



Week 2

DEEP LEARNING FOR COMPUTER VISION

Presented by **Asst. Prof. Dr. Tuchsanai Ploysuwan**



Understanding Images

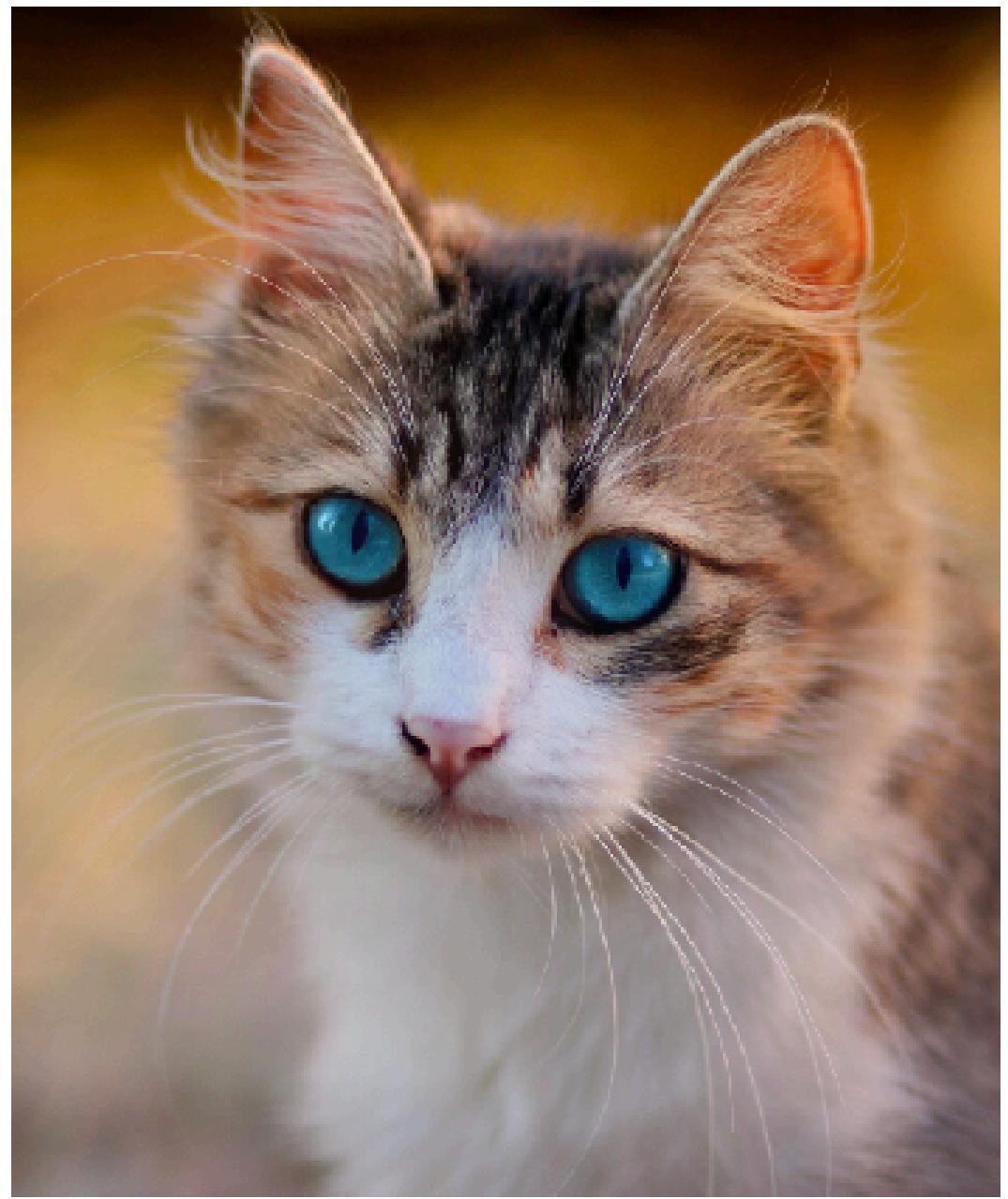
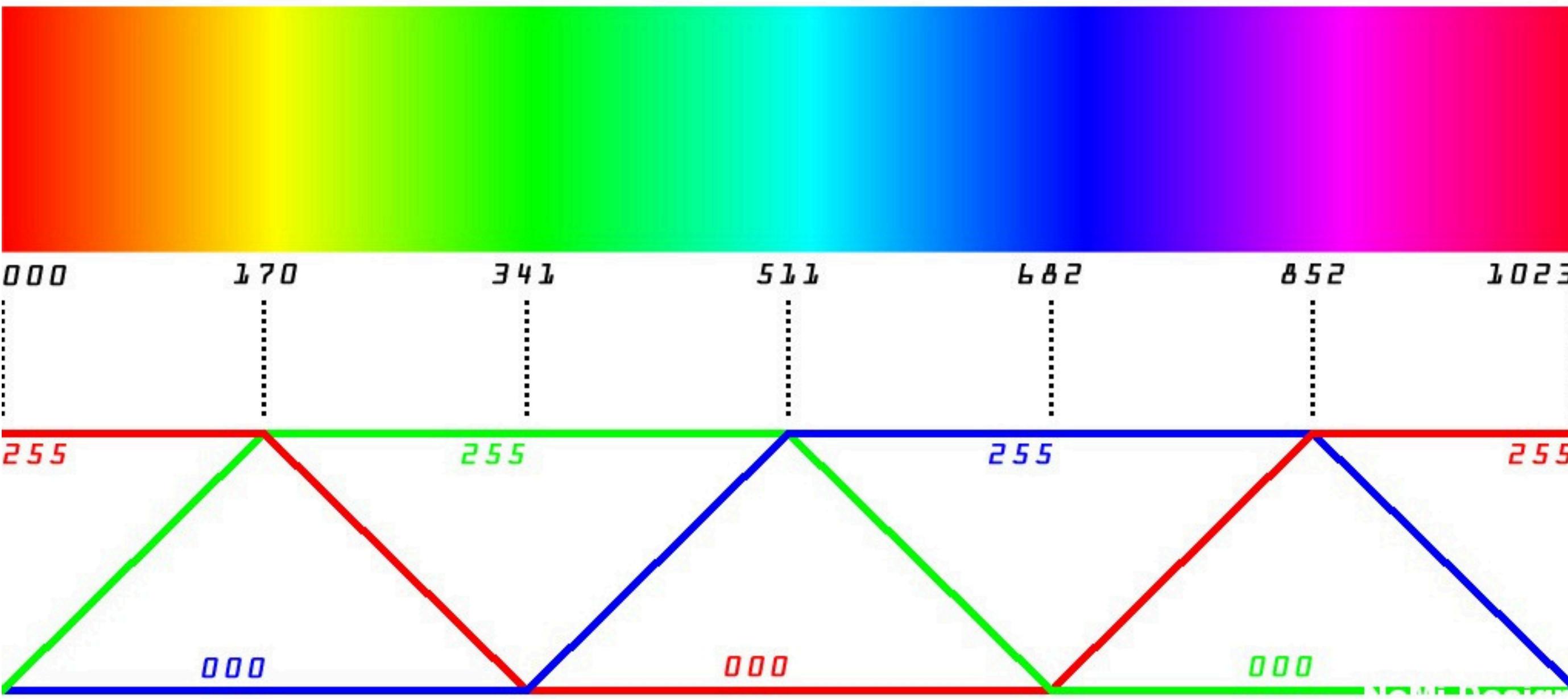


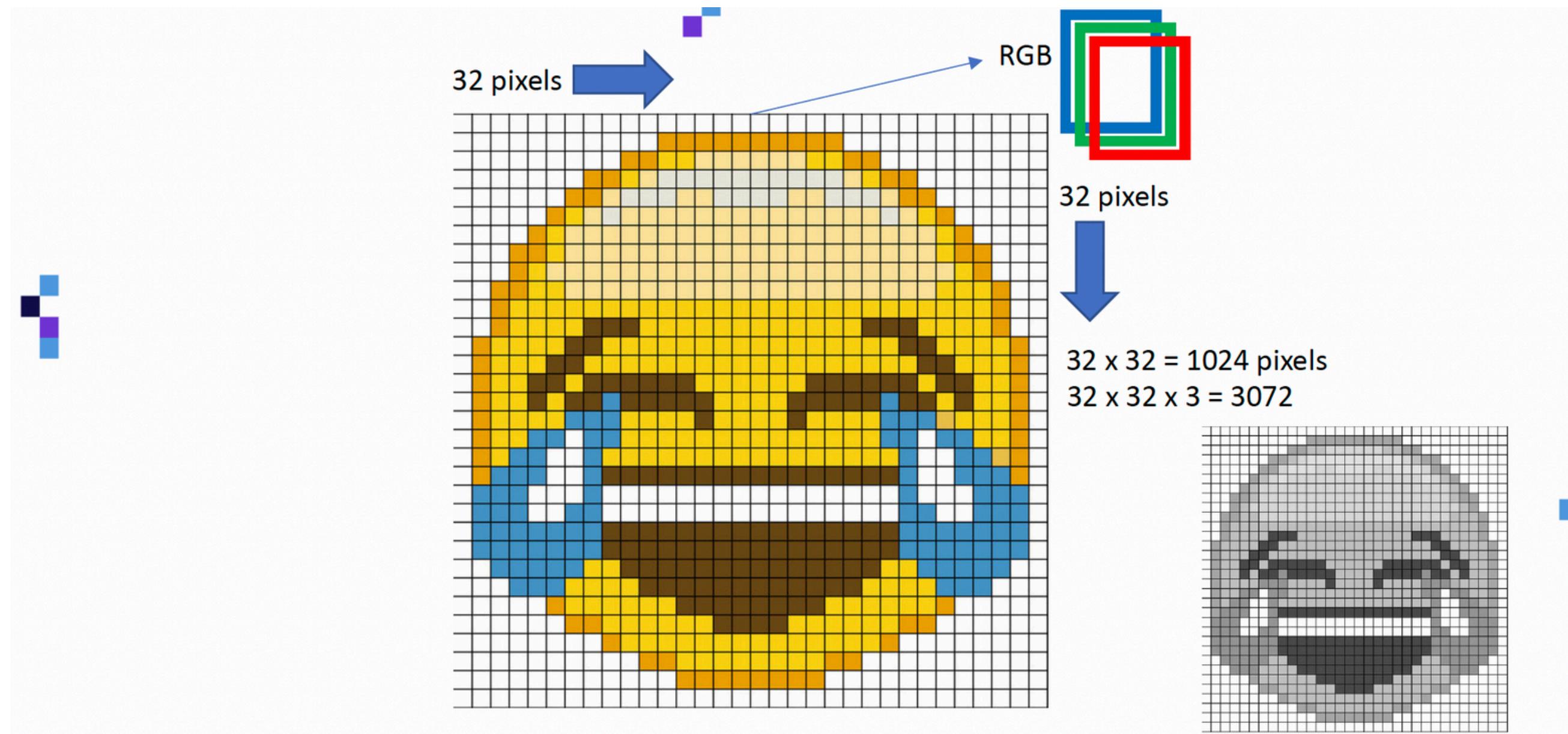
Image Basics



How do Computers ‘see’ Images?



