Computer Vision (CE316 and CE866): Convolution

Adrian F. Clark
⟨*alien@essex.ac.uk*⟩

Contents

List of Figures

1 Introduction

Now that we have some idea of how an image is represented and manipulated within a computer, we can start thinking more seriously about processing images. To motivate the technique that forms the basis of this document, let us think about identifying the location of the ball in robot football (discussed in the first lecture) or capturing the motion of humans for use in games or movies.

Motion capture rigs involve instrumenting the subject, typically by attaching reflective white spheres at joints (Figure 1). The subject is then viewed simultaneously from several high-frame-rate cameras. The locations of the markers are found and tracked from frame to frame, and the relative locations viewed from different cameras used to calculate the markers' locations in 3D. Commercial motion capture rigs are able to do this at video rates.[1] As well as being widely-used in the movie and games industries, this type of motion capture is important in medicine and sports (biomechanics) and biometrics (gait analysis).

Figure 1: Motion capture setup (photograph from the Wikipedia)

[1] Such systems, made by Vicon, are installed in both the Robot Arena and the Games Robotics Lab in CSEE.

In movies, the motion of the camera is calculated during post-production for live-action sequences performed against a chroma-keyed background; the most widely-used software package for doing this, 2D3's *Boujou*, stems from research carried out in the robotics research group at Oxford University. The calculated camera motions are then replicated in the accompanying computer animation and the two combined ("composited") together. (There are some shortcomings in this, for example some zooms and camera motions cannot easily be distinguished.) In games, the motion of real people is captured and used to animate characters in the game.

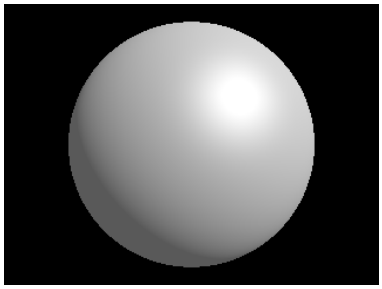


Figure 2: Appearance of a sphere illuminated from above right

In an ideal situation, each of the white spheres attached to the subject's joints would produce a strong reflection which the camera would detect as a single pixel. As the subject moves, we would get a series of single-pixel reflections in successive frames. The most obvious way of identifying the white pixel is to look for the highest-valued pixel in the image; but this does not work well if the reflection from the white sphere is not clearly lit. What we are likely to see in practice is a small region with a distinct variation of brightness, peaking in the specular reflection from the sphere, so simply thresholding the image will not lead to a single pixel that identifies the marker. Hence, to locate automatically points in an image for use in a depth estimation calculation, a good place to start is in enhancing white pixels, either isolated or forming part of a region. This will lead us into a consideration of *convolution* or *filtering*, which is almost certainly the most important image processing operation.

The discussion that follows is presented without any of the theory that underlies convolution commonly found in textbooks: the basic idea can be understood purely through principles and common-sense examples.

2 Enhancing isolated white points using convolution

Let us start by trying to identify isolated white points, *i.e.* ones surrounded by darker pixels. Fairly obviously, the only way to determine whether a pixel is lighter than its surroundings is to examine the values around it;

and we want to perform exactly the same series of operations to each pixel of the image in turn, scanning over the lines and pixels within each line in the way we saw when first considering OpenCV.

Hence, given a pixel at location (x, y) in the image P, we need to examine it and its eight nearest neighbours:

	$x-1$	x	$x+1$
$y-1$	$P(x-1,y-1)$	$P(x,y-1)$	$P(x+1,y-1)$
y	$P(x-1,y)$	$P(x,y)$	$P(x+1,y)$
$y+1$	$P(x-1,y+1)$	$P(x,y+1)$	$P(x+1,y+1)$

If $P(x,y)$ is light, we want its value to become much larger (and hence even lighter) than the other pixels shown above, and the easiest way to do that is to multiply it by a large number. But that is not enough; for if (say) $P(x-1,y)$ were also quite large, it would also be large after multiplication by the same number. So we need to multiply $P(x,y)$ by a large number *and simultaneously* suppress the values of those pixels around it. We can do this by placing a 3×3 'mask' of coefficients above the region of the image:

m_{11}	m_{12}	m_{13}
m_{21}	m_{22}	m_{23}
m_{31}	m_{32}	m_{33}

and multiply each pixel by the corresponding coefficient in this mask. Writing this out explicitly, we calculate

