CS1114 Section 6: Convolution
February 27th, 2013

1 Convolution

Convolution is an important operation in signal and image processing. Convolution operates on two signals (in 1D) or two images (in 2D): you can think of one as the "input" signal (or image), and the other (called the kernel) as a "filter" on the input image, producing an output image (so convolution takes two images as input and produces a third as output). Convolution is an incredibly important concept in many areas of math and engineering (including computer vision, as we'll see later).

Definition. Let's start with 1D convolution (a 1D "image," is also known as a signal, and can be represented by a regular 1D vector in Matlab). Let's call our input vector f and our kernel g, and say that f has length n, and g has length m. The convolution $f * g$ of f and g is defined as:

$$(f * g)(i) = \sum_{j=1}^{m} g(j) \cdot f(i - j + m/2)$$

One way to think of this operation is that we're sliding the kernel over the input image. For each position of the kernel, we multiply the overlapping values of the kernel and image together, and add up the results. This sum of products will be the value of the output image at the point in the input image where the kernel is centered. Let's look at a simple example. Suppose our input 1D image is:

$f =$

10	50	60	10	20	40	30

and our kernel is:

$g =$

1/3	1/3	1/3

Let's call the output image h. What is the value of $h(3)$? To compute this, we slide the kernel so that it is centered around $f(3)$:

10	50	60	10	20	40	30
	1/3	1/3	1/3			

For now, we'll assume that the value of the input image and kernel is 0 everywhere outside the boundary, so we could rewrite this as:

10	50	60	10	20	30	40
0	1/3	1/3	1/3	0	0	0

We now multiply the corresponding (lined-up) values of f and g, then add up the products. Most of these products will be 0, except for at the three non-zero entries of the kernel. So the product is:

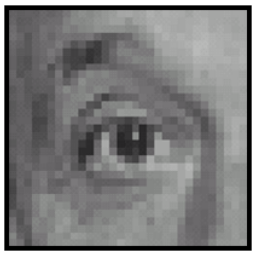
$$\frac{1}{3}50 + \frac{1}{3}60 + \frac{1}{3}10 = \frac{50}{3} + \frac{60}{3} + \frac{10}{3} = \frac{120}{3} = 40$$

Thus, $h(3) = 40$. From the above, it should be clear that what this kernel is doing is computing a windowed average of the image, i.e., replacing each entry with the average of that entry and its left and right neighbor. Using this intuition, we can compute the other values of h as well:

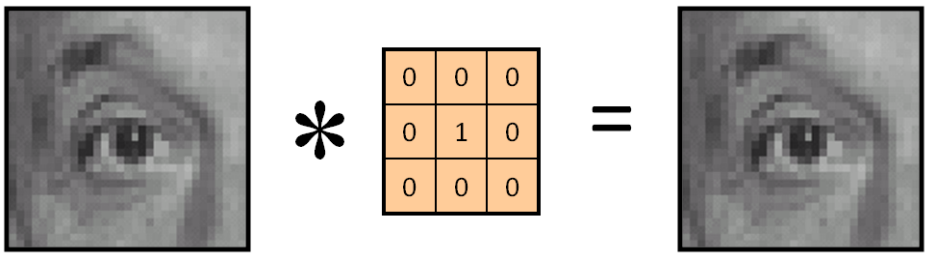
$$h = \boxed{20 \mid 40 \mid 40 \mid 30 \mid 20 \mid 30 \mid 23.333}$$

(Remember that we're assuming all entries outside the image boundaries are 0—this is important because when we slide the kernel to the edges of the image, the kernel will spill out over the image boundaries.)

We can also apply convolution in 2D—our images and kernels are now 2D functions (or matrices in Matlab; we'll stick with intensity images for now, and leave color for another time). For 2D convolution, just as before, we slide the kernel over each pixel of the image, multiply the corresponding entries of the input image and kernel, and add them up—the result is the new value of the image. Let's see the result of convolving an image with some example kernels. We'll use this image as our input: One very simple kernel is just a single



pixel with a value of 1. This is the identity kernel, and leaves the image unchanged:



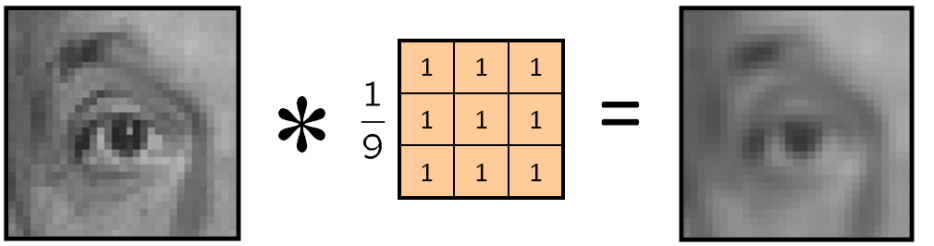
Original

Identical image

Another useful 2D kernel is an averaging or mean filter. Here's what convolving the image with a 3×3 mean filter looks like:

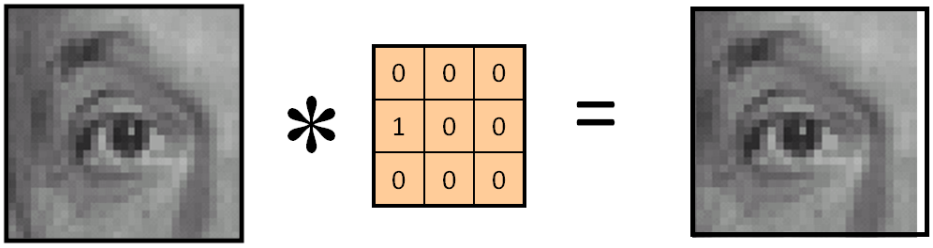
In this case, the mean kernel is just a 3×3 image where every entry is $1/9$ (why is it a good thing for the entries to sum to one?). This kernel has the effect of *blurring* the images—an image where the intensities are smoothed out.

What happens if we convolve the image with an identity kernel but where the one is shifted by one pixel? Then then output image is also shifted by one pixel:



Original

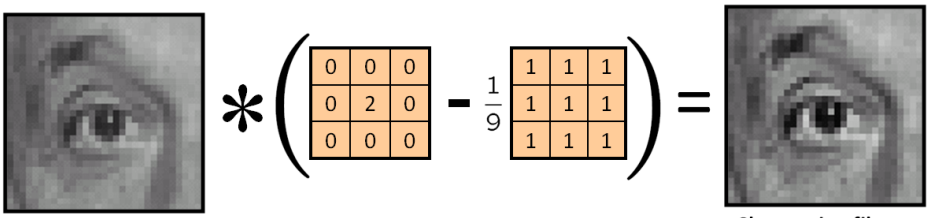
Blur (with a mean filter)



Original

Shifted left
By 1 pixel

We can create filters that do many other things, as well. An identity kernel that has a 2 in the center instead of a 1 multiplies all the image intensities by 2. Something interesting happens if we combine such a kernel with one that *subtracts* the average of the intensities in a 3×3 window: (Note that some of the entries in the resulting kernel will be negative.) This



Original

Sharpening filter
(accentuates edges)

results in an image where edges are accentuated; this is known as a *sharpening* operation, which is kind of like the opposite of blurring. This and many other kernels are built into image editing software such as Photoshop.

2 Convolving Images

Now it's your turn to play with image convolution. In Matlab, 2D convolution can be done with the `conv2` function. Copy over the files in the `/courses/cs1114/sections/convolution/`

directory, and open up Matlab. There are a couple of images that you should have copied over, (`wires.png`) and (`otter.png`). Open up one of the images (and convert to a matrix of doubles, as the Matlab convolution routines assume doubles). For instance:

```
>> f = im2double(imread('wires.png'));
```

Now create a kernel as a small matrix. For instance, you might create a 3×3 mean filter as:

```
>> g1 = [ 1 1 1 ; 1 1 1 ; 1 1 1 ] / 9;
```

Try creating `f` and `g1`, and convolving them to form an image `h`:

```
>> h = conv2(f, g1);
>> imshow(h);
```

How does the output image `h` look compared to `f`?

An averaging filter is one way to blur, but in many cases a nicer image results when you blur with a *Gaussian kernel*. In this kernel, values further from the pixel in question have lower weights. You can get a Gaussian kernel in Matlab using the `fspecial` function:

```
>> gaussian = fspecial('gaussian');
```

Blur the `wires` image with both the average and Gaussian kernels and see if you can notice any differences.