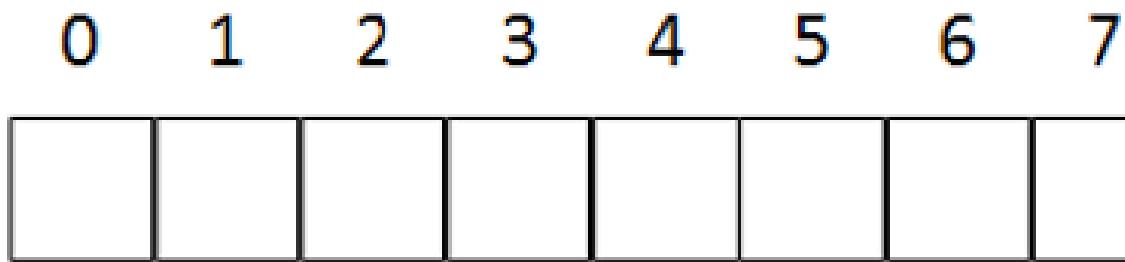
Pixels



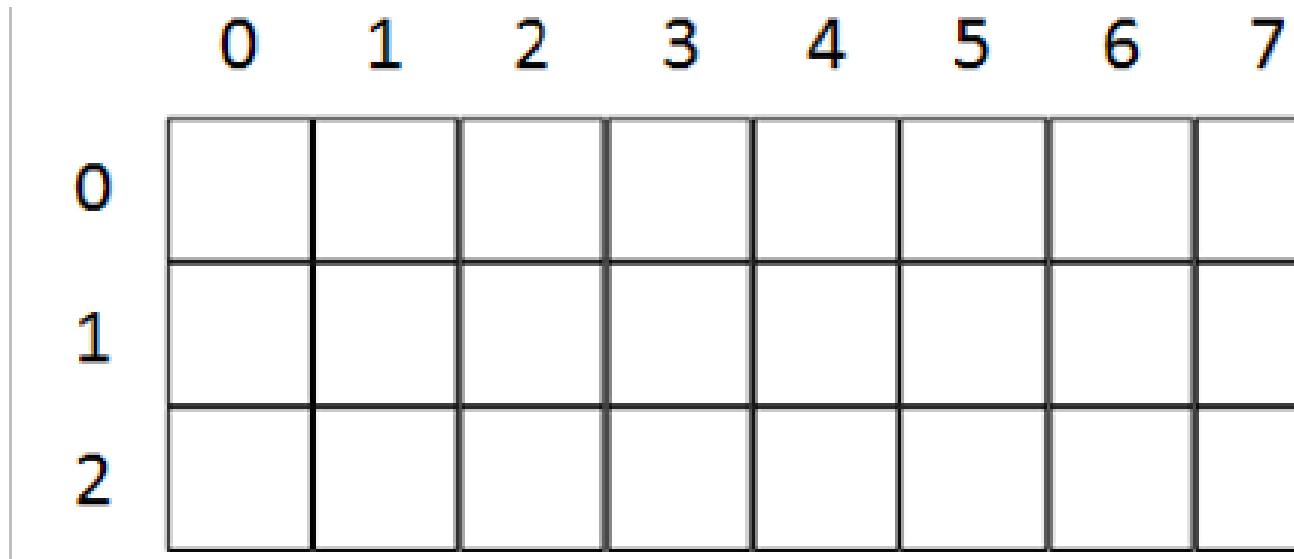
Digital Images Format

Images are stored in Multi-Dimensional Arrays

- A 1-Dimensional array looks like this



- A 2-Dimensional array looks like this



- A 3-Dimensional array looks like this

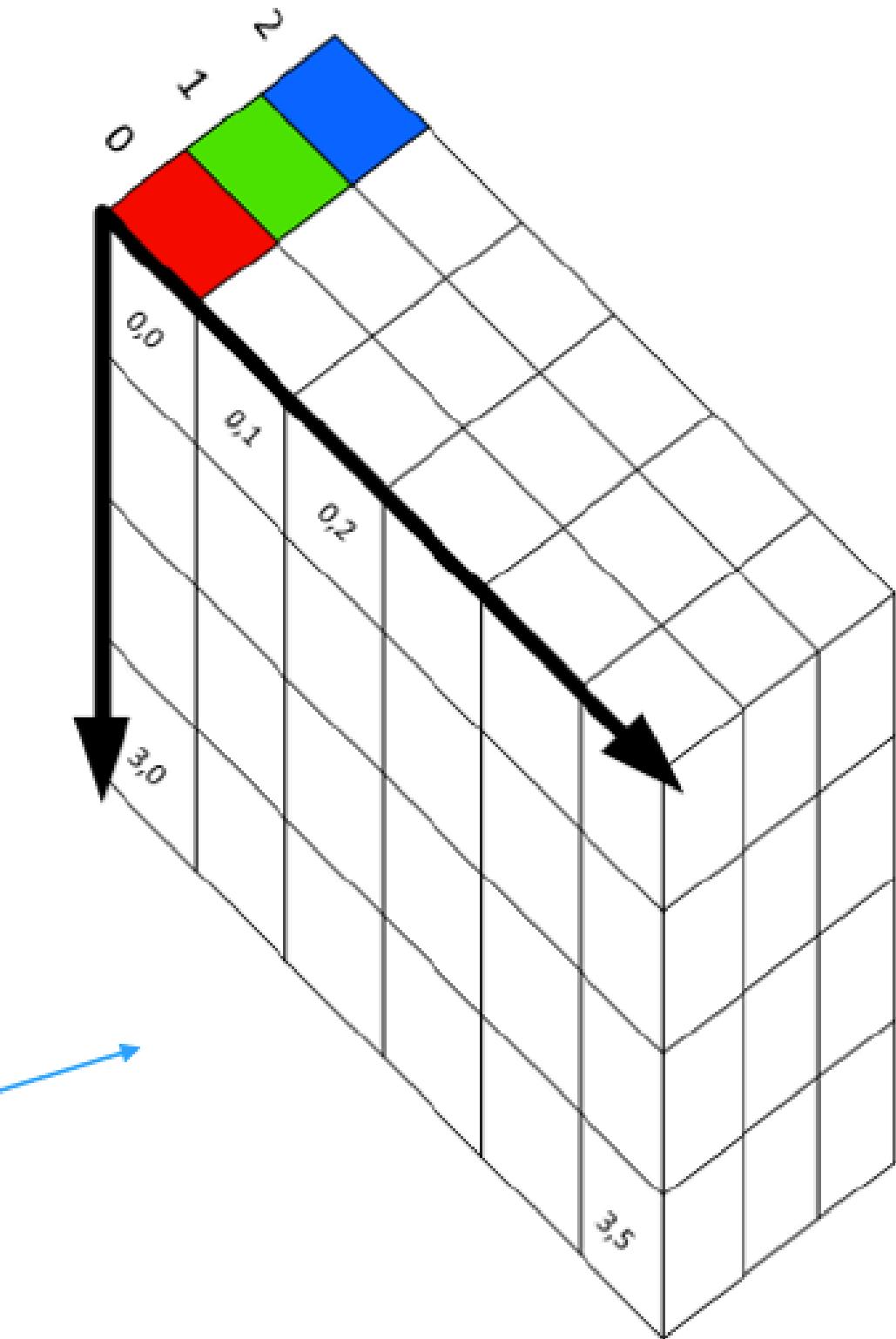
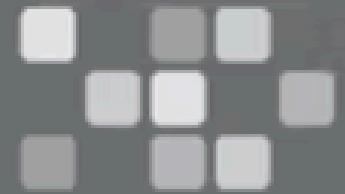


