



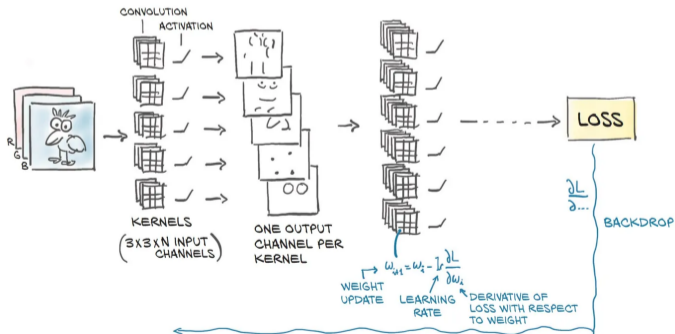
COMP 2211 Exploring Artificial Intelligence  
Digital Image Processing Fundamentals  
Dr. Desmond Tsoi

Department of Computer Science & Engineering  
The Hong Kong University of Science and Technology, Hong Kong SAR, China



# Convolutional Neural Network

- **Convolutional Neural Network** (CNN or ConvNet) is a class of Artificial Neural Networks applied to analyze visual imagery.
- Research has demonstrated that **preprocessing** images before feeding them to CNN would significantly improve the classification/recognition accuracy.
- To learn how to preprocess images and use this powerful tool to tackle Computer Vision tasks, you will first be introduced to digital image processing fundamentals.

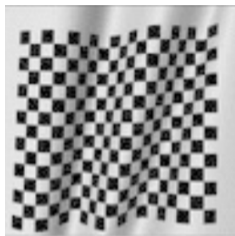


# Image Preprocessing

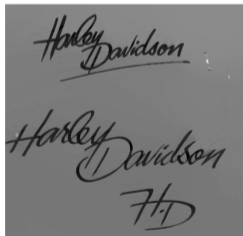
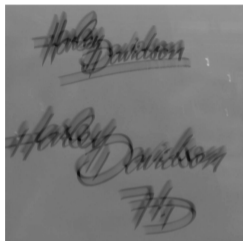
- **Image preprocessing** refers to **processing an image** so that the resulting image is **more suitable than the original** for a specific application.
- A preprocessing method that works well for one application may not be the best method for another application.
- Some preprocessing tasks:
  - Shading correction
  - De-blurring
  - De-noising
  - Contrast enhancement



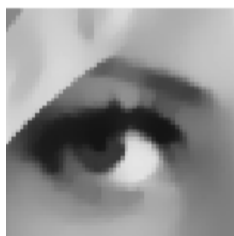
# Image Preprocessing



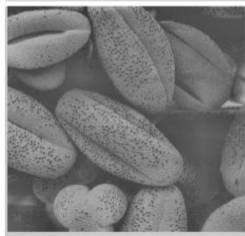
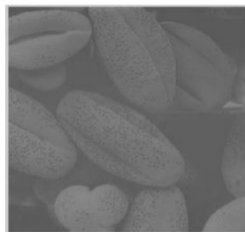
Shading Correction



De-blurring



De-noising

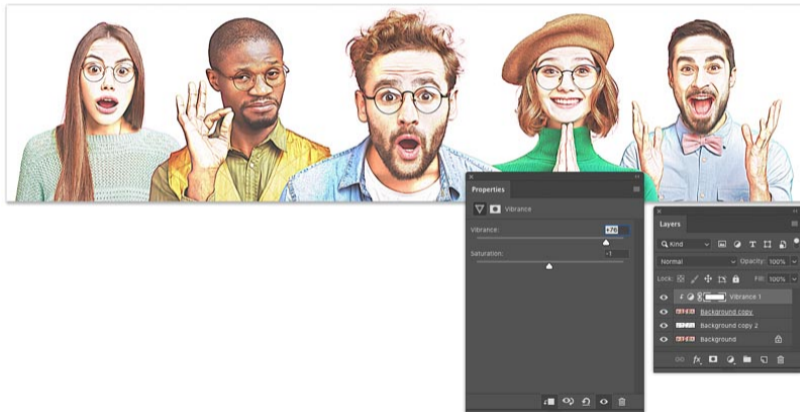


Contrast Enhancement



# Digital Image Processing

- Digital image processing (DIP) is the **method** to manipulate a digital image to either **enhance the quality or extract relevant information**.
- These methods provide the foundation to preprocess images that are more suitable for specific applications.



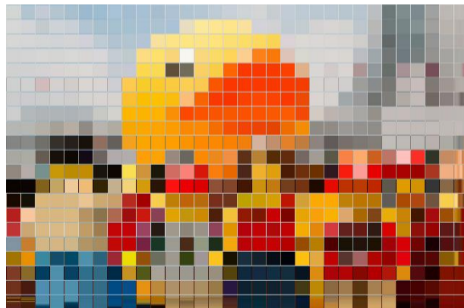
# Part I

## Digital Image Fundamentals



# Digital Image

- A **digital image** is a **two-dimensional grid of intensity values**, represented by  $I(x,y)$ , where  $x$  and  $y$  are coordinates, and the value of  $I$  at coordinates  $(x,y)$  is called intensity.
- **Pixels**: Short for Picture Element. A pixel is a **single point (dot)** in an image.
- **Dimensions**: Specified by the **width and height of the image**.
  - Image width is the number of columns in the image.
  - Image height is the number of rows in the image.



