1. Introduction to Computer Vision - Answers

1.1 Background subtraction

Exercise 1.1 Extracting a moving object. ■

It is important to change the type of image data prior to any processing in order to avoid overflow or rounding errors (you will see it especially during the next exercise session).

```
close all
% Load images
img1 = imread('street1.gif');
img2 = imread('street2.gif');

% Convert them to double
img1d = double(img1);
img2d = double(img2);

% Subtract second image form the first one and show
imgSub = img1d - img2d;
figure(1)
imagesc(imgSub);
colormap('gray')

% Check whats the result using imsubtract
imgSub2 = imsubtract(img1,img2);
figure(2)
imagesc(imgSub2);
colormap('gray')
```

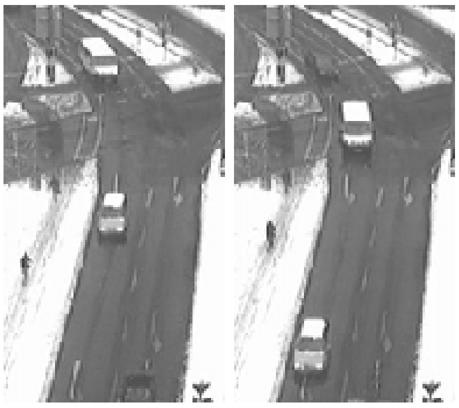

Figure 1.1: Input “street” images

As you can notice, the two difference images do not look the same. If you check the values of `imgSub`, you will find that it includes values from -255 to 255. If we visualize it using `imagesc()` function that scales the values to range from 0 to 1, -256 becomes

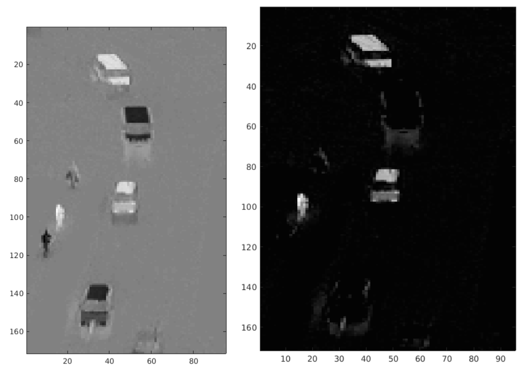

Figure 1.2: Output "street" images. Left: using basic matrix operation, right: using `imsubtract()`

0, pixels with value 0 now have value 0.5 and those with 256 are equal to 1.
Contrary to that `imsubtract()`, takes two uint8 matrices as input. Elements of the output that exceed the range of the integer type are truncated and fractional values are rounded. Because of that now background is black, but at the same time some of the cars are not visible.

Exercise 1.2 Pedestrian detection

We first get a list of all the images in the sequence1 directory.

```
img_folder = 'sequence1/';
img_ext = '*.jpg';
list_images = dir([img_folder img_ext]);
```

We then load all the images in list_images using imread and average them.

```
for i = 1:length(list_images)
    imgs(:,:,:,i) = double(imread([img_folder ...
        list_images(i).name]));
end
mean_img = mean(imgs, 4);
figure, imshow(uint8(mean_img))
```

Here is the expected averaged image:

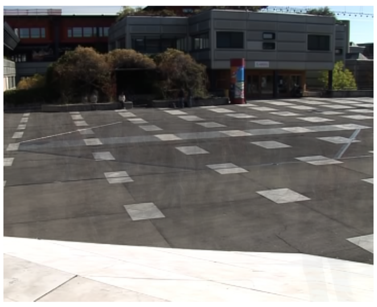

Figure 1.3: Mean image

Once we have the background image, we can just subtract a given image from it. We then average the resulting image over the third dimension and visualize the results in

Fig.1.4:

```
diff_img = mean_img - squeeze(imgs(:,:,:,frame_no));
avg_diff_img = mean(diff_img, 3);
figure, imshow(uint8(avg_diff_img))
```

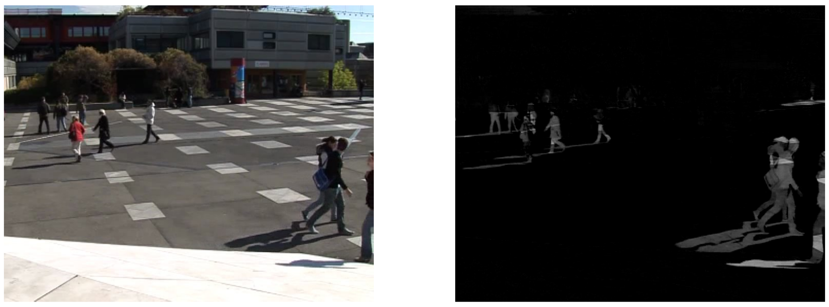

Figure 1.4: Original image corresponding to the 1st frame and the difference image obtained with background subtraction.

Modeling each pixel with a Gaussian distribution consists of computing the mean and standard deviation across all the images for each pixel. Here is the standard deviation image averaged across channels.

