Image Processing Lab 2

Mohammad Sorkhian

This lab has two main parts, spatial filtering, and thresholding. In the first section, we have various kernels to applying to given images. In the second part, we have to implement a threshold-calculating function based on the Otsu method and create the corresponding binary image. The obtained threshold values for img3 and img4 are 178 and 130, respectively. The developed functions and the outputs of kernel 1-8 have been presented along with this report.

Discussion

1.

- The first three kernels follow the same method, average. In this method, in each kernel, we have equal weight for all kernel pixels. In the output images, we can see that the second kernel has a more substantial blurring effect since it has a larger kernel, which considers more neighbors for calculating each pixel value. Kernel_2 and Kernel_3 are identical (weight=0.0204 and size=[7, 7]).
- Kernel_4 and Kernel_5 use a rotationally-symmetric-Gaussian-lowpass filter (fspecial('gaussian', hsize, sigma)) applying the most significant weight to the kernel center and decrease this value as distance of pixels increase from the center. hsize and sigma return kernel size and standard deviation, respectively. As we increase standard deviation or kernel size, we would experience a more smoothing effect.
- To understand different effects of Gaussian and simple averaging filters, we can compare the results of Kernel_1 with Kernel_4 and Kernel_2 with Kernel_5, which these sets are identical in kernel size [3, 3] and [7, 7] respectively. In the Gaussian filters, we can see that the blurring effect is noticeably lower than simple averaging filters with the same size, and it is because of their difference in their weighting system. In the Gaussian filter center pixel of the kernel has the most significant weights, and neighboring pixels weight has a negative relationship with the distance from the center. Interestingly, although the Gaussian filter of the second set has a higher

standard deviation (1.2) than the first set (0.5), we still experience less blurring effect in the Gaussian kernel than the simple averaging kernel.

- If we compare kernel_1 (average [3, 3]) with kernel_6 (median [3, 3]), it is clear that in the median filter, we have been able to maintain more details from the original picture. This filter sorts values of neighboring pixels and substitutes the center pixel value with the median value. With this approach, we would not have any intensity value that is not among the value of neighboring pixels, while in Kernel_1, we may end up with a filtered image that has some pixels value that may not be among their neighboring values. Finally, it is obvious that as we increase kernel size in the median filter from [3, 3] to [5, 5], we end up with a stronger blurring effect.

2. To discuss the effect of kernel size on the result, we can compare Kernels 1 and 2 with each other, which only are different in their kernel size. The second one has a larger kernel, and we have a more substantial blur effect. When we take a larger kernel for applying on the image, indeed, we are using more neighboring pixels value for calculating the value of each pixel, and this smooth the image more and cause we lose more details in the filtered image. Furthermore, we need to consider objects size in an image for setting the kernel size. It means when we have small objects, for maintaining sufficient details we need to choose smaller kernels in compare with larger objects

3. Kernel_6 and Kernel_7 represent the first derivative operation on an image in the x and y direction, respectively (Soble). Kernel_8 applies the second-order isotropic derivative operation in both x and y directions. This kernel is a Laplacian Gaussian filter (fspecial('log', hsize, sigma)). hsize is the filter size can be a vector specifying the number of rows and columns or a scaler for a square matrix. kernel_6 is sensitive to sharp and slow (ramps), intensity variation, but it is insensitive to constant ones. Kernel_8, is just sensitive to sharp intensity changes (onset and end of an intensity impulse). Therefore, kernel_8 could present a better edge detection in an image.

4. In the edges, we have a swift transition in intensities, and by this feature, we would be able to detect the edges in a picture. But unfortunately, noises have the same function and can be detected as edge and have a detrimental effect on edge

detection procedure. We need to use different techniques such as blurring to decrease the noise in an image and, after that, apply the edge detection technique.

5. As different light conditions affect pixels intensity level, it can cause some problems. For example, one application of Thresholding is in image segmentation. By this technique, we would be able to distinguish foreground and background regions (pixels). In a case that we have an object with varying illumination, some pixels may be labeled as foreground and others as background. To address this issue, instead of picking one threshold value and performing Global Thresholding for the image, we can divide it into different blocks (e.g., rectangular) and use different local-threshold values. Since these smaller regions are more likely to have uniform illumination, localizing the threshold value can better accommodate changing lighting conditions.

Matlab codes:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%   SPATIAL FILTER   %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %%%%% Section 2 (filter func)
clear all
img = imread('img1.png');
img = rgb2gray(img);
kernel_1 = (1/9)*ones(3);
kernel_2 = (1/49)*ones(7);
kernel_3 = fspecial('average',[7,7]);
kernel_4 = fspecial('gaussian',[3,3],0.5);
kernel_5 = fspecial('gaussian',[7,7],1.2);
result = filter_func(img, kernel_5);
figure, imshow(result)
% buitin_filter = imfilter(img, kernel_5);  % Use this builtin function for
checking output of our function
% figure, imshow(buitin_filter);

    %%%% Section 3 (Median filter)
clear all
img = imread('img1.png');
img = rgb2gray(img);
result = filter_median(img, 5);
figure, imshow(result);
% buitin_filter = medfilt2(img,[5 5]);      % Use this builtin function for
checking output of our function
% figure, imshow(buitin_filter);

