Computer Vision – Algorithms and Deep Learning

Eyal Gur

August 20, 2024

Part I
Classical Computer Vision

An *image* is a tensor of numbers, where each channel is a 2D array that represents the intensity of the pixels within that channel. A common pixel format is the 8-bit integer (byte image), giving a range of values from 0 (which is typically black) to 255 (typically white), and this range can be scaled to the range $[0, 1]$. Consequently, a grayscale image can be represented as a single 2D channel, while a color image can be represented as a 3D tensor, with each channel corresponding to a grayscale 2D image. The three channels in a color image – red, green, and blue (RGB) – can be combined and colorized to produce the final visual representation.

Since an image can be represented as a tensor, it can be viewed as a function F that maps a domain $D \subset \mathbb{R}^2$ (representing the pixel locations) to intensity values:

$$F(D) = (F_1(D), F_1(D), \ldots, F_C(D)),$$

where C is the number of channels, and F_i is the intensity map of channel i.

1 Image Filtering

There are two primary types of transformations that can be applied to an image: *filtering*, which involves transforming the pixel values (i.e., altering the range of F), and *warping*, which involves transforming the pixel locations (i.e., altering the domain D of F). Filtering is further categorized into *point processing* operations and *neighborhood processing* operations.

1.1 Point Processing

Point processing operations are transformations applied to each pixel of an image independently, without considering its neighboring pixels. For instance, given an image $\mathbf{x} \in \mathbb{R}^{h \times w \times m}$ with pixel values in the range $[0, 1]$, the element-wise operation $\mathbf{x} - 0.5$ uniformly darkens the image (with values outside the $[0, 1]$ range being clipped to the nearest boundary, either

0 or 1). Similarly, the operation $\mathbf{x} + 0.5$ uniformly brightens the image. The operation $1 - \mathbf{x}$ inverts the image colors.

The transformation $2\mathbf{x}$ increases the contrast of the image contrast by brightening all pixels, with the originally brighter pixels becoming even brighter relative to the darker ones. Conversely, the transformation $\mathbf{x}/2$ decreases contrast by darkening all pixels, making the originally brighter pixels less distinct from the darker ones.

Non-linear point operations can also be applied, such as $\mathbf{x}^2$, which darkens the image while leaving the white and black pixels unchanged. Another example is $\sqrt{\mathbf{x}}$, which brightens the image, again with white and black pixels remaining unaffected.

Figure 1 illustrates all of the discussed point operations and are implemented in the Google Colab notebook. Of course, there are many other types of point processing operations, but they all follow similar principles.

Figure 1: Point processing operations: applying linear or non-linear point operations for image filtering.

1.2 Linear Shift-Invariant Image Filtering

A linear shift-invariant filtering involves applying operations that take into account the neighborhood of each pixel, where each pixel is replaced by a linear combination of its neighboring pixels and itself.

1.2.1 Convolution In Image Processing

The Convolution Operation. An operation that combines two functions $f\colon \mathbb{R}^d \to \mathbb{C}$ and $g\colon \mathbb{R}^d \to \mathbb{C}$ to produce a third function $f * g\colon \mathbb{R}^d \to \mathbb{C}$. This operation is defined as the integral (or summation in the discrete case) of the product of the two functions after one function is reflected about the y-axis and then shifted (effectively flipping its domain). Mathematically, given two functions $f, g\colon \mathbb{R}^d \to \mathbb{C}$, their convolution is expressed as:

$$(f * g)(\mathbf{x}) = \int_{\mathbb{R}^d} f(\mathbf{y})\, g(\mathbf{x} - \mathbf{y})\, d\mathbf{y} = \int_{\mathbb{R}^d} f(\mathbf{x} - \mathbf{y})\, g(\mathbf{y})\, d\mathbf{y}.$$

A particularly interesting case is the two-dimensional discrete scenario, where the two functions being convolved can be represented as matrices. Given two matrices $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{m \times n}$, their convolution, according to the previous definition, is expressed as:

$$\mathbf{X} * \mathbf{Y} = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \mathbf{X}_{i,j} \mathbf{Y}_{m-1-i,n-1-j}.$$

Notice that the convolution operation is commutative. Additionally, this operation flips the order of the matrix $\mathbf{Y}$ and reverses its elements. If the order is not reversed, the operation is known as *correlation*.

In the special case where one of the matrices is symmetric around its vertical and horizontal axes, a situation known as *symmetric convolution*, the convolution simplifies to their Frobenius inner product.

Using the definition of the convolution operation, we can introduce the *kernel convolution* operation, often simply referred to as convolution or filtering in image processing. This operation can be seen as a generalization of the mathematical convolution when the two matrices involved do not have the same shape. The operation involves taking a kernel matrix and convolving it with portions of the image that match the shape of the kernel, then shifting the kernel to convolve with other parts of the image.