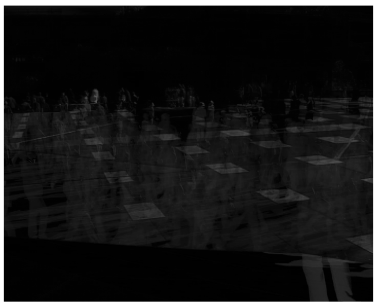

Figure 1.5: Standard deviation image

We can classify a pixel as background if its current intensity (I_t) lies within some confidence interval of its distribution's mean (μ_t):

$$\frac{|(I_t - \mu_t)|}{\sigma_t} > t \rightarrow \textit{Foreground}$$
$$\frac{|(I_t - \mu_t)|}{\sigma_t} < t \rightarrow \textit{Background}$$

where t is some threshold value and σ_t is the standard deviation of the pixel. A large threshold allows for a more dynamic background, while a smaller one would classify the pixel as foreground even for subtle changes. Expected background subtracted image is shown in Fig.1.6.

```
    th =  1.5;
    thresh_img = zeros(size(diff_img));
    thresh_img(diff_img ./ (std_img + 0.00001) > th) = 1;
    avg_thresh = mean(thresh_img, 3);
    figure, imshow(avg_thresh, [])
```

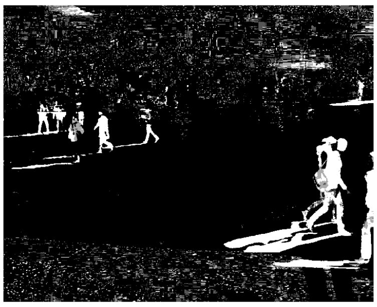

Figure 1.6: Background subtracted image with Gaussian distribution background model.

Note that the result is very noisy because the Gaussian distribution is not a good pick for all pixels.

1.2 Image Segmentation

Exercise 1.3 Image segmentation ■

We first load the image and visualize the histogram of the pixel values by the following:

```
im = imread('wdg.png');
figure, imshow(im)
im = rgb2gray(im);
figure, imhist(im)
```

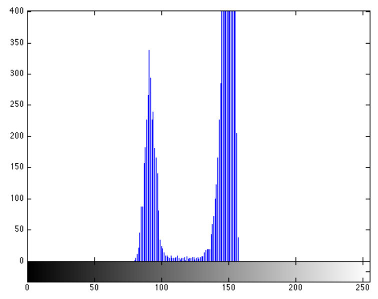

Figure 1.7: Histogram of the pixel values in “wdg.png” image

We then see that the foreground and background are clearly separable by setting a threshold between the two peaks in the histogram as follows:

```
low_thresh = 120;
high_thresh = 180;
seg_im = zeros(size(im));
seg_im( (im >= low_thresh) & (im <= high_thresh) ) = 1;
figure, imshow(seg_im)
```

The original and segmented images are shown in Fig. 1.8. The same procedure can be repeated also for “brain” and “shades” images with different threshold values. Fig. 1.9-1.10 depict the segmented images for these two images. As you can see, in most real-life examples selecting the threshold is not a trivial tasks and in some cases choosing one global thresholds is even not desirable. In a few weeks we will introduce some better methods for segmentation.

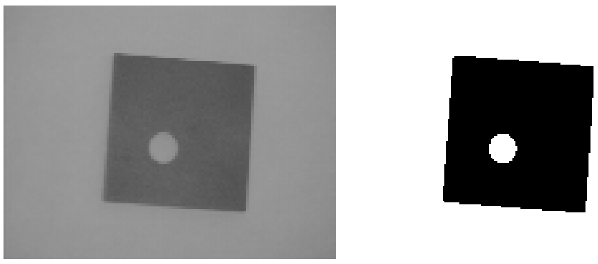

Figure 1.8: Input image (left), segmented image (right)

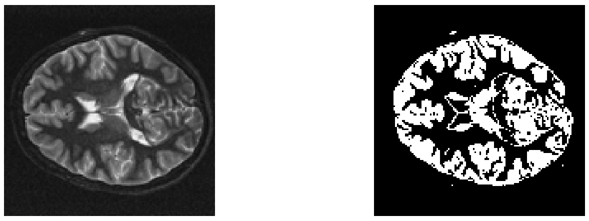

Figure 1.9: Input “brain” image (left), segmented image (right). The low threshold is set to 70, while the high threshold is set to 140.

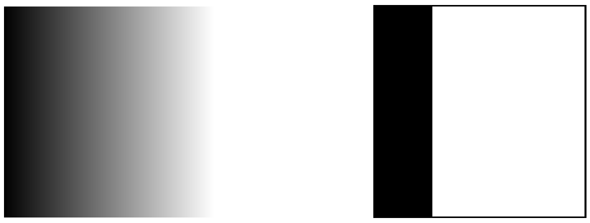

Figure 1.10: Input “shading” image (left), segmented image (right). The low threshold is set to 70, while the high threshold is set to 255.

