Universidad Intercontinental de la Empresa

Asignatura

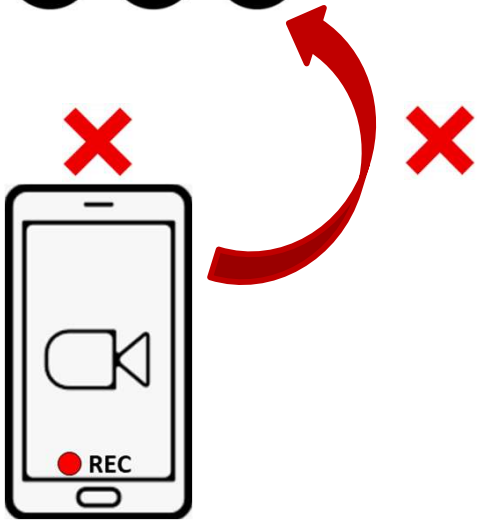
Computer Vision

Profesor

Rubén Ferreiroa


Recordar


Aula Virtual UIE


REC


Recordar

Participar

Universidad Intercontinental de la Empresa

Sesión 02

Unidad I

Digital Image Fundamentals

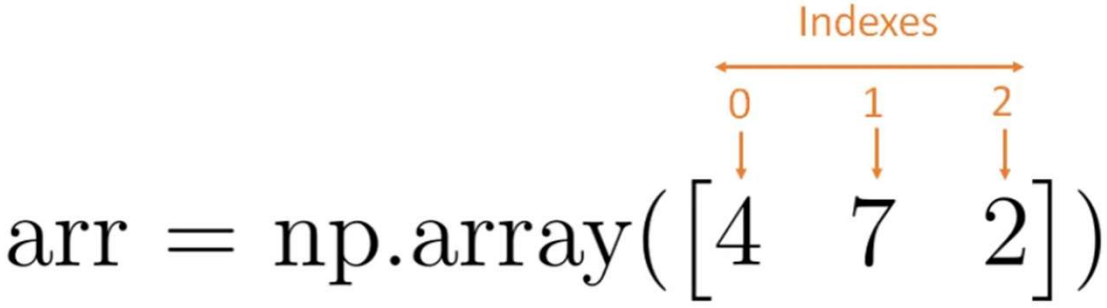
Tema 1

Nombre del tema

Crop Image

Slicing NumPy Arrays

One dimensional array



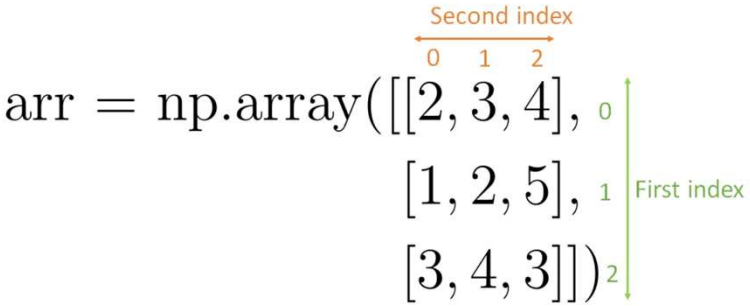
Let's say, I want to print the number 7 (which is the second element). I get it by indexing the array "arr" with a 1 in square brackets.

$$\text{print}(\text{arr}[1])$$

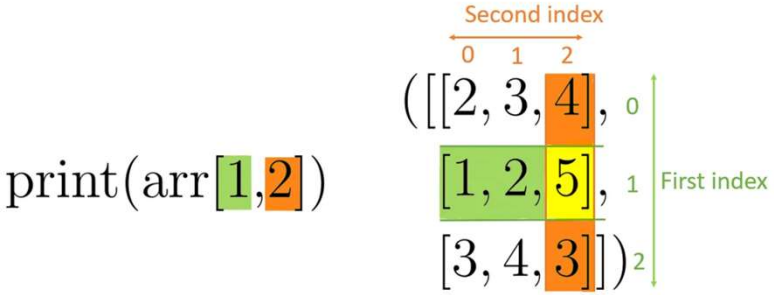
Crop Image
Slicing NumPy Arrays

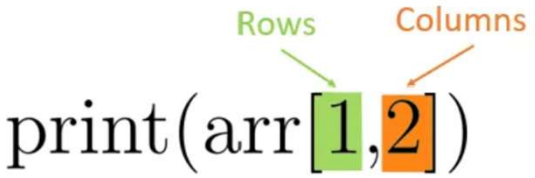
Two dimensional array

To get a single element from a 2 dimensional array, I have to provide two indexes.



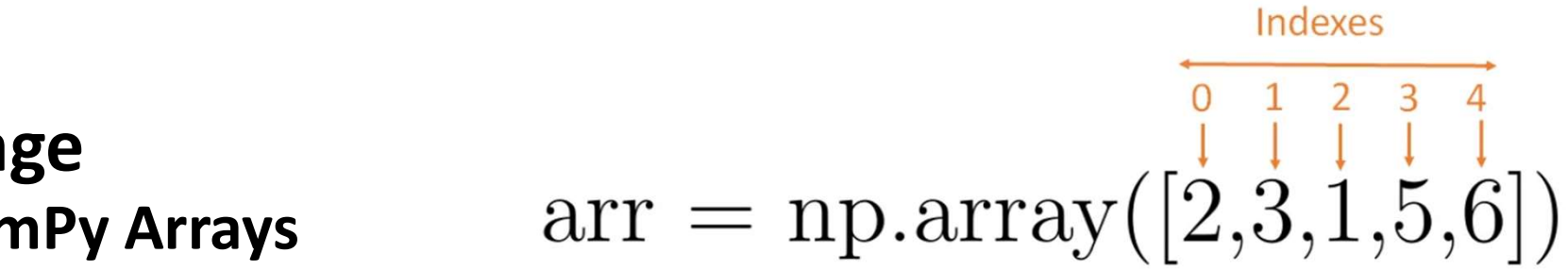
To get for example the number 5 from this array, you would have to index the array with the first and then the second index.



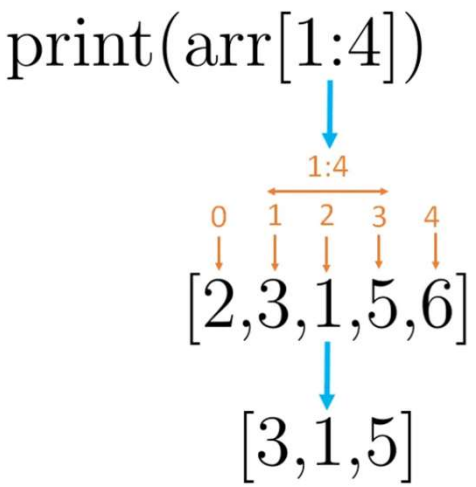


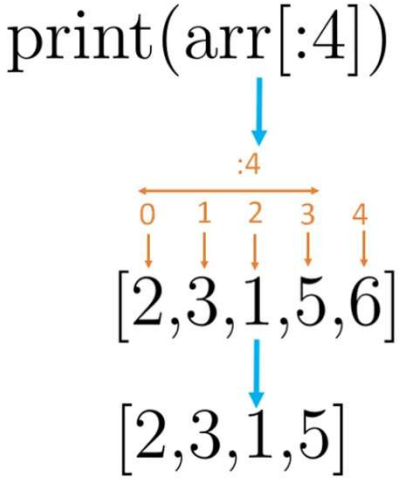
Crop Image

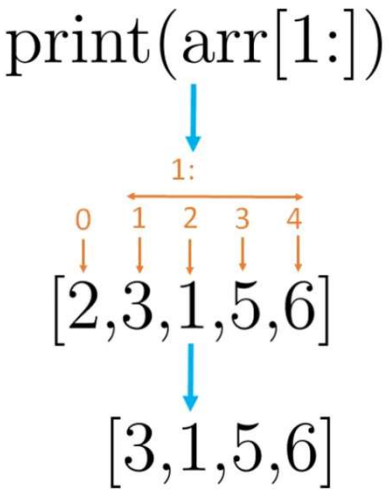
Slicing NumPy Arrays

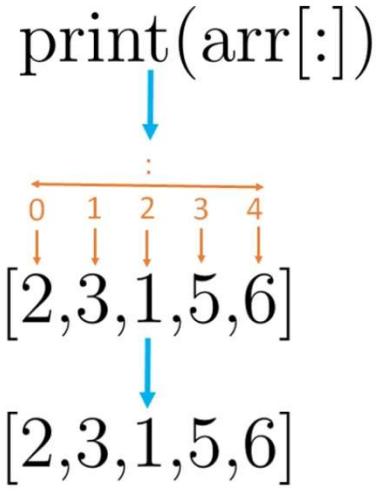


Select a part of an array (= slicing)



