2. Introduction to Computer Vision - Answers

2.1 Applying Convolutional Filters

Exercise 2.1 ■

Convolution is one of the most important basic building blocks of many computer vision algorithms, e.g. image filtering and convolutional neural networks. This exercise aims to develop an understanding of this operation. Fig. 2.1 gives a demonstration of convolution.

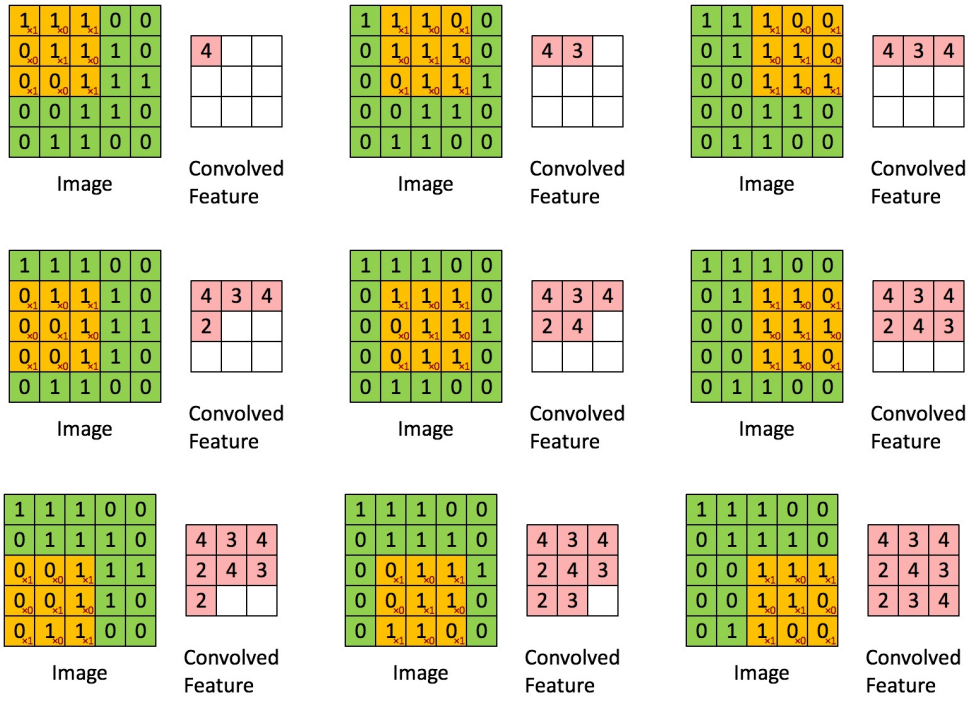

Figure 2.1: Convolution demo. The initial filter is flipped across x and y axis and then, slided across the image. At each pixel location, the inner product between the filter and neighboring image patch is computed. Figure courtesy of Andrew Ng, UFLDL Deep Learning Tutorial.

Border pixels have no neighbors at the edges. Therefore, these pixels should be handled in a different way. There is no single solution to this problem. However, the common practice is to take only the valid portion of convolution or to pad the image so that the border pixels have sufficient amount of neighbors to compute the inner product. Padding can be done with zeros, by repeating edge pixel values or by taking the mirror reflections of border elements.
Note that it is important to use `double` type instead of `uint8` as with `uint8`, the large values will be truncated to 255. This has been done in the main code using `im2double`. The following is an example implementation of convolution using the *valid* region of the image. The output of convolution is shown in Fig. 2.2.

```
function R = applyImageFilter( I, F )
```

```
R = zeros(size(I,1)-(size(F,1)-1)/2, ...
    size(I,2)-(size(F,2)-1)/2);

% Convolution only in the valid region
R = zeros(size(I,1)-(size(F,1)-1)/2, ...
    size(I,2)-(size(F,2)-1)/2);
% Loop through pixels
for row = (size(F,1)-1)/2+1:size(I,1)-(size(F,1)-1)/2
    for col = (size(F,2)-1)/2+1:size(I,2)-(size(F,2)-1)/2
        % Loop through filter elements, compute inner ...
           product
        for rowF = -(size(F,1)-1)/2:(size(F,1)-1)/2
            for colF = -(size(F,2)-1)/2:(size(F,2)-1)/2
                R(row, col) = R(row, col) + F(rowF + ...
                    (size(F,1)-1)/2 + 1, colF + ...
                    (size(F,1)-1)/2 + 1) * I(row - rowF, ...
                    col - colF);
            end
        end
    end
end
```