    %%%%% Section 4 (Sharpening filter)
clear all
img = imread('img2.png');
img = rgb2gray(img);
kernel_6 = [-1 0 1; -2 0 2; -1 0 1];
kernel_7 = [-1 -2 -1; 0  0  0 ; 1  2  1];
kernel_8 = fspecial('log',3);
result = filter_func(img, kernel_7);
figure, imshow(result);
% builtin_filter = imfilter(img, kernel_7);  % Use this builtin function for
checking output of our function
% figure, imshow(builtin_filter);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%   THRESHOLDING   %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 clear all
 img = imread('img3.png');
 img = rgb2gray(img);
 result = th_otsu(img);
 figure, imshow(result);
%  [counts,x] = imhist(img);
%  builtin_func = otsuthresh(counts);          % Use this builtin function for
checking output of our function
%  disp(round(builtin_func*256));
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%    FUNCTIONS     %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [img_final] = filter_func(image, kernel)
% This func takes kernel and image and apply the filter on the image
[h, w] = size(image);                                   % Fetch image size
dim_kernel = size(kernel);
s = (dim_kernel(1) - 1) / 2;
h_n = h+(2*s);
w_n = w+(2*s);
img_padded = zeros(h_n, w_n);
for t = 1:s
%     img_padded(t, s+1:s+w) = image(t, :);             % Padding from original
image (straight)
%     img_padded(h_n-t+1, s+1:s+w) = image(h-t+1, :);
%     img_padded(s+1:s+h, t) = image(:, t);
%     img_padded(s+1:s+h, w_n-t+1) = image(:, w-t+1);
    img_padded(t, s+1:s+w) = image(s-t+1, :);           % Padding from original
image (mirror)
    img_padded(h_n-t+1, s+1:s+w) = image(h-s+t, :);
    img_padded(s+1:s+h, t) = image(:, s-t+1);
    img_padded(s+1:s+h, w_n-t+1) = image(:, h-s+t);
end
for i = (s+1):s+h                                       % Add original image to
padded image
    for j = (s+1):s+w
        img_padded(i, j) = image(i-s, j-s);
    end
end
%kernel = flip(kernel);              % Use this for switching between correlation
and convolution
img_filtered = zeros(h_n, w_n);
for i = (s+1):s+h                   % Apply the kernel on the image
    for j = (s+1):s+w
        temp = 0;
        for ii = -s:s
            for jj = -s:s
                temp = temp + img_padded(i+ii, j+jj)* kernel(ii+s+1, jj+s+1);
            end
        end
        img_filtered(i, j) = temp;
    end
end
img_final = zeros(h, w);
for i = (s+1):s+h                   % Remove the padding
    for j = (s+1):s+w
        img_final(i-s, j-s) = img_filtered(i, j);
    end
end
img_final = img_final/255;          % Normalize values
end
```

```
function [img_final] = filter_median(image,kernel_size)
% This func takes two args, image and kernel size, and apply median filter on the
image
[h, w] = size(image);
s = (kernel_size - 1) / 2;
h_n = h+(2*s);
w_n = w+(2*s);
img_padded = zeros(h_n, w_n);
for t = 1:s
%     img_padded(t, s+1:s+w) = image(t, :);              % Padding from original
image (straight)
%     img_padded(h_n-t+1, s+1:s+w) = image(h-t+1, :);
%     img_padded(s+1:s+h, t) = image(:, t);
%     img_padded(s+1:s+h, w_n-t+1) = image(:, w-t+1);
    img_padded(t, s+1:s+w) = image(s-t+1, :);          % Padding from original
image (mirror)
    img_padded(h_n-t+1, s+1:s+w) = image(h-s+t, :);
    img_padded(s+1:s+h, t) = image(:, s-t+1);
    img_padded(s+1:s+h, w_n-t+1) = image(:, h-s+t);
end
for i = (s+1):s+h                                      % Add original image to
padded image
    for j = (s+1):s+w
        img_padded(i, j) = image(i-s, j-s);
    end
end
img_filtered = zeros(h_n, w_n);
for i = (s+1):s+h                    % Apply the kernel on the image
    for j = (s+1):s+w
        temp = [];
        for ii = -s:s
            for jj = -s:s
                temp = [temp, img_padded(i-ii, j-jj)];
            end
        end
        temp = sort(temp);
        mid_index = ((kernel_size^2-1)/2+1);
        img_filtered(i, j) = temp(mid_index);
    end
end
img_final = zeros(h, w);
for i = (s+1):s+h                    % Remove the padding
    for j = (s+1):s+w
        img_final(i-s, j-s) = img_filtered(i, j);
    end
end
img_final = img_final/255;           % Normalize values
end
```

```
function [img_final] = th_otsu(img)
% This func calculates threshold based on Otsu's method and generate
% a binary image
 Hist_arr = zeros(1,256);                  % Creat Histogram array
 [h, w] = size(img);                       % Read image width and height
for i = 1:h                                % Fill the Histogram array
    for j = 1:w
        Hist_arr(1, img(i, j)+ 1) = Hist_arr(1, img(i, j)+ 1) + 1 ;
    end
end

Hist_arr_pdf = Hist_arr / (h*w);  % Calculate PDF
var_max = 0;
threshold = 0;
for t = 1:256
    w0 = 0;
    w1 = 0;
    for i = 1:t
        w0 = w0+ Hist_arr_pdf(i);
    end
    for ii = t+1:256
        w1 = w1 + Hist_arr_pdf(ii);
    end
    m0 = 0;
    m1 = 0;
    for i = 1:t
        m0 = m0 + i * Hist_arr_pdf(i) / w0;
    end
    for ii = t+1:256
        m1 = m1 + ii * Hist_arr_pdf(ii) / w1;
    end
    var = w0 * w1 * (m0-m1)^2;
    if var > var_max
        threshold = t;
        var_max = var;
    end
end
disp(threshold);
img_bi = zeros(h,w);
for i = 1:h                          % Create Binary Image
     for j = 1:w
        if img(i, j) >= threshold
                img_bi(i, j) = 1;
        end
     end
end
img_final = img_bi;
end
```