Day 3: Image Filtering

CURIE Academy 2015: Computational Photography

Sign-Off Sheet

NAME: _____

NAME: _____

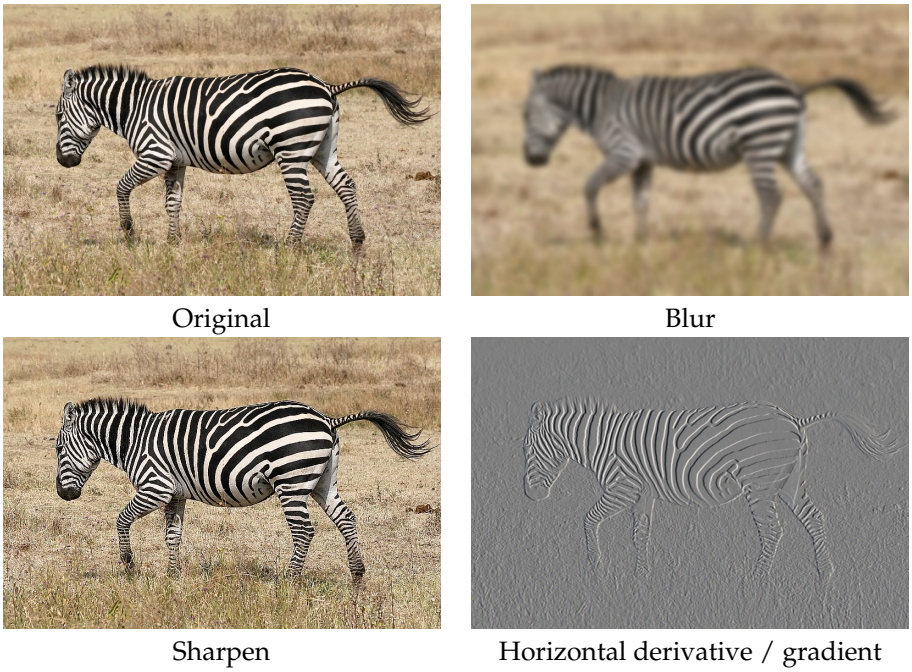
	Sign-Off Milestone	**TA Initials**
Exercise 1	Average Filter	
Exercise 2	Gradient Magnitude	
Exercise 3	(Optional) Motion Blur	
Exercise 4	(Optional) Sharpen Filter	

1 Introduction

Yesterday, we saw some examples of how to transform an image—for instance, by moving pixels around (remember your horizontal flip?), or by changing pixel colors (recall your image negation function). Today, we will look at another important way to transform an image known as **image filtering**. The basic idea of image filtering is that we take each pixel value in an image, and replace it with a new value calculated from the original value of the pixel and nearby pixels, according to some recipe for combining these values.

How do we define this pixel-value-combining recipe? One common way is with what we call a **linear filter**. With linear filters, each new pixel value is computed as the weighted sum of nearby pixel values. Linear filters can be described by a **filter** or **kernel**, which is a (usually very small) matrix of values specifying the contribution of each nearby value—you can think of this matrix as a small image, often 3×3 or 5×5. We often call the kernel values **weights** since they weight the contribution of each nearby pixel to the filtered result. Filtering with a kernel is also called **cross-correlation** and its mathematical symbol is $\star$.[1] With linear filtering, the exact same recipe is applied to each input pixel to produce the corresponding output pixel.

What can a linear filter do? It has the power to do many things! What it does depends on the weights we define in the kernels. With the right kernel we can blur or sharpen an image. We can also find the **derivatives** of an image (also known as "image gradients"). By derivatives, we mean how quickly the pixel values are changing in a given direction (for instance, the horizontal or vertical direction)—derivatives are large near strong image edges, and small in flat regions of an image. The images below illustrate filtering with various kernels.



Original | Blur

Sharpen | Horizontal derivative / gradient

Figure 1: Filtering examples.

Today, we will learn all about filters and how they can be used to do a ton of interesting things with images.

[1] Top secret note: You may also hear about "convolution" which is very similar to cross-correlation.

2 Example: Blurring an Image

Let's get some hands-on experience with filters by looking at blurring (local averaging) of images. Take a look at the image below. This image is corrupted with **noise**—lots of small local variations in color sort of like static on old televisions.

Figure 2: Noisy image.