Let $\mathbf{I} \in \mathbb{R}^{H \times W}$ be a 2D image (or a single channel of a multi-channel image), and let $\mathbf{K} \in \mathbb{R}^{k_H \times k_W}$ be a kernel matrix. Denote the *strides* along the height and width by $s_H, s_W \in \mathbb{N}$. The kernel convolution of $\mathbf{I}$ with the kernel $\mathbf{K}$ and stride (s_H, s_W) is defined as:

$$(\mathbf{K} * \mathbf{I})_{p,q} = \sum_{i=0}^{k_H-1} \sum_{j=0}^{k_W-1} \mathbf{K}_{ij} \mathbf{I}_{ps_H-1-i,qs_W-1-j} = \mathbf{K} * \mathbf{I}[ps_H - k_K : ps_H, qs_W - k_W : qs_W],$$

where we have applied the previously introduced definition of (mathematical) convolution. This operation is repeated to each channel. Commonly, the kernel a is square matrix with an odd dimension $2k + 1$, with the convention that the $(0, 0)$ point is its *center* element. In this case, the kernel convolution can be written conveniently as

$$(\mathbf{K} * \mathbf{I})_{p,q} = \sum_{i,j=-k}^{k} \mathbf{K}_{ij} \mathbf{I}_{ps_H-i,qs_W-j} = \mathbf{K} * \mathbf{I}[ps_H - k : ps_H + k, qs_W - k : qs_W + k].$$

The resulting convolved image $\mathbf{K} * \mathbf{I}$ (or single channel), referred to as the *filtered image*, has dimensions given by:

$$\left(\left\lfloor \frac{H - k_H}{s_H} \right\rfloor + 1 \right) \times \left(\left\lfloor \frac{W - k_W}{s_W} \right\rfloor + 1 \right).$$

Edge Handling. Kernel convolution often requires accessing pixels beyond the image boundaries. There are three primary methods to address this:

1. *Skipping*: Any pixel in the output image that requires values from outside the boundaries is skipped. This involves adjusting the kernel's position so that it only operates within the image area. As a result, the output image may be smaller, with cropped edges. This method is commonly used in machine learning.

2. *Kernel Cropping*: Any part of the kernel extending beyond the image boundaries is ignored during convolution, and the remaining values are normalized to compensate.

3. *Padding*: A widely used method where the image borders are extended to provide the necessary values for convolution. There are several types of padding:

   - Wrapping: the image is tiled, and values are taken from the opposite edge.
   - Mirroring: the image is mirrored at the edges, so accessing pixels beyond the edge reads from within the image.
   - Constant Padding: the added pixels have a constant value, usually zero.

   When padding by p_H pixels along the height and p_W pixels along the width, the output size of the convolved image is given by:

$$\left(\left\lfloor \frac{H - k_H + 2p_H}{s_H} \right\rfloor + 1 \right) \times \left(\left\lfloor \frac{W - k_W + 2p_W}{s_W} \right\rfloor + 1 \right).$$

Box Filtering. A simple linear filter where each pixel in the output image is computed as the average of its neighboring pixels in the input image. An $n \times n$ kernel matrix for this filter is $\textsc{ones}(n, n)/n^2$, and the strides are 1.

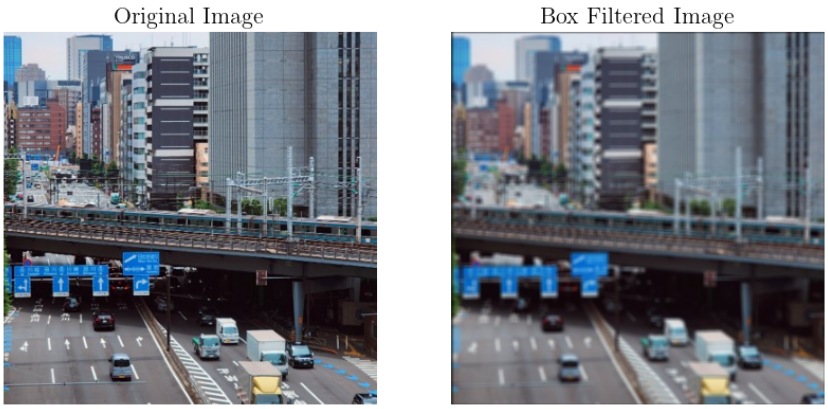
Figure 2 displays the result of applying a box filter, as implemented in the Google Colab notebook. This implementation automatically calculates the necessary constant padding to ensure that the output image retains the same dimensions as the input. The convolution is performed using `numpy`, although built-in functions are also available for this purpose.

1.2.2 Separable Filters

If the kernel matrix has dimensions $k_H \times k_W$ and the image has dimensions $H \times W \times C$ (where C is the number of channels), the computational cost of the kernel convolution is $k_H k_W HWC$. This is because the kernel requires $k_H k_W$ multiplications for each pixel.

This computational cost can be reduced if the kernel matrix is *separable*, meaning it can be expressed as the outer product of two vectors. For example, a box filter kernel of size $n \times n$ can be written as $\mathbf{1}_n \mathbf{1}_n^T$.

Figure 2: Box filtering by applying a 5×5 kernel matrix.



Suppose the kernel matrix $\mathbf{K} \in \mathbb{R}^{k_H \times k_W}$ can be expressed as $\mathbf{K} = \mathbf{k}\mathbf{l}^T$ where $\mathbf{k} \in \mathbb{R}^{k_H}$ and $\mathbf{l} \in \mathbb{R}^{k_W}$. In that case, it can be shown that $\mathbf{K} * \mathbf{I} = \mathbf{k} * \left(\mathbf{l}^T * \mathbf{I}\right)$, where $\mathbf{I}$ is a 2D image (or a single channel). Here, the 1D convolutions require a total of $k_H + k_W$ mmultiplications per pixel, resulting in a reduced overall cost of $(k_H + k_W)\, HWC$.

