Image Processing Techniques
Implementation and Theory

Oussama GUELFAA

01-04-2025

Contents

1 Introduction

This document presents the theoretical foundations and implementation of several image processing techniques. We will cover three main techniques:

- Intensity transformations (gamma correction and contrast stretching)
- Histogram equalization
- Histogram matching

For each technique, we will first present the theoretical basis, then explain the implementation in our Python project.

2 Intensity Transformations

Intensity transformations are operations that modify the pixel values of an image without changing their position. These transformations are generally represented by a transfer function (or Look-Up Table, LUT) that maps each input intensity level to an output intensity level.

2.1 Gamma Correction

2.1.1 Theory

Gamma correction is a non-linear transformation that modifies the intensity values of pixels according to the formula:

$$I_{out} = I_{in}^{\gamma} \tag{1}$$

where:

- I_{in} is the input pixel intensity (normalized between 0 and 1)
- I_{out} is the output pixel intensity
- γ is the correction parameter

When $\gamma < 1$, the dark areas of the image are brightened, which can be useful for bringing out details in the shadows. Conversely, when $\gamma > 1$, the bright areas are darkened, which can be useful for reducing overexposure.

2.1.2 Implementation

Our implementation of gamma correction is as follows:

```python
def apply_gamma_correction(image, gamma):
    """
    Apply gamma correction to the image.

    Args:
        image (ndarray): Grayscale image (values between 0 and 1)
        gamma (float): Gamma parameter

    Returns:
        ndarray: Image after gamma correction
    """
    # Check that the image is in float with values between 0 and 1
    if image.min() < 0 or image.max() > 1:
        print("Warning: Image should have values between 0 and 1. Normalization applied.")
        image = (image - image.min()) / (image.max() - image.min())

    # Apply gamma correction
    corrected = np.power(image, gamma)

    return corrected
```

Listing 1: Gamma correction implementation

2.2 Contrast Stretching

2.2.1 Theory

Contrast stretching is a transformation that increases the contrast of an image by stretching the intensity histogram. The general formula is:

$$I_{out} = \frac{1}{1 + \left(\frac{m}{I_{in}}\right)^E} \tag{2}$$

where:

- I_{in} is the input pixel intensity (normalized between 0 and 1)

- I_{out} is the output pixel intensity
- m is the median value (typically 0.5 for a normalized image)
- E is the stretching parameter

The larger E is, the more the contrast is enhanced. This transformation is particularly useful for improving the contrast of images with low dynamic range.

2.2.2 Implementation

Our implementation of contrast stretching is as follows:

```python
def apply_contrast_stretching(image, E, m=0.5):
    """
    Apply contrast stretching to the image.

    Args:
        image (ndarray): Grayscale image (values between 0 and 1)
        E (float): E parameter (controls stretching)
        m (float): Median value (default: 0.5)

    Returns:
        ndarray: Image after contrast stretching
    """
    # Check that the image is in float with values between 0 and 1
    if image.min() < 0 or image.max() > 1:
        print("Warning: Image should have values between 0 and 1. Normalization applied.")
        image = (image - image.min()) / (image.max() - image.min())

    # Avoid division by zero
    epsilon = 1e-10
    image_safe = np.maximum(image, epsilon)

    # Apply contrast stretching
    stretched = 1 / (1 + (m / image_safe) ** E)

    return stretched
```

Listing 2: Contrast stretching implementation

3 Histogram Equalization

3.1 Theory

Histogram equalization is a technique that transforms the image so that its histogram is as uniform as possible. This transformation generally improves the overall contrast of the image.

Histogram equalization is defined by the transformation:

$$T(x_k) = (L - 1) \cdot \mathrm{CDF}_I(k) \tag{3}$$

where:

- x_k is the intensity value k
- L is the maximum intensity value (256 for 8-bit images)
- $\mathrm{CDF}_I(k)$ is the cumulative distribution function of the image

The cumulative distribution function (CDF) is defined as the cumulative sum of the normalized histogram:

$$\mathrm{CDF}_I(k) = \sum_{j=0}^{k} p(x_j) \tag{4}$$

where $p(x_j)$ is the probability of occurrence of intensity j in the image, defined by:

$$p(x_j) = \frac{n_j}{n} \tag{5}$$

with n_j being the number of pixels with intensity j and n the total number of pixels in the image.

3.2 Implementation

Our implementation of histogram equalization is as follows:

```python
def equalize_histogram_custom(image, bins=256):
    """
    Custom implementation of histogram equalization.

    Args:
        image (ndarray): Grayscale image (values between 0 and 1)
        bins (int): Number of bins for the histogram
```

```

    Returns:
        ndarray: Image after histogram equalization
    """
    # Check that the image is in float with values between 0 and 1
    if image.min() < 0 or image.max() > 1:
        print("Warning: Image should have values between 0 and 1. Normalization applied.")
        image = (image - image.min()) / (image.max() - image.min())

    # Calculate the histogram
    hist, bin_edges = np.histogram(image.ravel(), bins=bins, range=(0, 1))

    # Calculate the CDF
    cdf = hist.cumsum()

    # Normalize the CDF
    cdf = cdf / cdf[-1]

    # Create the LUT (Look-Up Table) for the transformation
    # For each intensity value, associate its equalized value
    bin_centers = (bin_edges[:-1] + bin_edges[1:]) / 2

    # Create an array to store the equalized values
    equalized = np.zeros_like(image)

    # For each pixel in the image
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            # Find the bin index corresponding to the pixel value
            pixel_value = image[i, j]
            bin_index = min(int(pixel_value * bins), bins - 1)

            # Apply the equalization transformation
            equalized[i, j] = cdf[bin_index]

```

```
    return equalized
```

Listing 3: Histogram equalization implementation

4 Histogram Matching

4.1 Theory

Histogram matching (or histogram specification) is a technique that transforms the image so that its histogram matches a model histogram. Unlike histogram equalization, which aims to obtain a uniform histogram, histogram matching allows targeting any distribution.

