**Kernels, Convolution, Filtering**

Kernels in image processing

Image processing or “image filtering” refers to modifying an image. It’s a key topic in image editing and computer vision, where it may be used to reduce noise or enhance certain features, among other things. The basic unit of image filtering is the kernel, which is the topic of today’s post. Before we get into kernels proper, however, let’s first ensure that we have a good understanding of what an image actually is.

What is an image?

Consider the following photograph of a pastry I took in Amsterdam a few years ago (if I recall correctly, it was delicious):

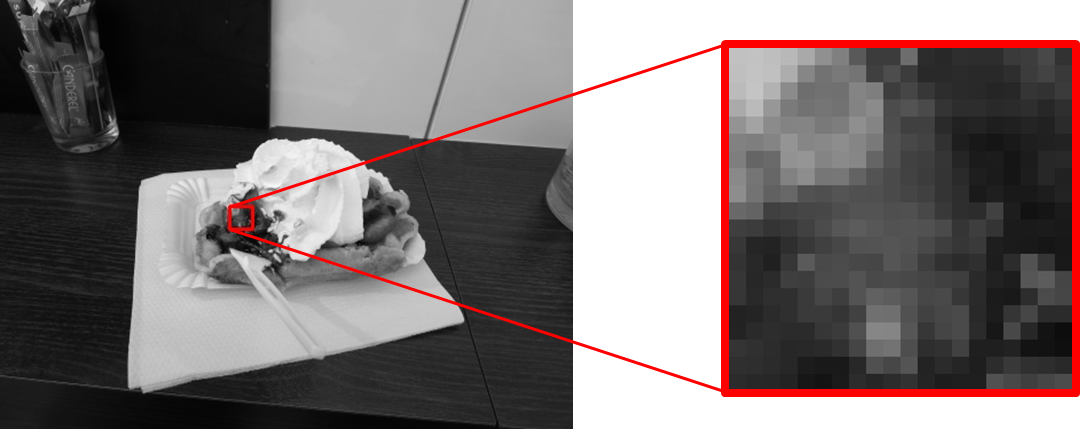


A digital image consists of pixels. In a color image, like the one above, each pixel is usually described in terms of the RGB color space. In other words, each pixel is represented by three channels: red, green, and blue. The bit depth, or color depth, of an image determines the range of values each channel can take on. In an 8-bit image, each channel can take on 28 = 256 values, from 0 to 255. 0 indicates that the intensity of a particular color is zero (black), and 255 indicates that the intensity of a particular color is maximum. So, for example, [R=0, G=0, B=0] would be black, [R=255, G=0, B=0] would be bright red, [R=0, G=0, B=255] would be bright blue, [R=127, G=0, B=127] would be an intermediate shade of purple, and so on.