An image of dimensions  $32 \times 21$  (i.e., image width = 32 pixels, image height = 21 pixels)

# Image Coordinate System

- A specific pixel is specified by its **coordinates**  $(x,y)$  where **x is increasing from left to right**, and **y is increasing from top to bottom**.
- The **origin**  $(0,0)$  is in the **top-left corner**.
- The following shows the coordinate system of digital images:






$(0,0)$	$(1,0)$	$(2,0)$	$(3,0)$	$\dots$	$(\text{width}-1,0)$
$(0,1)$	$(1,1)$	$(2,1)$	$(3,1)$	$\dots$	$(\text{width}-1,1)$
$\vdots$					
$(0, \text{height}-1)$	$(1, \text{height}-1)$	$(2, \text{height}-1)$	$(3, \text{height}-1)$	$\dots$	$(\text{width}-1, \text{height}-1)$

where width and height are the image width and image height, respectively.

- The legal range of x-coordinate is between 0 to width-1
- The legal range of y-coordinate is between 0 to height-1









# Grayscale Images

- Grayscale is a range of gray shades from black to white.
- Grayscale images are most commonly used in image processing because the data are smaller and allow us to process the images in a short time.
- A grayscale digital image is an image in which the value of each pixel carries only intensity information.
- A grayscale image contains 8-bit/pixel data, which has  $2^8 = 256$  different gray levels (0 for black, 127 for gray, and 255 for white).

Black	Gray-level = 0	
Dark gray	Gray-level = 64	
Medium gray	Gray-level = 127	
Light gray	Gray-level = 190	
White	Gray-level = 255	

## Color Images

- Color images have intensity from the darkest and lightest of 3 different colors, Red, Green, and Blue (RGB).
- The mixtures of these color intensities produce a color image.
- Since RGB images contain  $3 \times 8$ -bit intensities, they are also referred to as 24-bit color images.
- An 8-bit intensity range has 256 possible values, 0 to 255.
- Common colors represented in RGB:

Black	RGB = (0, 0, 0)	
White	RGB = (255, 255, 255)	
Red	RGB = (255, 0, 0)	
Green	RGB = (0, 255, 0)	
Blue	RGB = (0, 0, 255)	
Yellow	RGB = (255, 255, 0)	
Gray	RGB = (128, 128, 128)	
Dark Gray	RGB = (50, 50, 50)	

24-bit grayscale images are subset of RGB images where the RGB intensity are all equal (e.g., Gray and Dark Gray shown above).

# Access Images in Google Drive

- To access files/images in Google drive, you need to mount your Google Drive to Colab using the following code:

```
# Import drive from google.colab package
from google.colab import drive
# Import os and sys modules
import os, sys

# Mount Google Drive
drive.mount('/content/drive')
# Assume a folder "images" has been created, go to the folder "images"
os.chdir('/content/drive/My Drive/images')
# Add the path for interpreter to search
sys.path.append('/content/drive/My Drive/images')
```



# Read Images in Colab

- To **read an image**, you need to first import `matplotlib.image`:  
`import matplotlib.image as mpimg`
- Then use **`imread()` method** of the `matplotlib.image` module.

## Syntax

```
mpimg.imread(fname, format=None)
```

## Parameters:

- `fname`: The image file to read: a filename or a file-like object opened in read-binary mode.
- `format`: The image file format assumed for reading the data. If `format` is not given, the format is deduced from the filename. If nothing can be deduced, PNG is tried.
- Return value: `numpy.array`.
  - (M,N) for grayscale images
  - (M,N,3) for RGB images
  - (M,N,4) for RGBA images

**PNG images are returned as float arrays (0-1)**. All other formats are returned as int arrays, with a bit depth determined by the file's contents.



# Show Images in Colab

- To **show an image**, you need to first import `matplotlib.pyplot`:  
`import matplotlib.pyplot as plt`
- Then use **`imshow()` method** of the `matplotlib.pyplot` module.

## Syntax

```
plt.imshow(X, cmap, vmin, vmax)
```

## Parameters:

- **X**: The image data. Supported array shapes are
  - **(M,N)**: an image with scalar data. The values are mapped to colors using normalization and a colormap.
  - **(M,N,3)**: an image with RGB values (0-1 float or 0-255 int)
  - **(M,N,4)**: an image with RGBA values (0-1 float or 0-255 int), i.e., including transparency.

The first two dimensions (M,N) define the rows and columns of the image.

## Syntax

### Parameters:

- `cmap`: str (e.g., 'gray') or `Colormap`. The `Colormap` instance or registered colormap name used to map scalar data to colors. This parameter is ignored for RGB(A) data.
- `vmin`, `vmax`:
  - By default, `imshow` scales elements of the numpy array so that the smallest element becomes 0, the largest becomes 1, and intermediate values are mapped to the interval  $[0,1]$  by a linear function.
  - Optionally, `imshow` can be called with arguments, `vmin` and `vmax`. In such case all elements of the array smaller or equal to `vmin` are mapped to 0, all elements greater or equal to `vmax` are sent to 1, and the elements between `vmin` and `vmax` are mapped in a linear fashion into the interval  $[0,1]$ .
- Returns `AxesImage`  
`AxesImage` is an image attached to an `Axes`.

# Save Images in Colab

- To **save an image**, you need to first import `matplotlib.pyplot`:  
`import matplotlib.pyplot as plt`
- Then use **`imsave()` method** of the `matplotlib.pyplot` module.