Image File Formats



Raster images

Pixel-based graphics
Resolution dependent
Photos & web graphics

JPG

Web & print
photos and
quick previews

GIF

Animation &
transparency in
limited colors

PNG

Transparency
with millions
of colors

TIFF

High quality
print graphics
and scans

RAW

Unprocessed
data from
digital cameras

PSD

Layered Adobe
Photoshop
design files



Vector images

Curve-based graphics
Resolution independent
Logos, icons, & type

PDF

Print files and
web-based
documents

EPS

Individual
vector design
elements

AI

Original Adobe
Illustrator
design files

SVG

Vector files
for web
publishing

OpenCV Basics: Reading & Displaying Images

Installation

```
# Install OpenCV using pip  
!pip install opencv-python
```

Importing OpenCV

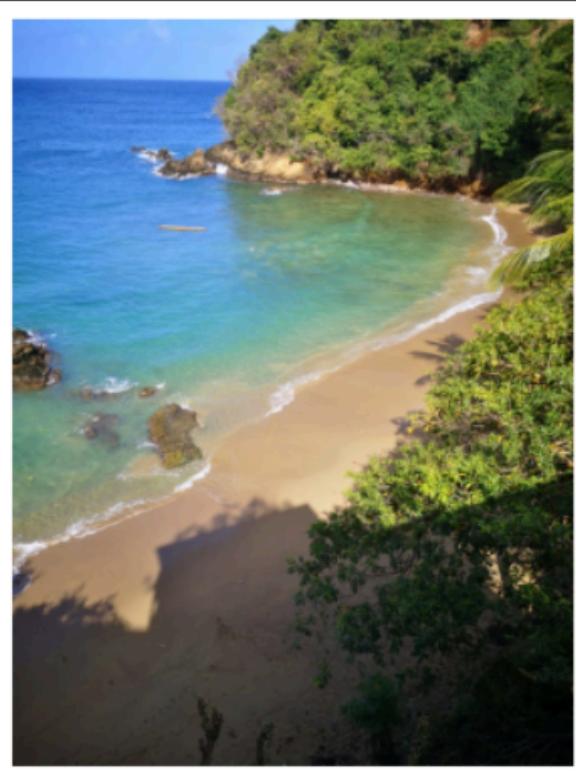
```
import cv2  
import numpy as np  
print(cv2.__version__)  
✓ 2.8s
```

Loading Images

```
# Load an image using 'imread' specifying the path to image  
image = cv2.imread('./images/castara.jpeg')  
✓ 0.0s
```

Displaying Images

```
from matplotlib import pyplot as plt  
  
# Show the image with matplotlib without axis  
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))  
plt.axis('off')  
plt.show()  
✓ 0.1s
```



Saving Images

```
cv2.imwrite('output.jpg', image)
```

True

```
cv2.imwrite('output.png', image)
```

OpenCV — Resize an Image

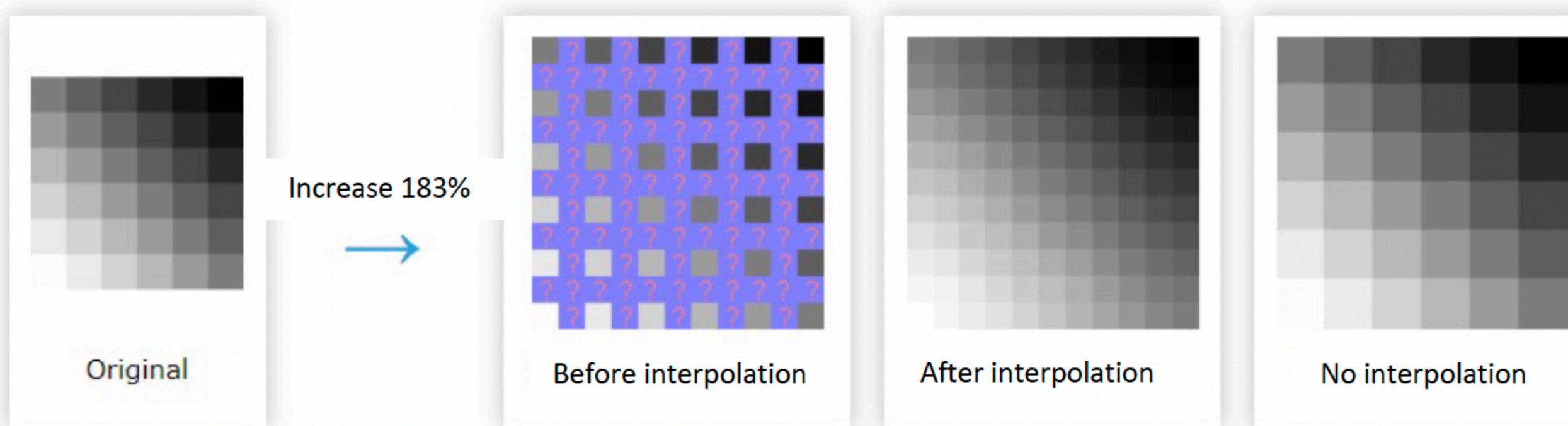
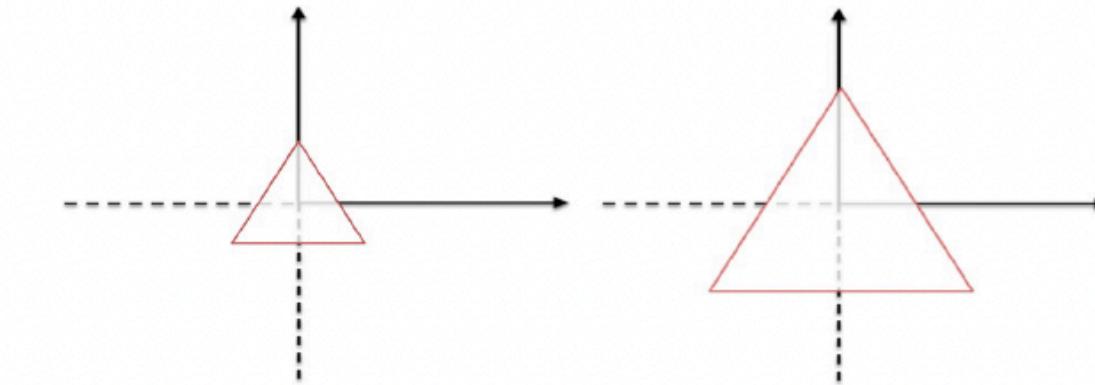
$$\begin{bmatrix} x_T \\ y_T \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

Scale factor:

>1 increase

<1 decrease

$s_x = s_y$ uniform scale factor (no distortion)



OpenCV — Resize an Image

```
# load our input image
image = cv2.imread('./images/Taj_Mahal.jpg')
print("width : {} pixels, Height : {} pixels".format(image.shape[1], image.shape[0]))
```

```
width : 796 pixels, Height : 712 pixels
```

```
# Let's make our image 3/4 of its original size
image_scaled = cv2.resize(image, None, fx=0.75, fy=0.75)
imshow("0.75x Scaling - Linear Interpolation", image_scaled)

# Let's double the size of our image
img_scaled2 = cv2.resize(image, None, fx=2, fy=2, interpolation=cv2.INTER_CUBIC)
imshow("2x Scaling - Inter Cubic", img_scaled2)

# Let's double the size of our image using inter_nearest interpolation
img_scaled3 = cv2.resize(image, None, fx=2, fy=2, interpolation=cv2.INTER_NEAREST)
imshow("2x Scaling - Inter Nearest", img_scaled3)

# Let's skew the re-sizing by setting exact dimensions
img_scaled4 = cv2.resize(image, (900, 400), interpolation=cv2.INTER_AREA)
imshow("Scaling - Inter Area", img_scaled4)
```

OpenCV options

- **INTER_NEAREST** - a nearest neighbor interpolation. It is widely used because it is the fastest
- **INTER_LINEAR** - a bilinear interpolation (it's used by default), generally good for zooming in and out of images
- **INTER_AREA** - uses the pixel area ratio. May be a preferred method for image reduction as it provides good results
- **INTER_CUBIC** - bicubic (4x4 neighboring pixels). It has better results
- **INTER_LANCZOS4** - Lanczos interpolation (8x8 neighboring pixels). Among these algorithms, it is the one with the best quality results

Comparison between interpolation algorithms – increasing the size

original (50x50)



area (400x400)



nearest (400x400)



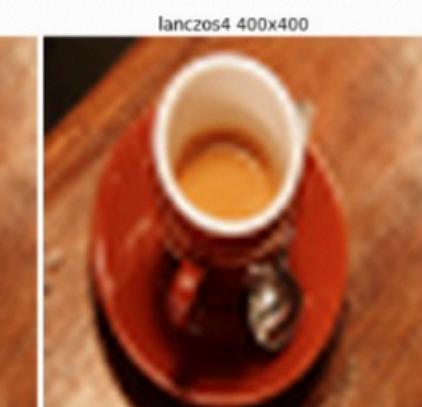
linear (400x400)



cubic (400x400)



lanczos4 (400x400)



original (400x400)



area (50x50)



nearest (50x50)



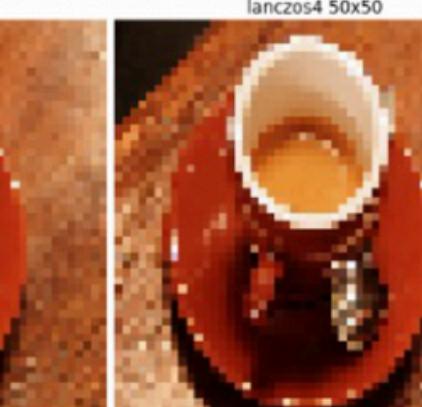
linear (50x50)



cubic (50x50)



lanczos4 (50x50)



Region-based segmentation

Let us first determine markers of the coins and the background. These markers are pixels that we can label unambiguously as either object or background. Here, the markers are found at the two extreme parts of the histogram of grey values:

```
>>> markers = np.zeros_like(coins)
```

Region-based segmentation

Let us first determine markers of the coins and the background. These markers are pixels that we can label unambiguously as either object or background. Here, the markers are found at the two extreme parts of the histogram of grey values:

```
>>> markers = np.zeros_like(coins)
```

Region-based segmentation

Let us first determine markers of the coins and the background. These markers are pixels that we can label unambiguously as either object or background. Here, the markers are found at the two extreme parts of the histogram of grey values:

```
>>> markers = np.zeros_like(coins)
```

Region-based segmentation

Let us first determine markers of the coins and the background. These markers are pixels that we can label unambiguously as either object or background. Here, the markers are found at the two extreme parts of the histogram of grey values:

```
>>> markers = np.zeros_like(coins)
```