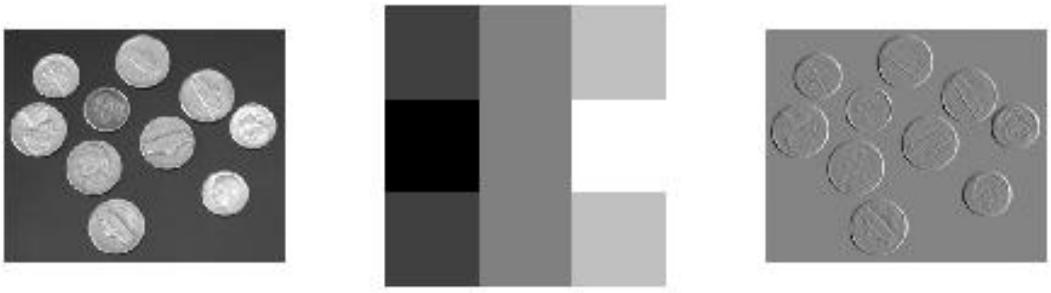

Figure 2.2: The original image, the filter and the resulting output image

If `imshow`, instead of `imagesc`, is used, the pixel values will not be scaled to the limits of the colormap, thus the contrast differences will not be as visible as in the case of `imagesc`.

2.2 Edges and Edge Detectors

Exercise 2.2 ■

The filter in Exercise 2.1 computes the partial derivative of the image in the horizontal direction. Using `impixelinfo` after `imshow` or `imagesc`, one can see that sharp changes in the pixel values occur rather horizontally. When going from a dark area to a bright area, pixel values increase. This is in line with the gradient magnitude and direction. The term $\frac{1}{8}$ is used to normalize the gradient value by computing the weighted average of the contributions from neighboring pixels.

Exercise 2.3 ■

In order to compute vertical and horizontal gradient components we just convolve the original image with Sobel filters S_x and S_y. We can do this by applying our function `applyImageFilter`. The magnitude and phase can be obtained in the following way:

```
% Construct Sobel filters
Sx = 1/8*[-1 0 1;-2 0 2;-1 0 1];
Sy = 1/8*[-1 -2 -1;0 0 0;1 2 1];
%Convolve them with the image
gradX = applyImageFilter(img,Sx);
gradY = applyImageFilter(img,Sy);

%Compute gradient magnitude
gradMag = sqrt(gradX.^2 + gradY.^2);
% Compute phase
gradPhase = atan2(gradY,gradX);

figure
subplot(1,3,1)
imagesc(gradX);
title('Horizontal gradient magnitude')
subplot(1,3,2)
imagesc(gradY);
title('Vertical gradient magnitude')
subplot(1,3,3)
imagesc(gradPhase);
```

Below we present the resulting horizontal and vertical components of the gradient value along with its phase.

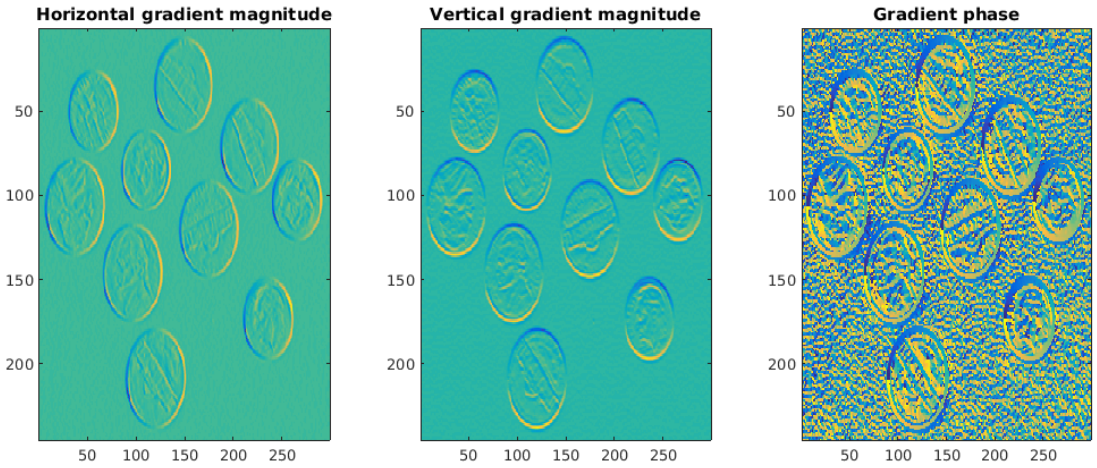

Figure 2.3: Horizontal and vertical gradient components and its phase.

