Enhancement of images with uneven illumination

Enhancing images with uneven illumination using ensemble learning

Group ID: #1

Til Mohr
tv.mohr@stud.uis.no

Alexander Mühleisen
???

ABSTRACT

KEYWORDS

image processing, image enhancement, uneven illumination, ensemble learning

1 INTRODUCTION

2 THEORY

In this section we will dive into different methods to enhance images with uneven illumination. We will start with a brief introduction to the problem and then discuss different methods to solve it, as well as how to evaluate the results.

2.1 Problem description

Uneven illumination refers to the irregular distribution of light intensity across an image. In essence, it disrupts the uniformity of the visual output, leading to disparities in brightness and contrast, often observable as glares or shadows. These disparities can mask essential features and details, making the subject of the image less identifiable. This becomes especially problematic when images need to be processed further for various computer vision tasks. In fields like optical microscopy, for example, consistent illumination is crucial for accurately identifying and segmenting microscopic entities. Uneven lighting can obscure crucial cellular structures or make similar-looking entities appear distinct, hampering accurate analysis [3].

To counter this issue, the goal is to enhance the image in a manner that simulates its capture under uniform illumination conditions. By doing so, we aim to restore a natural appearance to the image, preserving details and minimizing artifacts introduced by uneven lighting. This correction enables better analysis, ensuring that conclusions drawn are based on the actual subject and not on lighting imperfections [3].

2.2 Unsharp Masking

Unsharp masking is a sharpening technique that uses a blurred version of the original image to enhance edges and fine details. The name stems from the fact that the blurred image is subtracted from the original, leaving only the high-frequency components, which are then added back to the original image. This results into an image with sharper edges, more pronounced detail, and more contrast. This approach can be formulated as follows [2, 5, 7]:

$$g(x, y) = f(x, y) + \lambda \cdot (f(x, y) - Blur(f)(x, y)) \tag{1}$$

where $f(x, y)$ is the input image, $Blur(f)(x, y)$ is the blurred input image, and $\lambda > 0$ is a parameter that controls the strength of the sharpening effect. Typically, a Gaussian filter is used to blur the input image [2, 5, 7].

2.3 Retinex

From a theoretical research field, known as Retinex, which concerns itself with modelling the human visual system, a number of algorithms to enhance the visual appearance of images have appeared. One of these is called Multi Scale Retinex with Chromacity Preservation (MSRCP), which is an extension to the Multi Scale Retinex (MSR) algorithm, that builds on top of the Single Scale Retinex (SSR) algorithm. The SSR algorithm is characterized by the following formula [1, 6]:

$$\mathrm{R}_{n_i}(x, y) = \log(f_i(x, y)) - \log(f_i(x, y) * F_n(x, y)) \tag{2}$$

where $f_i(x, y)$ is the value of the input image at pixel (x, y) in channel i, and $F_c(x, y)$ is a Gaussian surround function with a $\sigma = n$. Building on top of SSR, the MSR algorithm is given by [1, 6]:

$$\mathrm{R}_{MSR_i}(x, y) = \sum_{n=1}^{N} \omega_n \cdot \mathrm{R}_{n_i}(x, y) \tag{3}$$

i.e. MSR is the weighted average of SSR at different scales. Experiments have shown that MSR alone often washes out the color of the image, and therefore the MSRCP algorithm was proposed, which first computes an intermediate image using MSR, and then stretches the colors of that image to use the full color range [6]. Finally, using both the original image and the intermediate image with color stretching, amplification factors are computed and applied to the original image to enhance it [6]. An implementation of this approach is shown in Listing 2.

2.4 Homomorphic Filtering

The intensity of an image at pixel (x, y) can be described as the product of the illumination $i(x, y)$ and the reflectance $r(x, y)$ [4, 8]:

$$f(x, y) = i(x, y) \cdot r(x, y) \tag{4}$$

In the frequency domain, illumination changes across the image are typically manifested by low frequencies, while high frequencies are associated with reflectance changes. Therefore, by applying the logarithm to the image, one can separate the illumination and reflectance components of the image [4, 8]:

$$\log(f(x, y)) = \log(i(x, y)) + \log(r(x, y)) \tag{5}$$

Applying the Fourier transform to this log-image, a filter $H(u, v)$ can be applied to attenuate the low frequencies, that is the frequencies responsible for illumination changes, and increasing the high frequencies responsible for detail. Afterwards, by applying the inverse Fourier transform and the exponential function, the image can be enhanced [4, 8]:

$$f(x, y) = \exp(\mathcal{F}^{-1}(\mathcal{F}(\log(f(x, y))) \cdot H(u, v))) \tag{6}$$