$$\begin{aligned} v \;=\; & P(x-1,y-1)m_{11} + P(x,y-1)m_{12} + P(x+1,y-1)m_{13} \\ + \; & P(x-1,y)m_{21} + P(x,y)m_{22} + P(x+1,y)m_{23} \\ + \; & P(x-1,y+1)m_{31} + P(x,y+1)m_{32} + P(x+1,y+1)m_{33} \end{aligned}$$

and use v to replace $P(x,y)$. We must be careful about the way we do this. In particular, we cannot perform this convolution 'in place' (*i.e.,* writing the results into the image we're reading values from) as we would end up using modified values for $P(x-1,y-1)$, $P(x,y-1)$, $P(x+1,y-1)$ and $P(x-1,y)$ in calculating the result for $P(x,y)$ — this is known as "recursive filtering" for obvious reasons. Hence, this type of processing with a mask always works from one image structure into another.

We have now decided what we are going to do but it remains to choose a set of values for the mask coefficients. The conventional choice is

-1	-1	-1
-1	8	-1
-1	-1	-1

and is known as the Laplacian (after the French mathematician Laplace). This mask fulfils our necessary criteria: the pixel in the centre of the mask is multiplied by a large number, while those around are multiplied by small ones. But why those particular values? The reason is that processing homogeneous region of the image (*i.e.,* one whose pixels all have identical values) will produce a response of zero. A few minutes' consideration should then tell you what processing with this mask will do to an isolated dark pixel: it will result in a large *negative* value.

3 Blurring

Having seen that the Laplacian enhances white pixels, we might ask ourselves what effects other choices of coefficients have. The obvious other extreme is to use

1	1	1
1	1	1
1	1	1

When we convolve this with an image, we are summing up the pixels in 3×3 regions around each pixel, effectively averaging them. (In practice, we would probably use $\frac{1}{9}$ instead of unity in each of the 9 positions in the mask, thereby calculating a true mean.) On a homogeneous region, this would have no effect: the average of 9 identical values is the same as each of them. But if one pixel was widely different from its neighbours, convolution with this mask would tend to reduce its effect; in other words, this can be used to perform noise reduction — though there are better ways, as we shall see shortly. On the other hand, a sharp boundary between uniformly light and dark regions would be smeared out by the region-averaging process — in other words, the image would be *blurred* in much the same way as focusing a camera poorly. Indeed, the above mask is usually called 'a 3×3 blur.'

On images captured by a modern digital camera, a 3×3 blur has a small but noticeable effect; the amount of blurring can be increased by using 5×5, 7×7 *etc*. masks. Mask sizes are almost always odd incidentally, so that the region considered is symmetrical around the pixel to be replaced.

People who have used Photoshop or similar tools may well have come across blurring and convolution. One popular technique they employ for enhancing images is *unsharp masking*. What this effectively does is subtract the blurred image from the original one, enhancing the edges, and add the result to the image.

4 Using the median

Our consideration of blurring masks tells us that we are effectively averaging regions — more precisely, calculating the arithmetic mean of the $9, 25, 49\ldots$ values for 3×3, 5×5, $7 \times 7\ldots$ masks. But the mean is only one measure of the behaviour of random variables; others are the median and the mode. (For a symmetric distribution with a single peak, all three coincide; but this is never achieved on real data — and, as has been said before, computer vision is an experimental discipline.)

The *mode* is the most commonly-occurring value, *i.e.* the peak of the histogram. Modal filtering has never become popular in image processing, perhaps because the numbers of values involved are so small, though extensive work has been done on it [Davies, 2012]. On the other hand, the *median*, the value in the middle position when all values are sorted into ascending order, has proved popular. The main reason for this is what happens in the vicinity of a pixel whose value is widely different from those surrounding it — just the kind of image feature the Laplacian finds. Although such features can arise naturally in images, they tend to arise

more commonly from timing problems in image or video capture circuitry — so-called "salt and pepper" noise — and we normally want to remove them. Convolving using the mean will smear out the effect of a single light or dark pixel over a region of the size of the mask; on the other hand, when the median is used, the value sorts to one extreme of the pixel values and hence has a much smaller effect on the median. So median filtering tends to be better at noise removal, one of the main reasons for contemplating averaging over a region, and introduces less blur too.

In terms of computation, the median takes more effort to calculate than the mean as it involves sorting numbers into order. As people who have studied algorithmics will know, there are many sort algorithms out there. The easiest to code is the 'bubble' sort but it is pretty inefficient. The algorithm with the best theoretical performance is the 'quicksort' and this is a popular algorithm to teach people — but more because it is a way of teaching recursion in programming than for any inherent property. The fact is that quicksort can be a very poor sorting algorithm if the data happen to be ordered unfortunately. A better choice for this kind of task is Shell's sort, which is almost as good as the quicksort in the best case and a lot better than quicksort in the worst case.