## Syntax

```
plt.imsave(fname, arr)
```

## Parameters:

- `fname`: a path or a file-like object to store the image in.
- `arr`: The image data. The shape can be one of  $M \times N$  (luminance),  $M \times N \times 3$  (RGB) or  $M \times N \times 4$  (RGBA). The first two dimensions ( $M, N$ ) define the rows and columns of the image.
- Returns `AxesImage`  
`AxesImage` is an image attached to an `Axes`.

```
# Assume Google Drive has been mounted & the path has been added for interpreter to search
```

```
# Import all the required libraries
```

```
import matplotlib.image as mpimg  
import matplotlib.pyplot as plt  
import numpy as np
```

```
# Read and show the image
```

```
img = mpimg.imread('snorlax.png')  
plt.imshow(img)
```

```
# Find the shape of input image
```

```
[height, width, layers] = np.shape(img)
```

```
# Print all the required information
```

```
print('Dimensions: {}x{}x{}'.format(height, width, layers))  
print('Total number of pixels:', width*height)
```

```
plt.imsave('snorlax2.png', img) # Save the image
```

Dimensions: 1000x1600x4

Total number of pixels: 1600000



## Part II

# Image Processing

Original Image



Cartoonized Image



# Image Processing using OpenCV

- **OpenCV** (Open Source Computer Vision Library) is an **open source computer vision and machine learning software library**.
- OpenCV was built to provide a **common infrastructure for computer vision applications** and to accelerate the use of machine perception in the commercial products.
- The library has **more than 2500 optimized algorithms**, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms.
- OpenCV supports a wide variety of programming languages such as Python, C++, Java, etc.



## Convert Color Images to Grayscale

- In certain problem, you will find it useful to **lose unnecessary information** from your images to **reduce space and computational complexity**.
- **Converting colored images to grayscale images** is an example. This is done, as color is not necessary to recognize and interpret an image.
- Grayscale can be good enough for recognizing certain objects, because color images contain more information than black and white images, they can add unnecessary complexity and take up more space in memory.



# Convert Color Images to Grayscale

- One way to **convert color images to grayscale** is to apply the following formula:

$$V = 0.299 \times R + 0.587 \times G + 0.114 \times B$$

- To perform the above using OpenCV, you need to first import cv2  
`import cv2`
- Then use **cvtColor()** method of the cv2 module.

## Syntax

```
cv2.cvtColor(image, code)
```

## Parameters:

- **image**: Image to be processed in n-dimensional array
- **code**: Conversion code for colorspace. For converting RGB to grayscale, we use cv2.COLOR\_RGB2GRAY
- **Return value**: Converted image.



```
# Assume Google Drive has been mounted & the path has been added for interpreter to search
```

```
# Import all the required libraries
```

```
import cv2
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
```

```
# Read and show the image
```

```
img = mpimg.imread('snorlax.png')
plt.figure()
plt.imshow(img)
```

```
# Convert color image to gray
```

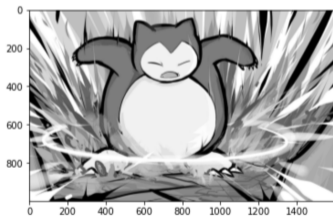
```
grayImg = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
```

```
# Show the image
```

```
plt.figure()
plt.imshow(grayImg, cmap='gray', vmin=0, vmax=1)
```

```
# Save the image
```

```
plt.imsave('snorlax-gray.png', grayImg)
```



## Image Affine Transformation

- An **affine transformation** is any transformation that **preserves collinearity, parallelism** as well as the **ratio of distances between the points** (e.g., midpoint of a line remains the midpoint after transformation).
- It does not necessarily preserve distances and angles.
- Geometric transformations, such as **translation, rotation, scaling, shearing**, etc., are all affine transformations.
- In general, the affine transformation can be expressed in the form of a linear transformation followed by a vector addition as follows:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} b_{00} \\ b_{10} \end{bmatrix} = \begin{bmatrix} a_{00}x + a_{01}y + b_{00} \\ a_{10}x + a_{11}y + b_{10} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & b_{00} \\ a_{10} & a_{11} & b_{10} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- $M = \begin{bmatrix} a_{00} & a_{01} & b_{00} \\ a_{10} & a_{11} & b_{10} \end{bmatrix}$  is a **transformation matrix**. To define what transformation you want to do, you need to define M.

# Image Translation

- To perform the image **affine transformation** using OpenCV, you need to first import cv2  
`import cv2`
- Then use **warpAffine()** method of the cv2 module.

## Syntax

```
cv2.warpAffine(src, M, dsize, flags, borderMode, borderValue)
```

## Parameters:

- src: input image
- M:  $2 \times 3$  transformation matrix
- dsize: size of the output image
- flags: combination of interpolation methods
- borderMode: pixel extrapolation method
- borderValue: value used in case of a constant border; by default, it is 0
- Return value: output image that has the size dsize and the same type as src

# Image Translation

- **Translation** is simply the **shifting** of object location.
- Suppose we have a point  $P(x, y)$  which is translated by  $(t_x, t_y)$ , then the coordinates after translation denoted by  $P'(x', y')$  are given by

$$x' = x + t_x$$

$$y' = y + t_y$$

- In matrix form

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

```

# Assume Google Drive has been mounted & the path has been added for interpreter to search

# Import all the required libraries
import cv2
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import numpy as np

# Read the image
img = mpimg.imread('snorlax.png')

# Convert the color image to gray and show it
grayImg = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
plt.figure();
plt.imshow(grayImg, cmap='gray', vmin=0, vmax=1)

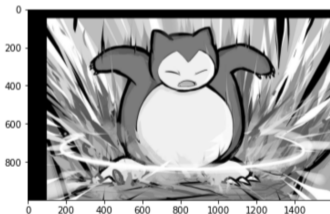
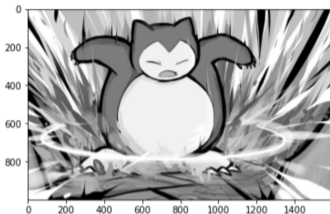
# Find the shape of the gray image
rows, cols = grayImg.shape

# Form the transformation matrix of translation
M = np.float32([[1, 0, 100],[0, 1, 50]])
# Perform the transformation
translatedImg = cv2.warpAffine(grayImg, M, (cols,rows))

# Show the image
plt.figure(); plt.imshow(translatedImg, cmap='gray', vmin=0, vmax=1)