Region-based segmentation

Let us first determine markers of the coins and the background. These markers are pixels that we can label unambiguously as either object or background. Here, the markers are found at the two extreme parts of the histogram of grey values:

```
>>> markers = np.zeros_like(coins)
```

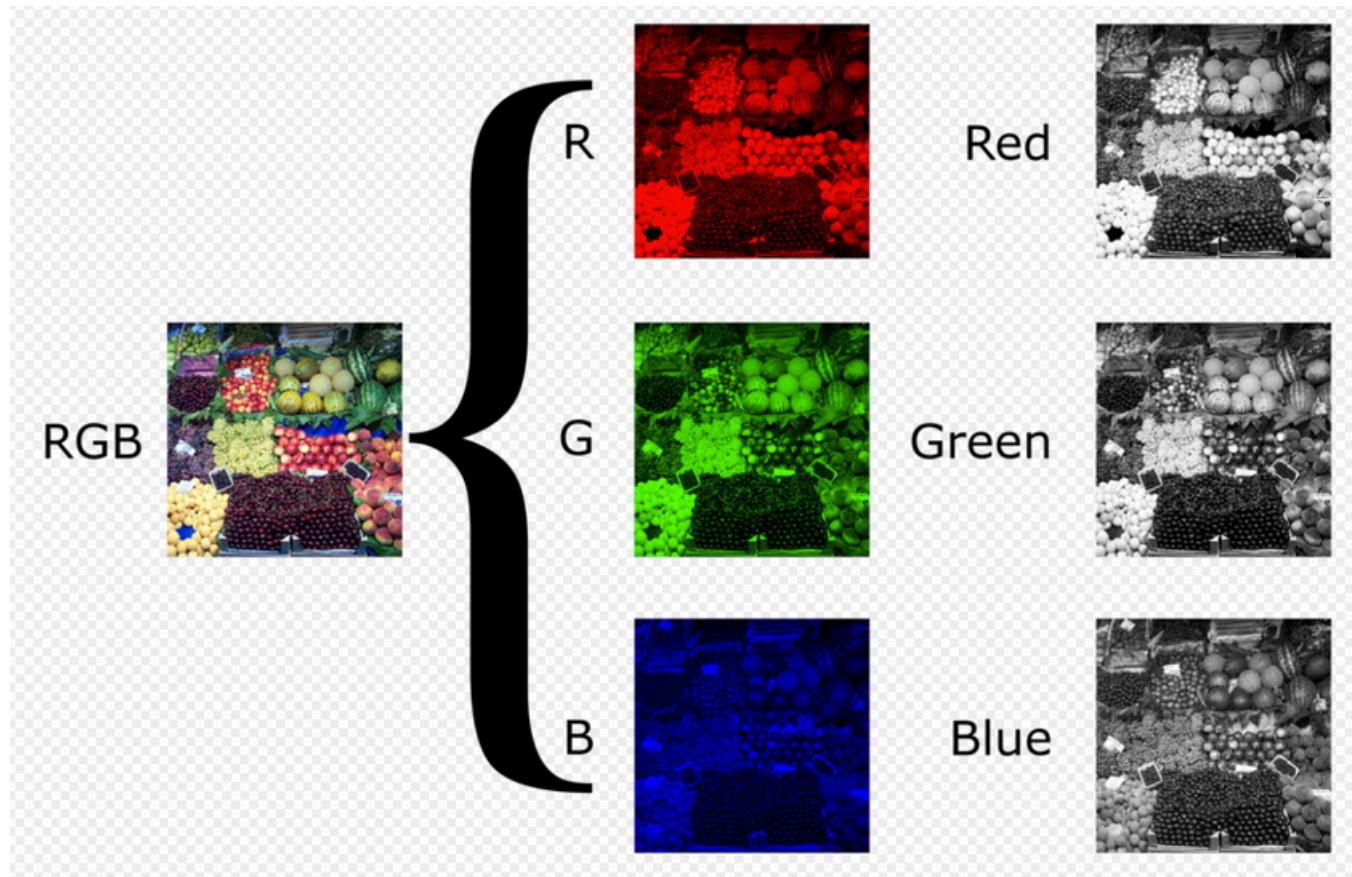
Region-based segmentation

Let us first determine markers of the coins and the background. These markers are pixels that we can label unambiguously as either object or background. Here, the markers are found at the two extreme parts of the histogram of grey values:

```
>>> markers = np.zeros_like(coins)
```

Grayscale Image

Sometimes referred to as black and white image



Luma coding in video systems [edit]

Main article: luma (video)

For images in color spaces such as Y'UV and its relatives, which are used in standard color TV and video systems such as PAL, SECAM, and NTSC, a nonlinear luma component (Y') is calculated directly from gamma-compressed primary intensities as a weighted sum, which, although not a perfect representation of the colorimetric luminance, can be calculated more quickly without the gamma expansion and compression used in photometric/colorimetric calculations. In the Y'UV and Y'IQ models used by PAL and NTSC, the rec601 luma (Y') component is computed as

$$Y' \equiv 0.299R' + 0.587G' + 0.114B'$$

where we use the prime to distinguish these nonlinear values from the sRGB nonlinear values (discussed above) which use a somewhat different gamma compression formula, and from the linear RGB components. The [ITU-R BT.709](#) standard used for [HDTV](#) developed by the [ATSC](#) uses different color coefficients, computing the luma component as

$$Y' \equiv 0.2126R' + 0.7152G' + 0.0722B'.$$

Although these are numerically the same coefficients used in sRGB above, the effect is different because here they are being applied directly to gamma-compressed values rather than to the linearized values. The [ITU-R BT.2100](#) standard for [HDR](#) television uses yet different coefficients, computing the luma component as

$$Y' \equiv 0.2627R' + 0.6780G' + 0.0593B'.$$

The Formula

$$\text{Grayscale} = 0.299R + 0.587G + 0.114B$$

Where:

- R = Red channel value (0-255)
- G = Green channel value (0-255)
- B = Blue channel value (0-255)

Why These Weights?

- Green (0.587): Human eyes are most sensitive to green
- Red (0.299): Second most impactful color
- Blue (0.114): Least impact on perceived brightness

Example Calculation

For a pink pixel:

R = 255
G = 192
B = 203

Calculation:

$$\begin{aligned}(0.299 \times 255) &+ \\(0.587 \times 192) &+ \\(0.114 \times 203) &= 76.245 + 112.704 + 23.142 \\&= 212 \text{ (rounded)}\end{aligned}$$

Key Points

- Result is rounded to nearest integer
- Output range is 0 (black) to 255 (white)
- Same value applied to all RGB channels

Grayscaling Images

```
import cv2
from matplotlib import pyplot as plt
✓ 0.4s
```

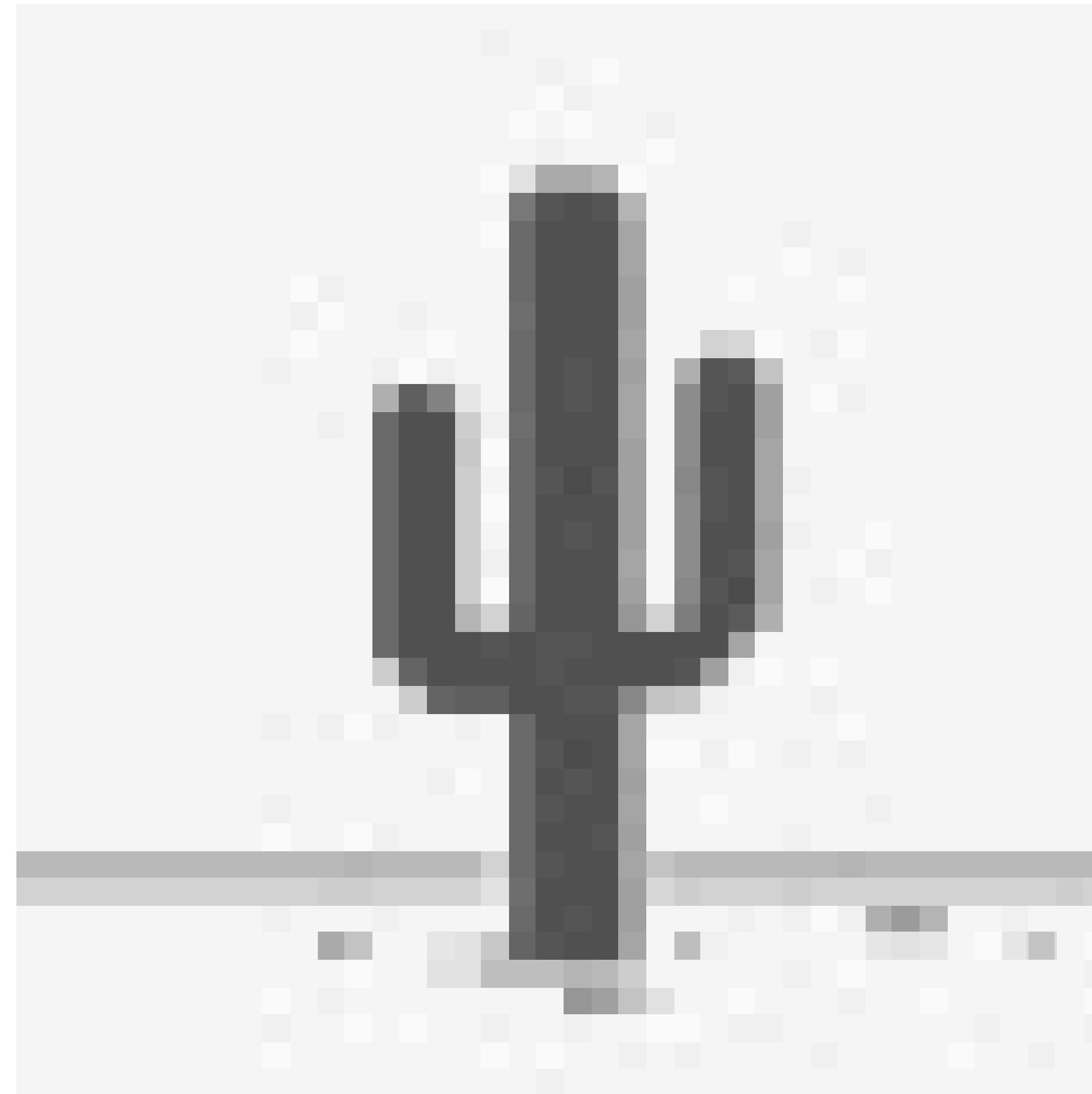
```
# Load our input image
image = cv2.imread('./images/castara.jpeg')
image.shape
✓ 0.0s
(1280, 960, 3)
```

```
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
plt.imshow(cv2.cvtColor(gray_image, cv2.COLOR_GRAY2RGB))
plt.axis('off')
plt.title('Grayscale Image')
plt.show()
```

```
✓ 0.1s
```