$\Longrightarrow$Now try to create some new kernels. Try creating `g2`, a filter that multiplies the image intensities by 2, and `g3`, a filter that sharpens the image (you should define `g3` in terms of `g1` and `g2`). Try applying both of these to an image, and take a look at the results.

$\Longrightarrow$Try out this kernel:

```
>> g4 = [ -1 -1 0 ; -1 3 0 ; 0 0 0 ];
```

Try applying this to an image (the otter image works better in this case). When you use imshow to view the result, the image will look mostly black—this is because the entries of this kernel sum to 0, instead of 1. To fix this, add 0.5 to the resulting image, e.g.

```
>> imshow(conv2(f, g4) + 0.5);
```

This is known as an *emboss* kernel.

$\Longrightarrow$Come up with your own kernel, and see what happens when you apply it to an image.

Let's consider the blur kernel `g1` again. What if we want to blur more? We could blur the image with `g1` once, then blur the result again with `g1`. This will give a "twice-blurred" image. This is equivalent to the operation: $((f * g_1) * g_1)$. Try this out, and see what the result looks like.

Alternatively, we could first filter `g1` with *itself*, then blur `f` with the result. This works because convolution is associative, so $((f * g_1) * g_1) = (f * (g_1 * g_1))$. Create a kernel `g11` that is `g1` filtered with itself, then convolve `f` with this new filter. We could blur more and more by convolving `g1` with itself multiple times. You can visualize the resulting kernels using `imagesc` (e.g., `imagesc(g11)`). Try this out.

3 Edge Detection

In this section we're going to play around with kernel design with the goal of detecting edges in images. Try your kernels on both images—a particular kernel may work well on one image but not the other.

For the purposes of convolution, images can be thought of as functions on pairs of integers. The image has a given intensity value at each x and y coordinate; we can imagine, as we have been, that all values outside the boundaries of the image are zero. Although images are not continuous functions, we can still talk about approximating their discrete derivatives.

1. A popular way to approximate an image's discrete derivative in the x or y direction is using the Sobel convolution kernels:

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

   $\Longrightarrow$Try applying these kernels to an image and see what it looks like.

   The image derivative (or its two-dimensional equivalent, the *gradient*) is the basis for many well-studied edge-detection algorithms.

2. For edges that go from light to dark, the Sobel operator gives a negative values. Matlab's `imshow` simply displays negative values as black. As with the emboss kernel, you could add 0.5 to the image, but in edge detection it's not necessarily important which way the intensity is changing, but how sharply it is doing so. How can you post-process the convolved image to improve the edge detection to include dark-to-light and light-to-dark edges?

3. The Sobel kernels only look for edges in one direction—x or y. Try to come up with a kernel that accomplishes in one convolution what the x and y Sobel kernels achieve with two.

4. How does your kernel compare the this single-kernel edge detector, which is called the Laplacian?

0	-1	0
-1	4	-1
0	-1	0

5. Now load `wires_noise.png` into Matlab and see how your kernel and the Laplacian behave on this image. This is the same image as the original `wires.png`, but it has some *noise* - small variations that reflect imperfections in the measurement of intensity values. You'll notice a problem—our edge detectors see these tiny local variations and consider them edges! Can use one of the kernels we've seen already in conjunction with the Laplacian to combat this problem?

4 Object recognition

Notice that there are two other images in the directory, `scene.png` and `butterfly.png`. Read in these two images (remembering to convert to double using `im2double`). You'll notice that the butterfly image appears in the scene image in two places. Suppose we don't know where the butterfly is in the image, and we want to find it—this is a simple version of the *object recognition* problem. It turns out that we can use convolution to solve this. Actually, we need a very similar operation called *normalized cross correlation* (the `normxcorr2` function in Matlab). The differences between this and convolution are subtle—you can think of it as a modified convolution, though the details are not too important right now.

Try running `normxcorr2` on the butterfly and scene images (the butterfly image will be the first argument to `normxcorr2` in this case), and look at the results using `imagesc`. You should see two bright spots corresponding to the two large butterflies in the scene image. It turns out that convolving one image with another, "real" subimage, will fire strongly where the two images are similar, and less strongly where they are different. So to detect a particular object in a big image, one simple way is to convolve the big image with the object image.

$\Longrightarrow$Why doesn't this work for the lightsticks that we'll be using to drive the robots? Can you think of a way to make this work?