We'd like to be able to remove or reduce this noise. How do we remove noise? Because the noise is random (each pixel's noise is independent of other pixels), one thing we can do is average nearby image values. The average of each small, local neighborhood should be an approximation of the image without noise. Let's say we want to replace each pixel with the average of the pixels in a 3×3 neighborhood of that pixel.

Let's do this. First, remember that we can think of an image as a 2D function—it maps pixel locations (x, y) to intensity or color values. So, $I(x, y)$ gives the pixel value of image I at location (x, y). Let I_{avg} be the average image we want to compute. Then we can write each output pixel as the average of the central pixel and eight surrounding pixels:

$$\begin{aligned} I_{avg}(x,y) = \frac{1}{9}(&I(x-1,y-1) + I(x-1,y) + I(x-1,y+1) \\ &+ I(x,y-1) + I(x,y) + I(x,y+1) \\ &+ I(x+1,y-1) + I(x+1,y) + I(x+1,y+1)) \end{aligned} \tag{1}$$

We can write this more compactly using summation notation:

$$J(x,y) = \frac{1}{9} \sum_{j=-1}^{1} \sum_{i=-1}^{1} I(x+i, y+j).$$

This is a classic case where we can write this even more elegantly as a filtering operation. Remember that the filter is just a tiny matrix or image that specifies that weights that you apply to the surrounding pixels. In the case of the 3×3 average operation, the filter looks like this:

$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$

Figure 3: 3×3 average filter.

The way to interpret this matrix is that if we center it on a pixel in our input image, we will replace that pixel's value with a weighted sum of neighboring values (in this case, with a weight of $\frac{1}{9}$ for each pixel). To make this concrete, here is the result of filtering an image I with the 3×3 average filter B:

0	0	0	0	0	0	0	0
0	0	0	90	90	90	0	0
0	0	0	90	0	90	0	0
0	0	0	90	90	90	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

I

$\star$

$\frac{1}{9}$

1	1	1
1	1	1
1	1	1

B

$=$

0	0	10	20	30	20	10	0
0	0	20	30	30	30	20	0
0	0	30	30	80	30	30	0
0	0	20	30	30	30	20	0
0	0	10	20	30	20	10	0
0	0	0	0	0	0	0	0

$I \star F$

In general, the equation for filtering image I with a $(2k+1) \times (2k+1)$ filter F is $J = F \star I$ which, for each pixel (x, y), is defined as:

$$J(x,y) = \sum_{j=-k}^{k} \sum_{i=-k}^{k} F(i,j)I(x+i, y+j).$$

Making and using filters in Matlab. Let's try this in Matlab. Because a kernel is a 2D matrix (similar to an image), we can easily create one on the command line. A matrix is created by entering its values between two brackets while separating rows with semicolons. For example, the command

```
>> A = [1 2 ; 3 4]
```

creates the matrix $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and assigns it to the variable A. You can also use the `ones` and `zeros` commands to create matrices of all ones and all zeros, respectively. For instance, the command

```
>> B = zeros(2,3)
```

creates a 2×3 matrix of all zeros.

Given an image I and a kernel stored in a variable F, you can filter an image with the Matlab function `imfilter` as follows:

```
>> J = imfilter(I,F,'replicate');
```

In this case, the filtered image will be stored in J.[2] Use the `imshow` command to look at the resulting filtered image. If you want to view two images I and J side-by-side use `imcompare(I,J)`.

Exercise 1 *Edit `blur.m`. Create a 3×3 average filter and run it on `zebra-noise.png` with `imfilter`. The filtered result should look similar to the blurred image in Figure 1. Try a 15×15 average filter (hint: use the `ones` function) and look at the result. What happens with the larger filter? Try different filter sizes to find a filter that reduces the noise without blurring the zebra too much.*

Hint 1 *An average filter should sum to 1. How do you check this? To get the sum of a 2D matrix M call `sum(sum(M))`. You call `sum` twice because you are summing over two dimensions.*

Checkpoint: get a TA to sign off for your answers in this section.

[2] You might be wondering what happens when you filter a pixel on the border of an image. When the kernel extends beyond the image border we need to "make up" (also known as "hallucinate") pixels outside the image border. The `replicate` option to `imfilter` will hallucinate pixels which have the same value as the nearest border pixel. There are many other ways to hallucniate pixels beyond the border.