As you can see gradient takes extreme values at the border of coins. Therefore it forms the basis of one of the conventional edge detection methods. As we go along the coin, the phase of gradient changes from -2π to 2π, which makes sense as we make a full circle following the coin border.

Exercise 2.4 ■

Gaussian filter is one of the basic and most important tools in image processing. When constructing a Gaussian filter we can specify its size (here denoted by `fSize`) and standard deviation of the underlying Gaussian function (here `fSigma`). Below we visualize Gaussian filters with different sizes and σ.

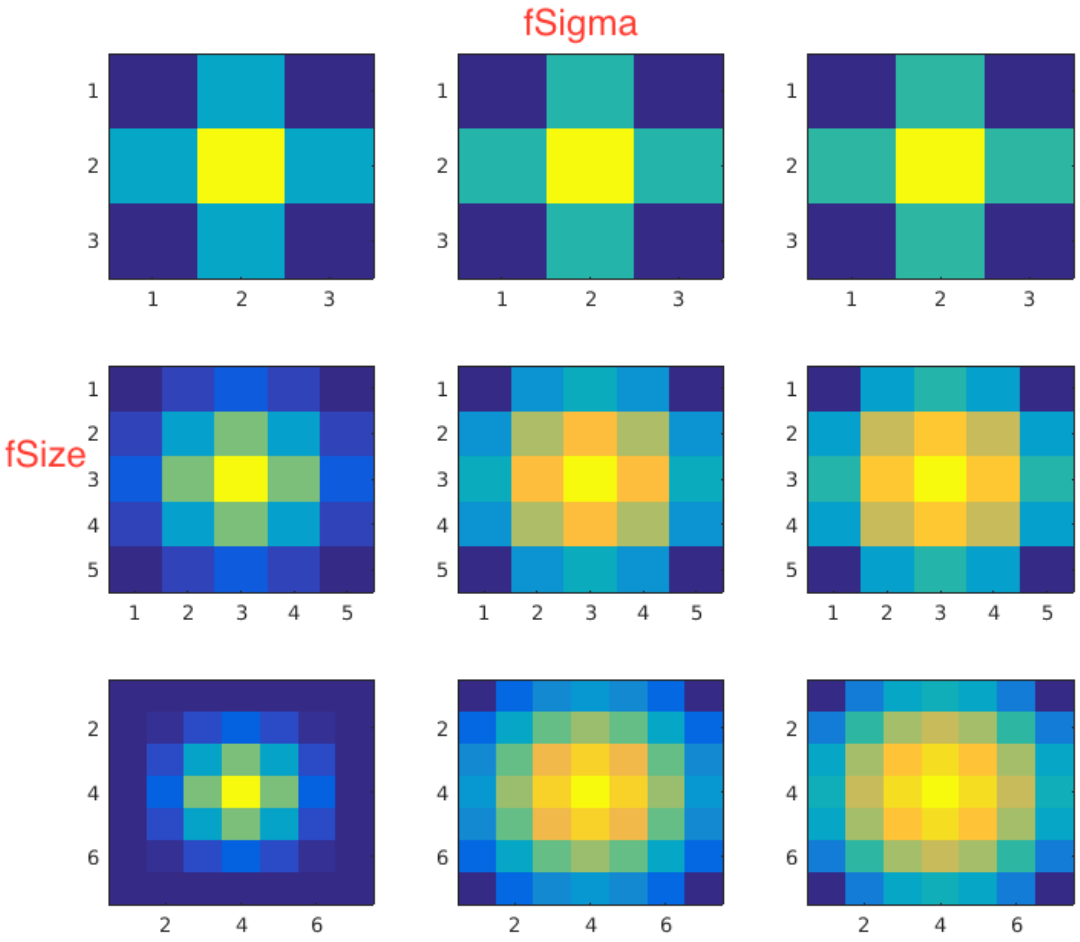

Figure 2.4: Gaussian filters of size 3,5 and 7 and σ 1,2.5 and 4.

Gaussian filter is used first of all for data smoothing, which is an important preprocessing step typically to reduce the amount of noise. Before proceeding to smoothing, one has to choose the size and σ of the filter. The effect of those two parameters are shown in Fig.2.5.
Generally, the bigger the size of filter, the more neighbourhood is taken into account while smoothing. The bigger σ, the bigger effects those neighbours have on the centre pixel. As usual, there is a trade-off between the amount of noise reduced and keeping small details of interest. As a rule of thumb, we usually use the filters of size $6\sigma \times 6\sigma$.