Gaussian Filter. The box filter computes the average pixel value at each pixel, treating all neighboring pixels equally. In contrast, the Gaussian filter averages pixel values with their neighbors, but it gives more weight to pixels closer to the center of the kernel, resulting in a more natural and realistic blur.

The x coordinate of the 1D Gaussian kernel is defined as:

$$\mathbf{G}^1(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{x^2}{2\sigma^2}\right),$$

where 0 is the center element of the kernel, and $\sigma > 0$ is the standard deviation that controls the level of smoothing. The (x, y) coordinates of the 2D Gaussian kernel are defined as:

$$\mathbf{G}(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right) = \mathbf{G}^1(x) \cdot \mathbf{G}^1(y), \tag{1}$$

where $(0, 0)$ is the center element of the kernel. The equality above also demonstrates that this kernel is separable, by treating the first multiplicand as a column vector and the second as a row vector. In other words, the 2D Gaussian kernel is outer product of two 1D Gaussian kernels, that is $\mathbf{G} = \mathbf{G}^1 \left(\mathbf{G}^1\right)^T$. For instance, an approximation of a 3×3 Gaussian kernel matrix is:

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

Theoretically, the Gaussian kernel is infinite, but in practice it is truncated to a certain radius from the center. Typically, the kernel values are considered negligible beyond about three standard deviations from the mean, allowing the kernel to be truncated at that point.

Figure 3 shows the result of applying a Gaussian filter, as implemented in the Google Colab notebook. This implementation accepts the kernel size and standard deviation as inputs and calculates the 1D vector for the convolution by leveraging the separability property for improved performance. The convolution is performed using `numpy`, although built-in functions are also available for this task.

Figure 3: Gaussian filtering by applying a 5×5 kernel matrix, with two different standard deviations determining the level of blurring.

Sharpening Filter. One example of such a filter in the 3×3 case is constructed as follows:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix},$$

which is also a separable filter (though there are other variants of the sharpening filter). With a stride of 1, this filter first amplifies the pixel by doubling its value, and then subtracts the average of the pixel values of its neighbors (subtracting a box filter). In regions with constant neighborhood values ("flat" areas), where subtracting the average from the doubled value results in no change, the filter has no effect. However, at edges, this process increases the pixel values in the output image, resulting in sharpening.

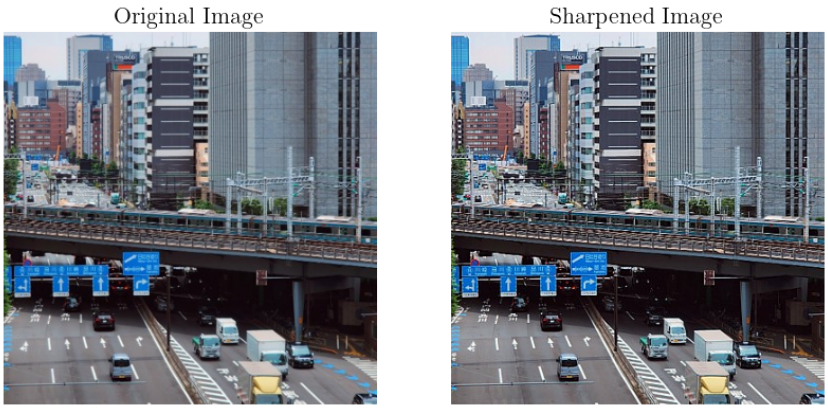
Figure 4 displays the result of applying a 3×3 sharpening filter, as implemented in the Google Colab notebook. This implementation utilizes the `cv2` package to perform the convolution using the `cv2.filter2D` function (we can also apply two 1D convolutions as discussed above).

1.2.3 Discrete Differentiation Filters

Discrete differentiation allows for detecting changes in pixel intensity, which correspond to edges and other important features in an image. For instance, consider the vector image $\mathbf{v} = (0, 0, 0, 1, 1, 1, 1, 0, 0, 0)$ and the kernel vector $\mathbf{k} = [1, -1]$. Using a stride of 1 and applying zero padding of size 1 (so that all elements of $\mathbf{v}$ are multiplied by the center of the kernel, which we define it to be the element 1), the convolution yields:

$$\mathbf{k} * \mathbf{v} = (0, 0, 0, 1, 0, 0, 0, -1, 0, 0, 0)\,.$$

Figure 4: Sharpening filter by applying a 3×3 kernel matrix.



This result demonstrates that the convolved output reflects the rate of change in pixel values within $\mathbf{v}$ when imagining moving from left to right through the input data. Similarly, taking the kernel $[-1, 1]$ reflects changes when moving from right to left (remember that during convolution, the multiplication is computed in a reversed order).