This process is illustrated in Figure 1.

$$f(x,y) \longrightarrow \boxed{\log} \longrightarrow \boxed{\mathcal{F}} \longrightarrow \boxed{H(u,v)} \longrightarrow \boxed{\mathcal{F}^{-1}} \longrightarrow \boxed{\exp} \longrightarrow g(x,y)$$

Figure 1: Homomorphic filtering pipeline.

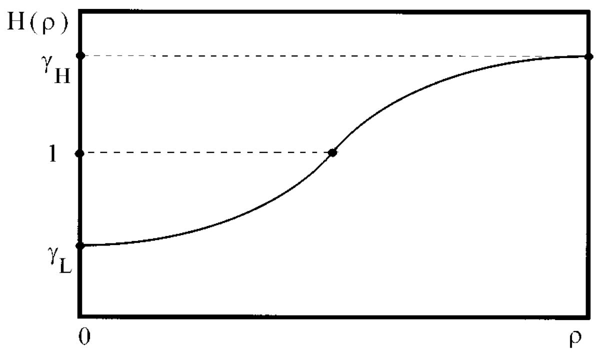

Figure 2: General form of the filter used in homomorphic filtering [8].

Many approaches to the linear filter $H(u,v)$ exist. Voicu et al. propose to use a second order Butterworth filter [8], to reduce the low frequencies and enhance the high frequencies:

$$H(u,v) = H'(\rho) = \gamma_1 - \gamma_2 \cdot \frac{1}{1 + 2.415 \cdot \left(\frac{\rho}{\rho_c}\right)^4}, \tag{7}$$

$$\text{where} \qquad \rho = \sqrt{u^2 + v^2} \tag{8}$$

where $\gamma_H, \gamma_L, \rho_c$ are parameters that can be tuned to achieve the desired effect, and $\gamma_1 \approx \gamma_H, \gamma_2 \approx \gamma_H - \gamma_L$ [8]. The resulting filter has the general form shown in Figure 2.

Finally, Fan et al. [4] propose to append a histogram equalization step to the homomorphic filtering pipeline, in order to improve the contrast of the image.

In order to enhance colored images using homomorphic filtering, this pipeline can be applied to a single channel, e.g. the illumination channel of HSI images, or all channels, as in RGB images. [4, 8]. An implementation of this approach is shown in Listing 3.

2.5 Evaluation of Enhancement

The quality of the enhancement can be evaluated in a few different ways. If the image was enhanced simply to improve its visual appearance, visual inspection often suffices. On the other hand, if the image was enhanced as a preprocessing step for some other computer vision task such as segmentation, the quality of the enhancement should be evaluated by measuring the performance of the computer vision task on the enhanced image. However, there are also some objective metrics that can be used to get an idea of how well an image has been enhanced:

2.5.1 RMS Contrast. Contrast is a measure of the difference in brightness between the darkest and brightest parts of an image, i.e. it is a measure of how well objects are distinguishable. After enhancing an image with uneven illumination, we hope to increase the contrast in the areas of the image that originally had the same illumination. Therefore, an enhanced image might not experience a global increase in contrast, and rather some local increases. The RMS contrast is defined as the variance of the pixel intensities across the entire image [3]:

$$\text{RMS Contrast} = \frac{1}{N \cdot M} \sum_{i=1}^{N} \sum_{j=1}^{M} (I(i,j) - \bar{I})^2 \tag{9}$$

2.5.2 Discrete Entropy. Entropy describes the amount of information in an image, where a high entropy means that the image contains a lot of information, and a low entropy means that the image contains little information, i.e. a flat image has zero entropy. Enhancing an image with uneven illumination should increase the amount of information in the image, and therefore increase the entropy. The discrete entropy is defined as [3, 9]:

$$\text{Discrete Entropy} = -\sum_i P_i \cdot \log_2(P_i) \tag{10}$$

where P_i is the probability that the difference between two adjacent pixels is i.