2.3 Separable and Non-separable filters

In this exercise we will create several Gaussian and random filters of increasing sizes and we will compare the running times of filtering with these filters. Intuitively, we expect that the Gaussian filtering will be much faster than the random filtering, due to the

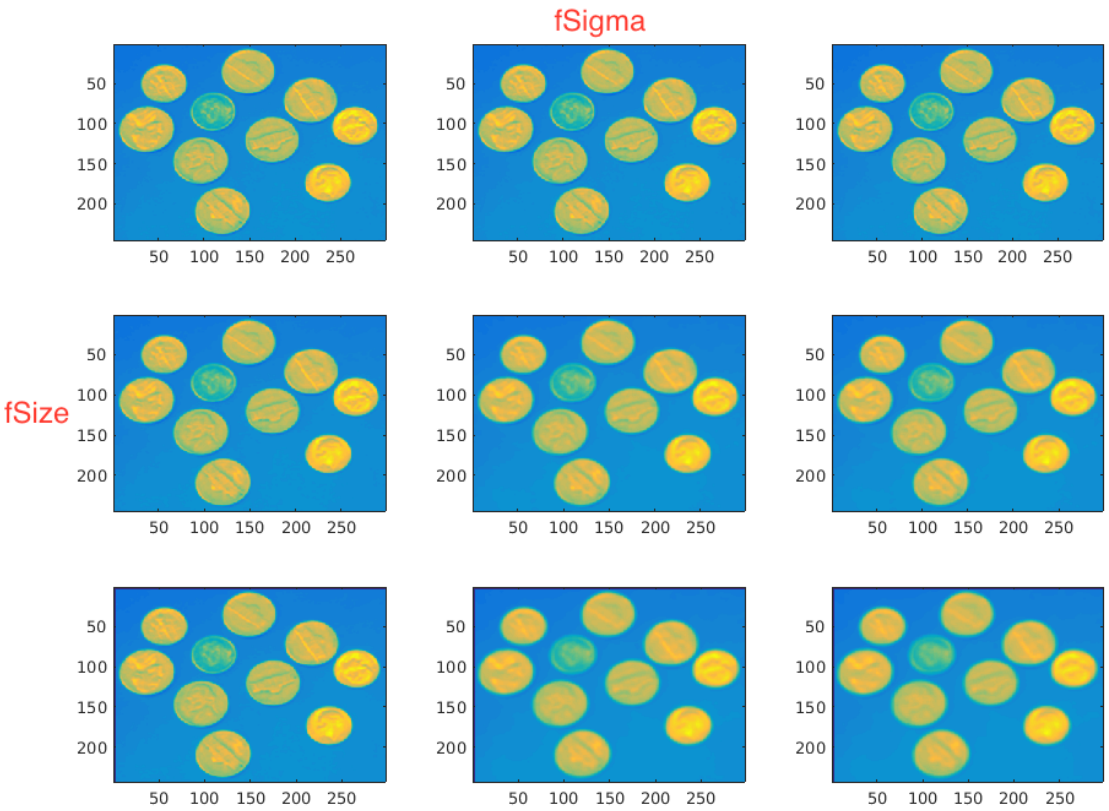

Figure 2.5: The results of smoothing image with Gaussian filters of size 3,5 and 7 and σ 1,2.5 and 4 respectively.

separability of the Gaussian filter. We run 10 times the same filtering procedure and we average the resulting times for more reliable comparison.

```
% range of filter sizes
s = 3:2:71;
ss = length(s);

% allocate memory of the running times
Ls = zeros(ss,1);       % filter size
LSep = zeros(ss,1);     % running time for separable filter
LNSep = zeros(ss,1);        % running times for ...
   non-separable filter

% loop over the filter size
iter = 10;
for l = 1:ss

    % initialize time for separable filtering
    tSep = 0;

    % create the Gaussian filter
    % usually the size of the filter is [6*sigma,6*sigma]
```

```
    ff = fspecial('gaussian', s(l), s(l)/6);

    % run the experiment ten times and
    % average the running times
    for i = 1:iter
        tic;
        imgi = imfilter(img,ff);
        tSep = tSep + toc;
    end
    LSep(l) = tSep / iter;


    % initialize time for non-separable filtering
    tNSep = 0;

    % create the random filter
    ff = rand(s(l));

    % run the experiment ten times and
    % average the running times
    for i = 1:iter
        tic;
        imgi = imfilter(img,ff);
        tNSep = tNSep + toc;
    end
    LNSep(l) = tNSep / iter;
end

% plot the running times as a function of the size
clf;
plot(s, LSep, 'b', s, LNSep, 'r');
legend('Separable','Non-separable');
xlabel('Filter size');
ylabel('Running time (in seconds)');
```