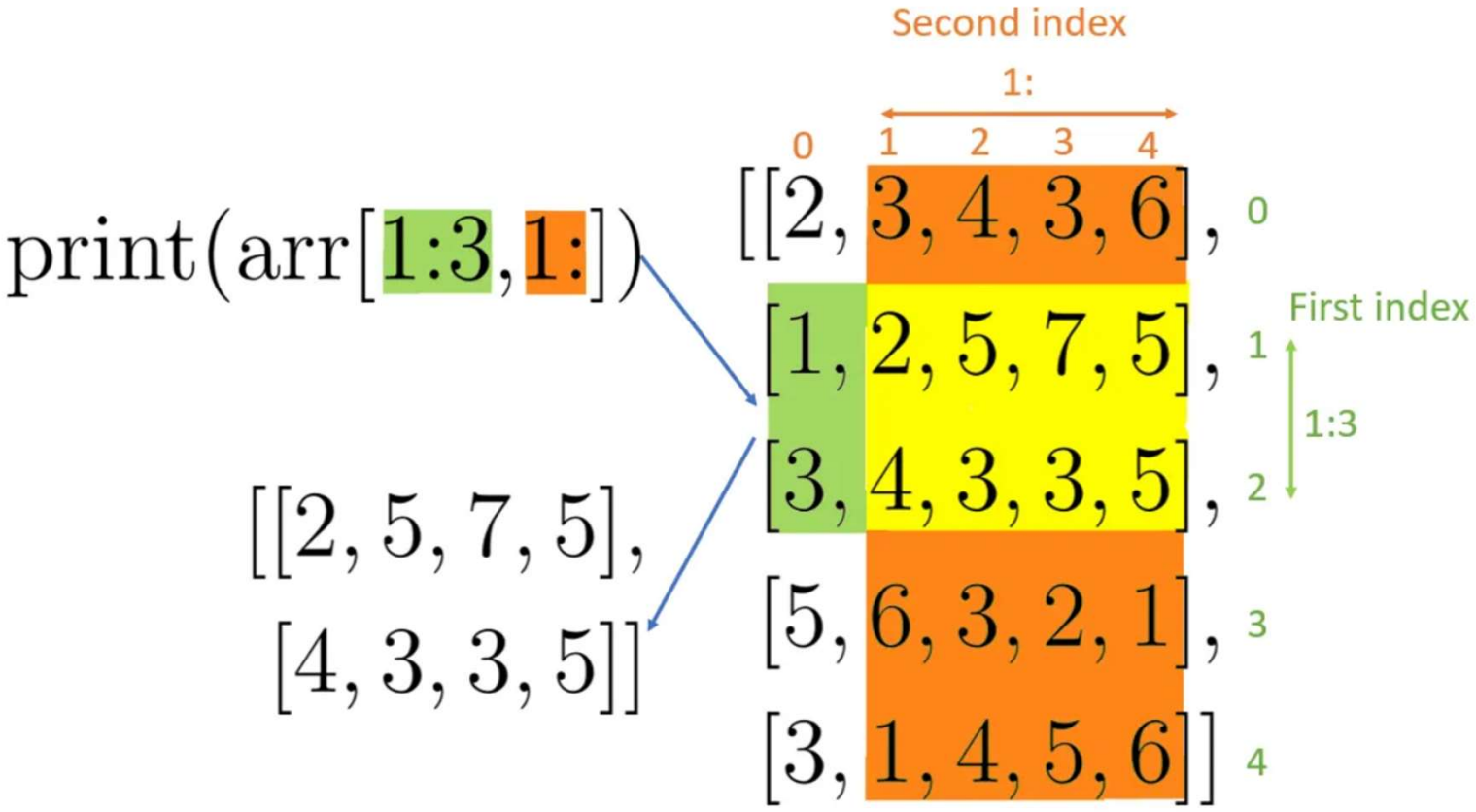






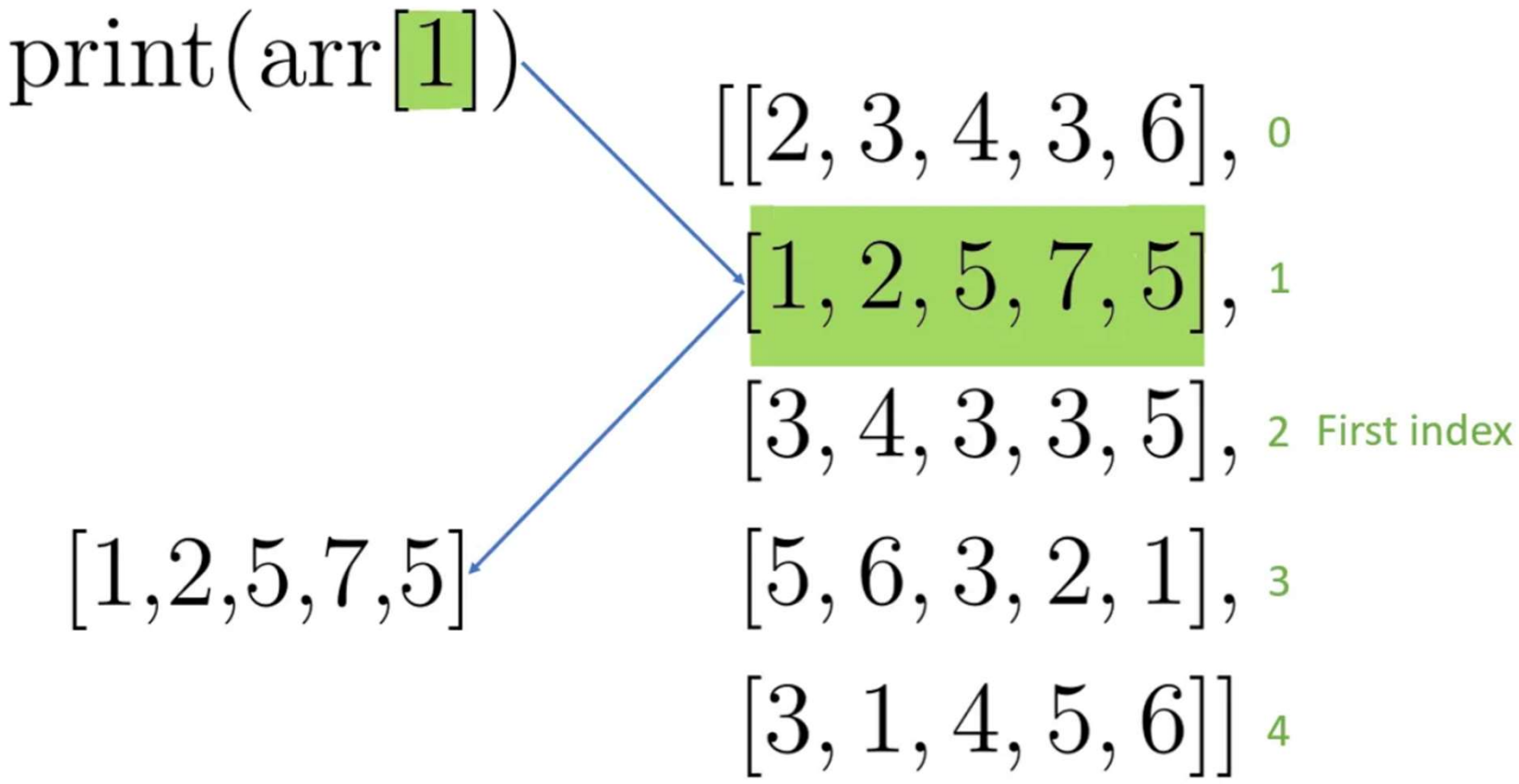
Crop Image

Slicing Numpy Arrays



Crop Image

Slicing Numpy Arrays



Crop Image

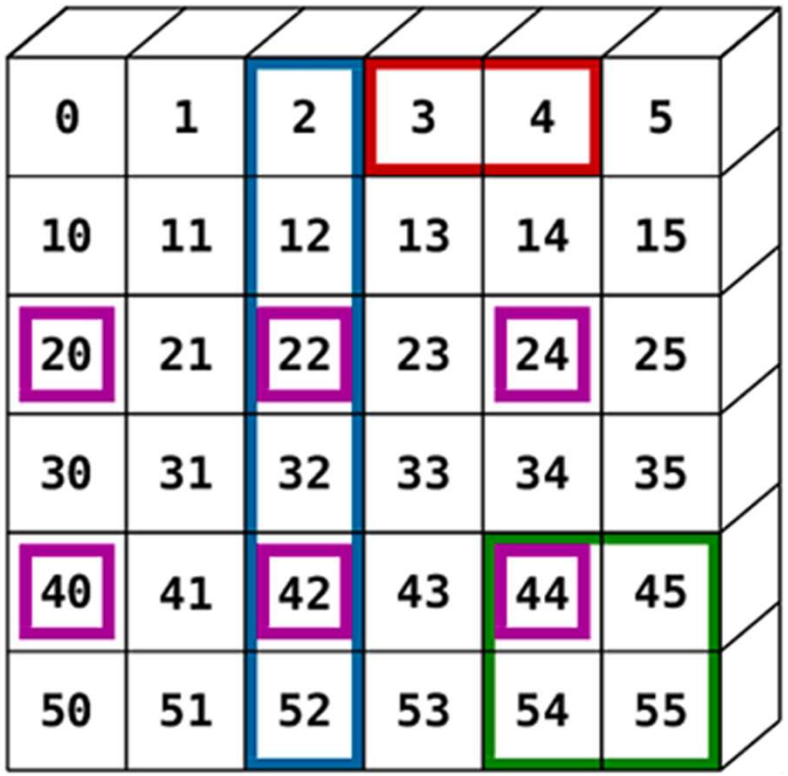
Cropping is the act of selecting and extracting the Region of Interest (or simply, ROI) and is the part of the image in which we are interested.

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 55],
       [54, 55]])

>>> a[:, 2]
a([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```



Crop Image

HANDS ON

What is image arithmetic?

Image arithmetic is simply matrix addition

Let's take a second and review some very basic linear algebra. Suppose we were to add the following two matrices:

$$\begin{bmatrix} 9 & 3 & 2 \\ 4 & 1 & 4 \end{bmatrix} + \begin{bmatrix} 0 & 9 & 4 \\ 7 & 9 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 9+0 & 3+9 & 2+4 \\ 4+7 & 1+9 & 4+4 \end{bmatrix} = \begin{bmatrix} 9 & 12 & 6 \\ 11 & 10 & 8 \end{bmatrix}$$

So it's obvious at this point that we all know basic arithmetic operations like addition and subtraction. But when working with images, *we need to keep in mind the numerical limits of our color space and data type*.

What is image arithmetic?

RGB images have pixels that fall within the range [0, 255]. What happens if we examine a pixel with an intensity of 250 and try to add 10 to it?

Under normal arithmetic rules, we would end up with a value of 260. However, since we represent RGB images as 8-bit unsigned integers who can only take on values in the range [0, 255], *260 is not a valid value*.

However, be sure to keep in mind that there is a difference between OpenCV and NumPy addition. *NumPy will perform modulus arithmetic and "wrap around."* On the other hand, *OpenCV will perform clipping* and ensure pixel values never fall outside the range [0, 255].

NUMPY
modulus operation and "wrap around" Under modulus rules, adding 10 to 255 would simply wrap around to a value of 9. **260 → 9**

OPENCV
clipping all pixels to have a minimum value of 0 and a maximum value of 255. **260 → 255**

What is image arithmetic?

perform modulus arithmetic and “wrap around.”

El operador módulo se utiliza cuando se desea comparar un número con el módulo y obtener el número equivalente restringido al rango del módulo.

Por ejemplo, digamos que quieres determinar qué hora sería nueve horas después de las 8:00 a. m. En un reloj de doce horas, no puedes simplemente sumar 9 a 8 porque obtendrías 17. Debes tomar el resultado, 17, y usarlo `mod`para obtener su valor equivalente en un contexto de doce horas:

```
Texto
```

```
8 o'clock + 9 = 17 o'clock
17 mod 12 = 5
```

`17 mod 12`devuelve `5`. Esto significa que nueve horas después de las 8:00 a. m. son las 5:00 p. m. Determinaste esto tomando el número `17`y aplicándolo a un `mod 12`contexto.

What is image arithmetic?

EXAMPLE 2.6: Comparing images using subtraction.

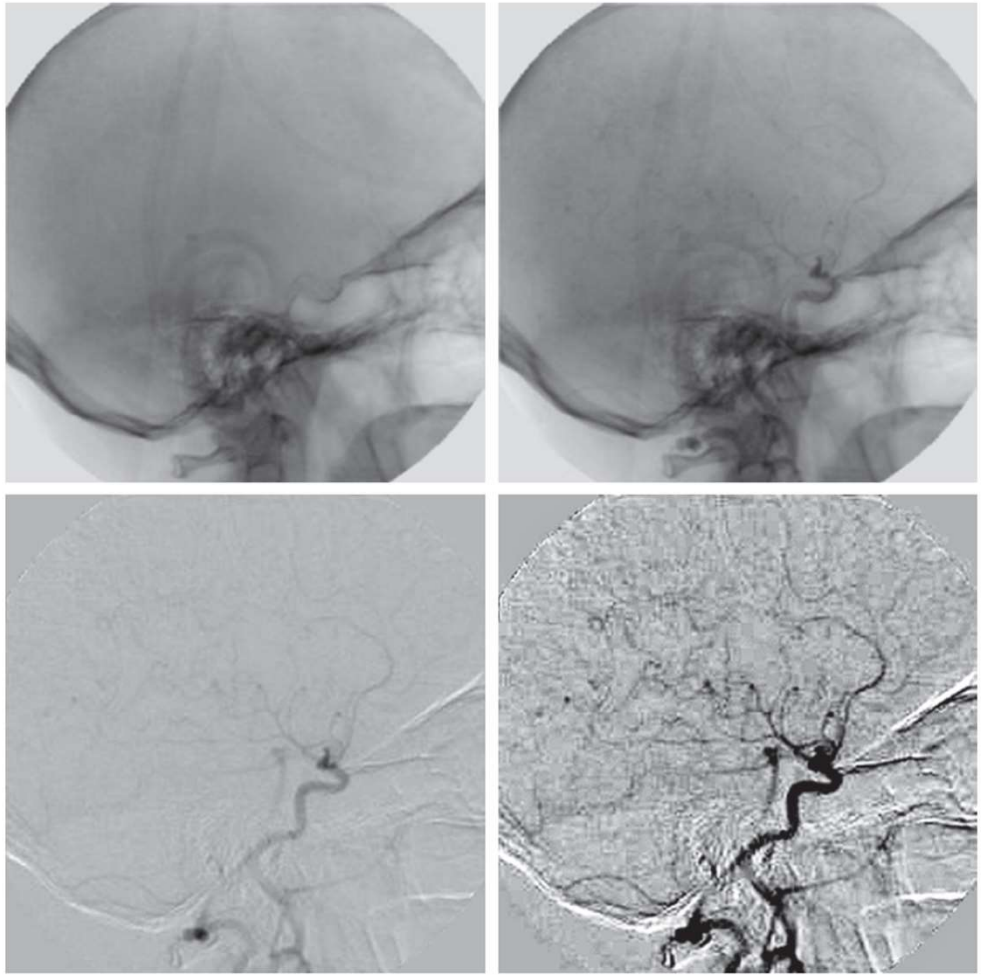
Image subtraction is used routinely for enhancing differences between images. For example, the image in Fig. 2.30(b) was obtained by setting to zero the least-significant bit of every pixel in Fig. 2.30(a). Visually, these images are indistinguishable. However, as Fig. 2.30(c) shows, subtracting one image from

a b c

FIGURE 2.30 (a) Infrared image of the Washington, D.C. area. (b) Image resulting from setting to zero the least significant bit of every pixel in (a). (c) Difference of the two images, scaled to the range [0, 255] for clarity. (Original image courtesy of NASA.)

What is image arithmetic?

Figure 2.32(a) shows a mask X-ray image of the top of a patient's head prior to injection of an iodine medium into the bloodstream, and Fig. 2.32(b) is a sample of a live image taken after the medium was



a b
c d

FIGURE 2.32 Digital subtraction angiography. (a) Mask image. (b) A live image. (c) Difference between (a) and (b). (d) Enhanced difference image. (Figures (a) and (b) courtesy of the Image Sciences Institute, University Medical Center, Utrecht, The Netherlands.)

What is image arithmetic?

HANDS ON

Resize Image

Interpolation

A method of constructing new data points within the range of a discrete set of known data points.

— Interpolation, Wikipedia

This table gives some values of an unknown function $f(x)$.

x	$f(x)$
0	0
1	0.8415
2	0.9093
3	0.1411
4	−0.7568
5	−0.9589
6	−0.2794

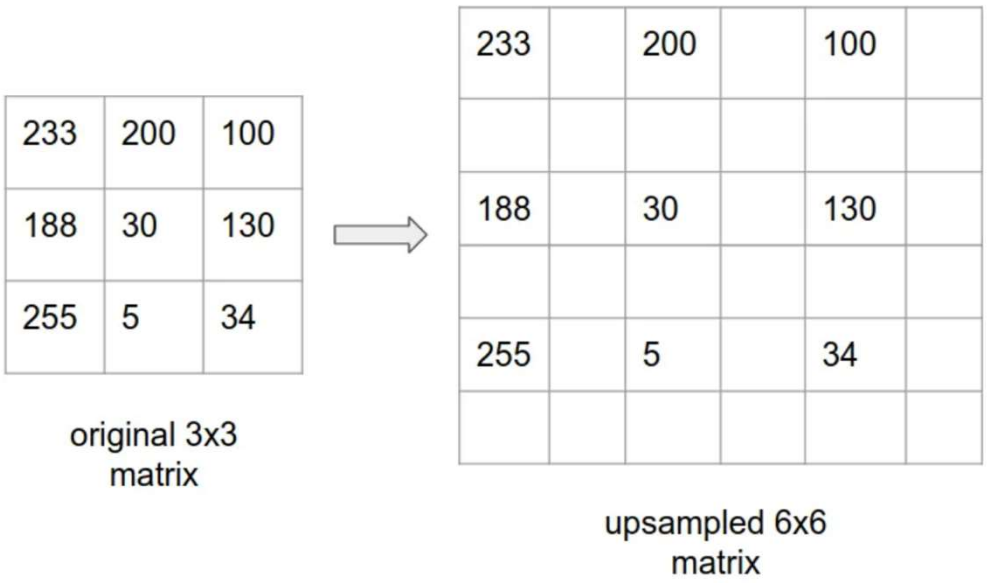
Interpolation provides a means of estimating the function at intermediate points, such as $x = 2.5$.

Resize Image

Basic Idea

A 2-d image is basically represented as a 2-dimensional matrix with each cell in the matrix containing a pixel value. So, when we say scaling up this matrix, it means creating a bigger matrix than the original one and fill up the missing pixel values in this bigger matrix.