These mask-based operators are absolutely central to image processing. In later chapters, we shall look at how they are used in feature detection, with emphasis on finding first lines and then corners in images.

5 Mathematical morphology

You might be asking yourself at this point whether there is anything else that can be used in convolution as well as the mean, mode and median. Two other interesting possibilities are the minimum and maximum values, and these give rise to grey-scale versions of what are known as *morphological* operators. Taking the minimum of a region will naturally tend to make light regions smaller; a 3×3 minimum will usually remove the outermost light pixels from around a region in an image, so this operation is called a *erode* or *shrink*. Conversely, taking the maximum will tend to grow light regions, by one pixel for a 3×3 region, two for a 5×5 region, and so on; this is known as a *dilate* or *expand*.[2] If these are difficult to visualize, think about what happens to a white object on a black background.

[2] A little care is necessary here as some texts consider the shrinking and expanding of a dark region in a white background, so you may encounter the opposite terminology to what I've used.

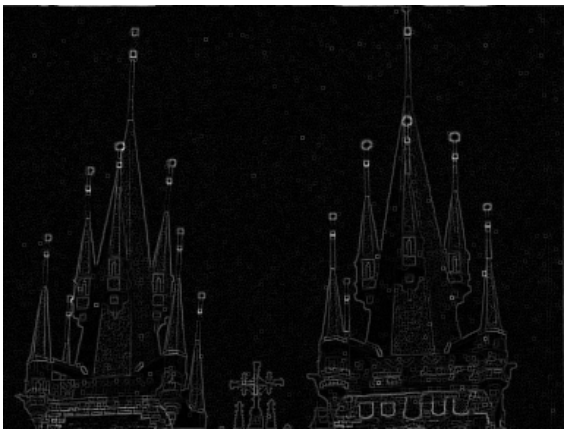
If we perform an erode followed by a dilate, a few moments thought should tell you that we will delete any isolated light pixels inside dark regions, and also grow dark regions inside light ones; this is known as an *opening*. The converse, dilate followed by erode, is a *closing*. Some of these operations are illustrated in Figure 3. It is clear that the specular reflections in Figure 3(b) are much more apparent than those in Figure 3(a), the original image, due to the expansion of light-coloured regions. Similarly, the reflections in Figure 3(c) are much less apparent than the original image. Finally, the result of the opening in Figure 3(d) essentially finds the outlines of the region.

These morphological versions of convolution are useful when one has identified regions of images that correspond to objects of interest, and

(a) Original image

(b) Expand

(c) Shrink



(d) Opening

Figure 3: Morphological expand, shrink and opening

those regions need to be 'tidied up.' They are rarely used directly on images with a view to extracting useful information directly as they lack sophistication.

6 Other types of masks

Having considered masks that enhance isolated spots or blur images, it is useful to look at a couple more masks to motivate the discussion in the next chapter.

Consider what effect on an image the mask

1	1	1
0	0	0
−1	−1	−1

will have. This takes the pixels in a line of the image and subtracts off it the pixels two lines below that. As usual, there is a zero response in uniform regions, where all the pixels have the same value. Now consider a *vertical* edge in the image directly below the centre of the mask: again, the response will be zero. But what if the edge is *horizontal*? If the image contains a lighter region (*i.e.,* with higher-valued pixels) above a darker one, this mask will produce a large positive response when it spans the

edge. Conversely, if the upper region is darker than the lower one, the mask produces a large negative response. So this mask tends to detect horizontal edges. By analogy, it is fairly easy to design a mask to detect vertical edges:

−1	0	1
−1	0	1
−1	0	1

What happens if we wish to detect edges that are neither vertical nor horizontal? We could design masks for the 45° angles. However, an edge running diagonally across the image will produce a response from both these masks, though less in both cases than would be obtained from its optimally-oriented edge — so a way ahead is to combine the responses from these two masks.

This approach is what is used in Sobel's edge detector. It uses two masks which differ a little from those above:

$$H = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \qquad V = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

and combines their responses in quadrature:

$$\sqrt{H^2 + V^2} \tag{1}$$

to calculate the overall response at a pixel. The Sobel detector is able to run at video rate on quite modest hardware, so is widely used in practice as an approximation to the scheme due to Canny, presented in the next chapter. However, the Canny edge detector is able to run at video rate on modern PC-class hardware too.