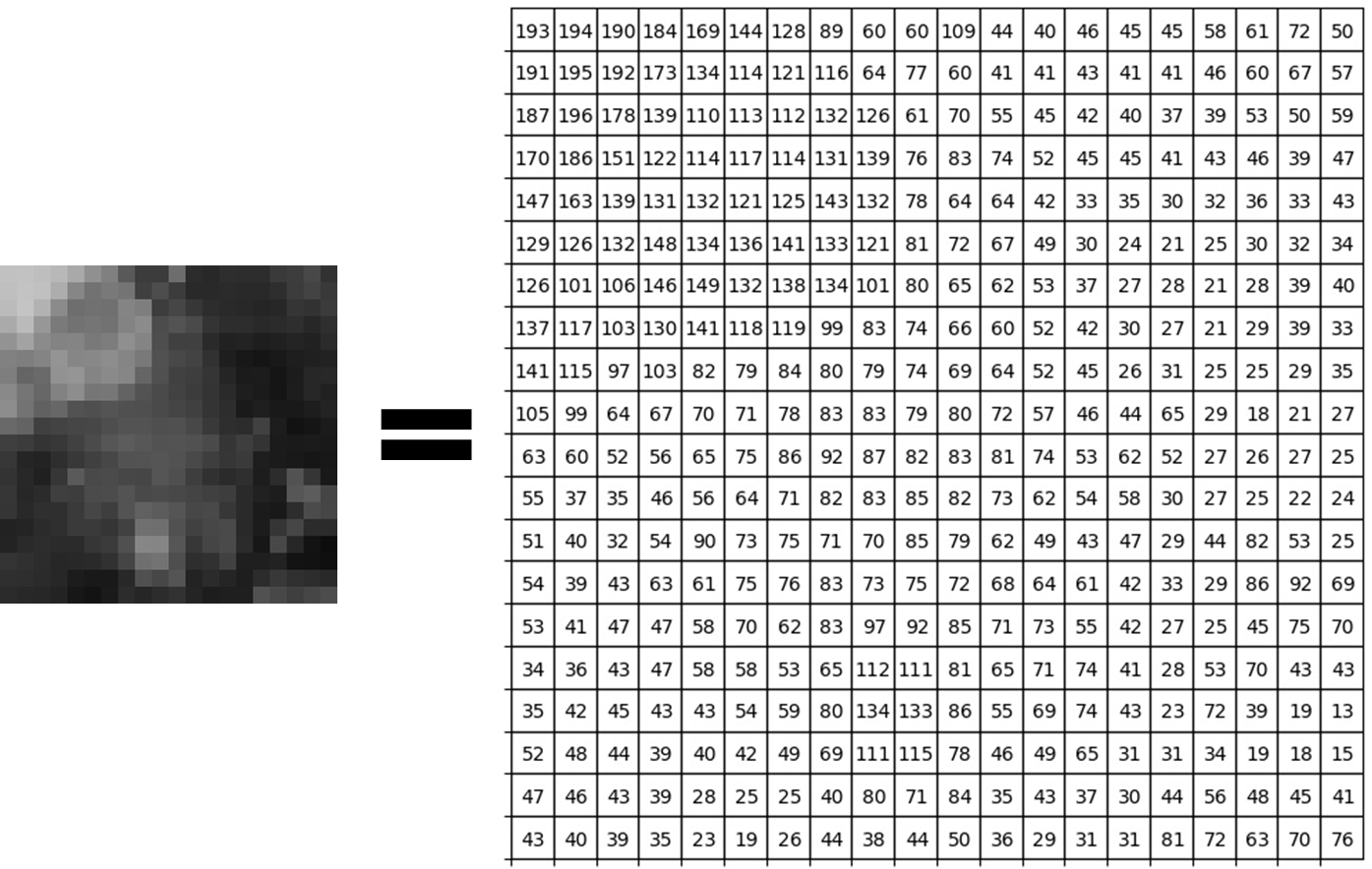
In a grayscale 8-bit image, there’s only one channel, and its intensity is represented by a value from 0 to 255, where 0 is black, 255 is white, and there are 254 shades of gray in between (which happens to be 204 more than the title of a certain novel). This is what the image above looks like when converted to grayscale:



If we zoom in on a 20 pixel x 20 pixel region of this image, we can discern the individual pixels that form the image, like so:



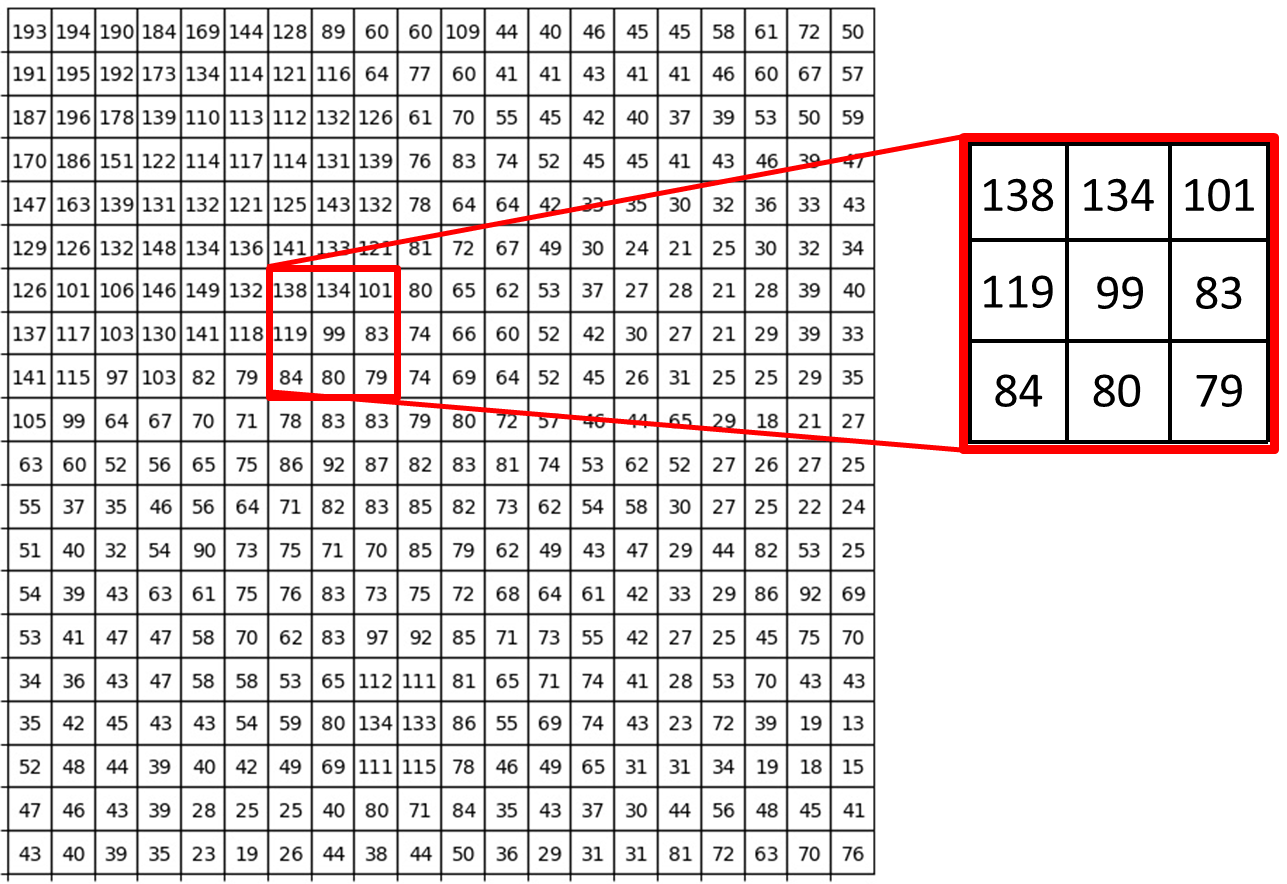
While we see each pixel as a hue of gray, remember that, as we just discussed, each pixel is actually represented digitally by a number from 0 to 255. That means the digital representation of the 20 x 20 selection of the image looks like this:



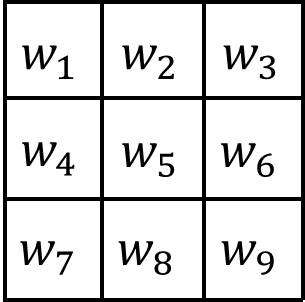
Notice how brighter pixels have higher values, corresponding to higher intensities. Conversely, darker pixels have lower values, corresponding to lower intensities. If we were looking at the color version of the image, each pixel would have three numbers (RGB) instead of just one. I chose the grayscale version as an example because it’s easier to visualize, but the RGB representation, or any other color space, for that matter, works essentially the same way, regardless of the number of channels.