Let us look at an image to make the idea more concrete:



Resize Image

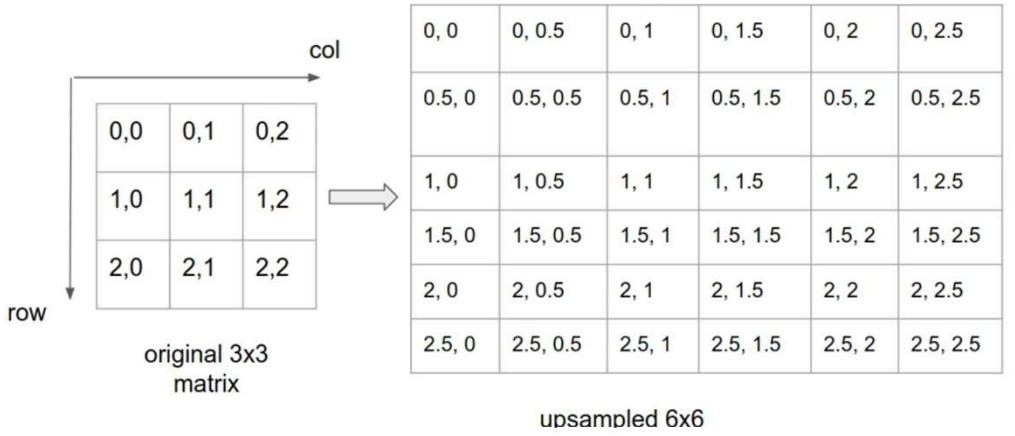
Basic Idea

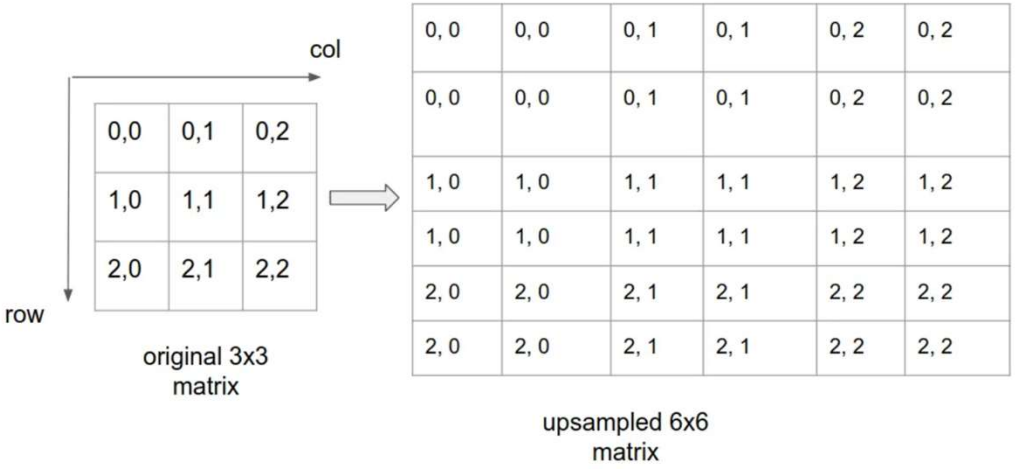
1. Consider an input matrix of size 3x3 with each cell containing some pixel values in the range 0–255. row, col indices starts from 0 to n-1 where n is the row/col length.
2. Now let's create a new 6x6 output(upscaled) empty matrix.
3. In order to start filling the pixel values for the new matrix, we first have to represent the output coordinate space interms of the input coordinate space i.e. for every (row, col) in the output matrix, what is the corresponding (row, col) in the input matrix? This is just the scaling factor which is 1/2 in our case.
4. To make it more clearer, the row scaling factor is 1/2 and column scaling factor is 1/2(row and col will have separate scaling factors but since our eg considers a square matrix, both are same here).
5. row 0, col 0 in output is mapped to row 0, col 0 in input, whereas row 1, col 1 in output is mapped to row 0.5, col 0.5 in input and so on.

Resize Image

Nearest Neighbour Interpolation

Now let's look at the mapped coordinate space:

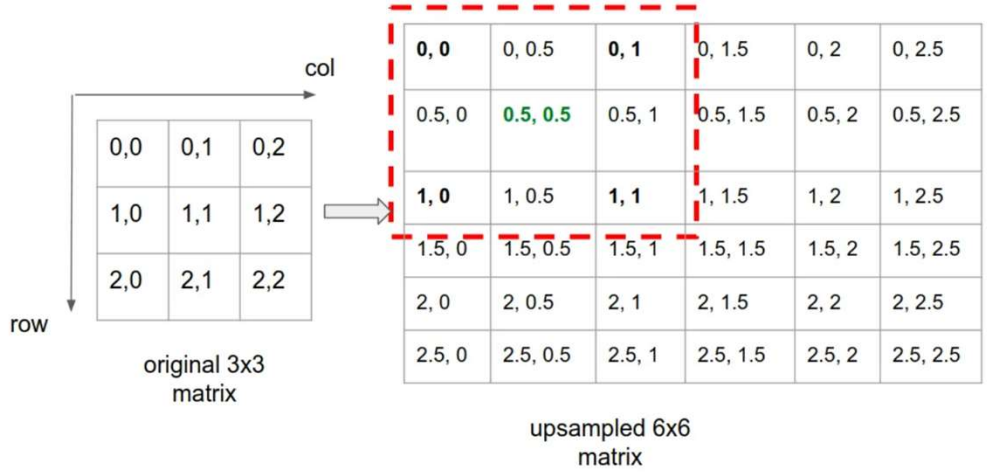




Here in the above image, a simple way to find the values for unknown cells such as (0.5, 0.5) or (2, 2.5) is to simply round them off to nearest integer. for eg: (2, 2.5) to (2, 2) or (1.5, 2.5) to (1, 2). This is called nearest neighbour interpolation.

Resize Image

Bilinear Interpolation:



1. For any unknown (row, col) cell in the upsampled matrix, pick the 4 nearest pixels. For eg: for cell (0.5, 0.5), the 4 nearest pixels are (0, 0), (0, 1), (1, 0), (1, 1).
2. Finding the pixel value at (0.5, 0.5) means first finding the pixels values at (0, 0.5) and (1, 0.5) and then using them to find the value at (0.5, 0.5)
3. 3. Do linear interpolation twice along x-direction — one at (0, 0.5) using <(0, 0), (0, 1)> and another at (1, 0.5) using <(1, 0), (1, 1)>
4. Then another interpolation along y-direction — at (0.5, 0.5) using (0, 0.5) and (1, 0.5)

But how is this linear interpolant is computed?

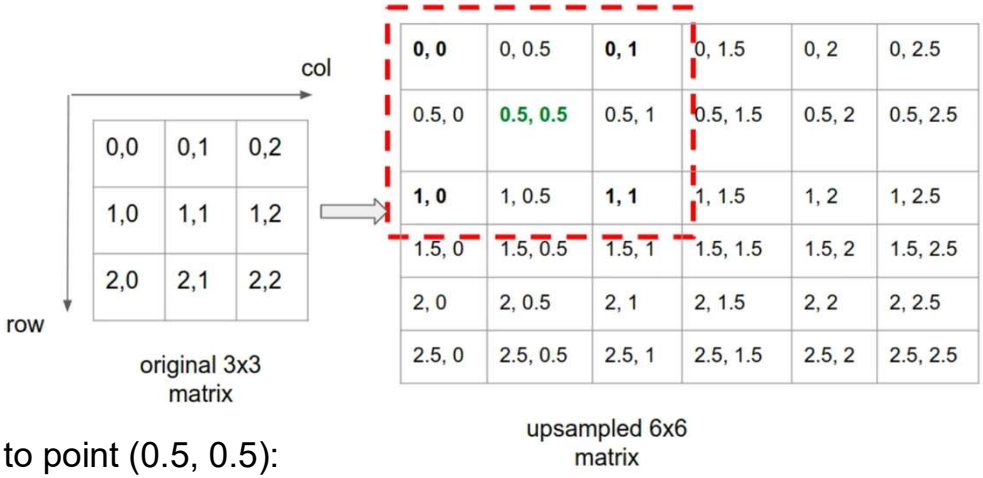
Let's perform the first interpolation along x-direction to compute the pixel value at (0, 0.5) using (0, 0) and (0, 1)

I(0, 0) = 233, I(0, 1) = 200 where I denotes the pixel intensity(refer above image for more clarity)

I(0, 0.5) = 233 * 0.5 + 200 * 0.5 = 116.5 + 100 = 216.5

Resize Image

Bilinear Interpolation



1. First, let's identify the 4 nearest neighboring pixels to point (0.5, 0.5):

   - (0,0) = 0.0
   - (0,1) = 1.0
   - (1,0) = 1.0
   - (1,1) = 1.1

2. The process requires two linear interpolations in x and a final one in y:
   A. First x-direction interpolation to find the value at (0, 0.5):

```
I(0, 0.5) = I(0,0) * 0.5 + I(0,1) * 0.5
I(0, 0.5) = 0.0 * 0.5 + 1.0 * 0.5 = 0.5
```

   B. Second x-direction interpolation to find the value at (1, 0.5):

```
I(1, 0.5) = I(1,0) * 0.5 + I(1,1) * 0.5
I(1, 0.5) = 1.0 * 0.5 + 1.1 * 0.5 = 1.05
```

   C. Final y-direction interpolation to find the value at (0.5, 0.5):

```
I(0.5, 0.5) = I(0, 0.5) * 0.5 + I(1, 0.5) * 0.5
I(0.5, 0.5) = 0.5 * 0.5 + 1.05 * 0.5
I(0.5, 0.5) = 0.25 + 0.525 = 0.775
```

Interpolation for upscaling images

Here are the most important image interpolation methods for upscaling images, along with their key advantages:

1. Nearest Neighbor Interpolation
   - Simplest and fastest method
   - Preserves hard edges
   - Good for pixel art and graphics with sharp edges
   - No new colors are introduced
   - Main disadvantage: Results in blocky/pixelated images

2. Bilinear Interpolation
   - Smoother results than nearest neighbor
   - Good balance between quality and computational cost
   - Works well for natural images with gradual transitions
   - Better preservation of detail than nearest neighbor
   - Main disadvantage: Can cause some blurring of edges

Interpolation for upscaling images

3. Bicubic Interpolation
   - Higher quality than bilinear
   - Better edge preservation
   - Sharper results than bilinear
   - Good for photographs and natural images
   - Main disadvantage: More computationally intensive than bilinear

4. Lanczos Interpolation
   - Excellent quality for photographic images
   - Superior edge preservation
   - Minimal artifacts
   - Better preservation of high-frequency details
   - Main disadvantage: Computationally expensive

5. B-Spline Interpolation
   - Very smooth results
   - Good for medical imaging
   - Reduces aliasing artifacts
   - Excellent for continuous tone images
   - Main disadvantage: Can over-smooth some details

Interpolation for upscaling images

6. Mitchell-Netravali
   - Good balance between sharpness and smoothness
   - Reduces ringing artifacts
   - Popular in professional image processing
   - Good for both natural and synthetic images
   - Main disadvantage: Medium computational cost

7. Deep Learning-Based Methods (e.g., SRCNN, ESRGAN)
   - Highest quality results
   - Can infer missing details
   - Excellent for facial features and text
   - Can reduce multiple types of artifacts
   - Main disadvantages:
     * Very computationally intensive
     * Requires specialized hardware
     * May introduce artifacts in certain cases

Interpolation for upscaling images

8. Sinc Interpolation
   - Theoretically perfect reconstruction for band-limited signals
   - Excellent preservation of frequency content
   - Used as a reference for other methods
   - Main disadvantages:
     * Infinite support (needs to be windowed)
     * Computationally expensive
     * Can introduce ringing artifacts

For practical applications:
- For real-time applications or quick previews: Nearest Neighbor or Bilinear
- For general photography: Bicubic or Lanczos
- For professional work: Mitchell-Netravali or Lanczos
- For specialized applications (AI upscaling): Deep Learning methods
- For medical imaging: B-Spline or specialized medical imaging algorithms

The choice of method often depends on:
1. The type of image content
2. The required processing speed
3. The available computational resources
4. The intended use of the output image
5. The acceptable trade-off between quality and processing time

Resize Image

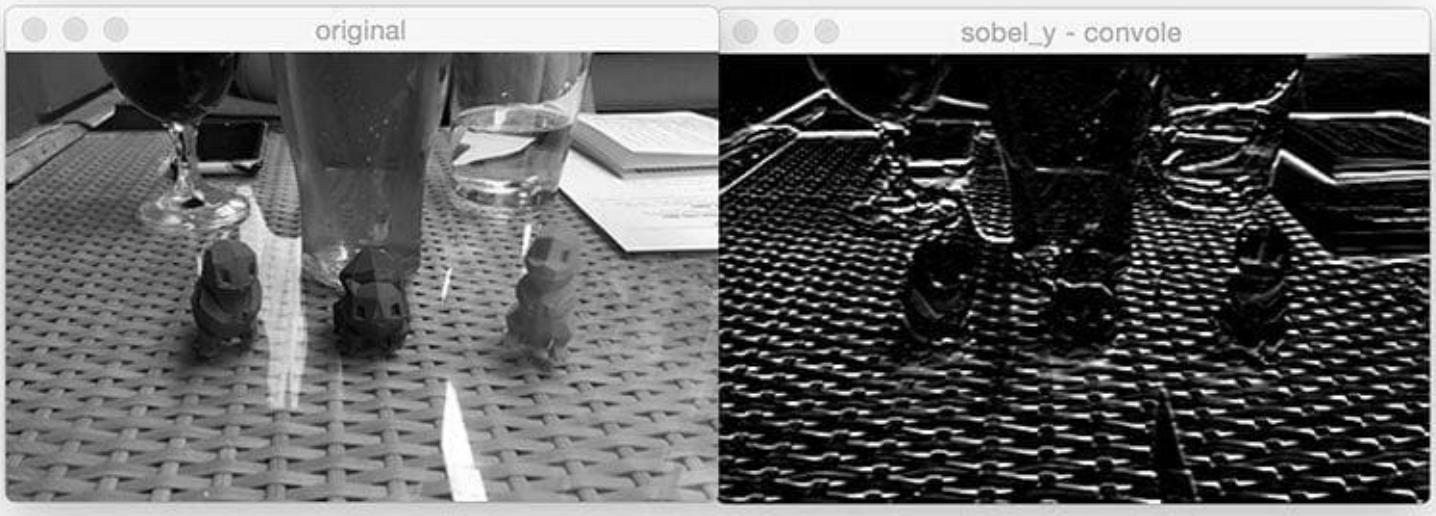
HANDS ON

Kernel Convolution: How Blurs & FILTERS WORK

In image processing, a kernel, convolution matrix, or mask is a *small matrix. It is used for blurring, sharpening, embossing, edge detection, and more.*

KERNEL: Convolutions with OpenCV and Python

- What are image convolutions?
- What do they do?
- Why do we use them?
- How do we apply them?
- And what role do convolutions play in deep learning?

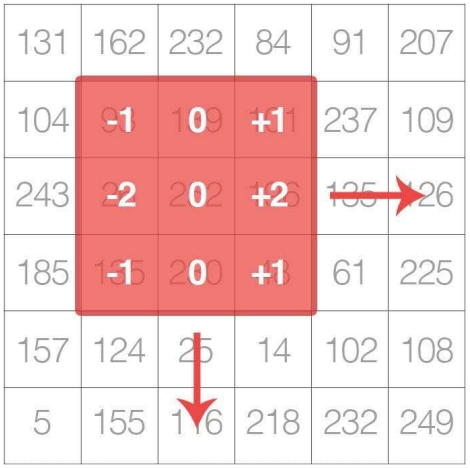


Convolutions with OpenCV and Python

- blurring? Yep, that's a convolution.
- edge detection? Yep, convolution.
- Sharpening? Yep, convolution
- Etc.

In reality, an (image) convolution is simply *an element-wise multiplication of two matrices followed by a sum*.

We can think of an *image as a big matrix* and *kernel or convolutional matrix as a tiny matrix* that is used for blurring, sharpening, edge detection, and other image processing functions.



