, ase_image))

        display_image(res)
```