The kernel

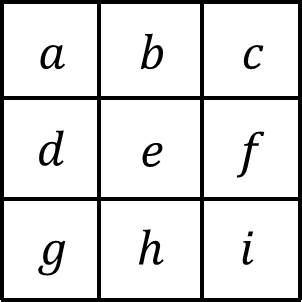
A kernel is essentially a mask or a filter that modifies the value of a pixel based on the values of its surrounding pixels. These surrounding pixels are termed the central pixel’s “neighborhood pixels.” Let’s zoom in further and examine an arbitrary 3×3 square neighborhood from our previous selection:



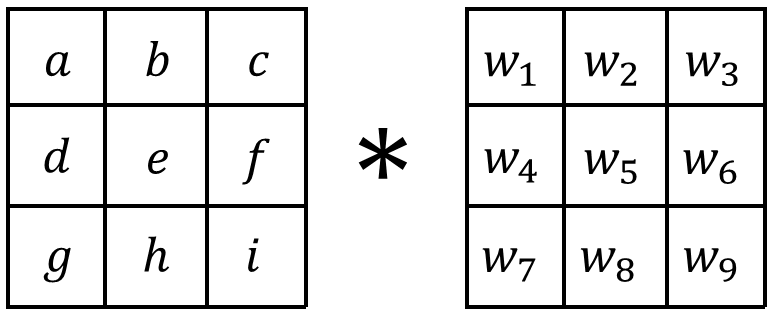
Here, we’re examining the neighborhood of the central pixel, which has a value of 99. The kernel will utilize the values from each pixel in the neighborhood (including the central pixel) to determine the new value for the central pixel. So, what does the kernel actually look like? A kernel is a matrix of the same shape as the neighborhood, and the value of each element of the kernel represents the weight given to the corresponding pixel from the neighborhood. A 3×3 kernel has the following basic form:



where *w*1​ through *w*9​ are the weights given to each pixel in the neighborhood. The pixel in the upper left corner would be given weight *w*1​, the central pixel would be given weight *w*5​, and so on. Representing the neighborhood in a similar fashion with the letters “a” through “g”:



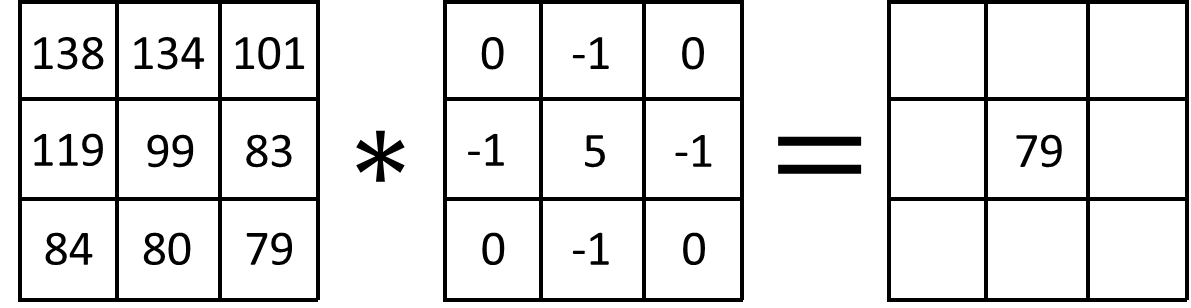
where *e* is the central pixel (corresponding to the pixel with a value of 99 from the example above) and the others are its neighborhood pixels. Then the convolution of the neighborhood with the kernel is written as:



where ∗ is the convolution symbol. The new value of the central pixel is the weighted sum of all the pixels in the neighborhood. In other words, the new value is given by the sum of the elements resulting from element-wise multiplication of the two matrices:

*enew  = w1a + w2b + w3c + w4d + w5e + w6f + w7g + w8h + w9i*

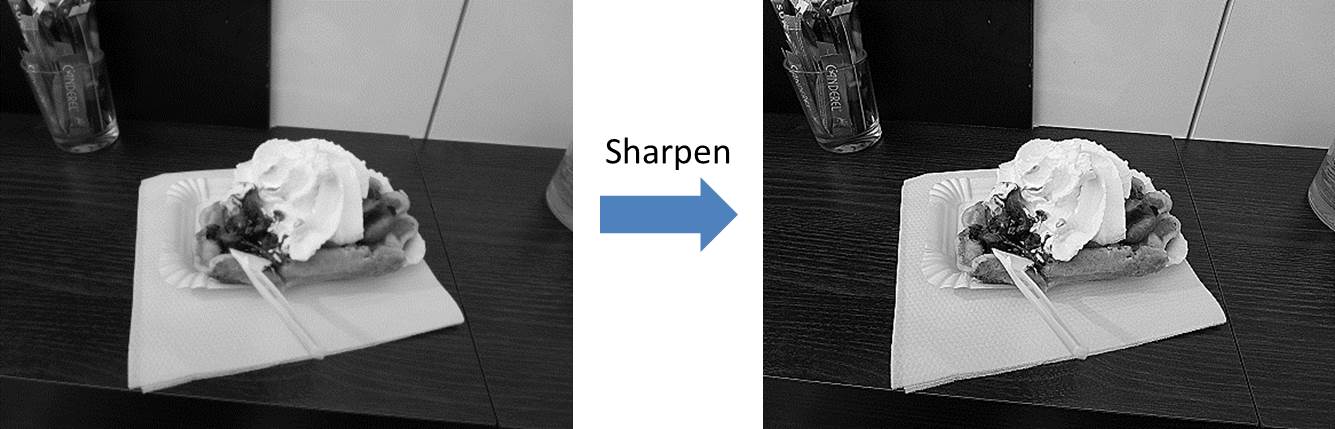
Going back to our example, let’s apply a standard 3×3 sharpen kernel. A sharpen kernel ignores the four corner pixels, subtracts the value of each pixel directly adjacent the central pixel, and multiplies the value of the central pixel by 5:



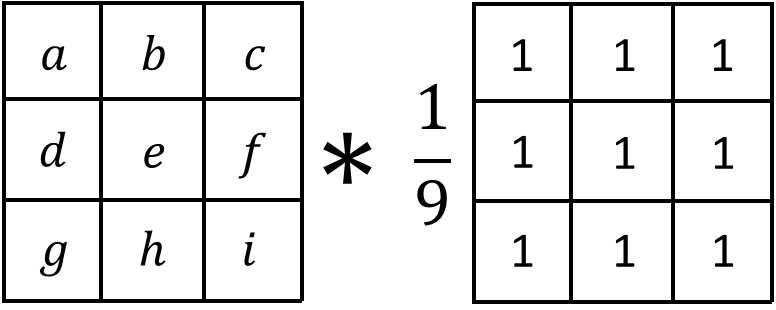
Writing this out:

(−1)(134) + (−1)(83) + (−1)(80) + (−1)(119) + (5)(99) = 79(−1)(134) + (−1)(83) + (−1)(80) + (−1)(119) + (5)(99) = 79.

This is done for every pixel of the source image to find the new value of each pixel in the resulting image. Here’s what applying a sharpen filter to an entire image looks like:



The comparison doesn’t require much commentary. By increasing the intensity of each pixel relative to its neighbors, the resulting image appears considerably sharper than the original. What if we want to blur the image? This is what a standard “box blur” convolution looks like:



With a box blur, we set the value of the central pixel equal to the average of all the pixels in its neighborhood (hence the division by 9). In essence, this “dilutes” each pixel. This is what it looks like applied to a whole image:

A blue arrow pointing to a glass

Description automatically generated

What if we want to blur the image more? Well, this is what happens when the size of the kernel (and the neighborhood) is increased to 5×5:

A blue arrow pointing to a blue object

Description automatically generated with medium confidence

Generally, increasing the size of the kernel/neighborhood amplifies the effect of the kernel.