Understanding Image Convolutions

In image processing, a convolution requires three components:

1. An input image.
2. A kernel matrix that we are going to apply to the input image.
3. An output image to store the output of the input image convolved with the kernel.

Convolution itself is actually very easy. All we need to do is:

1. Select an (x, y) coordinate from the original image.
2. Place the center of the kernel at this (x, y) coordinate.
3. Take the element-wise multiplication of the input image region and the kernel, then sum up the values of these multiplication operations into a single value. The sum of these multiplications is called the kernel output.
4. Use the same (x, y) coordinates from Step 1, but this time, store the kernel output in the same (x, y)-location as the output image

Understanding Image Convolutions

$$O_{i,j} = \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} \star \begin{bmatrix}197 & 50 & 213\\ 3 & 181 & 203\\ 231 & 2 & 93\end{bmatrix} = \begin{bmatrix}1/9 \times 197 & 1/9 \times 50 & 1/9 \times 213\\ 1/9 \times 3 & 1/9 \times 181 & 1/9 \times 203\\ 1/9 \times 231 & 1/9 \times 2 & 1/9 \times 93\end{bmatrix}$$

Figure 4: Convolving a *3 x 3* input image region with a *3 x 3* kernel used for blurring.

$$O_{i,j} = \sum \begin{bmatrix}21 & 5 & 23\\ 0 & 20 & 22\\ 25 & 0 & 10\end{bmatrix} = 126$$

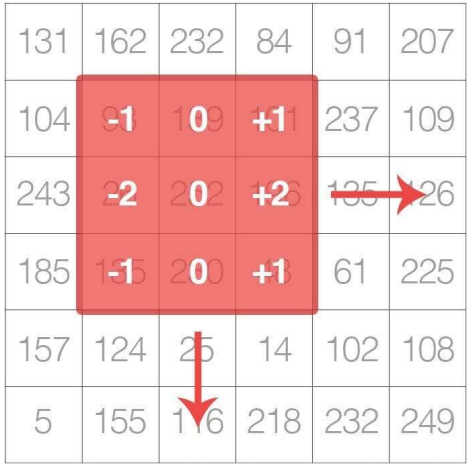
Figure 5: The output of the convolution operation is stored in

After applying this convolution, we would set the pixel located at the coordinate (i, j) of the output image O to O_i,j = 126.

That's all there is to it!

Understanding Image Convolutions

Convolution is simply the sum of element-wise matrix multiplication between the kernel and neighborhood that the kernel covers of the input image.



Implementing Convolutions with OpenCV and Python

Convolutions with OpenCV and Python

```
7.  def convolve(image, kernel):
8.      # grab the spatial dimensions of the image, along with
9.      # the spatial dimensions of the kernel
10.     (iH, iW) = image.shape[:2]
11.     (kH, kW) = kernel.shape[:2]
12.
13.     # allocate memory for the output image, taking care to
14.     # "pad" the borders of the input image so the spatial
15.     # size (i.e., width and height) are not reduced
16.     pad = (kW - 1) // 2
17.     image = cv2.copyMakeBorder(image, pad, pad, pad, pad,
18.         cv2.BORDER_REPLICATE)
19.     output = np.zeros((iH, iW), dtype="float32")
```

1. Original Image and Kernel
Convolution requires two main elements:
- A grayscale image
- A kernel (convolution matrix)

Imagen				
50	50	50	50	50
50	100	100	100	50
50	100	150	100	50
50	100	100	100	50
50	50	50	50	50

Kernel		
-1	-1	-1
-1	8	-1
-1	-1	-1

Implementing Convolutions with OpenCV and Python

1. Original Image and Kernel

Convolution requires two main elements:
- A grayscale image
- A kernel (convolution matrix)

Image

50	50	50	50	50
50	100	100	100	50
50	100	150	100	50
50	100	100	100	50
50	50	50	50	50

Kernel

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

2. The Border Problem

When sliding the kernel, border pixels don't have enough neighbors for complete calculation.
This leads to reduced output dimensions.

Image

50	50	50	50	50
50	100	100	100	50
50	100	150	100	50
50	100	100	100	50
50	50	50	50	50

Kernel

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

Image with Padding

50	50	50	50	50	50	50
50	50	50	50	50	50	50
50	50	100	100	100	50	50
50	50	100	150	100	50	50
50	50	100	100	100	50	50
50	50	50	50	50	50	50
50	50	50	50	50	50	50

3. Solution: Padding

We add a border to the original image to maintain dimensions. Common types:
- Border replication
- Zero padding
- Wrap around

4. Convolution Process

The process involves:
- Centering the kernel on each pixel
- Multiplying overlapping values
- Summing the results
- Storing the output in the new image

Image

50	50	50	50	50
50	100	100	100	50
50	100	150	100	50
50	100	100	100	50
50	50	50	50	50

Kernel

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

Convolution Calculation Example

For center pixel (150):

Sum = (-1×100) + (-1×100) + (-1×100) +

(-1×100) + (8×150) + (-1×100) +

(-1×100) + (-1×100) + (-1×100)

= 150

Implementing Convolutions with OpenCV and Python

→ Launch Jupyter Notebook on Google Colab

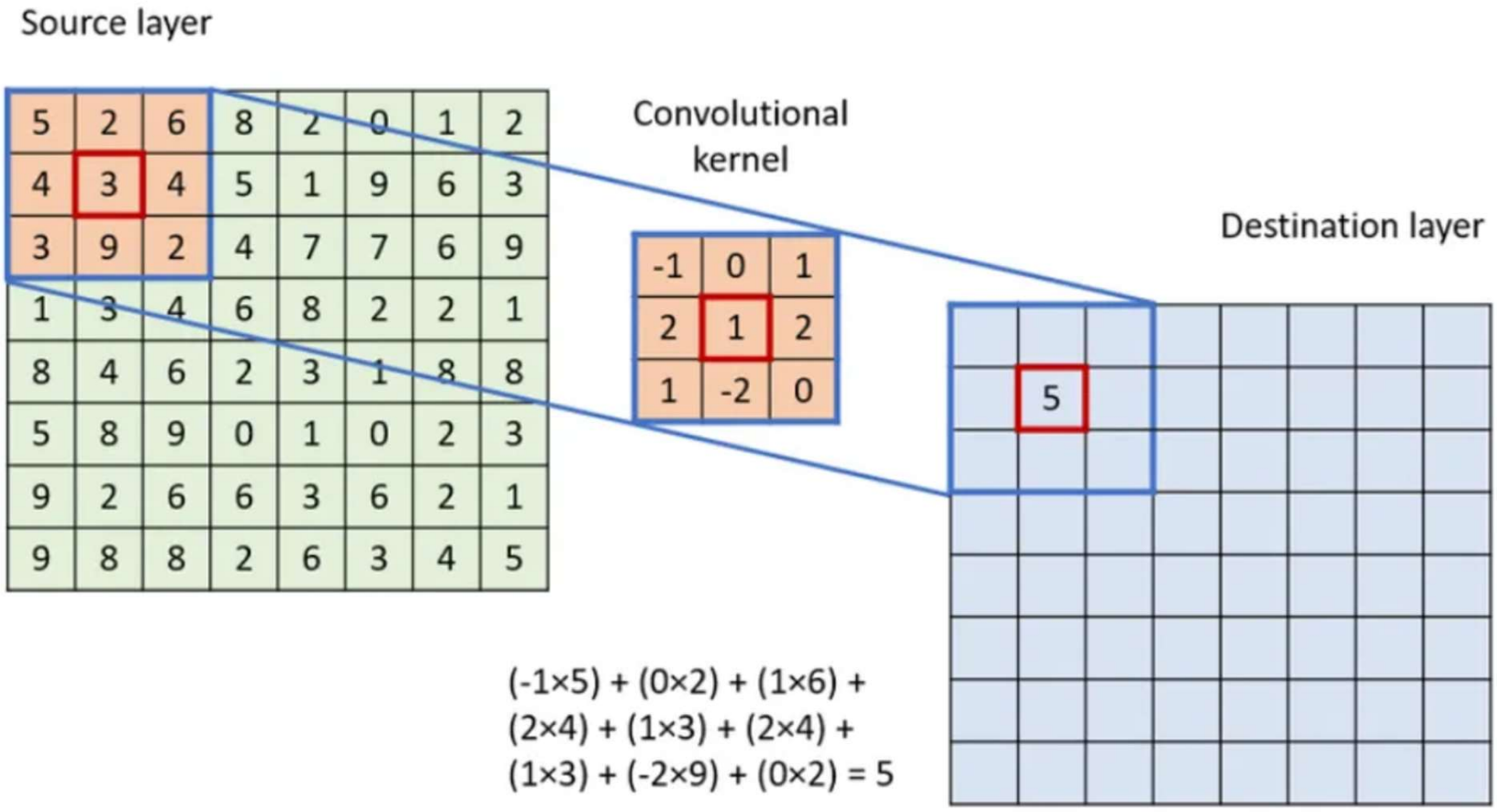
Convolutions with OpenCV and Python

```
    # loop over the input image, "sliding" the kernel across
    # each (x, y)-coordinate from left-to-right and top to
    # bottom
    for y in np.arange(pad, iH + pad):
        for x in np.arange(pad, iW + pad):
            # extract the ROI of the image by extracting the
            # *center* region of the current (x, y)-coordinates
            # dimensions
            roi = image[y - pad:y + pad + 1, x - pad:x + pad + 1]

            # perform the actual convolution by taking the
            # element-wise multiplicate between the ROI and
            # the kernel, then summing the matrix
            k = (roi * kernel).sum()

            # store the convolved value in the output (x,y)-
            # coordinate of the output image
            output[y - pad, x - pad] = k
```

Kernel Convolution: How Blurs & Filters Work

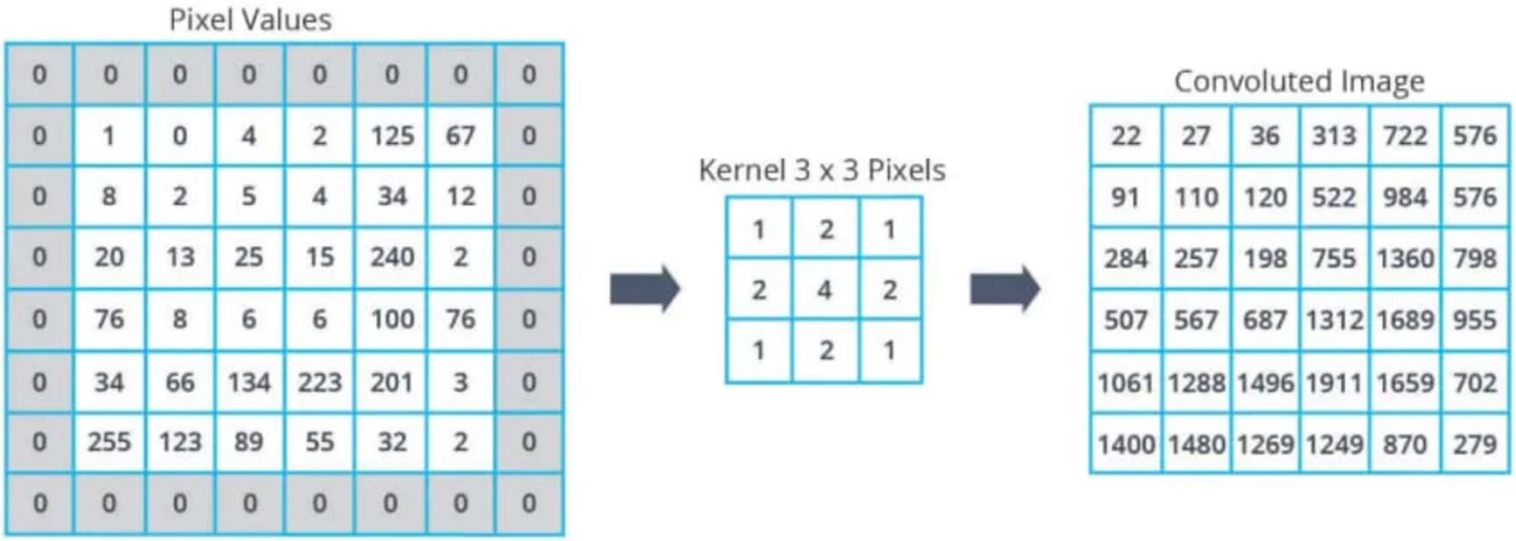


Kernel Convolution: How Blurs & Filters Work

The zeros placed around the original image matrix (a technique called *zero padding*) serve several important purposes in convolution operations:

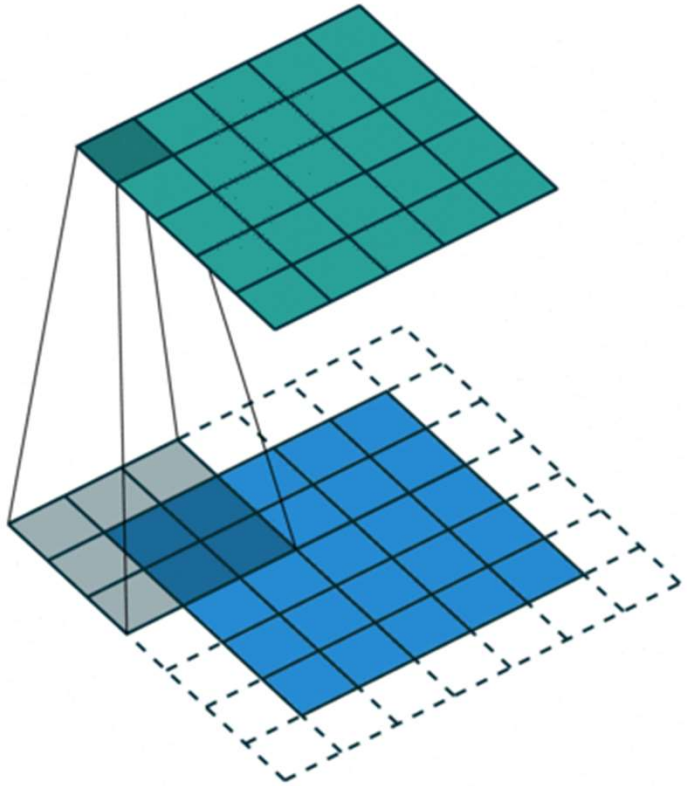
1. Edge Preservation
   - Without padding, we wouldn't be able to apply the kernel to the border pixels of the image
   - Zero padding allows us to compute convolution values for every pixel in the original image. This helps preserve the spatial dimensions of the image
2. Mathematical Necessity
   - For a 3x3 kernel, we need 9 values to perform each convolution operation
   - Edge pixels don't have all neighboring pixels (for example, corner pixels only have 3 neighbors)
   - Adding zeros ensures we can perform the complete convolution calculation for every pixel
3. Output Size Control
   - Without padding, the output image would shrink after each convolution
   - If we have an input image of size n×n and a kernel of size k×k, the output would be (n-k+1)×(n-k+1)
   - By adding padding of (k-1)/2 zeros on each side (where k is the kernel size), we maintain the original image dimensions