We can show that the discrete differentiation kernel $\mathbf{k} = [1, -1]$ is derived from the definition of the derivative. Recall the definition:

$$\frac{\partial f}{\partial x}(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h},$$

where $f \colon \mathbb{R} \to \mathbb{R}$ is some continuous (and differentiable) function. In the discrete case, for a discrete signal $\mathbf{v}$, the discrete analogue is obtained by removing the limit:

$$\frac{\partial \mathbf{v}}{\partial x}(x) = \frac{\mathbf{v}(x+h) - \mathbf{v}(x)}{h}, \tag{2}$$

where $h \in \mathbb{R}$ is small enough, and $\mathbf{v}(x+h)$ represents the value of $\mathbf{v}$ at the coordinate $x+h$. An approximation of $\partial \mathbf{v}/\partial x$ can be obtained by choosing specific values of h. For example, substituting $h = 1$ in (2) we get:

$$\frac{\partial \mathbf{v}}{\partial x}(x) \approx \mathbf{v}(x+1) - \mathbf{v}(x).$$

Iterating through the coordinates of $\mathbf{v}$ with a stride of 1 and zero padding, we indeed obtain:

$$\frac{\partial \mathbf{v}}{\partial x} = (0, 0, 0, 1, 0, 0, 0, -1, 0, 0, 0) = \mathbf{k} * \mathbf{v}. \tag{3}$$

Notice that $\mathbf{k} = [1, -1]$ represents the coefficient of $\mathbf{v}(x)$ and $\mathbf{v}(x+1)$ in the approximation (3), with -1 corresponding to $\mathbf{v}(x)$ and 1 corresponding to $\mathbf{v}(x+1)$, but in a reversed order. Since the coefficient of $\mathbf{v}(x-1)$ is 0, then the kernel matrix $[1, -1, 0]$ is equivalent to $\mathbf{k} = [1, -1]$, up to padding. Similarly, the kernel $[-1, 1]$ (right to left derivative) is obtained by looking at the equivalent definition of (2):

$$\frac{\partial \mathbf{v}}{\partial x}(x) = \frac{\mathbf{v}(x-h) - \mathbf{v}(x)}{h},$$

and plugging $h = -1$.

We can derive other discrete differentiation kernels using the same methodology. For instance, consider the alternative definition (2):

$$\frac{\partial \mathbf{v}}{\partial x}(x) = \frac{\mathbf{v}(x + 0.5h) - \mathbf{v}(x - 0.5h)}{h}, \tag{4}$$

which leads to the kernel $[0.5, 0, -0.5]$ by plugging $h = 2$. This suggests that the kernels $[1, 0, -1]$ and $[-1, 0, 1]$ are discrete approximations of twice the derivative.

The Prewitt Filter. A type of discrete differentiation and separable filter is defined for vertical and horizontal directions. Utilizing the discrete approximation matrix $[1, 0, -1]$, which represents twice the derivative as derived above, we define the 2D kernels of the Prewitt filter as follows:

$$\mathbf{K}_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \mathbf{K}_y = \mathbf{K}_x^T.$$

This means that $\mathbf{K}_x$ first approximates the derivative along the rows, and then increases the pixel intensity of the derivatives along the columns. It is evident that if, for example, the vertical kernel matrix $\mathbf{K}_x$ returns a positive pixel value after convolution, it indicates a transition from dark to light when moving from right to left. Conversely, if it returns a negative value, it signifies a transition from light to dark.

The Sobel Filter. Like the Prewitt filter, this is a discrete differentiation and separable filter that measures intensity transitions when moving from left to right or from top to bottom. However, it detects more complex transitions, might leading to a more cluttered convolved image. It is defined for vertical and horizontal directions as follows:

$$\mathbf{K}_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} \quad \text{and} \quad \mathbf{K}_y = \mathbf{K}_x^T.$$

Other Discrete Filters. The Scharr Filter is defined as

$$\mathbf{K}_x = \begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix} \quad \text{and} \quad \mathbf{K}_y = \mathbf{K}_x^T,$$

and it also measured light transitions from left to right or from top to bottom. The Roberts Filter is defined as

$$\mathbf{K}_x = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{K}_y = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix},$$

and it measures light transition along diagonals.

1.3 Edge Detection

Image edges are sharp discontinuities or sudden changes in pixel intensity (rate of change). These discontinuities can be detected using discrete approximations of the first-order or second-order derivatives of the pixels, representing the rate of change in their intensity. These derivatives can are visualized, for example using a *gradient map*, thereby enabling edge detection.

1.3.1 Calculating First-Order Gradient Maps

To compute such maps, follow these steps:

1. Select a discrete differentiation filter for the vertical and horizontal directions, $\mathbf{K}_x$ and $\mathbf{K}_y$ (although other directions can also be used).

2. Convolve the image $\mathbf{I} \in \mathbb{R}^{H \times W}$ with these kernels to produce two filtered images:

$$\frac{\partial \mathbf{I}}{\partial x} \approx \mathbf{K}_x * \mathbf{I} \in \mathbb{R}^{H \times W} \quad \text{and} \quad \frac{\partial \mathbf{I}}{\partial y} \approx \mathbf{K}_y * \mathbf{I} \in \mathbb{R}^{H \times W}.$$

3. Calculate the gradient magnitude (the gradient map)

$$\|\nabla \mathbf{I}\| \equiv \sqrt{\left(\frac{\partial \mathbf{I}}{\partial x}\right)^2 + \left(\frac{\partial \mathbf{I}}{\partial y}\right)^2} \in \mathbb{R}^{H \times W},$$

   where the operations are performed element-wise.