```

<matplotlib.image.AxesImage at 0x7fc1460d1850>



## Image Reflection

- **Reflection** refers to **mirroring (or flipping) images** along x-axis or y-axis.
- To make sure the resulting image fits the image coordinate, after flipping image along x-axis, we need to translate it by the amount of number of rows in y-axis. Similarly, after flipping image along y-axis, we need to translate it by the amount of columns in x-axis.
- Suppose we have a point  $P(x, y)$  which is reflected along the x-axis, then the coordinates after reflection denoted by  $P'(x', y')$  are given by

$$x' = x$$

$$y' = -y + rows$$

In matrix form

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & rows \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- Similarly, a point  $P(x, y)$  which is reflected along the y-axis, the coordinates after reflection denoted by  $P'(x', y')$  are given by

$$x' = -x + cols$$

$$y' = y$$

In matrix form

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -1 & 0 & cols \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

```
# Assume Google Drive has been mounted & the path has been added for interpreter to search
```

```
# Import all the required libraries
```

```
import cv2; import numpy as np
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
```

```
img = mpimg.imread('snorlax.png') # Read the image
# Convert the color image to gray and show it
grayImg = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
plt.figure(); plt.imshow(grayImg, cmap='gray', vmin=0, vmax=1)
```

```
rows, cols = grayImg.shape # Find the shape of the gray image
```

```
# Form the transformation matrix of x-axis reflection
```

```
M = np.float32([[1, 0, 0], [0, -1, rows]])
```

```
# Perform the transformation
```

```
xaxisreflection = cv2.warpAffine(grayImg, M, (cols,rows))
```

```
plt.figure(); plt.imshow(xaxisreflection, cmap='gray', vmin=0, vmax=1)
```

```
# Form the transformation matrix of y-axis reflection
```

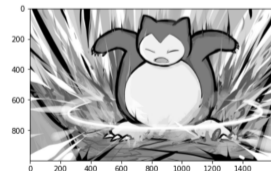
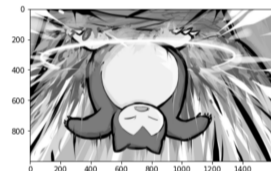
```
M = np.float32([[-1, 0, cols], [0, 1, 0]])
```

```
# Perform the transformation
```

```
yaxisreflection = cv2.warpAffine(grayImg, M, (cols,rows))
```

```
plt.figure(); plt.imshow(yaxisreflection, cmap='gray', vmin=0, vmax=1)
```

<matplotlib.image.AxesImage at 0x7fc1460a5850>



# Image Rotation

- **Image rotation** refers to **rotating an image  $\theta$  degree along  $(x_0, y_0)$** .
- Suppose we have a point  $P(x, y)$  which is rotated along the center of the image, then the coordinates after rotation denoted  $P'(x', y')$  are given by

$$\begin{aligned}x' &= (x - x_0)\cos\theta + (y - y_0)\sin\theta + x_0 \\y' &= -(x - x_0)\sin\theta + (y - y_0)\cos\theta + y_0\end{aligned}$$

In matrix form

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & -x_0\cos\theta - y_0\sin\theta + x_0 \\ -\sin\theta & \cos\theta & x_0\sin\theta - y_0\cos\theta + y_0 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$



# Assume Google Drive has been mounted & the path has been added for interpreter to search

```
import cv2, math; import numpy as np # Import all the required libraries
import matplotlib.image as mpimg; import matplotlib.pyplot as plt
```

```
img = mpimg.imread('snorlax.png') # Read the image
# Convert the color image to gray and show it
grayImg = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
plt.figure(); plt.imshow(grayImg, cmap='gray', vmin=0, vmax=1)
```

```
rows, cols = grayImg.shape # Find the shape of image
```

```
# Form the transformation matrix of rotation
# The angle for math.sin and math.cos should be in radian.
# 45 degree = pi/4 radian
```

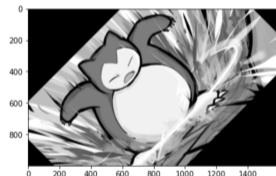
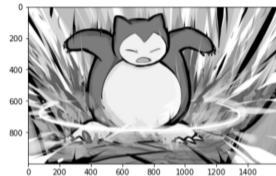
```
angle = math.pi/4
M = np.float32([[math.cos(angle), math.sin(angle),
-(cols//2)*math.cos(angle)-(rows//2)*math.sin(angle) + (cols//2)],
[-math.sin(angle), math.cos(angle),
(cols//2)*math.sin(angle)-(rows//2)*math.cos(angle) + (rows//2)])])
```

```
# Another way to generate the required transformation matrix
# The angle for getRotationMatrix2D should be in degree
# M = cv2.getRotationMatrix2D((cols//2,rows//2), 45, 1.0)
```

```
# Perform the transformation
```

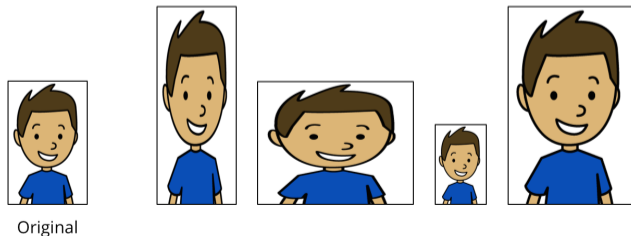
```
rotate45 = cv2.warpAffine(grayImg, M, (cols,rows))
plt.figure(); plt.imshow(rotate45, cmap='gray', vmin=0, vmax=1)
```

<matplotlib.image.AxesImage at 0x7f2b2ef27290>



# Image Resizing/Scaling

- **Resizing/Scaling images** is a critical preprocessing step in computer vision.
- Machine learning models train faster on smaller images. Moreover, many deep learning model architectures require that our images are the same size, and our raw collected images may vary in size.
- Resizing is a common approach to make the input images the same size, and it works well unless you have a very different aspect ratio from the expected input shape.