In this specific example: The original image is 6x6 (excluding padding). A border of zeros is added making it 8x8. This allows the 3x3 kernel to slide over every original pixel while having sufficient neighbors for the calculation. The result maintains meaningful dimensions that correspond to the original image structure



Kernel Convolution: How Blurs & Filters Work

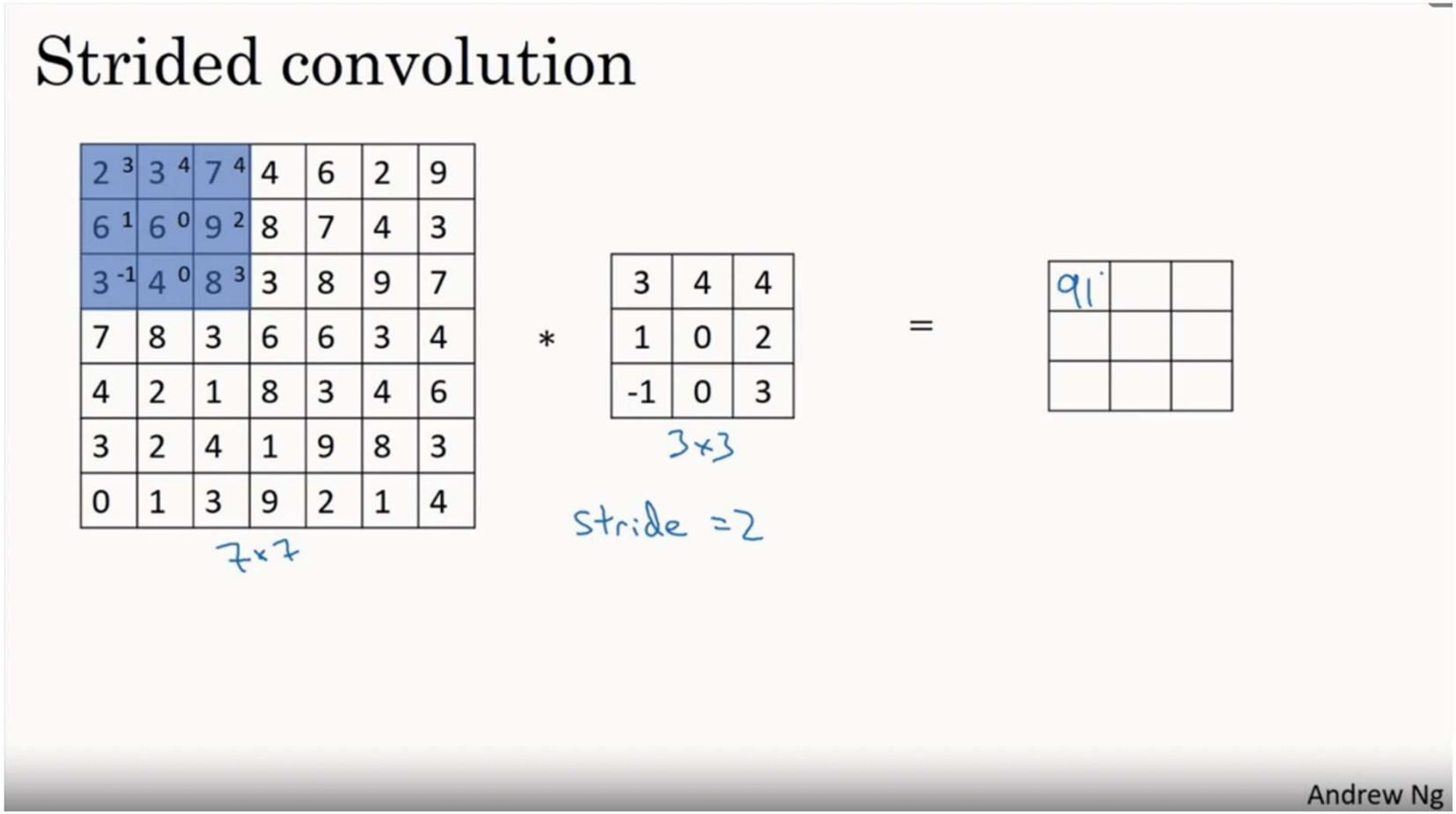
Stride defines by what step does to kernel move, for example stride of 1 makes kernel slide by one row/column at a time and stride of 2 moves kernel by 2 rows/columns.



In plain English, a stride is a step that you take when walking or running.

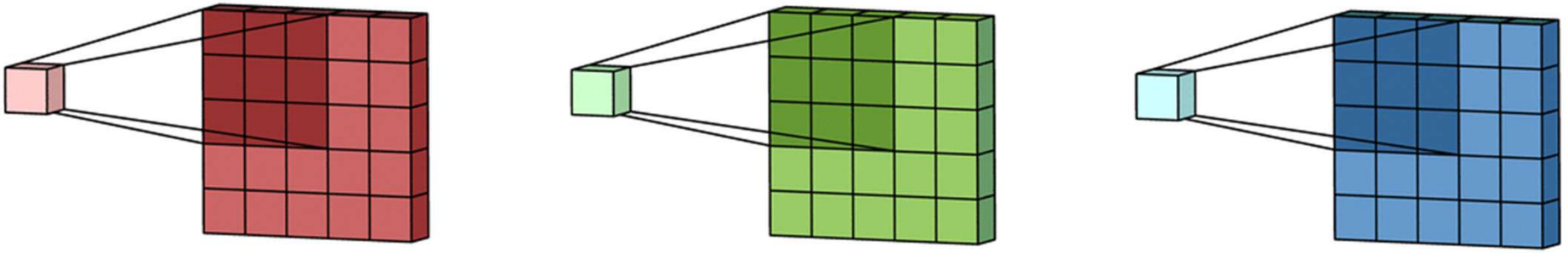
Kernel Convolution: How Blurs & Filters Work

Stride 2



Kernel Convolution: How Blurs & Filters Work

Kernel movement without padding and stride 1



Kernel Convolution

Important Concepts

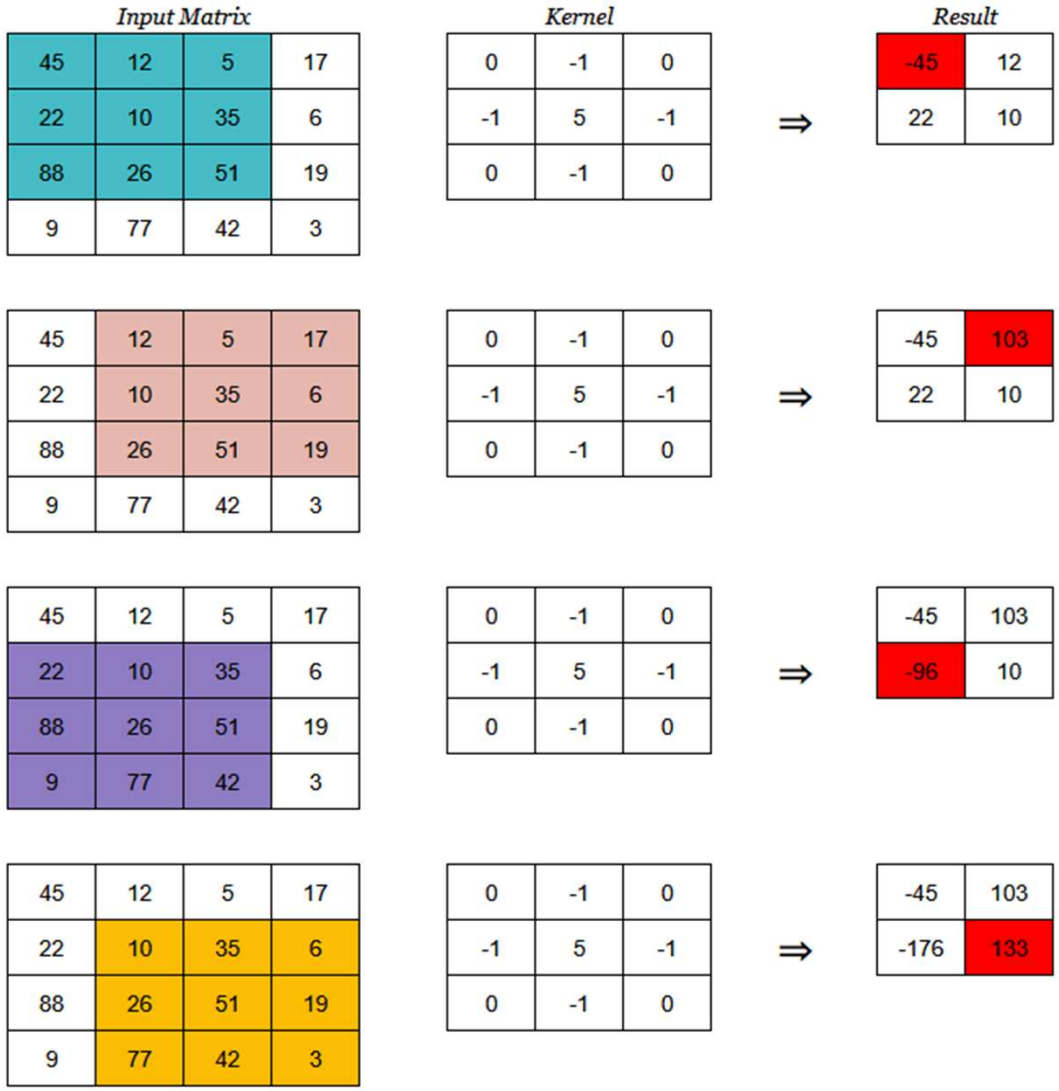
- Kernel Size: The kernel size defines the field of view of the convolution. A common choice for 2D is 3 — that is 3x3 pixels.
- Stride: The stride defines the step size of the kernel when traversing the image.
- Padding: The padding defines how the border of a sample is handled.

Kernel Convolution: How Blurs & Filters Work

A kernel is a small 2D matrix whose contents are based upon the operations to be performed. A kernel maps on the input image by simple matrix multiplication and addition, the output obtained is of lower dimensions and therefore easier to work with.

Original	*Gaussian Blur*	*Sharpen*	*Edge Detection*
$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$
			

Kernel Convolution: How Blurs & Filters Work



Kernel Convolution: How Blurs & Filters Work

HANDS ON

Blurring filter

Blurring filter

1. What is it?
   - An effect where pixels blend with their neighbors
   - Similar to an out-of-focus photo where sharp details are lost

2. How it Works
   - Uses a low-pass filter
   - Each pixel combines with surrounding pixel intensities
   - Results in an image with less detail and noise

3. Importance
   - While usually unwanted in photography
   - It's fundamental in computer vision
   - Used as a preprocessing step in many image processing tasks

4. Technical Impact
   - *Reduces high-frequency content (noise and edges)*
   - *Decreases image detail*
   - *Makes it easier to identify larger structural elements*

Blurring filter

Here are a few similar analogies to help explain the concept:

- Think of it like reading a map: When you look at a map from a distance, you lose the names of small streets and minor details, but the main highways and city boundaries become more obvious. This is exactly how blurring helps computer vision focus on major features.

- Or imagine looking at a forest: When you unfocus your eyes slightly, individual leaves and branches blur together, but the shapes of entire trees and the forest's outline become more distinct. This is similar to how *blurring helps computer vision identify larger objects by reducing distracting details.*

- Another way to think about it is like an impressionist painting: The artist doesn't paint every leaf on a tree or every brick in a building. By "blurring" these details, they actually make the overall scene more recognizable. Computer vision uses blurring in the same way to better understand the big picture.

- Think of it like aerial photography: From very high up, you can't see individual cars or people, but this actually makes it easier to identify larger structures like buildings, roads, and parks. This is similar to how blurring helps computer vision systems identify major features in an image.

Blurring filter

Four main smoothing and blurring options that you'll often use in your own projects

1. Average blurring
2. Gaussian blurring
3. Median filtering
4. Bilateral filtering

Average blurring (cv2.blur)
Takes an area of pixels surrounding a central pixel, averages all these pixels together, and replaces the central pixel with the average.

To accomplish our average blur, we'll actually be convolving our image with an M x N normalized filter where both M and N are both odd integers.

This kernel is going to slide from left-to-right and from top-to-bottom for each and every pixel in our input image. The pixel at the center of the kernel (and hence why we have to use an odd number, otherwise there would not be a true "center") is then set to be the average of all other pixels surrounding it.

Average blurring (cv2.blur)

Let's go ahead and define a 3 x 3 average kernel that can be used to blur the central pixel with a 3 pixel radius:

$$K = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

We could also define a 5 x5 average kernel:

$$K = \frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

This kernel takes more pixels into account for the average, and will blur the image more than a 3 x 3 kernel since the kernel covers more area of the image

Average blurring (cv2.blur)

Hence, this brings us to an important rule: as the size of the kernel increases, so will the amount in which the image is blurred.

HANDS ON

Gaussian blurring (cv2.GaussianBlur)

Gaussian blurring is similar to average blurring, but instead of using a simple mean, we are now using a weighted mean, where neighborhood pixels that are closer to the central pixel contribute more "weight" to the average.

The end result is that our image is less blurred, but more "naturally blurred," than using the average method discussed in the previous section. Furthermore, based on this weighting we'll be able to preserve more of the edges in our image as compared to average smoothing.

Just like an average blurring, Gaussian smoothing also uses a kernel of M x N, where both M and N are odd integers.