Exercise 1.4 Color Image segmentation ■

In some segmentation tasks it is easier to consider only one channel of the colour image instead of all of them. In this case, after displaying individual channels it seems that apples can be seen best in a green channel. We investigate its histogram and obtain binary image by selecting pixels that fall between th1 and th2. The binary image is then an input for the supplied function that counts connected components and displays them. Note that we can set the threshold also for the smallest size of a detected component. Thanks to that we can remove small groups of pixels which are noise.

```
close all
clear all
% Load image
img = imread('apples.jpg');
% Read individual channels
R = img(:,:,1);
G = img(:,:,2);
B = img(:,:,3);

% Show channels separately
```

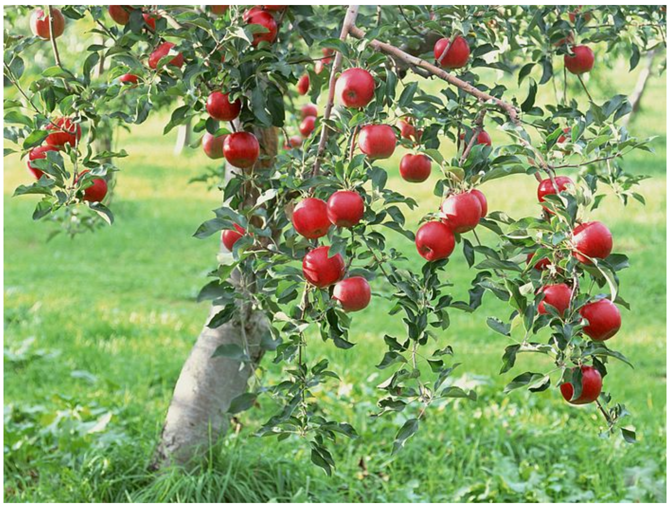

Figure 1.11: An original input image.

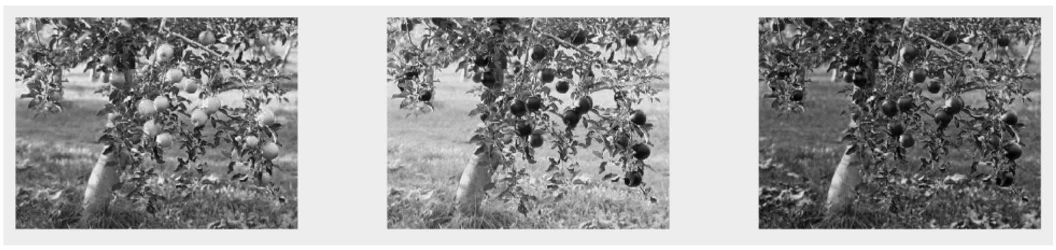

Figure 1.12: Apple tree image decomposed into three channels.

```
figure
subplot(1,3,1);
imshow(R);
subplot(1,3,2);
imshow(G);
subplot(1,3,3);
imshow(B);

% Investigate histogram
figure
imhist(G);

% Get binary image
th1 = 0;
th2 = 50;
thImg = zeros(size(R));
thImg(find(G>th1 & G<th2)) = 1;
figure
imagesc(thImg);
colormap('gray');

% Find and count connected components
sizeTh = 150;
```

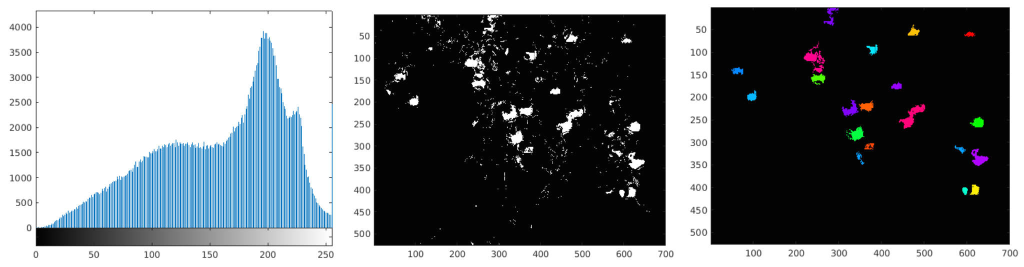

Figure 1.13: Left: histogram of green channel. Middle: a binary image. Right: connected components.

```
[imageColor, nConnectedComponents] = ...
   CountConnectedComponents(thImg,sizeTh);
figure
imagesc(imageColor);
```

Generally, the algorithm detected almost all of the apples. However, some of them are missing or merged with another fruit. One of the limitations of this approach is that it is not robust to clutter (e.g. an apple can be split into two if there is a leaf occluding it). Moreover, selecting thresholds for intensity and size of component is often an arbitrary decision and may be troublesome.
The method could be improved by creating a "template" image of an apple and searching for a similar pattern through the image. The locations where similarity response is large can be assumed to be apples. This task is much easier for a human than a computer because we have some prior information about how the apple looks, such as the shape and colour. This information helps us detect it even in the presence of occlusions or noise.