# Image Resizing/Scaling

- To perform the image **resizing/scaling** using OpenCV, you need to first import cv2  
`import cv2`
- Then use **resize()** method of the cv2 module.

## Syntax

```
cv2.resize(src, dsize, dst, fx = 0, fy = 0, interpolation = INTER_LINEAR)
```

## Parameters:

- src: Input image
- dsize: The size for the output image
- dst (optional): The output image with size dsize
- fx (optional): The scale factor along the horizontal axis
- fy (optional): The scale factor along the vertical axis
- interpolation: The algorithm used to reconstruct the new pixels
  - cv2.INTER\_NEAREST (nearest neighbor interpolation)
  - cv2.INTER\_LINEAR (bilinear interpolation)
  - cv2.INTER\_CUBIC (bicubic interpolation)
- Returns AxesImage

```
# Assume Google Drive has been mounted & the path
# has been added for interpreter to search

# Import all the required libraries
import cv2
import matplotlib.image as mpimg
import matplotlib.pyplot as plt

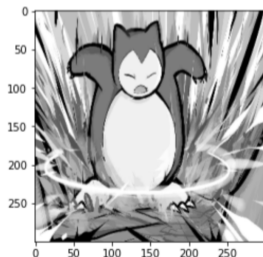
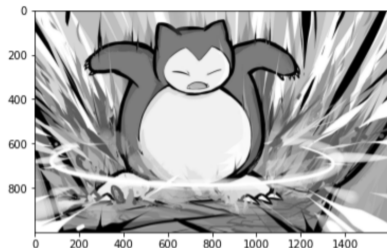
img = mpimg.imread('snorlax.png') # Read the image

# Convert the color image to gray and show it
grayImg = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
plt.figure()
plt.imshow(grayImg, cmap='gray', vmin=0, vmax=1)

# Perform the transformation
resizedImg = cv2.resize(grayImg, (300, 300),
                          interpolation=cv2.INTER_LINEAR)

# Show the image
plt.figure()
plt.imshow(resizedImg, cmap='gray', vmin=0, vmax=1)
```

<matplotlib.image.AxesImage at 0x7f149d2057d0>



# Image Cropping

- Sometimes, you may want to **crop the region of interest (ROI)** for further processing.
- For instance, in a face detection application, you may want to drop the face from an image.
- To crop an image, you can use the same method as **numpy array slicing**.
- To slice an array, you need to specify the start and end index of the first as well as the second dimension.

## Syntax

```
croppedImg = sourceImg[start_row:end_row, start_col:end_col]
```

```
# Assume Google Drive has been mounted, & the path
# has been added for interpreter to search

# Import all the required libraries
import cv2
import matplotlib.image as mpimg
import matplotlib.pyplot as plt

img = mpimg.imread('snorlax.png') # Read the image

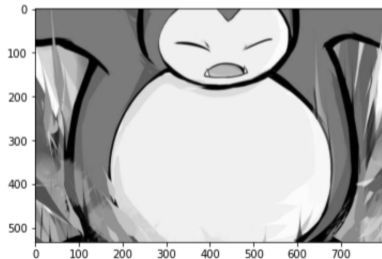
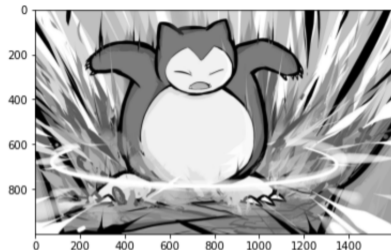
# Convert the color image to gray and show it
grayImg = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

# Show the image
plt.figure()
plt.imshow(grayImg, cmap='gray', vmin=0, vmax=1)

# Obtain part of the image
croppedImg = grayImg[200:733, 300:1100]

# Show the image
plt.figure()
plt.imshow(croppedImg, cmap='gray', vmin=0, vmax=1)
```

<matplotlib.image.AxesImage at 0x7f1491def690>



# Image Padding

- Image padding introduces new pixels around the edges of an image.
- Types of padding
  - Constant padding
  - Reflection padding
  - Replication padding

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	1	2	3	0	0
0	0	4	5	6	0	0
0	0	7	8	9	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

BORDER\_CONSTANT

5	4	4	5	6	6	5
2	1	1	2	3	3	2
2	1	1	2	3	3	2
5	4	4	5	6	6	5
8	7	7	8	9	9	8
8	7	7	8	9	9	8
5	4	4	5	6	6	5

BORDER\_REFLECT

1	1	1	2	3	3	3
1	1	1	2	3	3	3
1	1	1	2	3	3	3
4	4	4	5	6	6	6
7	7	7	8	9	9	9
7	7	7	8	9	9	9
7	7	7	8	9	9	9

BORDER\_REPLICATE

# Image Padding

- To **pad an image**, you need to first import cv2  
`import cv2`
- Then use **copyMakeBorder()** method of the cv2 module.