However, since we are weighting pixels based on how far they are from the central pixel, we need an equation to construct our kernel. The equation for a Gaussian function in one direction is:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{x^2}{2\sigma^2}}$$

Gaussian blurring (cv2.GaussianBlur)

And it then becomes trivial to extend this equation to two directions, one for the x-axis and the other for the y-axis, respectively:

$$G(x, y) = \frac{1}{2\pi\sigma} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where x and y are the respective distances to the horizontal and vertical center of the kernel and \sigma is the standard deviation of the Gaussian kernel.

when the size of our kernel increases so will the amount of blurring that is applied to our output image. However, the blurring will appear to be more “natural” and will preserve edges in our image better than simple average smoothing

Finding out the values of a Gaussian Kernel

For a filter (kernel is also known as filter) size x by y, the number of rows are x and the number of columns are y. Generally, The filter size consists of odd numbers . In this case ,the values for rows are -{(x-1)/2} to +{(x-1)/2} and the values of columns are -{(y-1)/2} to +{(y-1)/2} . So, for filter size 5x5 , the row values are -2 to +2 and column values are -2 to +2. So, if we want to calculate the 1st value of the kernel , we have to put the row value and column value of the kernel for that position in the Gaussian Function. Here , it is (-2,-2) . So, x=-2 , y=-2 . By putting the value of x, y in the equation we can find the value h(x,y) = 0.0029 which is approximately 0.003 . Following the same procedure , we can find the values of other positions . The entire table is shown below:

$$G_\sigma = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

Rows \ Columns	-2	-1	0	1	2
-2	0.003	0.013	0.022	0.013	0.003
-1	0.013	0.059	0.097	0.059	0.013
0	0.022	0.097	0.159	0.097	0.022
1	0.013	0.059	0.097	0.059	0.013
2	0.003	0.013	0.022	0.013	0.003

5×5 Gaussian kernel

Gaussian blurring (cv2.GaussianBlur)

HANDS ON

What is Thresholding?

Thresholding is a fundamental segmentation technique that converts a grayscale image into a binary image.

- Pixels are converted to either 0 (black) or 255 (white)
- Helps separate objects from the background
- Useful for edge detection and shape identification

Key Concept:
Think of thresholding as drawing a line that separates dark and light pixels.

Types of Thresholding

1. Simple Thresholding
   - Manually define a threshold value T
   - Works well under controlled lighting
   - Pixels > T become white (255)
   - Pixels ≤ T become black (0)
2. Otsu's Method
   - Automatically calculates optimal T value
   - Perfect for bimodal images
   - No manual adjustment needed

Simple Thresholding - Example

Simple thresholding process:

```
# Python with OpenCV
import cv2

# Load image in grayscale
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Apply simple threshold
T = 200  # threshold value
ret, thresh = cv2.threshold(image, T, 255, cv2.THRESH_BINARY)
```

Pixels above 200 will become white, others black

Otsu's Method - Advantages

- Automatically calculates the optimal threshold value
- Analyzes the image histogram
- Perfect for images with two distinct pixel groups
- No prior knowledge of lighting conditions needed

```
# Apply Otsu's method
ret, thresh = cv2.threshold(image, 0, 255,
                            cv2.THRESH_BINARY + cv2.THRESH_OTSU)
```

Practical Tips

- Pre-process images with slight blur
- Experiment with different threshold methods
- Consider your image lighting conditions
- For complex cases, use adaptive thresholding

Remember:

There's no perfect method for all situations. Your choice depends on your specific case.

Thresholding

HANDS ON

Real-World Applications

Common Uses:

- Document scanning and OCR
- Quality control in manufacturing
- Medical image analysis
- License plate recognition
- Fingerprint detection

Pro Tip:

Always test your thresholding method with various images under different conditions.

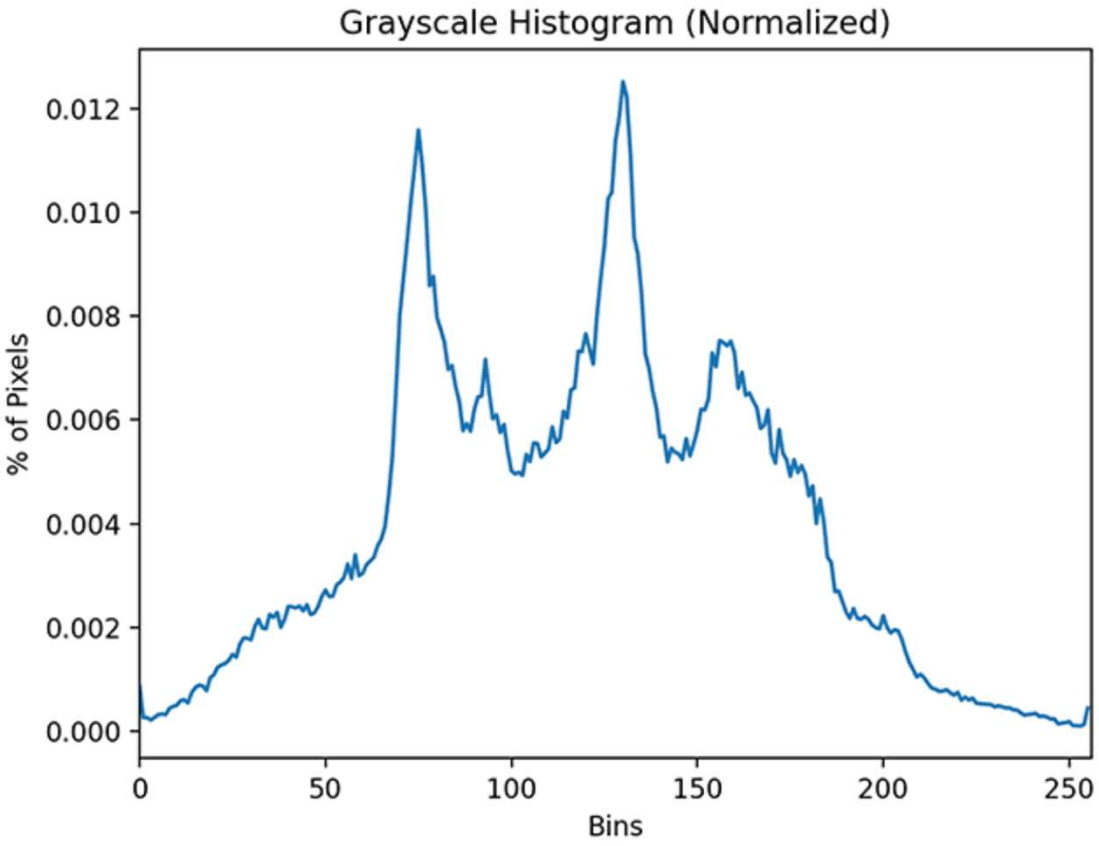
What is an image histogram?

A **histogram represents the distribution of pixel intensities** (whether color or grayscale) in an image. It can be visualized as a graph (or plot) that gives a high-level intuition of the intensity (pixel value) distribution. We are going to assume a RGB color space in this example, so these pixel values will be in the range of 0 to 255.

When plotting the histogram, the x-axis serves as our “bins.” If we construct a histogram with 256 bins, then we are effectively counting the number of times each pixel value occurs.

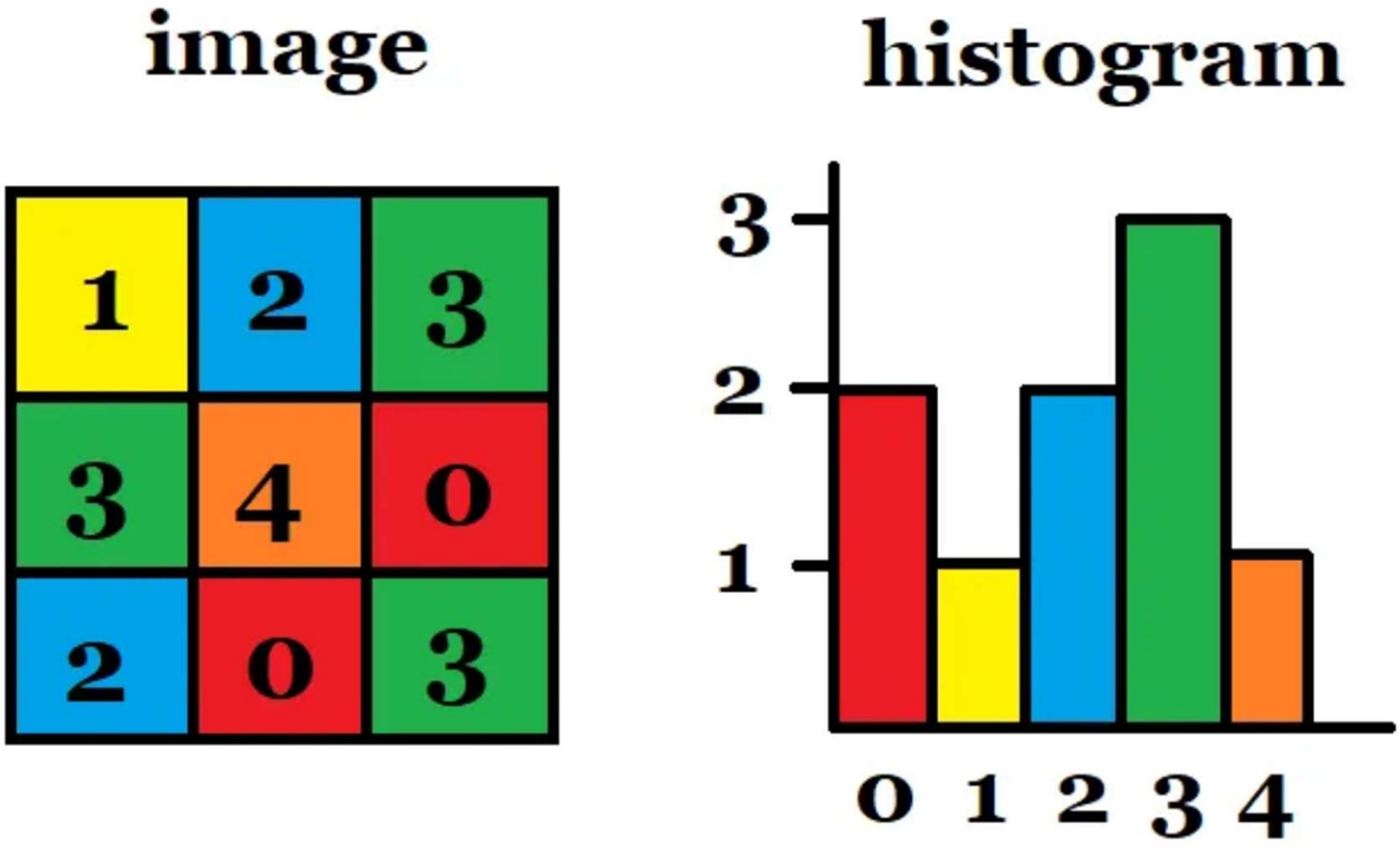
In contrast, if we use only 2 (equally spaced) bins, then we are counting the number of times a pixel is in the range [0, 128] or [128, 255].

What is an image histogram?



- we have plotted a histogram with 256 bins along the x-axis and the percentage of pixels falling into the given bins along the y-axis. Examining the histogram, note that there are three primary peaks.
- The first peak in the histogram is around x=65 where we see a sharp spike in the number of pixels — clearly there is some sort of object in the image that has a very dark value.
- We then see a much slower rising peak in the histogram, where we start to ascend around x=100 and finally end the descent around x=150. This region probably refers to a background region of the image.
- Finally, we see there is a very large number of pixels in the range x=150 to x=175. It's hard to say exactly what this region is, but it must dominate a large portion of the image.
- By simply examining the histogram of an image, you get a general understanding regarding the contrast, brightness, and intensity distribution. If this concept seems new or foreign to you, don't worry — we'll be examining more examples like this one later in this lesson.

What is an image histogram?



Using OpenCV to compute histograms

cv2.calcHist(images, channels, mask, histSize, ranges)

- images: This is the image that we want to compute a histogram for. Wrap it as a list: [myImage]
- channels: A list of indexes, where we specify the index of the channel we want to compute a histogram for. To compute a histogram of a grayscale image, the list would be [0]. To compute a histogram for all three red, green, and blue channels, the channels list would be [0, 1, 2].
- mask: Remember learning about masks in my Image Masking with OpenCV guide? Well, here we can supply a mask. If a mask is provided, a histogram will be computed for masked pixels only. If we do not have a mask or do not want to apply one, we can just provide a value of None.
- histSize: This is the number of bins we want to use when computing a histogram. Again, this is a list, one for each channel we are computing a histogram for. The bin sizes do not all have to be the same. Here is an example of 32 bins for each channel: [32, 32, 32].
- ranges: The range of possible pixel values. Normally, this is [0, 256] (that is not a typo — the ending range of the cv2.calcHist function is non-inclusive so you'll want to provide a value of 256 rather than 255) for each channel, but if you are using a color space other than RGB [such as HSV], the ranges might be different.)

Using OpenCV to compute histograms

HANDS ON

Image gradients

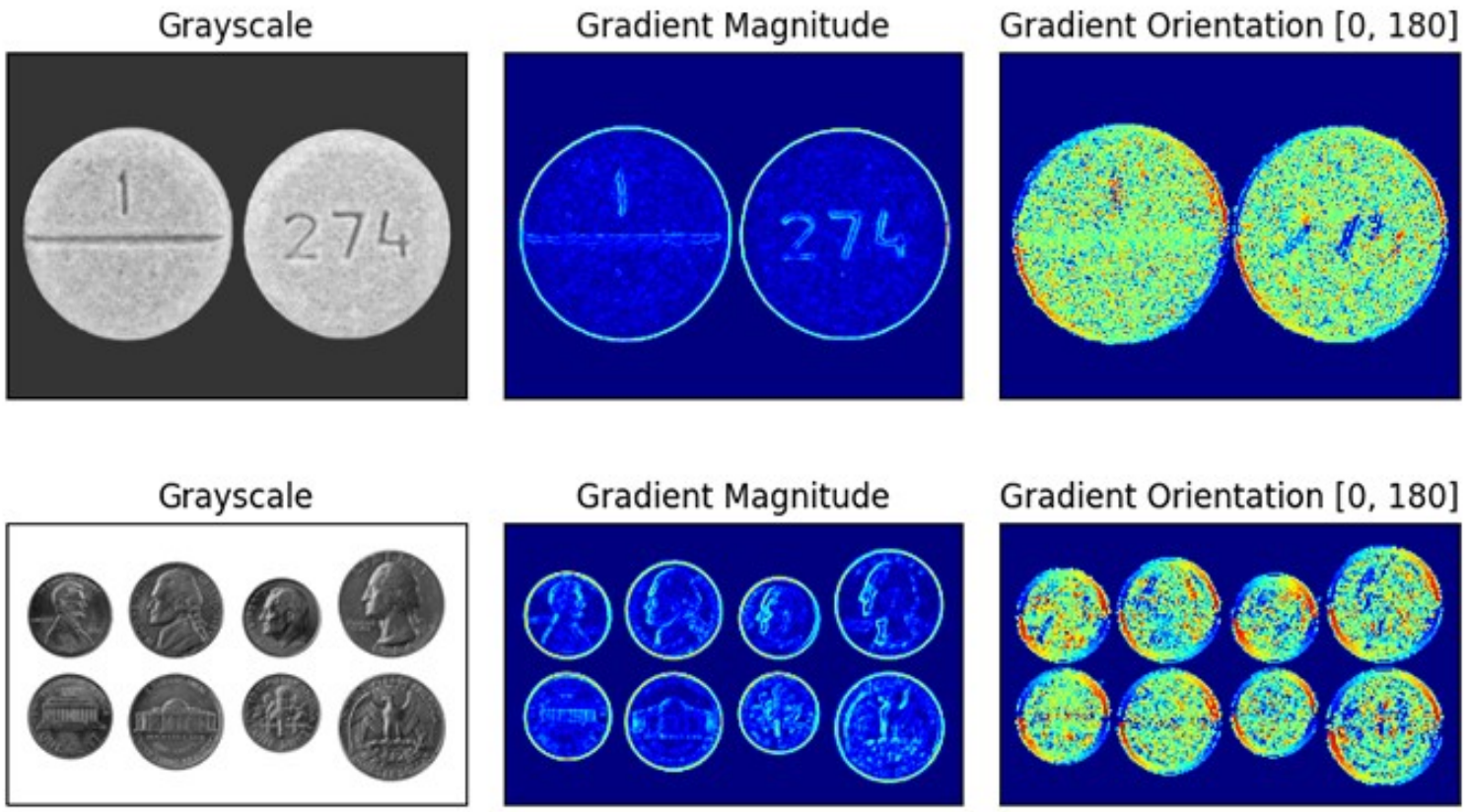


Image Gradients: Learning Objectives

- Understand the importance of image gradients in computer vision.
- Learn how to compute gradients using Sobel and Scharr operators in OpenCV.
- Explore practical applications of gradients for edge detection and feature extraction.