**Difference Between 1D and 2D Kernels:**

In image processing, kernels are matrices used for various operations such as blurring, sharpening, edge detection, and more. The main difference between 1D and 2D kernels lies in their dimensionality and application.

The Gaussian kernel is separable, meaning you can apply the 1D kernel along rows then along columns to obtain the same result as the 2D kernel.

1. 1D Kernels:

- Dimensionality: 1D kernels are essentially one-dimensional matrices, often represented as a single row or column.

- Application: They are primarily used for operations along one axis of the image, such as blurring or sharpening along the horizontal or vertical direction.

- Example: For blurring along the horizontal direction, a 1D kernel might be something like `[0.1, 0.8, 0.1]`, where each value represents the weight assigned to neighboring pixels in the horizontal direction.

2. 2D Kernels:

- Dimensionality: 2D kernels are two-dimensional matrices.

- Application: They are used for operations that involve both horizontal and vertical directions simultaneously, such as general blurring, edge detection, or sharpening.

- Example: The popular Sobel operator, used for edge detection, consists of two 3x3 kernels – one for detecting edges in the horizontal direction and the other for the vertical direction.

Key Considerations:

- Computational Complexity: 1D kernels are computationally simpler compared to 2D kernels because they involve fewer operations.

- Effectiveness: 2D kernels are more versatile and can capture more complex patterns and features in images due to their ability to consider both horizontal and vertical relationships.

In summary, 1D kernels are suitable for operations along a single axis, while 2D kernels are used for more comprehensive image processing tasks involving both horizontal and vertical dimensions.

**Convolution Algorithm:**

* Calculate the dimensions of input image and kernel.
* Calculate dimensions of output image:
  + output\_height = img\_height-kernel\_height+1
  + output\_width = img\_ width -kernel\_ width +1
* Initialize the output image
* Apply convolution:
  + For each and every pixel, perform this:
    - output[y ,x] = np.sum(img[y:y+kernel\_height, x:x+kernel\_width] \* kernel )

**Sobel Kernel:**

The 3x3 sobel kernel is given as:

A two squares with numbers and letters

Description automatically generated with medium confidence

The Sobel operator is a widely used edge detection algorithm in image processing. It computes an approximation of the gradient of the image intensity function, which highlights edges by emphasizing areas of high intensity change.

The Sobel operator consists of two 3x3 convolution kernels, one for detecting edges in the horizontal direction (often called the Sobel X kernel) and one for detecting edges in the vertical direction (often called the Sobel Y kernel). These kernels are applied to the image via convolution.

When applied to an image, these kernels highlight horizontal and vertical edges respectively. The Sobel X kernel emphasizes changes in intensity along the horizontal direction, while the Sobel Y kernel emphasizes changes along the vertical direction.

By computing the gradient magnitude and direction from the responses of these two kernels, the Sobel operator can effectively detect edges in an image, making it useful for various image analysis and computer vision tasks.

Algorithm:

* Create the Sobel kernel.
* Apply convolution to first calculate gradient along x and then along y.

**Gaussian Kernel:**

We know the equation of Gaussian function is :

A mathematical equation with numbers and symbols

Description automatically generated

The kernel generated from this function would be called Gaussian Kernel. Where x, y are the distances from the center of the kernel.

A Gaussian kernel is a mathematical function used in image processing and computer vision for tasks such as blurring, noise reduction, and edge detection. It's based on the Gaussian distribution, which is a bell-shaped curve that describes the probability distribution of a continuous random variable.

In the context of image processing, a Gaussian kernel is often represented as a two-dimensional matrix of numbers. When applied to an image via convolution, it blurs the image by averaging the pixel values in the neighborhood of each pixel, with more weight given to pixels closer to the center according to the Gaussian distribution.