## Syntax

```
cv2.copyMakeBorder(src, top, bottom, left, right, borderType, value)
```

## Parameters:

- src: Source image
- top: The border width in number of pixels in top direction
- bottom: The border width in the number of pixels in bottom direction
- left: The border width in the number of pixels in left direction
- right: The border width in the number of pixels in the right direction
- borderType: The kind of border to be added
  - cv2.BORDER\_CONST
  - cv2.BORDER\_REFLECT
  - cv2.BORDER\_REPLICATE
- value (optional): The color of border if border type is cv2.BORDER\_CONSTANT
- Returns the resulting image



```

# Assume Google Drive has been mounted
# & the path has been added for
# interpreter to search

# Import all the required libraries
import cv2
import matplotlib.image as mpimg
import matplotlib.pyplot as plt

# Read the image
img = mpimg.imread('snorlax.png')

# Create subplots
fig = plt.figure(figsize=(12,9))
fig.tight_layout();
fig.subplots_adjust(wspace=0.2, hspace=0.2)

# Add the image to the first row, first col
ax1= fig.add_subplot(2, 2, 1)
ax1.title.set_text('Original')
grayImg = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
plt.imshow(grayImg, cmap='gray', vmin=0, vmax=1)

```

```

padImgConst = cv2.copyMakeBorder(grayImg,
                                  50, 50, 50, 50,
                                  cv2.BORDER_CONSTANT, 128)

```

```

# Add the image to the first row, second col
ax2 = fig.add_subplot(2, 2, 2)
ax2.title.set_text('BORDER_CONSTANT')
plt.imshow(padImgConst, cmap='gray',
            vmin=0, vmax=1)

```

```

padImgRef = cv2.copyMakeBorder(grayImg,
                                300, 300, 300, 300,
                                cv2.BORDER_REFLECT)

```

```

# Add the image to the second row, first col
ax3 = fig.add_subplot(2, 2, 3)
ax3.title.set_text('BORDER_REFLECT')
plt.imshow(padImgRef, cmap='gray',
            vmin=0, vmax=1)

```

```

padImgRep = cv2.copyMakeBorder(grayImg,
                                 300, 300, 300, 300,
                                 cv2.BORDER_REPLICATE)

```

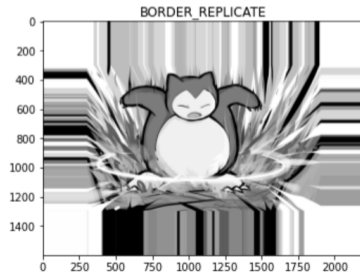
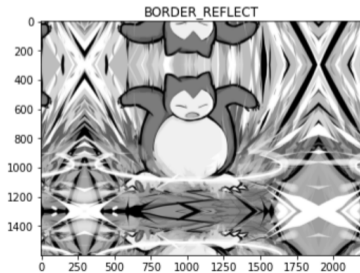
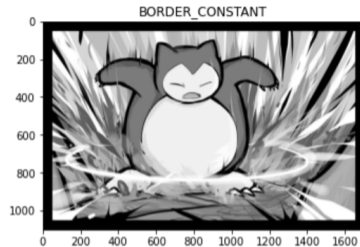
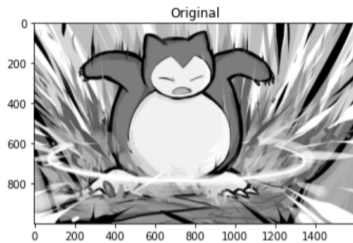
```

# Add the image to the second row, second col
ax4 = fig.add_subplot(2, 2, 4)
ax4.title.set_text('BORDER_REPLICATE')
plt.imshow(padImgRep, cmap='gray',
            vmin=0, vmax=1)

```

# Example

<matplotlib.image.AxesImage at 0x7fc14535b410>



# Image Histogram

- An **image histogram** is a graphical representation of **the number of pixels in an image as a function of their intensity**.
- To find the **histogram of an image**, you need to first import cv2  
`import cv2`
- Then use **calcHist()** method of the cv2 module.

## Syntax

```
cv2.calcHist(images, channels, mask, histSize, ranges[, hist[, accumulate]])
```

## Parameters:

- **image**: Image of type uint8 or float32 represented as “[img]”
- **channels**: It is the index of channel for which we calculate histogram. For grayscale image, its value is [0] and color image, you can pass [0], [1], or [2] to calculate histogram of each channel respectively.
- **mask**: mask image. To find histogram of full image, it is given as ‘None’.
- **histSize**: This represents the number of bins. For full scale, we pass [256]
- **ranges**: This is the range of intensities. Normally, it is [0,256].
- **Return value**: Histogram of the image.

```
# Assume Google Drive has been mounted & the path has been added for interpreter to search
```

```
# Import all the required libraries
```

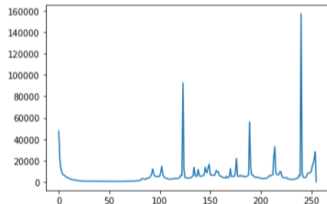
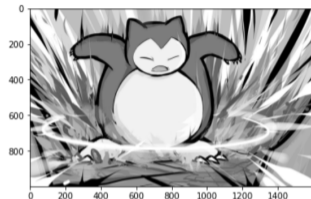
```
import cv2
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
```

```
img = mpimg.imread('snorlax.png') # Read the image
```

```
# Convert the color image to gray and show it
grayImg = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
plt.figure()
plt.imshow(grayImg, cmap='gray', vmin=0, vmax=1)
```