Images & Kernel Convolutions



Images & Kernel Convolutions

Images & Kernel Convolutions

Kernel Convolutions

Kernel is an MxN matrix

3x3 Kernel Example

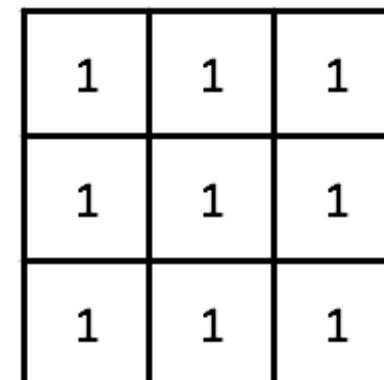
1	1	1
1	1	1
1	1	1

Kernel Convolutions

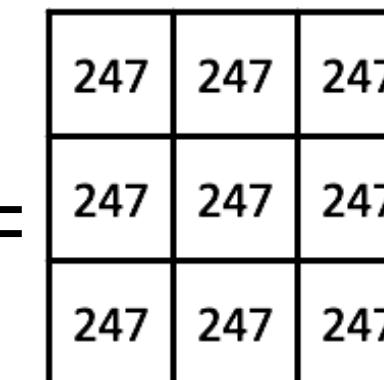
Window



Kernel



Multiplied



Sum = 9

Sum = 2223

New Pixel Value

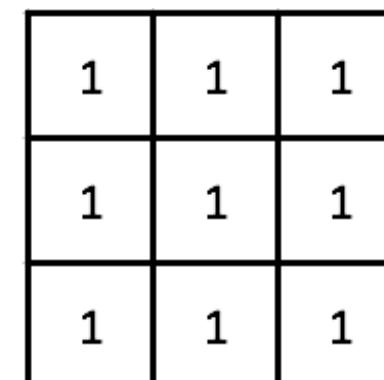
$$2223 / 9 = 247$$

Kernel Convolutions

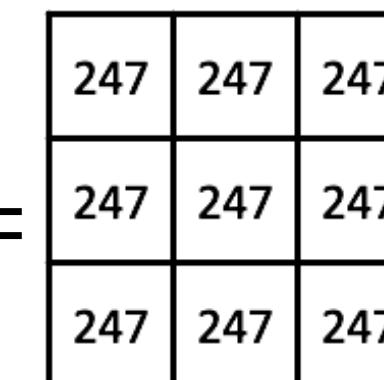
Window



Kernel



Multipled



$$\text{Sum} = 9$$

Sym = 2223

New Pixel Value

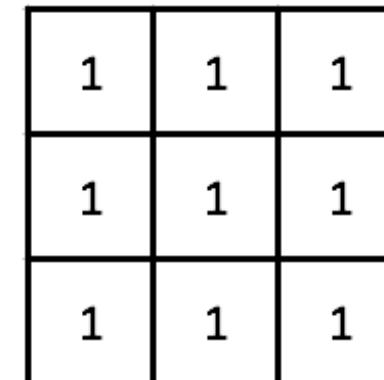
$$2223 / 9 = 247$$

Kernel Convolutions

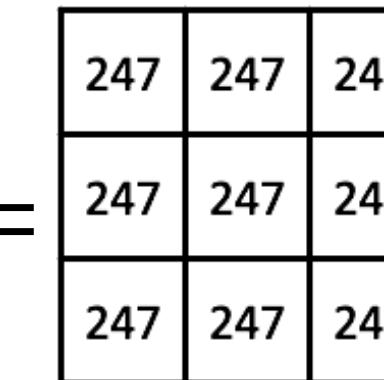
Window



Kernel



Multiplied



Sum = 9

Sum = 2223

New Pixel
Value
 $223 / 9 = 24$

Kernel Convolutions

Windows

7	247	120
7	247	83
7	247	83

X

Kernel

1	1	1
1	1	1
1	1	1

Multiplied

247	247	120
247	247	83
247	247	83

$$\text{Sum} = 9$$

Sum = 1768

New Pixel
Value
 $68 / 9 = 19$
196

Kernel Convolutions

Window

247	120	95
247	83	83
247	83	83

x

Kernel

1	1	1
1	1	1
1	1	1

Sum = 9

Multipled

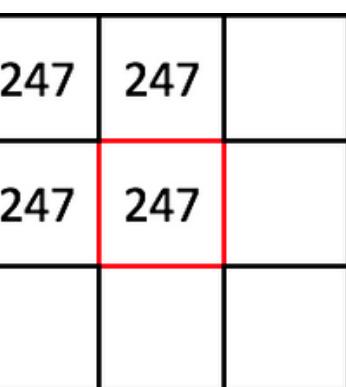
247	120	95
247	83	83
247	83	83

Sum = 1288

New Pixel
Value
 $88 / 9 = 14$
143

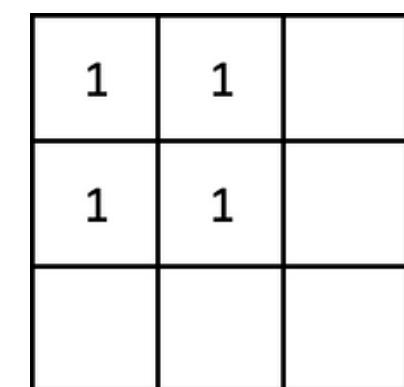
Kernel Convolutions

Window



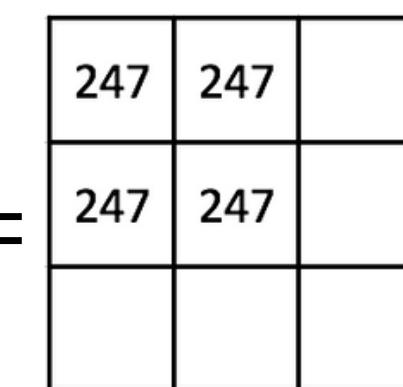
x

Kernel



1

Multiplied



Sum = 4

Sum = 988

New Pixel
Value
 $98 / 4 = 24$
247

Kernel Convolutions

Kernel

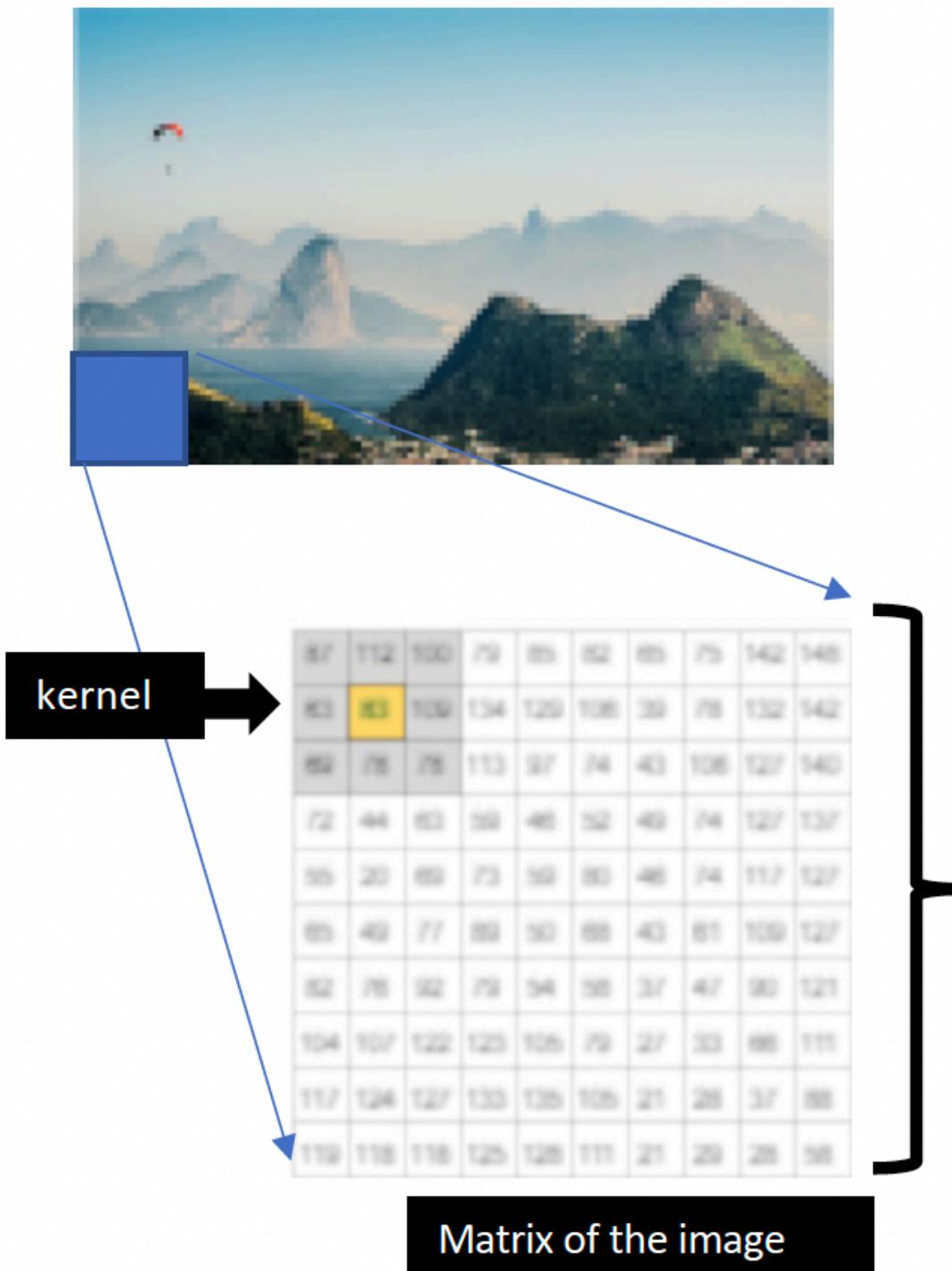
1	1	1
1	1	1
1	1	1



247	247	247	247	247	247	247	247	247	247	247	247	247	247	247	247	247	247	247	247
247	247	247	247	247	247	247	247	247	247	247	247	247	247	247	247	247	247	247	247
247	247	247	247	247	247	247	247	247	233	216	216	230	247	247	247	247	247	247	247
247	247	247	247	247	247	247	247	247	215	180	178	211	246	247	247	247	247	247	247
247	247	247	247	247	247	247	247	247	196	143	141	192	245	247	247	247	247	247	247
247	247	247	247	247	247	247	247	247	192	138	135	182	231	234	241	247	247	247	247
247	247	247	247	247	242	224	224	174	138	135	164	196	198	224	247	247	247	247	247
247	247	247	247	236	199	199	156	138	135	146	160	163	207	247	247	247	247	247	247
247	247	247	247	229	175	175	138	138	135	135	138	141	195	247	247	247	247	247	247
247	247	247	247	228	173	173	138	138	135	135	138	141	195	247	247	247	247	247	247
247	247	247	247	228	173	161	125	125	123	123	134	149	203	247	247	247	247	247	247
247	247	247	247	234	181	151	109	107	109	118	146	175	221	247	247	247	247	247	247
247	247	247	247	241	206	175	127	107	109	136	182	210	238	247	247	247	247	247	247
247	247	247	247	247	231	212	158	119	121	166	221	238	247	247	247	247	247	247	247
247	247	247	247	247	247	247	247	192	138	135	190	244	247	247	247	247	247	247	247
247	247	247	247	247	247	247	247	192	138	135	190	244	247	247	247	245	245	246	247
247	247	247	247	246	246	246	240	188	137	141	191	242	245	247	245	245	243	244	242
247	247	247	247	246	246	246	240	207	171	175	206	242	245	247	245	245	243	244	242
247	247	247	247	246	246	246	240	225	207	212	225	243	245	247	247	247	247	244	242
247	247	247	247	247	247	247	247	247	247	242	239	239	244	247	247	247	247	247	247

CONVOLUTION

- Convolution is a mathematical operation performed on two matrices, which produces a third matrix that is the result of the operation
- The primary matrix is the image to be treated, and the treatment of this matrix of the original image is done by another matrix called “kernel”, or mask
- Depending on the kernel values it is possible to obtain filters of different types, such as blur and sharpen



Kernel Convolutions

Mean Blur

Kerne

1	1	1
1	1	1
1	1	1



247	247	247	247	247	247	247	247	247	247	247	247	247	247	247	247	247	247	247
247	247	247	247	247	247	247	247	247	247	247	247	247	247	247	247	247	247	247
247	247	247	247	247	247	247	247	247	233	216	216	230	247	247	247	247	247	247
247	247	247	247	247	247	247	247	247	215	180	178	211	246	247	247	247	247	247
247	247	247	247	247	247	247	247	247	196	143	141	192	245	247	247	247	247	247
247	247	247	247	247	247	247	247	247	192	138	135	182	231	234	241	247	247	247
247	247	247	247	247	247	247	247	247	242	224	224	174	138	135	164	196	198	224
247	247	247	247	247	247	247	247	247	236	199	199	156	138	135	146	160	163	207
247	247	247	247	247	247	247	247	247	229	175	175	138	138	135	135	138	141	195
247	247	247	247	247	247	247	247	247	228	173	173	138	138	135	135	138	141	195
247	247	247	247	247	247	247	247	247	228	173	161	125	125	123	123	134	149	203
247	247	247	247	247	247	247	247	247	234	181	151	109	107	109	118	146	175	221
247	247	247	247	247	247	247	247	247	241	206	175	127	107	109	136	182	210	238
247	247	247	247	247	247	247	247	247	231	212	158	119	121	166	221	238	247	247
247	247	247	247	247	247	247	247	247	192	138	135	190	244	247	247	247	247	247
247	247	247	247	247	247	247	247	247	192	138	135	190	244	247	247	245	245	246
247	247	247	247	247	246	246	246	240	188	137	141	191	242	245	247	245	245	243
247	247	247	247	247	246	246	246	240	207	171	175	206	242	245	247	245	245	244
247	247	247	247	247	246	246	246	240	225	207	212	225	243	245	247	247	247	242
247	247	247	247	247	247	247	247	247	246	239	239	244	247	247	247	247	247	247

