Lecture 4 - Convolution

Gidon Rosalki

2025-11-9

1 Convolution

Convolution is a linear operator, functioning as the weighted sum of neighbours, applied *identically on all pixels*. An example is averaging a pixel with its 3×3 neighbourhood. Here is the generic convolution function

$$h(x, y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} f(i, j) \, g(x - i, y - j)$$

Where h returns the blurred pixels, the kernel is provided by the function f, and g returns the original pixels. So, the image g is blurred by the *kernel* f, giving image h. If we set f to be a 3×3 grid, where each value is $\frac{1}{9}$, then we get the uniform average blur effect.

We can also create a weighted blur, with the kernel:

$$f = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Blur is necessary *before* we sample (discussed in the Fourier lectures).

1.1 Different types of convolution

1.1.1 Blur

A blur replaces a pixel, with an average of its neighbours. We can also blur only in specific axes:

$$\begin{aligned} \text{Horizontal blur } &= \frac{1}{3} \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \\ \text{Vertical blur } &= \frac{1}{3} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \end{aligned}$$

1.1.2 Shift

The following kernel will *shift* the image to the left:

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

1.1.3 Edge

We can compute the difference between neighbours of a pixel (approximately the derivative) and find the vertical edges:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0.5 & 0 & -0.5 \\ 0 & 0 & 0 \end{bmatrix}$$

or the horizontal edges:

$$\begin{bmatrix} 0 & 0.5 & 0 \\ 0 & 0 & 0 \\ 0 & -0.5 & 0 \end{bmatrix}$$

An equivalent operation would be to shift the image, and subtract from the original.

1.1.4 CNNs

In IML we discussed Convolutional Neural Networks. These apply standard convolutions between layers, but there is a significant difference to the convolutions we discuss here. In CNNs, the weights of the kernels are learned, to give the best performance, but for us, we have standard kernels of fixed weights.

1.1.5 Boundary handling

How do we handle boundaries? At the edges / corners, then our kernel is touching pixels outside of the image. One approach is that used in Fourier, which is the *cyclic approach*. This is rarely used in practice, since generally pixels are similar to adjacent pixels, but the pixels across the left, and right, of the image, are likely to be very different, and so this is unhelpful in our case. We can also use 0 padding, where every index out of range is 0. This is commonly used in CNNs. Finally we have reflection, where

$$F\left(-a\right) = F\left(a\right)$$

This is our most common solution in image processing.

1.2 Continuity

Convolution is also defined as being continuous:

$$\left(f * g\right)\left(x\right) = \int_{-\infty}^{\infty} f\left(a\right) g\left(x-a\right) da$$

We want continuous convolution, because our physical world *is* continuous. Consider the response of sensors to colour. There is always a significant, and large amount of overlap between the wavelengths to which each colour responds.

What other courses use convolutions? IML used them for neural networks. They are also used in Algorithms for fast polynomial multiplication (FF). The complexity of a discrete convolution is $O\left(n^2\right)$, but can be improved to $O\left(n\log\left(n\right)\right)$.

Convolution in the spatial domain $\left(f\left(x,y\right), g\left(x,y\right)\right)$ is **equivalent** to *pointwise multiplication* in the frequency domain $\left(F\left(u,v\right), G\left(u,v\right)\right)$:

$$\Phi\left(f * g\right) = F \cdot G$$
$$\Phi\left(f \cdot g\right) = F * G$$

To perform convolution by using Fourier:

$$f * g = \Phi^{-1}\left(F \cdot G\right) = \Phi^{-1}\left(\Phi\left(f\right) \cdot \Phi\left(g\right)\right)$$

The FFT can thus massively reduce the complexity of convolution, from $O\left(n^2\right) \to O\left(n\log\left(n\right)\right)$. Consider blurring, if we blur an original image by performing a straight convolution, then we get $f * g\left(x,y\right)$ at $O\left(n^2\right)$, but if we take the Fourier transform, convolve that, and then take the inverse, then we get exactly the same result, but at $O\left(n\log\left(n\right)\right)$.