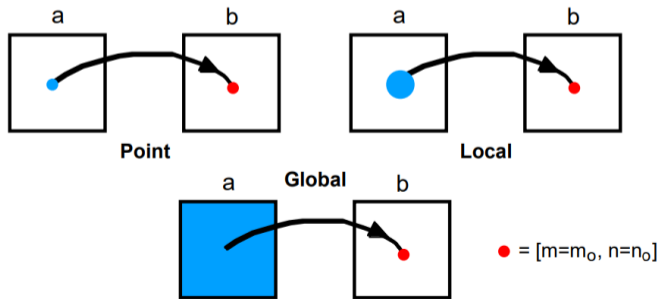
```
# Convert pixel values from [0,1] to [0,255]
grayImgUint = grayImg*255
grayImgUint = grayImgUint.astype(np.uint8)
```

```
# Calculate histogram
hist = cv2.calcHist([grayImgUint], [0], None, [256], [0,255])
plt.figure()
plt.plot(hist) # Plot and show the histogram
plt.show()
```



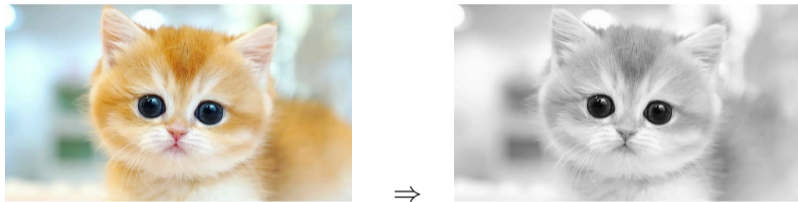
# Characteristics of Image Operations

- There are many ways to classify **image operations**.
- One way for doing so is to be based on the “region” used to process the pixels.
  1. **Point**: The **output value at a specific coordinate** is dependent only on the input value at the **same coordinate**.
  2. **Local**: The **output value at a specific coordinate** is dependent on the **input values in the neighbourhood** of that same coordinate.
  3. **Global**: The **output value at a specific coordinate** is dependent on **all the values in the input image**.



## Question

What type of image operation is color to grayscale conversion? :D

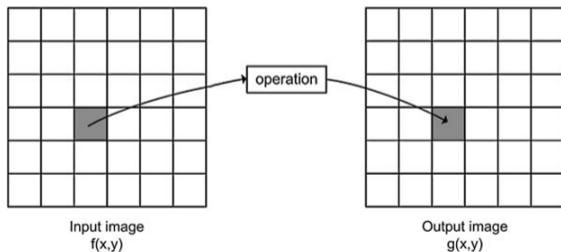


Answer:

Point operation! Since the output value at a specific coordinate of the grayscale image is dependent only on the input value at the same coordinate of the color image.

## Point-based Operations (Examples)

- **Brightness adjustment**: Make images brighter or dimmer
- **Contrast stretching**: Adjust the contrast of images
- **Gamma correction**: Grayscale non-linear transformation
- **Grayscale threshold**: Convert a grayscale image into a black and white binary image
- **Histogram equalization**: Transformation where an output image has approximately the same number of pixels at each gray level



# Brightness Adjustment

- To **adjust the brightness** of an image using OpenCV, you need to first import cv2  
`import cv2`
- Then use `convertScaleAbs()` method of the cv2 module.

## Syntax

```
cv2.convertScaleAbs(image, alpha = 1, beta = 0)
```

`convertScaleAbs` does the following:

$$I_{\text{new}}(x,y) = \min(\alpha * I(x,y) + \beta, 255)$$

$$I_{\text{new}}(x,y) = \max(I_{\text{new}}(x,y), 0)$$

Parameters:

- `image`: Image to be processed in n-dimensional array
- `alpha`: The scale factor. It is 1 by default
- `beta`: The delta added to the scaled values. It is 0 by default.
- Return value: Converted image.



```
# Assume Google Drive has been mounted & the path has been added for interpreter to search
```

```
# Import all the required libraries
```

```
import cv2
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
```

```
img = mpimg.imread('snorlax.png') # Read the image
```

```
# Convert the color image to gray and show it
```

```
grayImg = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
plt.figure(); plt.imshow(grayImg, cmap='gray', vmin=0, vmax=1)
```

```
# Convert pixel values from [0,1] to [0,255]
```

```
grayImgUint = grayImg*255
grayImgUint = grayImgUint.astype(np.uint8)
```

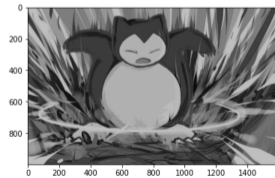
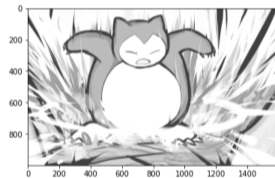
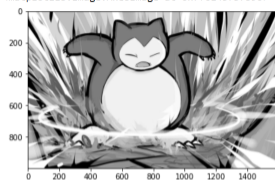
```
# Produce an image that is brighter than the original
```

```
brighterImg = cv2.convertScaleAbs(grayImgUint, alpha = 1.0, beta = 64)
plt.figure(); plt.imshow(brighterImg, cmap='gray', vmin=0, vmax=255)
```

```
# Produce an image that is dimmer than the original
```

```
dimmerImg = cv2.convertScaleAbs(grayImgUint, alpha = 1.0, beta = -64)
plt.figure(); plt.imshow(dimmerImg, cmap='gray', vmin=0, vmax=255)
```