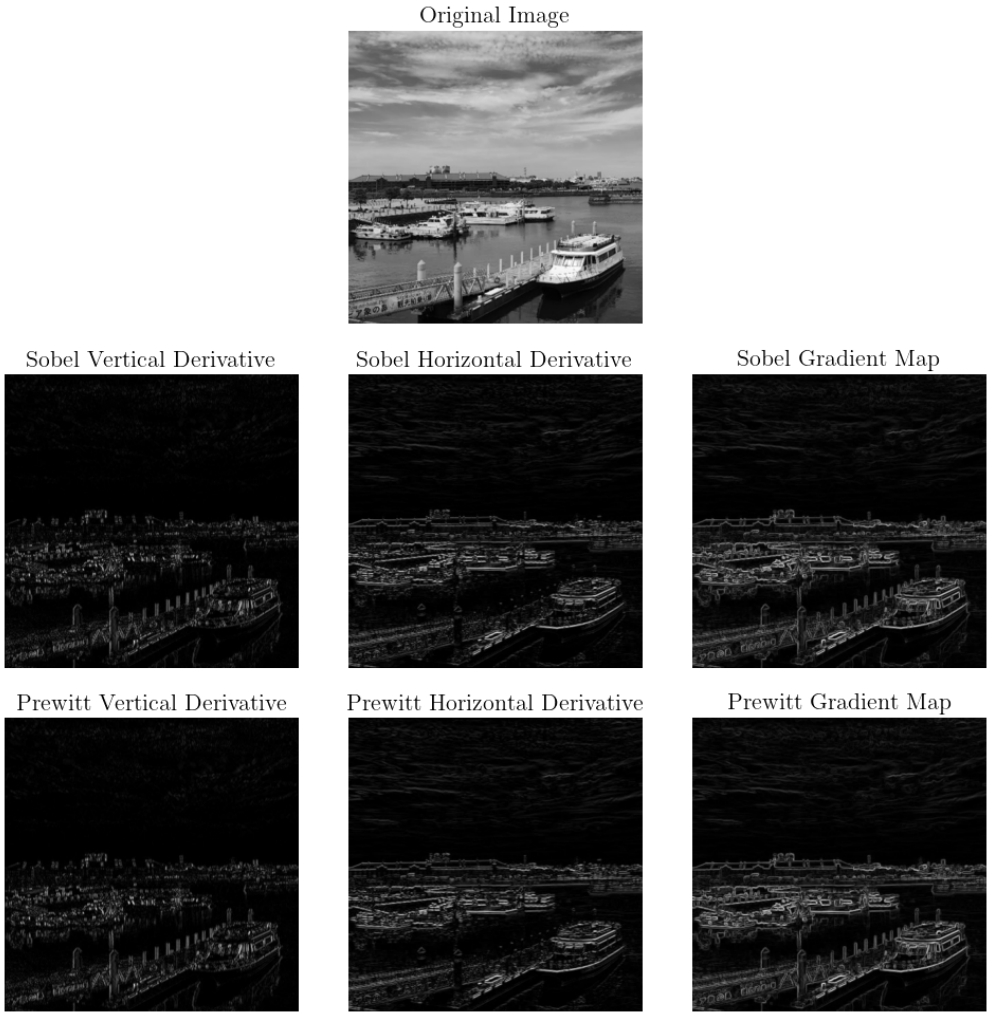
Figure 5 shows the result of applying the Sobel and Prewitt filters for image edge detection, as implemented in the Google Colab notebook. In this implementation, the filters are applied to generate the filtered images (the derivatives), which are then used to create the gradient maps for edge detection. It is important to note that the filtered images, unlike the gradient maps, contain both positive and negative values. However, when plotting the vertical or horizontal edges (the derivatives), we take the absolute values of the filtered images, since the sign of the derivative (positive or negative) is not relevant for edge detection, as it only indicates whether the edge transitions from light to dark or from dark to light. Additionally, the original image used is in grayscale, as edge detection focuses on transitions in pixel intensity rather than on any specific color channel.

1.3.2 Derivative of Gaussian (DoG)

A problem occurs when attempting to detect edges in a noisy image. In this scenario, the changes in intensity might not indicate actual edges but rather discontinuities caused by the noise. Consequently, plotting the gradient map could highlight unnecessary noise that does not necessarily correspond to true edges in the image. For instance, see Figure 6 with its implementation in the Google Colab notebook, that shows the output Sobel gradient map of a noisy input image.

To address this issue, we can first apply a Gaussian filter $\mathbf{G}$ to the noisy image $\mathbf{I}$, and calculate the gradient map of this filtered image instead using some discrete differentiation

Figure 5: Gradient maps for edge detection: Displaying the gradient maps for edge detection created using the Sobel and Prewitt filters. The absolute values of the vertical and horizontal derivatives (i.e., the filtered images from each kernel matrix) are also displayed.



kernels $\mathbf{K}_x$ and $\mathbf{K}_y$. The blurring effect of the Gaussian filter helps to smooth out random noise, allowing us to focus on true changes in pixel intensity.