3 Gradients

We can also use filters to find the horizontal and vertical partial derivatives of an image. The filters are shown below. Each filter estimates the partial derivative with respect to an axis by the method of central differences. We call the horizontal partial derivative I_x and the vertical partial derivative I_y.

$\frac{1}{2}$

0	0	0
-1	0	1
0	0	0

$\frac{1}{2}$

0	-1	0
0	0	0
0	1	0

Figure 4: **Left:** horizontal partial derivative filter. **Right:** vertical partial derivative filter.

Negative values. These filters are different from the ones we have seen so far: they have negative weights. So far we have told you that pixel values range from 0 to 255. But, we cannot compute gradients correctly unless we can store negative values in our matrices. Our solution is to convert our matrix to one which stores double-precision floating-point numbers. Floating-point numbers can store both positive and negative real numbers. Furthermore, we will divide the values by 255 so that black is 0 and white is 1.

The gradient magnitude. Once we know the horizontal and vertical partial derivatives we can calculate the gradient magnitude.

$$gradient\_magnitude(I) = (I_x^2 + I_y^2)^{1/2}.$$

In order to calculate the gradient magnitude correctly in Matlab you will need to use **element-wise operations**. An element-wise operation is a math operation that is applied element by element. In Matlab, an operation preceeded by a `.` is element-wise. For example, `A^2` will perform a matrix multiply whereas `A.^2` will square each element of `A`. Use element-wise operations when calculating the gradient magnitude.

Exercise 2 *Edit `im2real.m`. Use `double(I)` to convert matrix I to floating point. Also, divide it by 255 so that white is 1.*

Edit `gradientmag.m`. Use two filters to compute the horizontal gradient and vertical gradient. Then compute the gradient magnitude using the equation above.

Test your functions on `zebra.png`. First call `im2real`, then call `gradientmag`. The gradient magnitude should look like the image below.

Figure 5: Gradient magnitude.

Checkpoint: get a TA to sign off for your answers in this section.

4 (Optional) Can we motion blur? Yup, we can do that.

Suppose you are a special effects engineer working at Industrial Light & Magic, and you decide you want to add motion blur to a shot to make it more dynamic. You want to achieve a look something like this, where pixels are smeared horizontally:

Input image

Motion blurred image

Exercise 3 Challenge: *Edit* `motionblur.m`. *Come up with a filter for motion blurring an image horizontally (by say at least 25 pixels). Apply this filter to the image* `'jetski.png'`. *If you get stuck you can look at the hint below, but don't look at it until you've tried to solve it yourself first.*

Hint 2 *Start by looking at the motion blurred image and thinking about which neighbors influence a single pixel. Is it all the neighbors or just the horizontal neighbors? What kind of filter would be a weighted sum of horizontal neighbors only?*

Checkpoint: get a TA to sign off for your answers in this section.

5 (Optional) Sharpening images

Aside from blurring / averaging, something else linear filters can do is **sharpen** images. Note that the blur or average filters you have used remove detail from images, making sharp edges fuzzy. Sharpening does the opposite—it will tend to make sharp edges even more distinct. We can actually reason about how to do this by considering what happens when we *subtract* a blurry image from the original image. We can implement this subtract-blurry-from-original image operation as a filter, because filtering is a commutative and associative operation. In other words, if we have two filters A and B, the difference of the two filtered images is equivalent to the image filtered once with the difference of the two filters:

$$I \star A - I \star B$$

is the same as

$$I \star (A - B)$$

Hence, we can implement the subtract-blurry-from-original image with the following filter:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

The first filter just has a 1 in the middle; this won't change the image at all. We then subtract an average filter from this and use the difference to filter the image, which produces the following result on the zebra image:

Note that the image above has negative values because the subtract-blurred-from-original filter has negative entries. So the image above is visualized so that gray values mean 0, white values are positive, and black values are negative values. You can think of this image as containing only the *details* of the original image–the sharp edges. To sharpen the original image, we want to *add* this image to the original. Let's see if we can do this as a single filter.

Exercise 4 Challenge: *Edit* `sharpen.m`. *Create a* 3×3 *sharpening filter by modifying the detail filter above. Try your sharpening filter on* `zebra.png`. *It should look like the sharpened image in Figure 1.*

Hint 3 *Does your filter have negative weights? If so, call* `im2real` *once on your image (usually right after you* `imread` *it).*

Checkpoint: get a TA to sign off for your answers in this section.