1.3 Properties of convolution

$$\text{Commutative: } f * g = g * f$$
$$\text{Associative: } f * \left(g * h\right) = \left(f * g\right) * h$$
$$\text{Distributive: } f * \left(g + h\right) = f * g + f * h$$

So as we can see, convolution is a linear operation over 1D vectors, and 2D images. Since it is a linear operation, therefore we can apply it as a matrix multiplication.

1.4 Noise

When smoothing / blurring, the sum of the weights in the kernel add to 1. This ensures that the average grey level after applying the kernel is the same.

Definition 1.1 (White noise)**.** *Noise where all the frequencies are the same, thus meaning that the FT is flat.*

Noise is random for every pixel, and does not depend on any other pixel. Blurring reduces noise, it smooths it all out. It also removes details from the image, but can overall make the image look better to the human eye.

We sometimes use the *median* of a neighbourhood to clean noise. Consider salt and pepper noise, which is the addition of many white and black blobs to an image. By taking a median, we can clean out most of that noise, with a minimal impact on the original quality, where a blur will simply blur the entire image.

1.5 Derivatives

Since images are not continuous, taking the derivative of an image is an interesting question. We are measuring the rate of change between adjacent pixels, turned into a discrete state.

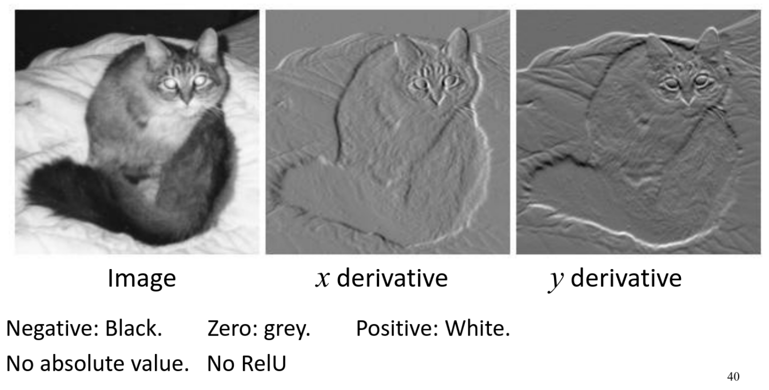

Figure 1: Derivative example

Due to its non continuous nature, we have many approximations of the derivative. A popular blur kernel is Sobel:

$$\frac{\partial f}{\partial x} = \frac{1}{8}\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

$$\frac{\partial f}{\partial y} = \frac{1}{8}\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

A good derivative filter is edge detection in one direction, and a **blur** in the orthogonal direction.

The *gradient* is the vecotor of x and y derivatives:

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)$$

The gradient has size:

$$|\nabla f| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

it also has direction:

$$\alpha = \tan^{-1}\left(\frac{\left(\frac{\partial f}{\partial x}\right)}{\left(\frac{\partial f}{\partial y}\right)}\right)$$

To find the second derivative, we apply the convolution again. The *Laplacian* is the sum of the partial derivatives:

$$(1\ \ -2\ 1) + \begin{pmatrix} 1 \\ -2 \\ 1 \end{pmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Or in its alternative form

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

The Laplacian describes the difference between a pixel, and its neighbourhood.

To put all this together, to find an edge, we take the *maximum* of the first derivative, which is also the point of where the second derivative crosses the x axis, also known as the point of *zero crossing.* The problem with this is that the second derivative is very noisy, which brings us back to the point earlier of blurring first. When we want to find edges, we should smooth the image first, to reduce the noise of the second derivative. We generally smooth with the 2D Gaussian.

We can also use these techniques to *sharpen* an image, by subtracting the second derivative, from the original function:

$$f(x) - f''(x)$$