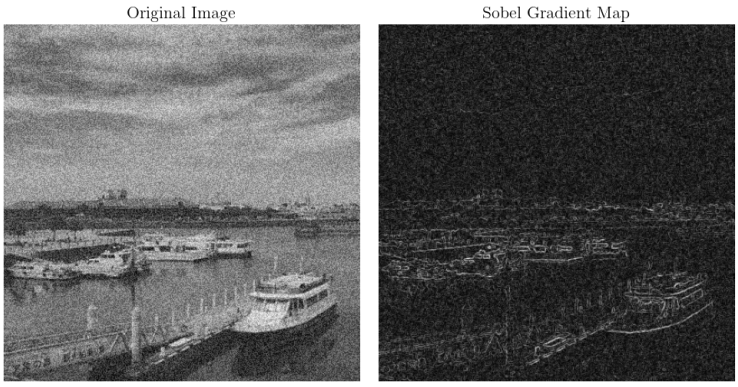
Let $\mathbf{G}$ the Gaussain filter, $\mathbf{K}_x$ and $\mathbf{K}_y$ be discrete differentiation kernels, and $\mathbf{I}$ the noisy image. Mathematically, the resulting operations are:

$$\frac{\partial\left(\mathbf{G} * \mathbf{I}\right)}{\partial x} \approx \mathbf{K}_x * \left(\mathbf{G} * \mathbf{I}\right) = \left(\mathbf{K}_x * \mathbf{G}\right) * \mathbf{I} \approx \frac{\partial \mathbf{G}}{\partial x} * \mathbf{I}, \tag{5}$$

and similarly for $\mathbf{K}_y$. This implies that instead of first convolving $\mathbf{G}$ with $\mathbf{I}$ and then convolving the result with $\mathbf{K}_x$, we can convolve $\partial\mathbf{G}/\partial x$ (which is a constant matrix independent of the choice of discrete differentiation kernel) directly with $\mathbf{I}$.

The matrix $\partial\mathbf{G}/\partial x$ is called the *Derivative of Gaussian* (DoG) kernel matrix, and it automatically blurs (smooths) the image, and calculates the derivatives. Mathematically,

Figure 6: Sobel gradient generated by a noisy input image.



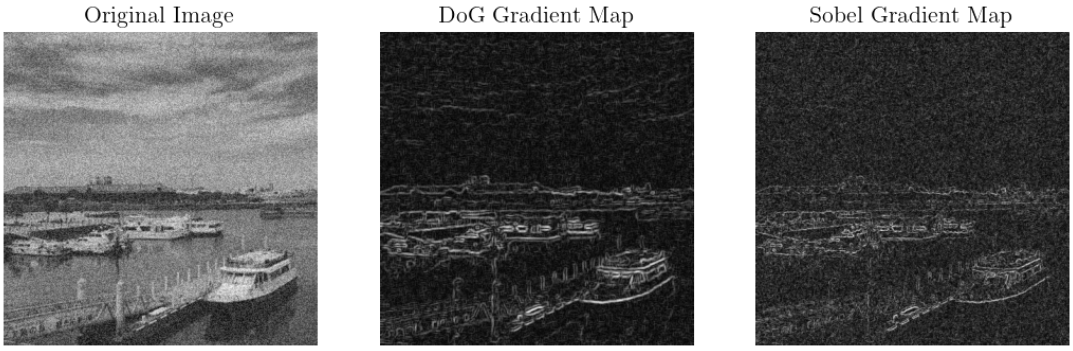
following (1), the 2D DoGs are

$$\frac{\partial \mathbf{G}}{\partial x}(x,y) = -\frac{x}{\sigma^2}\mathbf{G}(x,y) \quad \text{and} \quad \frac{\partial \mathbf{G}}{\partial y}(x,y) = -\frac{y}{\sigma^2}\mathbf{G}(x,y)\,. \tag{6}$$

Recall that we began this discussion with differentiation kernels $\mathbf{K}_x$ and $\mathbf{K}_y$ in (5). It can indeed be shown that for discrete differentiation kernel matrices, such as the Sobel kernels and others, the relationship $\mathbf{K}_x * \mathbf{G} \approx \partial\mathbf{G}/\partial x$ (and similarly for $\mathbf{K}_y$) holds true. These can act as an approximation of the DoG matrices.

Figure 7 displays the gradient map for edge detection of a noisy input image, generated using the DoG filter and the Sobel filter for comparison, as implemented in the Google Colab notebook. It is evident that the DoG filter yields significantly better results. The DoG filter is implemented by performing two 1D convolutions in both the x and y directions for faster results. This can be efficiently computed as follows: for the x direction, the process involves first convolving with the 1D DoG kernel from (6) as a column vector, followed by convolving with the 1D Gaussian kernel (1) treated as a row vector. For the y direction, we first convolve with the 1D Gaussian kernel as a column vector, and then convolve with the 1D DoG kernel treated as a row vector.

Figure 7: Sobel gradient generated by a noisy input image.



1.3.3 Laplacian Filters

When using first-order filters (such as Sobel, Prewitt, or DoG) for edge detection, the edges are highlighted in the resulting gradient map, but the exact coordinates of the edges are not precisely defined. Second-order filters allow for more accurate localization of the edges.