Kernel Convolutions

Blur

Kernel

1	1	1
1	1	1
1	1	1

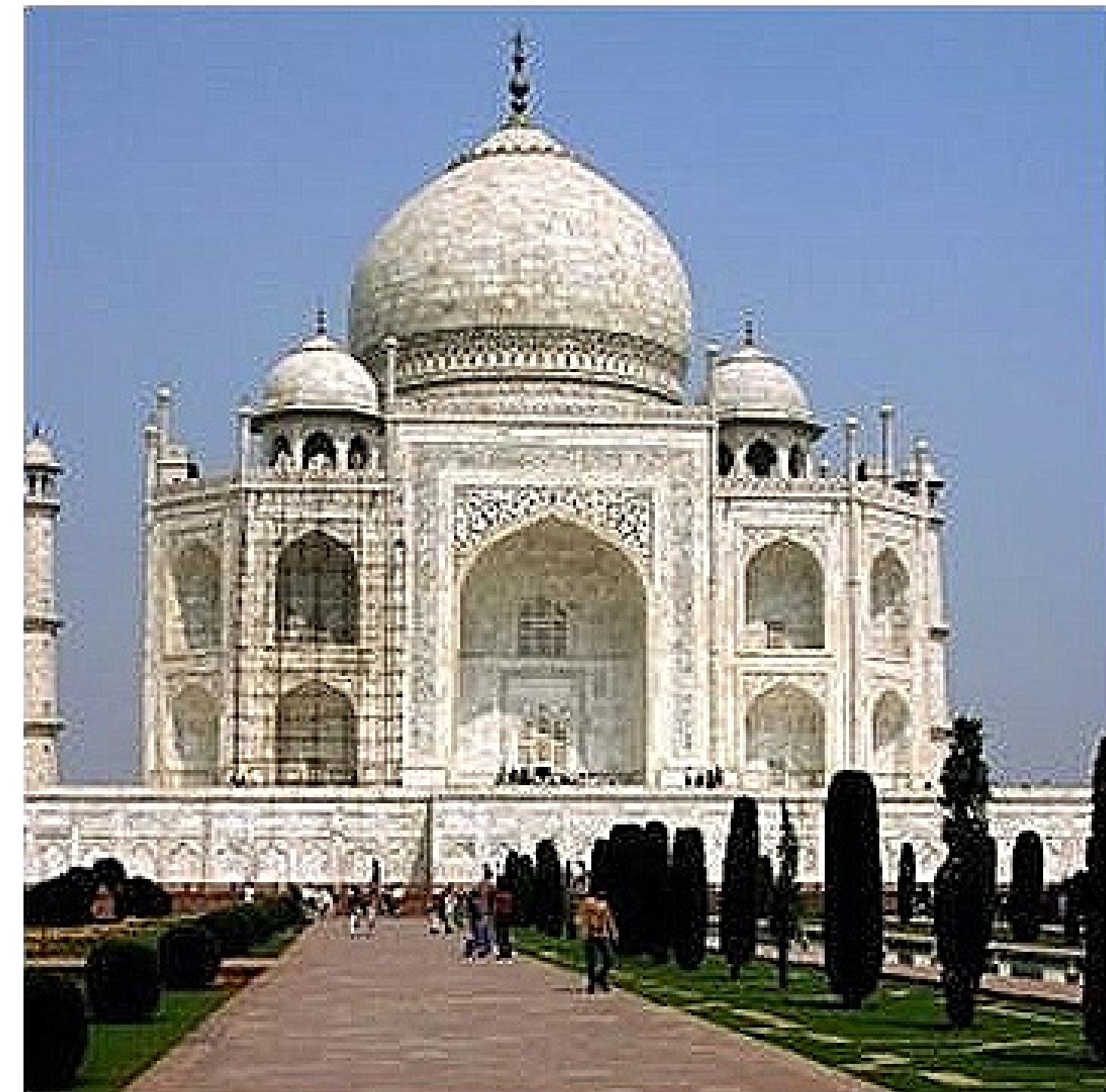


Kernel Convolutions

Sharpen

Kernel

0	-1	0
-1	5	-1
0	-1	0

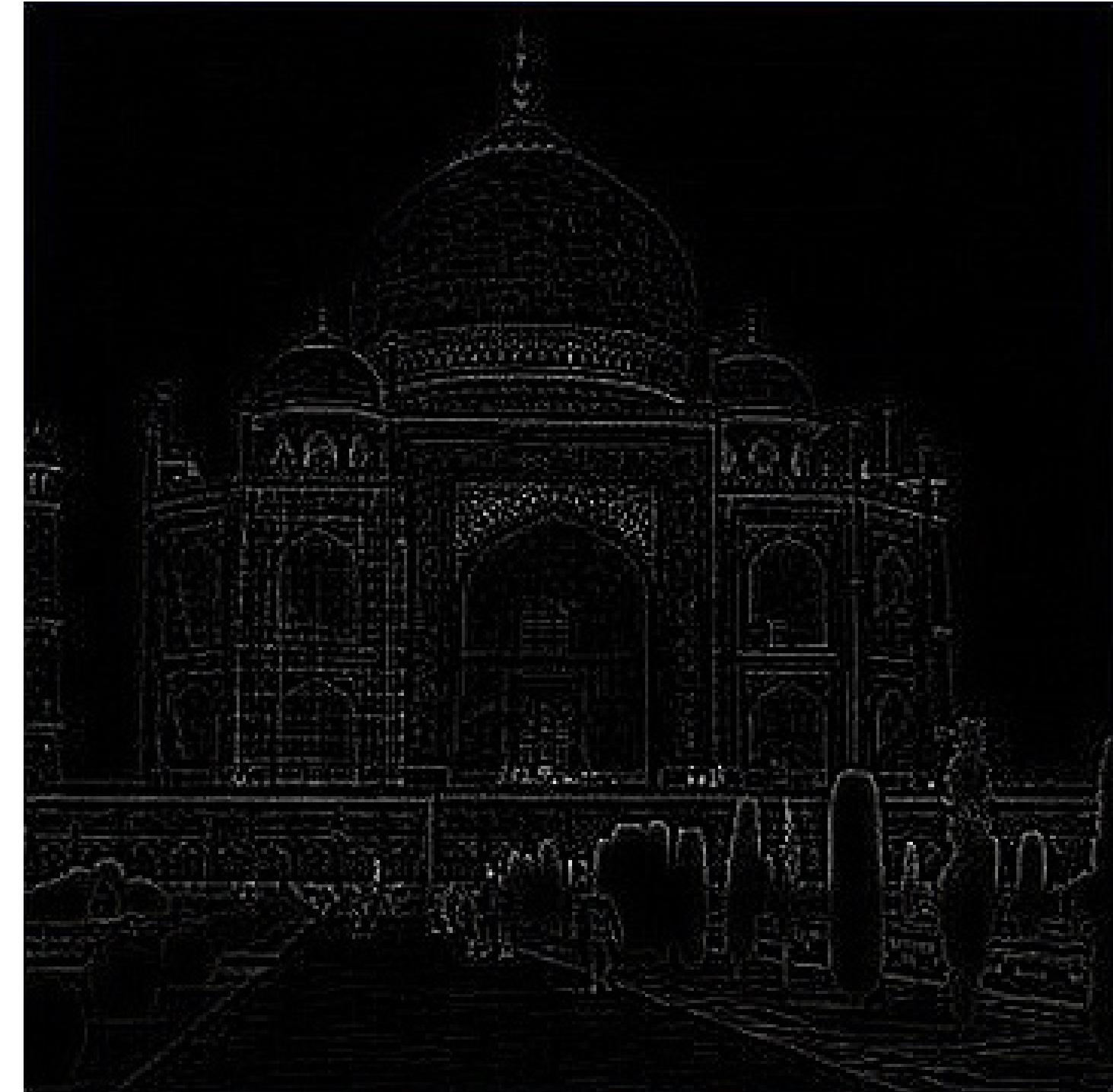


Kernel Convolutions

Edge
Highlight
Kernel



0	1	0
1	-4	1
0	1	0

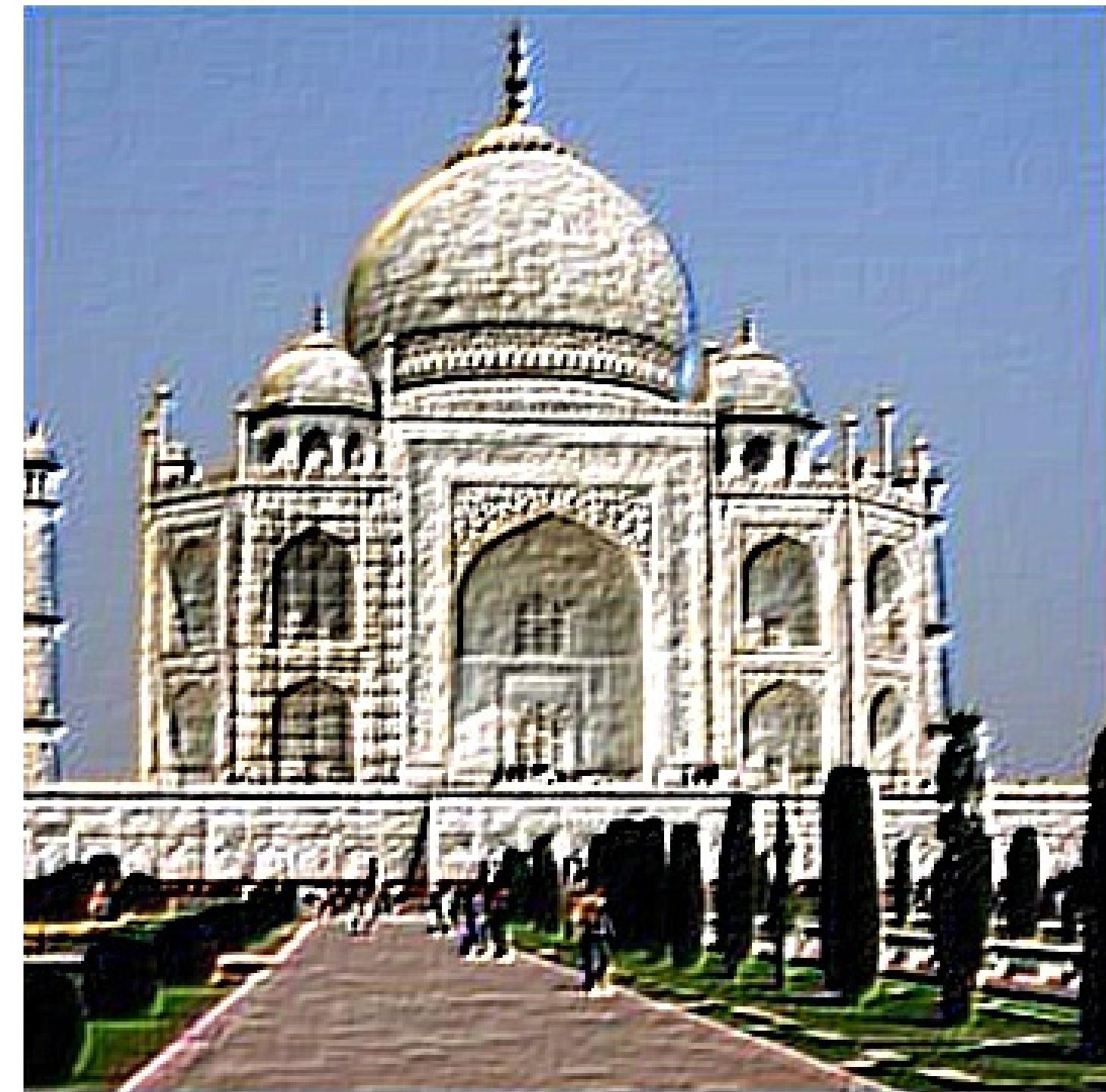


Kernel Convolutions

Emboss

Kernel

-2	-1	0
-1	1	1
0	1	2



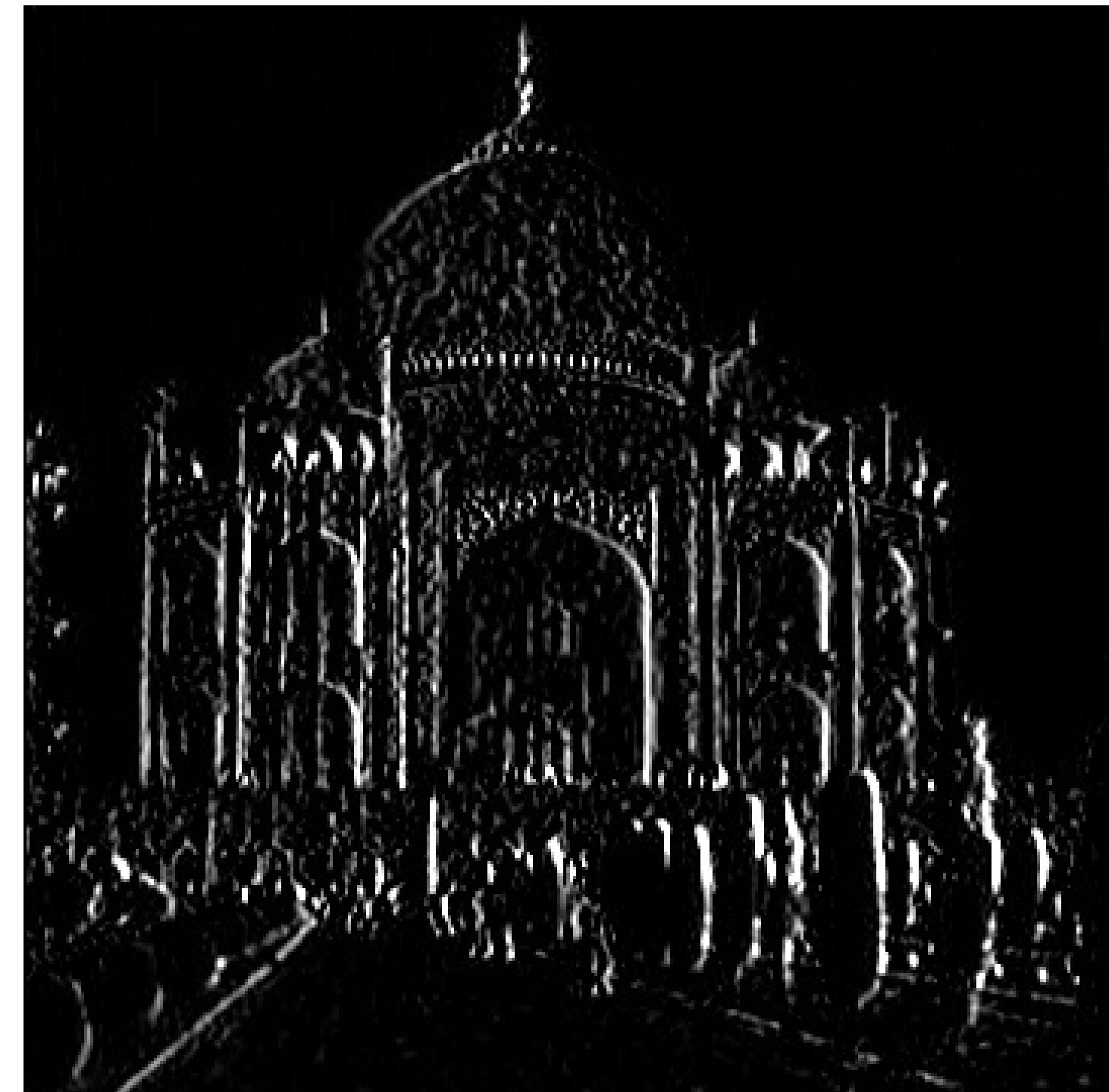
Kernel Convolutions

Horizontal “Derivative”



Kernel

1	0	-1
2	0	-2
1	0	-1



Kernel Convolutions

Vertical “Derivative”



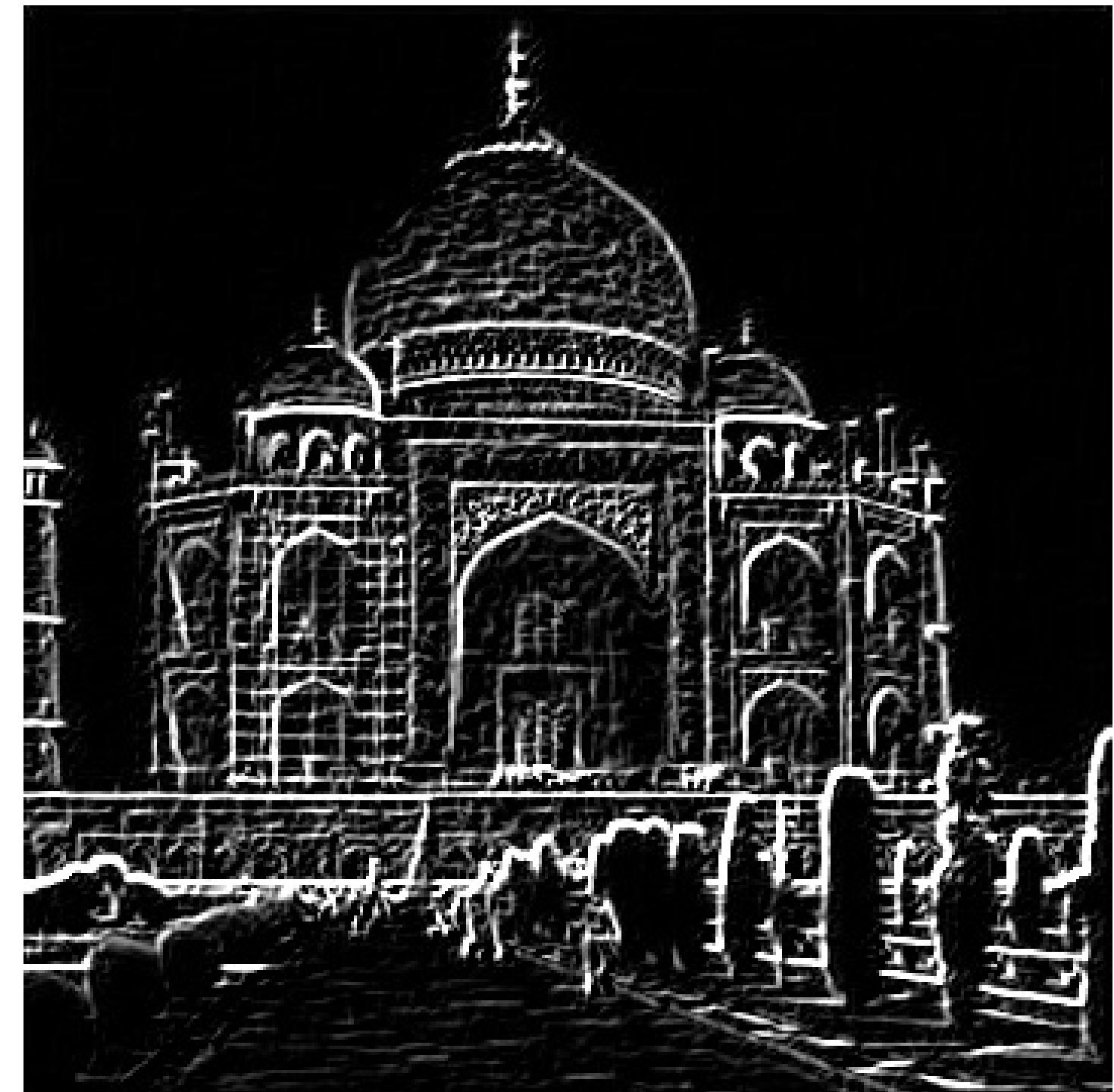
Kernel

1	2	1
0	0	0
-1	-2	-1



Kernel Convolutions

Overlay them to get **Sobel Operator**



Gaussian filter

It is another linear spatial filter used for smoothing, noise reduction or blurring. Gaussian filtering is more effective at smoothing images. It works similar to human visual perception when processing on images. The kernel coefficients diminish with increasing distance from the kernel's center. Normally, the kernel matrix are considered symmetric and size of the filter is odd. The values inside the kernel are computed by the Gaussian function, which is as follows:

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

Two dimensional gaussian function

In gaussian kernel, central pixels have a higher weighting than those on the periphery. The kernel for gaussian filter can be represented as

$$\begin{bmatrix} c & a & c \\ a & b & a \\ c & a & c \end{bmatrix}$$

General form of gaussian kernel ($c < a < b$)

A 3×3 Gaussian Kernel with standard deviation = 1, is as shown below.