3 METHODOLOGY

4 RESULTS

5 DISCUSSION AND CONCLUSIONS

REFERENCES

[1] Kobus Barnard and Brian Funt. 1998. Investigations into multi-scale retinex. *Proc. Colour Imaging in Multimedia'98* (1998), 9–17.

[2] Guang Deng. 2010. A generalized unsharp masking algorithm. *IEEE transactions on Image Processing* 20, 5 (2010), 1249–1261.

[3] Nilanjan Dey. 2019. Uneven illumination correction of digital images: A survey of the state-of-the-art. *Optik* 183 (2019), 483–495.

[4] Chun-Nian Fan and Fu-Yan Zhang. 2011. Homomorphic filtering based illumination normalization method for face recognition. *Pattern Recognition Letters* 32, 10 (2011), 1468–1479.

[5] Koichi Morishita, Shimbu Yamagata, Tetsuo Okabe, Tetsuo Yokoyama, and Kazuhiko Hamatani. 1988. Unsharp masking for image enhancement. US Patent 4,794,531.

[6] Ana Belén Petro, Catalina Sbert, and Jean-Michel Morel. 2014. Multiscale retinex. *Image Processing On Line* (2014), 71–88.

[7] Zenglin Shi, Yunlu Chen, Efstratios Gavves, Pascal Mettes, and Cees GM Snoek. 2021. Unsharp mask guided filtering. *IEEE Transactions on Image Processing* 30 (2021), 7472–7485.

[8] Liviu I Voicu, Harley R Myler, and Arthur Robert Weeks. 1997. Practical considerations on color image enhancement using homomorphic filtering. *Journal of Electronic Imaging* 6, 1 (1997), 108–113.

[9] Zhengmao Ye, Habib Mohamadian, and Yongmao Ye. 2007. Discrete entropy and relative entropy study on nonlinear clustering of underwater and arial images. In *2007 IEEE International Conference on Control Applications*. IEEE, 313–318.

A LISTINGS

A.1 Unsharp Masking

```python
import numpy as np
import cv2


def process_image(self, image: np.ndarray) -> np.ndarray:
    blurred = cv2.GaussianBlur(image, (self.ksize, self.ksize), 0)
    sharpened = cv2.addWeighted(image, 1 + self.alpha, blurred, -self.alpha, 0)
    return sharpened
```