What Are Image Gradients?

- Definition:

Image gradients represent the intensity change of pixels in an image. They are critical for detecting edges, textures, and object boundaries.

- Interpretation:

A gradient is a vector with magnitude (strength of change) and direction (angle of change).

Computed in horizontal (x) and vertical (y) directions.

What is an Image Gradient?

A directional change in image intensity at each pixel position

- Helps detect edges in images
- Measures how quickly pixel values change
- Essential for computer vision applications

Image Gradient : The Basic Process

Computing gradients involves looking at neighboring pixels

- Examine 3x3 pixel neighborhoods
- Calculate changes in x and y directions
- Consider intensity differences between adjacent pixels

Figure 2: When computing image gradients, we need to find the difference in image pixel intensities along both the *x* and *y* direction within the given kernel size.

Image Gradient : Gradient Components

We calculate two main components:

- Gx: Horizontal change (east-west pixels)
- Gy: Vertical change (north-south pixels)
- Together they tell us the direction and strength of the change

$G_y = I(x, y + 1) - I(x, y - 1)$

$G_x = I(x + 1, y) - I(x - 1, y)$

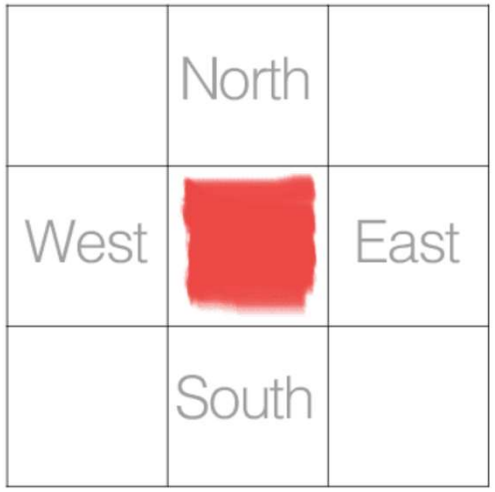


Figure 3: Labeling the north, south, east, and west pixel surrounding the center red pixel.

- **North:** $I(x, y - 1)$
- **South:** $I(x, y + 1)$
- **East:** $I(x + 1, y)$
- **West:** $I(x - 1, y)$

Image Gradient : Mathematical Formula

Key calculations:

- $G_x = I(x+1, y) - I(x-1, y)$
- $G_y = I(x, y+1) - I(x, y-1)$
- Where $I(x,y)$ represents pixel intensity at position (x,y)

Image Gradient : Final Calculations

From the components, we can determine:

- Gradient Magnitude = sqrt($Gx^2 + Gy^2$)
- Gradient Direction = arctan2(Gy, Gx)
- These values help us understand edge strength and direction

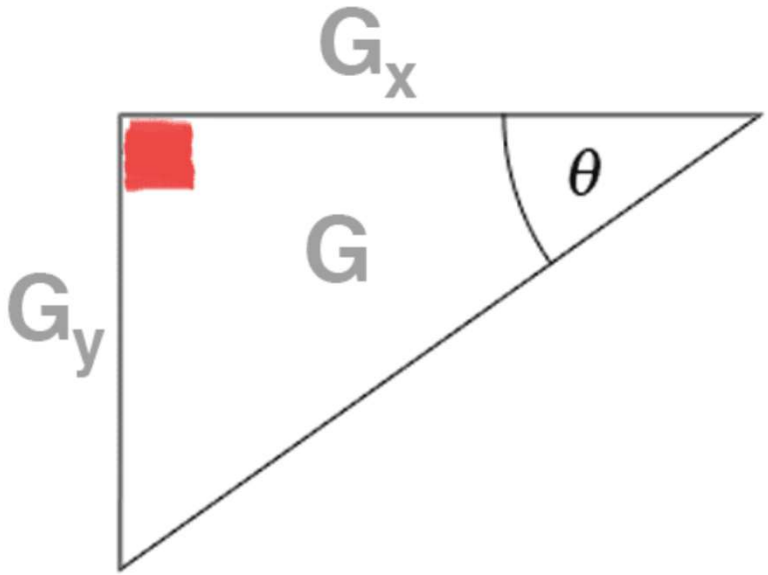


Figure 6: The basic trigonometry of computing the gradient magnitude and orientation.

Example 1

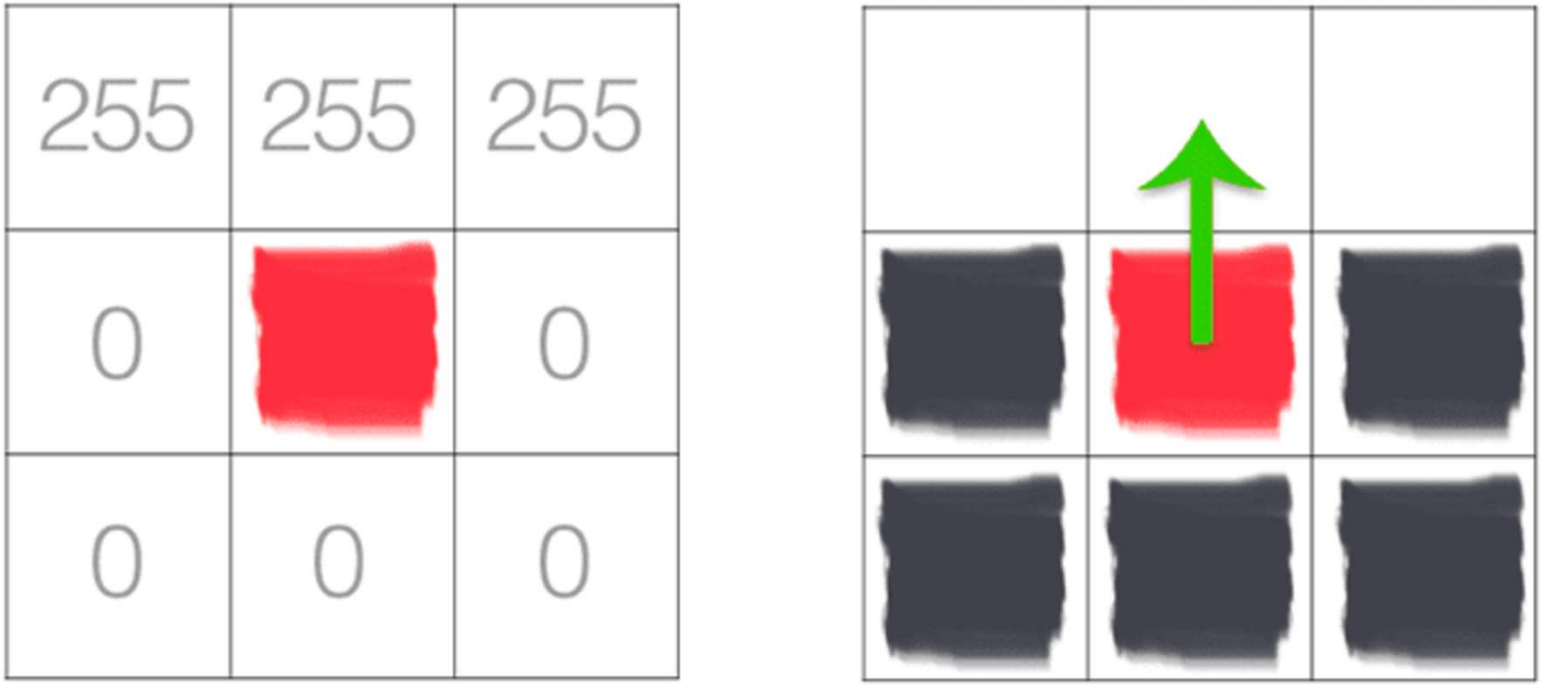


Figure 7: *Left:* The pixel values of the 8-neighborhood. *Right:* A visualization of the 90-degree gradient orientation.

$G_x = 0 - 0 = 0$ $G_y = 0 - 255 = -255$

$$G = \sqrt{0^2 + (-255)^2} = 255$$

$$\theta = \arctan2(-255, 0) \times \left(\frac{180}{\pi}\right) = -90^\circ$$

Example 2

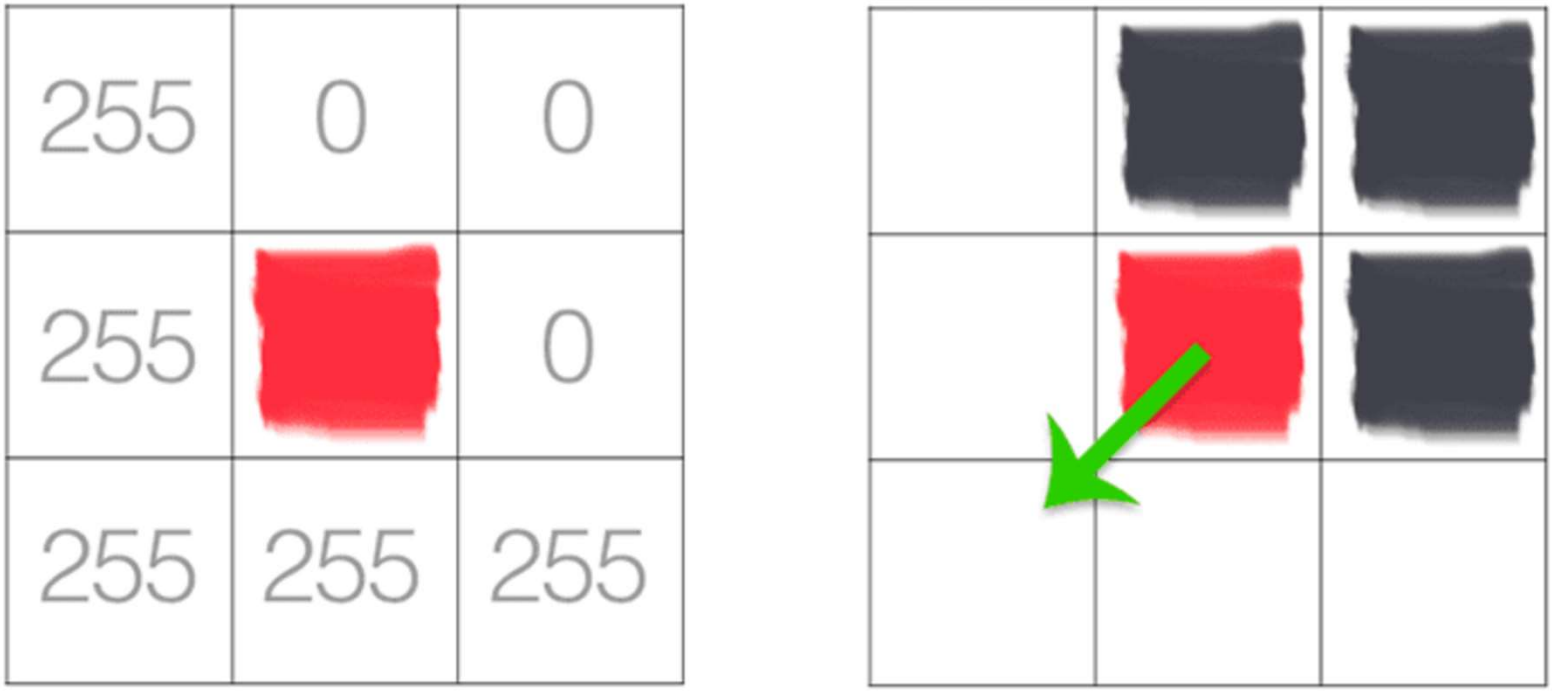


Figure 8: *Left:* Our raw pixel intensity values. *Right:* The gradient orientation of −135 degrees.

$G_x = 0 - 255 = -255$ and $G_y = 255 - 0 = 255$

$$G = \sqrt{(-255)^2 + 255^2} = 360.6 \qquad \theta = \arctan2(255, -255) \times \left(\frac{180}{\pi}\right) = 135^\circ$$

Gradient Operators: Sobel vs. Scharr

- **Sobel Operator:**

  ✓ A convolution kernel used to approximate gradients.
  ✓ Highlights edges in horizontal and vertical directions.

Kernel Example:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \quad \text{and} \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

- **Scharr Operator:**

  ✓ An improved version of Sobel, better at detecting edges at 45° and 135°.

Kernel Example:

$$G_x = \begin{bmatrix} +3 & 0 & -3 \\ +10 & 0 & -10 \\ +3 & 0 & -3 \end{bmatrix} \quad \text{and} \quad G_y = \begin{bmatrix} +3 & +10 & +3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix}$$

Gradient Operators: Sobel

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

Sobel Operator. Equation by author in LaTeX.

Images don't have perfect horizontal or vertical edges, so we need to compute the total magnitude of the gradient. This can done as follows:

$$G = \sqrt{G_x^2 + G_y^2}$$

Magnitude of gradient. Equation by author in LaTeX.