In Figure 2.6 you see a comparison of the running times after executing the above code. It is obvious from the figure that separable filtering is very efficient and does not depend on the filter size. This is not the case, however, with non-separable filtering, where there is a large correlation between filter size and running time. For only very small filter sizes, we observe similar running times between the two filters.

2.4 Bonus: Implement Separable Convolution

The code for one-dimensional convolution can be directly adjusted from the solution of exercise 2.1: Here, F_1 and F_2 are the two one-dimensional filters that make up the final two-dimensional filter F.

```
    function R = applyImageFilter(I, F1, F2)
    [n,m] = size(I);
    k = length(F1); % column vector
    l = length(F2); % row vector
```

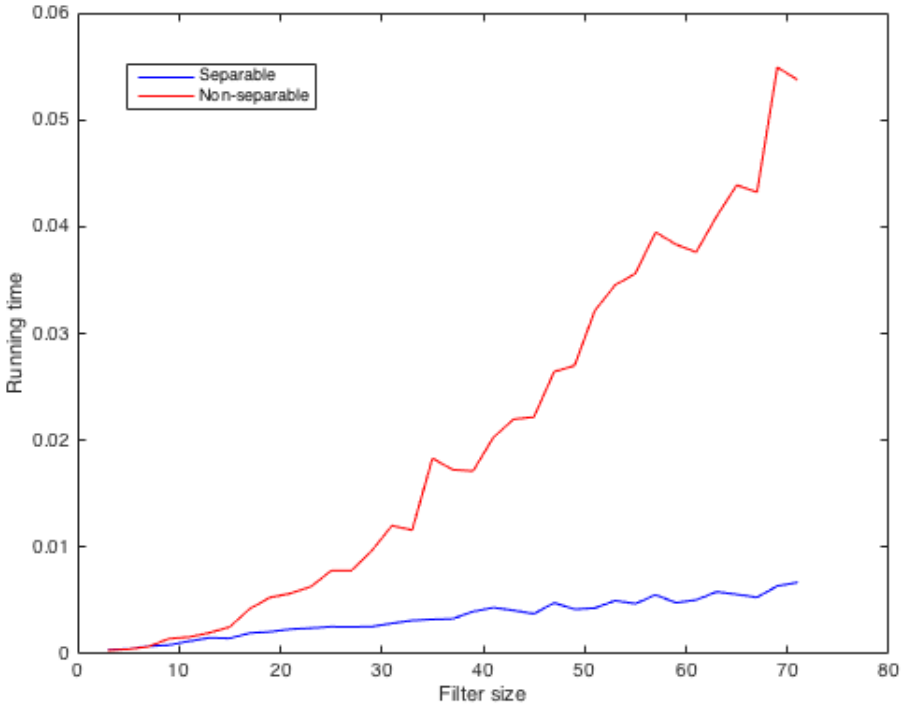

Figure 2.6: Comparison of running times between separable and non-separable filtering for different filter sizes.

```
% auxiliary offset variables
offset1 = (k - 1) / 2;
offset2 = (l - 1) / 2;

% allocate memory for the final result
R = zeros(n,m);

% 1-d convolution with F1 (column vector)
for col = offset2+1:m-offset2
    for row = offset1+1:n-offset1
        % Loop through filter elements, compute inner ...
           product
        for rowF = -offset1:offset1
            R(row,col) = R(row,col) + ...
               F1(rowF+offset1+1) * I(row+rowF,col);
        end
    end
end

% 1-d convolution with F2 (row vector)
% we filter R1 again, save it to an auxiliary variable
R_temp = R;
```

```
R = zeros(n,m);
for row = offset1+1:n-offset1
    for col = offset2+1:m-offset2
        % Loop through filter elements, compute inner ...
           product
        for colF = -offset2:offset2
            R(row,col) = R(row,col) + ...
               F2(colF+offset2+1) * R_temp(row,col+colF);
        end
    end
end

% valid convolution
R = R(offset1+1:n-offset1,offset2+1:m-offset2);
```