We begin by introducing *discrete second-order differentiation filters*, which are similar to the first-order filters discussed in Section 1.2.3. These filters are linear shift-invariant and can be derived in a similar manner to the first-order case. A common approximation of the second-order derivative of a twice-differentiable function $f\colon \mathbb{R} \to \mathbb{R}$ is given by:

$$\frac{\partial^2 f}{\partial x^2} \approx \frac{f\left(x+h\right) - 2f\left(x\right) + f\left(x-h\right)}{h^2}, \tag{7}$$

though other approximations also exist. Following a similar derivation as in Section 1.2.3 for first-order differentiation of discrete signals, we arrive at the discrete second derivative approximation kernel $\mathbf{k}^2 = [1, -2, 1]$.

To illustrate this, consider the vector image $\mathbf{v} = (0,0,0,1,1,1,1,0,0,0)$ from Section 1.2.3 with the discrete differentiation kernel vector $\mathbf{k} = [1, -1]$, where the convolution yielded:

$$\mathbf{k} * \mathbf{v} = (0,0,0,1,0,0,0,-1,0,0,0)\,.$$

Now, applying the kernel vector $\mathbf{k}$ to the convolved data again, with a stride of 1 and zero padding of size 1, we obtain:

$$\mathbf{k} * (\mathbf{k} * \mathbf{v}) = (0,0,0,1,-1,0,0,0,-1,1,0,0,0)\,,$$

and it becomes clear that $\mathbf{k} * (\mathbf{k} * \mathbf{v}) = \mathbf{k}^2 * \mathbf{v}$, , which is a discrete approximation of the second-order derivative of $\mathbf{v}$, when imagining moving along $\mathbf{v}$ from right to left.

More On Discrete Approximations. There are many possible generalizations of this concept. For instance, with first-order filters, we approximated the derivatives along the x and y directions and then combined these results in a gradient map by summing their squares and taking the square root. However, it is also possible to combine these results differently – such as by simply adding the two derivatives together using the kernel matrix $\mathbf{K}_x + \mathbf{K}_y$. Since this kernel inherently accounts for both directions, constructing a gradient map is unnecessary (recall that the gradient map was a method for combining results from different directions into a single 2D representation). To better illustrate this, consider a 2D image $\mathbf{I}$. Following (4), we have:

$$\frac{\partial \mathbf{I}}{\partial x}(x,y) + \frac{\partial \mathbf{I}}{\partial y}(x,y) \approx \frac{\mathbf{I}\left(x+0.5h,0\right) - \mathbf{I}\left(x-0.5h,0\right) + \mathbf{I}\left(0,y+0.5h\right) - \mathbf{I}\left(0,y-0.5h\right)}{h}.$$

Plugging $h = 2$ (as done in the derivations in Section 1.2.3), we obtain the discrete approximation first-order kernel matrix:

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

Similarly, we can also take the sum of the derivatives along the x and y axes, together with the sum of directional derivatives along the two directions $(1, \pm 1)$, resulting with the kernel

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

which is also the kernel obtained by summing the two Sobel kernels (for the two directions), up to a scalar multiplication.

Deriving Laplacian Kernels. Second-order filters are filters that utilize this concept of approximations but with second-order derivatives of discrete signals. Specifically, Laplacian filters approximate the sum of the unmixed second-order derivatives, known as the *Laplacian*:

$$\frac{\partial^2 \mathbf{I}}{\partial x^2}(x, y) + \frac{\partial^2 \mathbf{I}}{\partial y^2}(x, y) \tag{8}$$

Following (7), a second-order approximation kernel matrix corresponding to (8) for a 2D image is:

$$\mathbf{K}^2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix},$$

known as the *Five-Point Stencil* filter, as it consists of a grid including the central point and its four neighbors. An alternative second-order approximation of the sum in (8) involves using all neighboring points, hence it is called the *Nine-Point Stencil* filter, defined as:

$$\mathbf{K}^2 = \begin{bmatrix} 0.25 & 0.5 & 0.25 \\ 0.5 & -3 & 0.5 \\ 0.25 & 0.5 & 0.25 \end{bmatrix}$$

Figure 8 shows the two Laplacian filters alongside the gradient map of the Sobel filter as implemented in the Google Colab notebook. It is clear that while the first-order Sobel gradient map highlights the edges, the second-order Laplacian filters have *zero crossings* at the edges, enabling more precise edge localization, though this method can be less convenient. Since the output of the Laplacian filter can be negative, and we only care about the absolute rate of change, we plot their map of absolute values (as we also did for the Sobel and Prewitt directional maps).

1.3.4 Laplacian of Gaussain (LoG)

Similarly to the first-order DoG discussed in Section 1.3.2, for noisy images, we can first apply a Gaussian filter for smoothing, and then use second-order derivative kernels for zero-crossing edge detection. Like the derivation of the DoG filter, this process involves taking the second-order derivative of the Gaussian filter and convolving it with the noisy image.

The second-order derivative of the Gaussian filter is known as the Laplacian of Gaussian (LoG), and it is defined for a 2D Gaussian kernel as follows:

$$\frac{\partial^2 \mathbf{G}}{\partial x^2}(x, y) + \frac{\partial^2 \mathbf{G}}{\partial y^2}(x, y) = \frac{x^2 - \sigma^2}{\sigma^4}\mathbf{G}(x, y) + \frac{y^2 - \sigma^2}{\sigma^4}\mathbf{G}(x, y) = \frac{x^2 + y^2 - \sigma^2}{\sigma^4}\mathbf{G}(x, y).$$

Figure 8: Two variations of the second-order Laplacian filter, compared with the first-order Sobel filter for edge detection.