Histogram matching is defined by the transformation:

$$x_2 = \mathrm{CDF}_2^{-1}(\mathrm{CDF}_1(x_1)) \tag{6}$$

where:

- x_1 is the intensity value in the source image
- x_2 is the corresponding intensity value in the target image
- CDF_1 is the cumulative distribution function of the source image
- CDF_2 is the cumulative distribution function of the model histogram

The principle is as follows:

1. Calculate the histogram and CDF of the source image
2. Define a reference histogram (in our case, a bimodal histogram)
3. Calculate the CDF of the reference histogram
4. For each intensity level x_1 in the source image:
    - Find the value of $\mathrm{CDF}_1(x_1)$
    - Find the value x_2 such that $\mathrm{CDF}_2(x_2) = \mathrm{CDF}_1(x_1)$
    - Replace x_1 with x_2 in the resulting image

Since intensity values are discrete, interpolation is necessary to find the exact value of x_2.

4.2 Implementation

Our implementation of histogram matching is as follows:

```python
def match_histogram_custom(image, reference_hist, bins=256):
    """
    Custom implementation of histogram matching.

    The transformation is defined by:
    x2 = cdf2^(-1)(cdf1(x1))

    where:
    - x1 is the intensity value in the source image
    - x2 is the corresponding intensity value in the target image
    - cdf1 is the cumulative distribution function of the source image
    - cdf2 is the cumulative distribution function of the model histogram

    Args:
        image (ndarray): Grayscale image (values between 0 and 1)
        reference_hist (ndarray): Reference histogram
        bins (int): Number of bins for the histogram

    Returns:
        ndarray: Image after histogram matching
    """
    # Check that the image is in float with values between 0 and 1
    if image.min() < 0 or image.max() > 1:
        print("Warning: Image should have values between 0 and 1. Normalization applied.")
        image = (image - image.min()) / (image.max() - image.min())

    # Calculate the histogram of the source image
    hist_source, bin_edges_source = np.histogram(image.ravel(), bins=bins, range=(0, 1))

    # Calculate the CDF of the source image
    cdf_source = compute_cdf_from_hist(hist_source)
```

```python

    # Calculate the CDF of the reference histogram
    cdf_reference = compute_cdf_from_hist(reference_hist)

    # Create the LUT (Look-Up Table) for the transformation
    # For each value of cdf_source, find the corresponding value in cdf_reference
    bin_centers = (bin_edges_source[:-1] + bin_edges_source[1:]) / 2

    # Create an array to store the transformed values
    matched = np.zeros_like(image)

    # For each pixel in the image
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            # Find the bin index corresponding to the pixel value
            pixel_value = image[i, j]
            bin_index = min(int(pixel_value * bins), bins - 1)

            # Get the CDF source value for this pixel
            cdf_value = cdf_source[bin_index]

            # Find the index in the reference CDF that best matches this value
            idx = np.argmin(np.abs(cdf_reference - cdf_value))

            # Convert the index to intensity value
            matched[i, j] = bin_centers[idx]

    return matched
```

Listing 4: Histogram matching implementation

4.3 Creating a Bimodal Histogram

For histogram matching, we need a reference histogram. We chose to create a bimodal histogram, which is a combination of two Gaussian distributions:

```python
def create_bimodal_histogram(bins=256, peak1=0.25, peak2=0.75, sigma1=0.05, sigma2=0.05, weight1=0.5, weight2=0.5):
    """
    Create a bimodal reference histogram.

    Args:
        bins (int): Number of bins for the histogram
        peak1 (float): Position of the first peak (between 0 and 1)
        peak2 (float): Position of the second peak (between 0 and 1)
        sigma1 (float): Standard deviation of the first peak
        sigma2 (float): Standard deviation of the second peak
        weight1 (float): Weight of the first peak (between 0 and 1)
        weight2 (float): Weight of the second peak (between 0 and 1)

    Returns:
        tuple: (reference_hist, bin_centers) where reference_hist is the bimodal histogram
                and bin_centers are the bin centers
    """
    # Normalize weights
    total_weight = weight1 + weight2
    weight1 = weight1 / total_weight
    weight2 = weight2 / total_weight

    # Create bins
    bin_edges = np.linspace(0, 1, bins + 1)
    bin_centers = (bin_edges[:-1] + bin_edges[1:]) / 2

    # Create the bimodal histogram (sum of two Gaussians)
    reference_hist = weight1 * np.exp(-0.5 * ((bin_centers - peak1) / sigma1) ** 2) / (sigma1 * np.sqrt(2 * np.pi))
    reference_hist += weight2 * np.exp(-0.5 * ((bin_centers - peak2) / sigma2) ** 2) / (sigma2 * np.sqrt(2 * np.pi))

```

```
    # Normalize the histogram
    reference_hist = reference_hist / np.sum(
        ↪ reference_hist)

    return reference_hist, bin_centers
```

Listing 5: Creating a bimodal histogram

5 Conclusion

In this document, we have presented the theoretical foundations and implementation of three important image processing techniques:

- Intensity transformations (gamma correction and contrast stretching)
- Histogram equalization
- Histogram matching

These techniques are essential for improving the visual quality of images and for preparing images for more advanced processing. They form the basis of many image processing and computer vision algorithms.

Our Python implementation makes it easy to apply these techniques to grayscale images, with the ability to visualize the results and compare different approaches.