1	2	1
2	4	2
1	2	1

source: geeksforgeeks

Laplacian of Gaussian Filter

A Laplacian filter is one of edge detectors used to compute the second spatial derivatives of an image. It measures the rate at which the first derivatives changes. In other words, Laplacian filter highlights the regions where the pixel intensities dramatically change. Due to this characteristic of Laplacian filter, it is often used to detect edges in an image. We will see how the filter find edges with visual illustration later.

The mathematical definition of Laplacian is as follows.

$$\begin{aligned} \text{Laplacian } \nabla^2 f &= \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \\ &= f(x+1, y) + f(x-1, y) - 2f(x, y) + f(x, y+1) + f(x, y-1) - 2f(x, y) \\ &= f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1) - 4f(x, y) \end{aligned}$$

Given the definition, discretized 3×3 Laplacian filter (because we are dealing with images and they are discrete) for an image f is defined as an array below:

0	-1	0
-1	4	-1
0	-1	0

Figure 8. 3×3 Laplacian filter

Note that, the defined kernel above, which is not identical to the mathematical definition of Laplacian due to the opposite signs, uses a negative peak because it is more commonly used and straightforward. However, it is still valid.

Spatial filters are implemented through kernels (masks/arrays), with odd dimensions. Filters can be::

- Low-pass – used for blur
- High-pass – used to sharpen



```

import cv2
import numpy as np
import matplotlib.pyplot as plt
import math