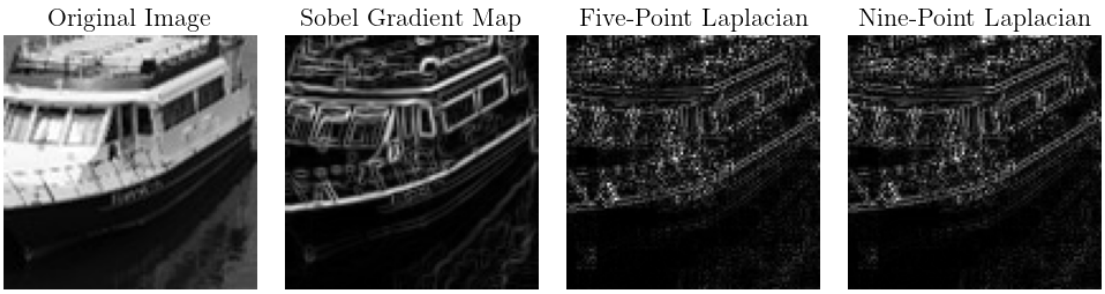
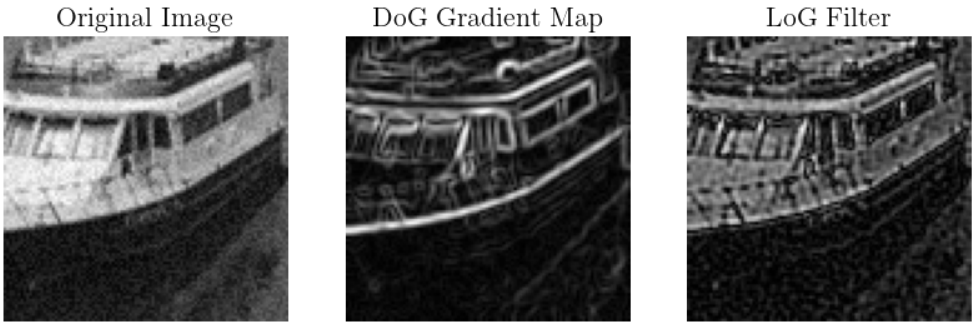


Figure 9 shows the results of applying the LoG filter. Since the LoG filter can produce negative values, the absolute values are plotted, as we are primarily interested in the magnitude of the change. The figure compares the LoG output to the DoG gradient map, and it is clear that the LoG filter exhibits zero-crossings at the edges. The implementation in the Google Colab notebook performs the LoG filtering using two 1D convolutions, as discussed in the DoG implementation in Section 1.3.2.

Figure 9: The absolute values of the LoG filter (zero crossings) and the gradient map produced by the DoG filter when applied to a noisy image.



2 Image Pyramids and Frequency Domains

In signal processing, *sampling* refers to the conversion of a continuous-time signal into a discrete-time signal. A *sample* is the value of the signal at a specific point in time. For time-varying functions, let $s(t)$ represent a continuous signal that is sampled every T seconds, known as the *sampling interval*. The resulting sampled function is expressed as the sequence $s(nT)$ for $n \in \mathbb{N}$. The *sampling frequency* f_s is the average number of samples taken per second, given by $f_s = 1/T$, typically measured in samples per second, or *hertz*. For instance, a sampling rate of 48 kHz corresponds to 48000 samples per second.

Undersampling occurs when we do not collect enough samples from the continuous signal, leading to a loss of information about the original signal. As a consequence, undersampling

can cause the signal to be mistaken for a lower-frequency one, a phenomenon known as *aliasing*. There are two primary strategies to address aliasing: *oversampling* the signal, which incurs a higher sampling cost, and *smoothing* the signal by filtering out details that cause aliasing. While smoothing results in some loss of information, it is preferable to the distortions introduced by aliasing.

In image processing, the term *frequency* describes the rate at which pixel values change across the image. Specifically:

- *Low-frequency components* correspond to slow or gradual changes in pixel values, such as smooth gradients or areas of uniform color.
- *High-frequency components* are associated with rapid changes in pixel values, typically found in areas with sharp edges or detailed patterns. For example, at a sharp edge in an image, where a black area meets a white area, the pixel intensity changes abruptly over a few pixels, representing a high-frequency component.

Figure 10 (as implemented in the Google Colab notebook) illustrates the impact of aliasing during downsampling and the use of smoothing as an anti-aliasing technique. Each downsampling step removes every other row and column, which leads to an image that appears blocky and pixelated due to aliasing. These artifacts arise because high-frequency details like sharp edges are sampled too sparsely, causing visual distortions and misinterpretation as lower frequencies. Conversely, when downsampling is combined with smoothing (such as using a Gaussian blur filter), many fine details are lost, but the severity of aliasing artifacts is reduced.

By reducing the intensity of high-frequency components, blurring ensures that these components do not get misrepresented (aliased) when the image is downsampled. The Gaussian blur acts as a *low-pass filter*, allowing only lower-frequency components (which can be more accurately sampled) to pass through.

Figure 10: Downsampling an image by removing every other row and column at each step. The top row shows the results without anti-aliasing, while the bottom row includes a Gaussian blur filter serving as a low-pass filter for anti-aliasing.