We can also calculate the angle of the edge through the use of arctan:

$$\Theta = \arctan\left(\frac{G_y}{G_x}\right)$$

Angle of edge. Equation by author in LaTeX.

It's important to note that the Sobel operator is only really used for grayscale images as it measures changes in intensity.

Implementing Scharr and Sobel Filters with OpenCV

opencv_sobel_scharr.py

Image Gradients with OpenCV (Sobel and Scharr)

```
19.  # set the kernel size, depending on whether we are using the Sobel
20.  # filter or the Scharr operator, then compute the gradients along
21.  # the x and y axis, respectively
22.  ksize = -1 if args["scharr"] > 0 else 3
23.  gX = cv2.Sobel(gray, ddepth=cv2.CV_32F, dx=1, dy=0, ksize=ksize)
24.  gY = cv2.Sobel(gray, ddepth=cv2.CV_32F, dx=0, dy=1, ksize=ksize)
25.
26.  # the gradient magnitude images are now of the floating point data
27.  # type, so we need to take care to convert them back a to unsigned
28.  # 8-bit integer representation so other OpenCV functions can operate
29.  # on them and visualize them
30.  gX = cv2.convertScaleAbs(gX)
31.  gY = cv2.convertScaleAbs(gY)
32.
33.  # combine the gradient representations into a single image
34.  combined = cv2.addWeighted(gX, 0.5, gY, 0.5, 0)
35.
36.  # show our output images
37.  cv2.imshow("Sobel/Scharr X", gX)
38.  cv2.imshow("Sobel/Scharr Y", gY)
39.  cv2.imshow("Sobel/Scharr Combined", combined)
40.  cv2.waitKey(0)
```

Computing both the Gx and Gy values is handled on Lines 23 and 24 by making a call to cv2.Sobel. Specifying a value of dx=1 and dy=0 indicates that we want to compute the gradient across the x direction. And supplying a value of dx=0 and dy=1 indicates that we want to compute the gradient across the y direction.

Implementing Scharr and Sobel Filters with OpenCV

Note: In the event that we want to use the Scharr filter instead of the Sobel filter we simply specify our --scharr command line argument to be > 0 — from there the appropriate ksize is set (Line 22).

However, at this point, both gX and gY are now of the floating point data type. If we want to visualize them on our screen we need to convert them back to 8-bit unsigned integers. Lines 30 and 31 take the absolute value of the gradient images and then squish the values back into the range [0, 255].

Finally, we combine gX and gY into a single image using the cv2.addWeighted function, weighting each gradient representation equally.

Lines 37-40 then show our output images on our screen.

Implementing Scharr and Sobel Filters with OpenCV

HANDS ON

What is Edge Detection?

- A fundamental technique in computer vision
- Helps identify boundaries in images
- Used in object detection, image segmentation, and pattern recognition

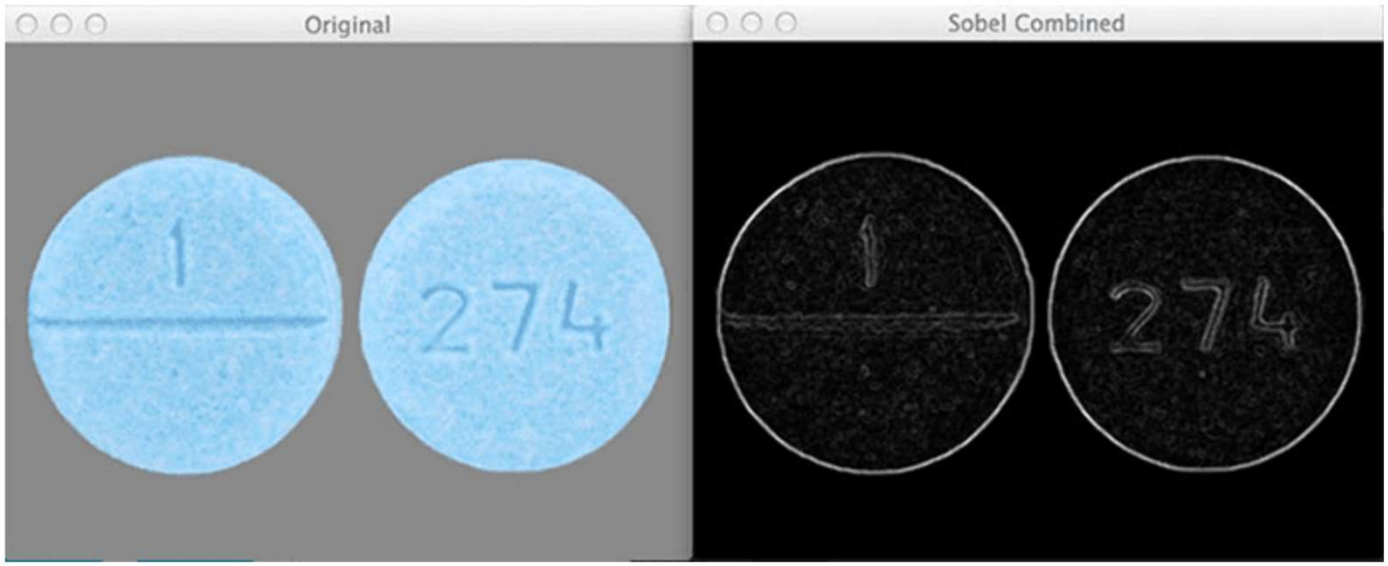


Figure 1: *Left:* Our original input image. *Right:* Computing the image gradient for the image on the left. Notice how the outline of the pills is revealed, but we're also detecting "noise" inside the pill themselves.

Why Do We Need Edge Detection?

- Reduces image data while preserving structural properties
- Helps understand image content
- Essential for many computer vision applications

Line segments detected from an Image

Types of Edges We Can Find

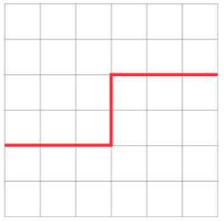


Figure 3: An example of a step edge, as if we are taking the next step on a flight of stairs.

1. Step Edge
   - Sudden change in intensity
   - Like climbing stairs - abrupt change
   - Easy to detect
2. Ramp Edge
   - Gradual change in intensity
   - Like going up a slope
   - More challenging to detect
3. Ridge Edge
   - Two ramp edges back-to-back
   - Like climbing up and down a mountain
   - Has a plateau at the top
4. Roof Edge
   - Similar to ridge but with no plateau
   - Like a mountain peak
   - Sharp at the top

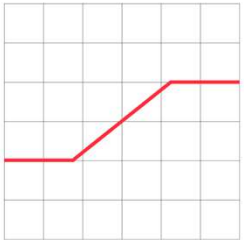


Figure 4: An example of a ramp edge, as if we are driving our car up a steep hill.

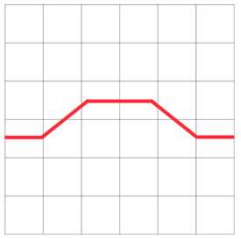


Figure 5: A ridge edge is like a ramp edge, only as our car gets to the top of the hill, it drives along the plateau a bit, before driving back down the hill.

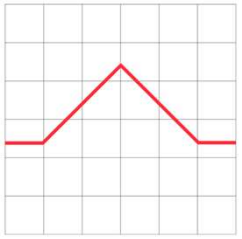


Figure 6: In a roof edge, there is no plateau at the top. We simply drive to the top of the hill and immediately head back down.

The Canny Edge Detection Process

Four main steps:

1. **Gaussian Smoothing**
   - Reduces image noise
   - Blurs the image slightly
   - Prepares image for edge detection
2. **Gradient Calculation**
   - Finds intensity changes
   - Calculates magnitude and direction
   - Uses Sobel operators
3. **Non-Maximum Suppression**
   - Thins the edges
   - Keeps only the strongest edges
   - Makes edges one pixel wide
4. **Hysteresis Thresholding**
   - Uses two thresholds (high and low)
   - Strong edges: Above high threshold
   - Weak edges: Between thresholds
   - Connects weak edges to strong ones

The Canny Edge Detection Process

Step 1: Gaussian smoothing

Smoothing an image allows us to ignore much of the detail and instead focus on the actual structure (Explained in previous slide)

Step 2: Gradient magnitude and orientation

Now that we have a smoothed image, we can compute the gradient orientation and magnitude (Explained in previous slide)

The Canny Edge Detection Process

Step 3: Non-maxima suppression

it's simply an edge thinning process

After computing our gradient magnitude representation, the edges themselves are still quite noisy and blurred, but in reality there should only be one edge response for a given region, not a whole clump of pixels reporting themselves as edges.

To remedy this, we can apply edge thinning using non-maxima suppression. To apply non-maxima suppression we need to examine the gradient magnitude G and orientation \theta at each pixel in the image and:

The Canny Edge Detection Process

Step 3: Non-maxima suppression

To apply non-maxima suppression we need to examine the gradient magnitude G and orientation \theta at each pixel in the image and:

- Compare the current pixel to the 3 x 3 neighborhood surrounding it
- Determine in which direction the orientation is pointing:
  - If it's pointing towards the north or south, then examine the north and south magnitude
  - If the orientation is pointing towards the east or west, then examine the east and west pixels
  - If the center pixel magnitude is greater than both the pixels it is being compared to, then preserve the magnitude; otherwise, discard it

The Canny Edge Detection Process

Step 3: Non-maxima suppression

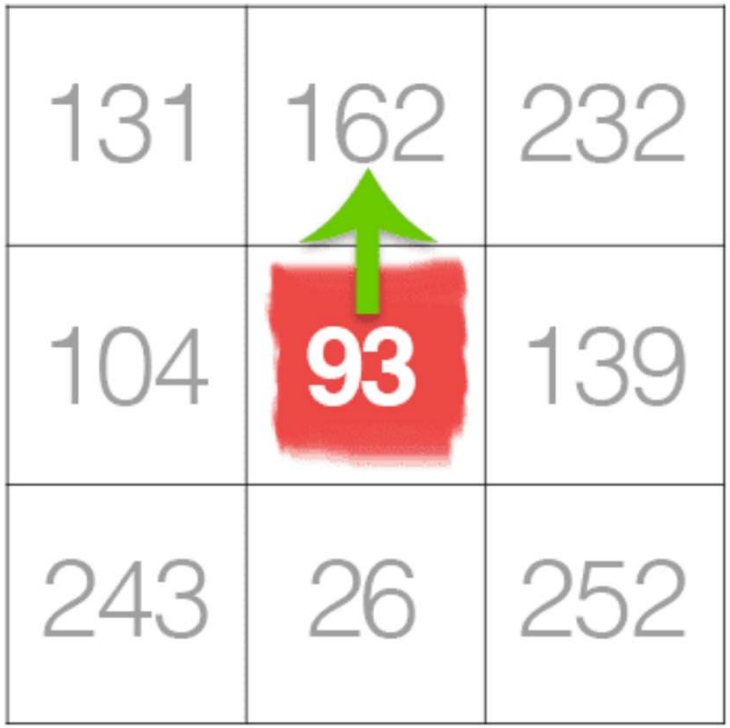


Figure 7: Let's pretend the gradient orientation for this image is 90 degrees. Applying non-maxima suppression, we would need to examine the north and south pixels and take the maximum value — which is 162.

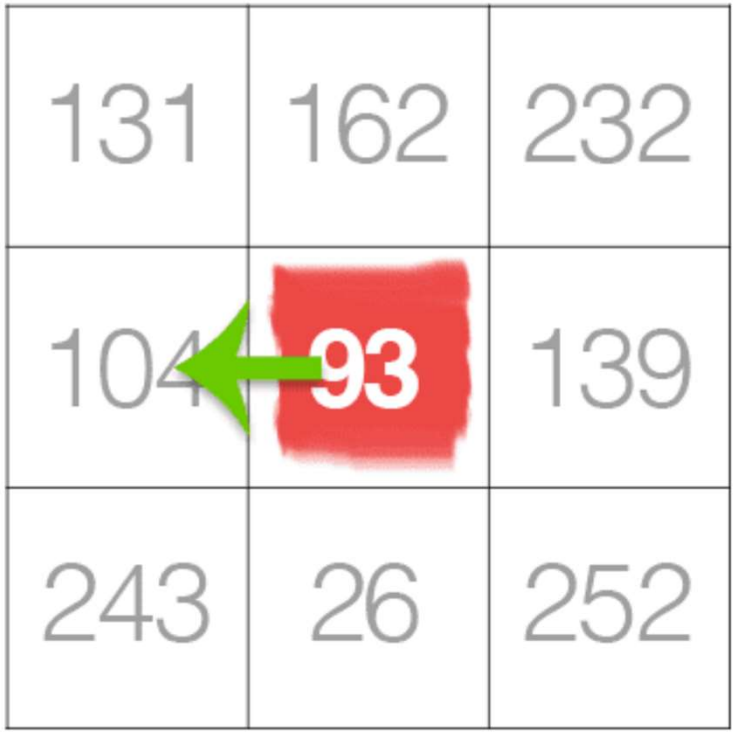


Figure 8: Applying non-maxima suppression in this image ends in us discarding the 93 and keeping the 104 and 139 values.

The Canny Edge Detection Process

Step 4: Hysteresis thresholding

Even after applying non-maxima suppression, we may need to remove regions of an image that are not technically edges, but still respond as edges after computing the gradient magnitude and applying non-maximum suppression.

To ignore these regions of an image, we need to define two thresholds:

T_{upper} T_{lower}.

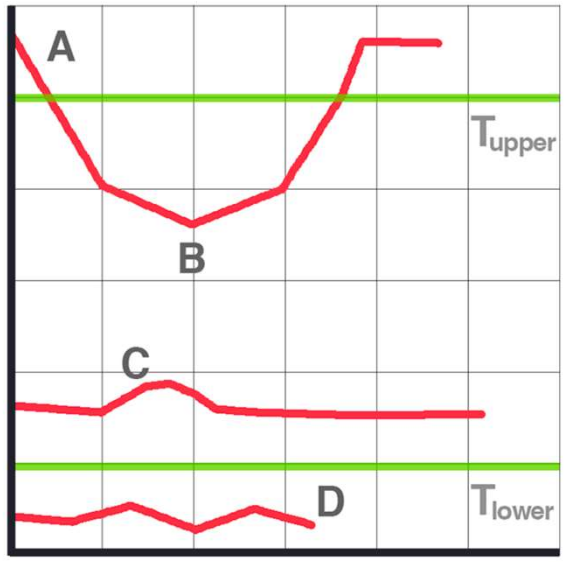


Figure 9: An example of applying hysteresis to a set of edges.

The Canny Edge Detection Process

HANDS ON

THE END