# Load an image
image = cv2.imread('./images/Taj_Mahal.jpg', cv2.IMREAD_GRAYSCALE)

# Define different kernels
kernels = {
    "Blur": np.ones((5, 5), np.float32) / 25,
    "Sharpen": np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]]),
    "Edge Detection": np.array([[1, 1, 1], [-1, 8, -1], [-1, -1, -1]]),
    "Emboss": np.array([[2, 0, 0], [0, 1, 1], [0, 1, 2]]),
    "Horizontal Derivative (Sobel X)": np.array([[1, 0, 1], [-2, 0, 2], [-1, 0, 1]]),
    "Vertical Derivative (Sobel Y)": np.array([[1, -2, -1], [0, 0, 0], [1, 2, 1]]),
    "Gaussian Blur": np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]], np.float32) / 16,
    "Laplacian Edge Detection": np.array([[0, -1, 0], [-1, 4, -1], [0, -1, 0]])
}

# Calculate the number of subplots needed
num_kernels = len(kernels) + 1 # +1 for the original image
num_cols = 3 # Define number of columns you want in the grid
num_rows = math.ceil(num_kernels / num_cols) # Calculate rows required

# Apply each kernel to the image
filtered_images = {name: cv2.filter2D(image, -1, kernel) for name, kernel in kernels.items()}

# Plot original and filtered images
plt.figure(figsize=(12, 10))
plt.subplot(num_rows, num_cols, 1)
plt.imshow(image, cmap='gray')
plt.title("Original Image")
plt.axis("off")

for i, (name, filtered_img) in enumerate(filtered_images.items(), 2):
    plt.subplot(num_rows, num_cols, i)
    plt.imshow(filtered_img, cmap='gray')
    plt.title(name)
    plt.axis("off")

plt.tight_layout() # Adjust layout to prevent overlap
plt.show()

```



kernels in image processing