The Gaussian kernel is often normalized so that the sum of its elements equals 1. This ensures that the brightness of the image remains roughly the same after applying the blur.

Gaussian blurring is a widely used technique in image processing for reducing noise and detail in an image while preserving important structural features.

Algorithm:

* + Create a gaussian kernel of some shape.
  + Calculate the mid point of the kernel.
  + Calculate the constant term in the formula.
  + For each index i,j in the range(-mid to mid+1):
    - Calculate the exponential term.
    - Multiply the constant term and exponential term and stores it in kernel like kernel [y+half, x+half] = value
  + Now normalize it using this:
    - Kernel /=np.sum(kernel)

**Sharpen Kernel:**

[In image processing, a sharpen kernel is a small matrix used for enhancing the differences in adjacent pixel values, which makes the image look more vivid](https://cppsecrets.com/users/15271114971181051151051141181054950495164103109971051084699111109/Python-Computer-vision-Algorithm-sharpen-kernel.php). [This is accomplished by doing a convolution between the kernel and an image](https://cppsecrets.com/users/15271114971181051151051141181054950495164103109971051084699111109/Python-Computer-vision-Algorithm-sharpen-kernel.php).

A typical sharpen kernel might look like this:

0 -1 0

-1 5 -1

0 -1 0

[This kernel ignores the four corner pixels, subtracts the value of each pixel directly adjacent to the central pixel, and multiplies the value of the central pixel by 5](https://cppsecrets.com/users/15271114971181051151051141181054950495164103109971051084699111109/Python-Computer-vision-Algorithm-sharpen-kernel.php)[3](https://nrsyed.com/2018/02/17/kernels-in-image-processing/). [The process of applying this kernel to an image will cause the pixel intensities to be higher and therefore more prominent to the human eye](https://cppsecrets.com/users/15271114971181051151051141181054950495164103109971051084699111109/Python-Computer-vision-Algorithm-sharpen-kernel.php)[4](https://www.analyticsvidhya.com/blog/2021/08/sharpening-an-image-using-opencv-library-in-python/). [This results in a sharper and more detailed image](https://cppsecrets.com/users/15271114971181051151051141181054950495164103109971051084699111109/Python-Computer-vision-Algorithm-sharpen-kernel.php).

**Algorithm:**

* Create a sharpen kernel.
* Convolute it.

Perwitt Kernel:  
The Prewitt operator is a discrete differentiation operator used in image processing, particularly within edge detection algorithms. It computes an approximation of the gradient of the image intensity function. The Prewitt kernel is a 3x3 convolution kernel used in the Prewitt operator. It consists of two 3x3 matrices, one for horizontal changes and one for vertical changes in intensity.

The Prewitt kernels are as follows:

A black text on a white background

Description automatically generated

These kernels are convolved with the image to calculate the gradient in the horizontal and vertical directions, which can then be used to detect edges in the image.

Algorithm:

* Create this kernel.
* Convolate the image to first calculate gradient along x and then along y.

**BOX BLUR/FILTER:**

A box filter, also known as a box blur or a boxcar filter, is a simple image processing filter used for blurring an image or reducing noise. It works by averaging the pixel values in a rectangular neighborhood around each pixel in the image.

The size of the rectangular neighborhood (or the size of the box) determines the strength of the blur effect. A larger box size results in a more pronounced blur, while a smaller box size produces a softer blur.

Mathematically, the box filter can be represented as a convolution operation, where each output pixel is computed as the average of the pixel values within the neighborhood defined by the box.

Box filters are widely used in image processing for tasks such as smoothing, noise reduction, and pre-processing before more complex operations like edge detection or feature extraction. However, they can sometimes produce artifacts, such as loss of detail or a "blocky" appearance, especially with larger box sizes.

Algorithm:

* Create a box kernel.
* Convolute it.