7 Matched filtering

There are many cases where we might wish to find arbitrary-shaped regions within an image. To give a concrete example of this, Figure 4 shows a micrograph of a cell membrane where proteins in or on the membrane give it a characteristic 'lumpy' appearance. In modelling the biological processes of cells, one needs to know how much the proteins clump together into regions; and to do that, one needs to determine where each protein lies in the image.

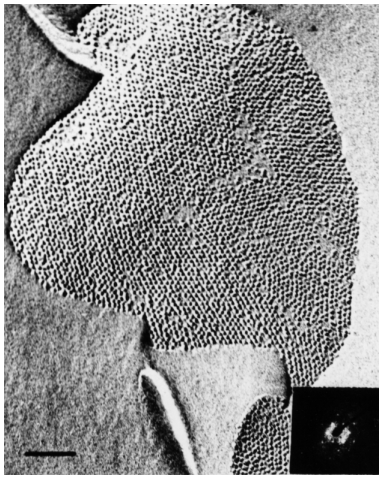


Figure 4: Proteins on a cell membrane

If, as in this case, we are able to identify the general appearance of the feature that we are looking for in the image, we can design a so-called *matched filter* which will produce a large response when that feature is encountered. In this case, we observe that the proteins appear to be illuminated from the upper left corner of the image, so a mask that has a positive response to its upper left and a negative response in its lower right will tend to enhance these features. Hence, a suitable convolution mask will be something like:

2	1	0
1	0	−1
0	−1	−2

(2)

As usual, the coefficients in the mask sum to zero so that it will produce a response of zero for a uniform region of the input image.

When a suitable matched filter is not obvious or the processing needs to be made automatic, we need an alternative strategy. One suitable technique is to identify and extract from the image a typical feature. This extracted region forms a *template* that we want to look for in the image. We can then centre the template over each pixel in the image in turn and work out the similarity between the template and image region — and we have already considered two ways of determining similarity when we looked at content-based image retrieval, namely simple differencing and correlation. Readers who have a knowledge of video coding will recognize that template matching forms part of the motion estimation techniques of the H.261 and MPEG standards.

8 Implementing convolution

It is not tricky to code up a routine that implements straightforward convolution: it is only a matter of multiplying pixels by their coefficients and adding up the result. The biggest problem is around the edges of the image, for it is not clear what should be done. There are several possible approaches, ranging from padding out the edges of the image with zero values (or, equivalently, ignoring mask pixels that 'fall off' the image), mirroring the image, and so on.

However, only one way of handling the edges agrees with the underlying Fourier theory (which takes too long to go into in this lecture course) and that is to wrap those mask coefficients that 'fall off' one edge of the image around to the other edge. Fortunately, all programming languages have a way of calculating the remainder when one number is divided by another and we can use this to work out subscripts; in C and Python, this is the '%' operator. With this, the routine that implements convolution in EVE is shown overleaf. You will see that this supports different types of convolution (mean, median *etc*.) and the similar morphological operators via minimum and maximum.

References

Davies, E. Roy (2012). *Machine Vision: Theory, Algorithms, Practicalities*. 4th edition. Morgan Kaufman.

```
def convolve (im, mask, statistic='sum'):
    ny, nx, nc = sizes (im)
    my, mx, mc = sizes (mask)
    yo = my // 2
    xo = mx // 2

    # Create an output image of the same size as the input.
    result = image (im)

    # We need a special case for the 'min' statistic to erase the mask
    # elements that are zero.
    nzeros = len ([x for x in mask.ravel() if x == 0])

    # Loop over the pixels in the image.  For each pixel position, multiply
    # the region around it with the mask, summing the elements and storing
    # that in the equivalent pixel of the output image.
    v = numpy.zeros ((my*mx*mc))
    vi = 0
    for yi in xrange (0, ny):
        for xi in xrange (0, nx):
            for ym in xrange (0, my):
                yy = (ym + yi - yo) % ny
                for xm in xrange (0, mx):
                    xx = (xm + xi - xo) % nx
                    v[vi] = im[yy,xx,0] * mask[ym,xm,0]
                    vi += 1
            if   statistic == 'sum':    ave = numpy.sum (v)
            elif statistic == 'mean':   ave = numpy.mean (v)
            elif statistic == 'max':    ave = numpy.max (v)
            elif statistic == 'min':    ave = numpy.min (v[nzeros:])
            elif statistic == 'median': ave = numpy.median (v)
            result[yi,xi,0] = ave
            vi = 0
    return result
```