Listing 1: Unsharp masking

A.2 Retinex

```python
from typing import List, Optional

import numpy as np
import cv2


def get_ksize(sigma: float) -> int:
    # opencv calculates ksize from sigma as
    # sigma = 0.3*((ksize-1)*0.5 - 1) + 0.8
    # then ksize from sigma is
    # ksize = ((sigma - 0.8)/0.15) + 2.0
    return int(((sigma - 0.8) / 0.15) + 2.0)


def get_gaussian_blur(
    img: np.ndarray, ksize: Optional[int] = None, sigma: float = 5.0
) -> np.ndarray:
    if ksize is None:
        ksize = get_ksize(sigma)
    # Gaussian 2D-kernel can be seperable into 2-orthogonal vectors
    # then compute full kernel by taking outer product or simply mul(V, V.T)
    sep_k = cv2.getGaussianKernel(ksize, sigma)
    # if ksize >= 11, then convolution is computed by applying fourier transform
    return cv2.filter2D(img, -1, np.outer(sep_k, sep_k))


def ssr(img: np.ndarray, sigma: float) -> np.ndarray:
    return np.log10(img) - np.log10(get_gaussian_blur(img, ksize=0, sigma=sigma) + 1.0)


def msr(img: np.ndarray, sigma_scales: List[float] = [15, 80, 250]) -> np.ndarray:
    msr = np.zeros(img.shape)
    for sigma in sigma_scales:
        msr += ssr(img, sigma)
    msr = msr / len(sigma_scales)
    # computed MSR could be in range [-k, +l], k and l could be any real value
    # so normalize the MSR image values in range [0, 255]
    msr = cv2.normalize(msr, None, 0, 255, cv2.NORM_MINMAX, dtype=cv2.CV_8UC3)
    return msr



def color_balance(img: np.ndarray, low_per: float, high_per: float) -> np.ndarray:
    tot_pix = img.shape[1] * img.shape[0]
    # no.of pixels to black-out and white-out
    low_count = tot_pix * low_per / 100
    high_count = tot_pix * (100 - high_per) / 100
    # channels of image
    ch_list = []
    if len(img.shape) == 2:
        ch_list = [img]
    else:
        ch_list = cv2.split(img)
    cs_img = []
    # for each channel, apply contrast-stretch
    for i in range(len(ch_list)):
        ch = ch_list[i]
        # cummulative histogram sum of channel
        cum_hist_sum = np.cumsum(cv2.calcHist([ch], [0], None, [256], (0, 256)))
        # find indices for blacking and whiting out pixels
        li, hi = np.searchsorted(cum_hist_sum, (low_count, high_count))
        if li == hi:
            cs_img.append(ch)
            continue
        # lut with min-max normalization for [0-255] bins
        lut = np.array(
            [
                0 if i < li else (255 if i > hi else round((i - li) / (hi - li) * 255))
                for i in np.arange(0, 256)
            ],
            dtype="uint8",
        )
        # constrast-stretch channel
        cs_ch = cv2.LUT(ch, lut)
        cs_img.append(cs_ch)
    if len(cs_img) == 1:
        return np.squeeze(cs_img)
    elif len(cs_img) > 1:
        return cv2.merge(cs_img)
    raise Exception("Color balance failed")


def msrcp(
    img: np.ndarray,
    sigma_scales: List[float] = [15, 80, 250],
    low_per: float = 1,
    high_per: float = 1,
) -> np.ndarray:
    # Intensity image (Int)
    int_img = (np.sum(img, axis=2) / img.shape[2]) + 1.0
    # Multi-scale retinex of intensity image (MSR)
    msr_int = msr(int_img, sigma_scales)
    # color balance of MSR
    msr_cb = color_balance(msr_int, low_per, high_per)
    # B = MAX/max(Ic)
    B = 256.0 / (np.max(img, axis=2) + 1.0)
    # BB = stack(B, MSR/Int)
    BB = np.array([B, msr_cb / int_img])
    # A = min(BB)
    A = np.min(BB, axis=0)
    # MSRCP = A*I
    msrcp = np.clip(np.expand_dims(A, 2) * img, 0.0, 255.0)
    return msrcp.astype(np.uint8)
```

Listing 2: Retinex

A.3 Homomorphic Filtering

```python
import numpy as np
import cv2
from util import BGR2HSI, HSI2BGR


def filter(value, gamma_1: float = 1.0, gamma_2: float = 0.6, rho: float = 2.0):
    return gamma_1 - gamma_2 * (1 / (1 + 2.415 * np.power(value / rho, 4)))


def process_image(image: np.ndarray) -> np.ndarray:
    """
    Process image using the model.

    Parameters
    ----------
    image : np.ndarray
        Image to be processed, as BGR.

    Returns
    -------
    np.ndarray
        Processed image, as BGR.
    """
    # Convert image to HSI space
    image = image.astype(np.float32)
    hsi = BGR2HSI(image)

    # Extract intensity channel and apply homomorphic filtering
    i = hsi[:, :, 2]
    i_log = np.log2(i + 1.0)
    i_log_fft_shifted = np.fft.fftshift(np.fft.fft2(i_log))
    i_log_fft_shifted_filtered = np.zeros_like(i_log_fft_shifted)
    for i in range(i_log_fft_shifted.shape[0]):
        for j in range(i_log_fft_shifted.shape[1]):
            i_log_fft_shifted_filtered[i, j] = i_log_fft_shifted[i, j] * filter(
                np.sqrt(
                    (i - i_log_fft_shifted.shape[0] / 2) ** 2
                    + (j - i_log_fft_shifted.shape[1] / 2) ** 2
                )
            )
    i_log_filtered = np.real(np.fft.ifft2(np.fft.ifftshift(i_log_fft_shifted_filtered)))
    i_filtered = np.exp2(i_log_filtered) - 1.0
    # Replace intensity channel with filtered one
    hsi_filtered = hsi.copy()
    hsi_filtered[:, :, 2] = i_filtered

    # Convert image back to BGR space
    image = HSI2BGR(hsi)
    image = np.clip(image, 0, 255)
    image = image.astype(np.uint8)

    # Equalize histogram of value channel
    image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    image[:, :, 2] = cv2.equalizeHist(image[:, :, 2])

    # Convert image back to BGR space
    image = cv2.cvtColor(image, cv2.COLOR_HSV2BGR)
    return image
```

Listing 3: Homomorphic filtering