<matplotlib.image.AxesImage at 0x7fc145737b90>



# Contrast Stretching

- **Contrast stretching** is an image enhancement method which attempts to improve an image by **stretching the range of intensity values**.
- One way to perform contrast stretching is to **stretch minimum and maximum intensity values present to the possible minimum and maximum intensity values**.
- Example:
  - Assume 0-255 taken as standard minimum and maximum intensity for 8-bit images.
  - If the minimum intensity value ( $I_{min}$ ) present in the image is 100, then it is stretched to the possible minimum value 0.
  - Likewise, if the maximum intensity value ( $I_{max}$ ) is less than the possible maximum intensity value 255, then it is stretched out to 255.
  - General formula for contrast stretching:

$$I_{new} = \frac{I - I_{min}}{I_{max} - I_{min}} \times 255$$

where

- $I$  is the current pixel intensity value
- $I_{min}$  is the minimum intensity value present in the whole image
- $I_{max}$  is the maximum intensity value present in the whole image
- $I_{new}$  is the output intensity value rounded up to the nearest integer value

```
# Assume Google Drive has been mounted & the path has been added for interpreter to search
```

```
# Import all the required libraries
```

```
import cv2; import numpy as np  
import matplotlib.image as mpimg  
import matplotlib.pyplot as plt
```

```
# Read the image
```

```
img = mpimg.imread('snorlax-low-contrast.png')
```

```
# Convert the color image to gray and show it
```

```
grayImg = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)  
plt.figure(); plt.imshow(grayImg, cmap='gray', vmin=0, vmax=1)
```

```
# Convert pixel values from [0,1] to [0,255]
```

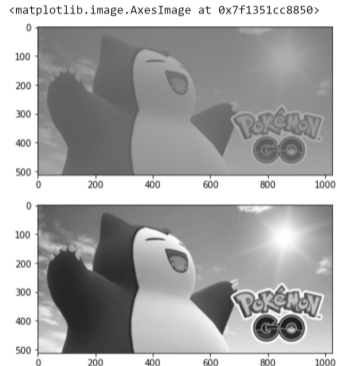
```
grayImgUint = grayImg*255  
grayImgUint = grayImgUint.astype(np.uint8)
```

```
# Find min and max pixel values and perform normalization
```

```
min = np.min(grayImgUint)  
max = np.max(grayImgUint)  
imageContrastEnhance = ((grayImgUint-min)/(max-min))*255
```

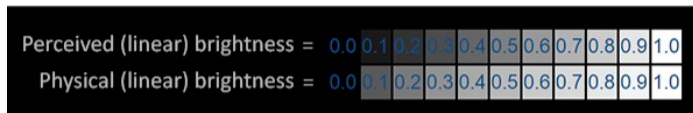
```
# Show the image
```

```
plt.figure(); plt.imshow(imageContrastEnhance.astype(np.uint8), cmap='gray', vmin=0, vmax=255)
```



## Gamma Correction

- **Gamma** defines the relationship between a **pixel's numerical value and its actual luminance**.



- Without gamma, shades captured by digital cameras would not appear as they did to our eyes (on a standard monitor).
- Gamma is also referred to as gamma correction, gamma encoding or gamma compression, but these all refer to a similar concept.
- A gamma encoded image has to have “gamma correction” applied when it is viewed – which effectively converts it back into light from the original scene.

Gamma correction can be performed by **adjusting gamma value ( $\gamma$ )**.

- $\gamma < 1$  will make the image appear darker
- $\gamma > 1$  will make the image appear lighter
- $\gamma = 1$  will have no effect on the input image

# Assume Google Drive has been mounted & the path has been added for interpreter to search

# Import all the required libraries

```
import cv2; import numpy as np
import matplotlib.image as mpimg; import matplotlib.pyplot as plt
```

```
img = mpimg.imread('snorlax.png') # Read the image
```

# Convert the color image to gray and show it

```
grayImg = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
plt.figure(); plt.imshow(grayImg, cmap='gray', vmin=0, vmax=1)
# Convert pixel values from [0,1] to [0,255]
grayImgUint = grayImg*255; grayImgUint = grayImgUint.astype(np.uint8)
```

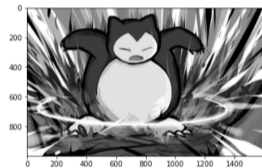
# Prepare look-up-table and perform gamma correction

```
gamma = 0.5; invGamma = 1/gamma
table = [((i / 255) ** invGamma) * 255 for i in range(256)]
table = np.array(table, np.uint8)
processedImg1 = cv2.LUT(grayImgUint, table)
plt.figure(); plt.imshow(processedImg1, cmap='gray', vmin=0, vmax=255)
```

# Prepare look-up-table and perform gamma correction

```
gamma = 2.2; invGamma = 1/gamma
table = [((i / 255) ** invGamma) * 255 for i in range(256)]
table = np.array(table, np.uint8)
processedImg2 = cv2.LUT(grayImgUint, table)
plt.figure(); plt.imshow(processedImg2, cmap='gray', vmin=0, vmax=255)
```

<matplotlib.image.AxesImage at 0x7f13523ee590>



# Image Thresholding

- **Image thresholding** is a simple form of image segmentation.
- It is a way to **create a binary image from a grayscale image or full-color image**.
- This is typically done in order to separate “object” or foreground pixels from background pixels to aid in image processing.
- The formula of image thresholding with **threshold T** is defined as

$$I_{new} = \begin{cases} 0 & I < T \\ 255 & \textit{otherwise} \end{cases}$$

```
# Assume Google Drive has been mounted & the path has been added for interpreter to search
```

```
# Import all the required libraries
```

```
import cv2  
import matplotlib.image as mpimg  
import matplotlib.pyplot as plt
```

```
# Read the image
```

```
img = mpimg.imread('snorlax.png')
```

```
# Convert the color image to gray and show it
```

```
grayImg = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)  
plt.figure(); plt.imshow(grayImg, cmap='gray', vmin=0, vmax=1)
```

```
# Convert pixel values from [0,1] to [0,255]
```

```
grayImgUint = grayImg*255  
grayImgUint = grayImgUint.astype(np.uint8)
```

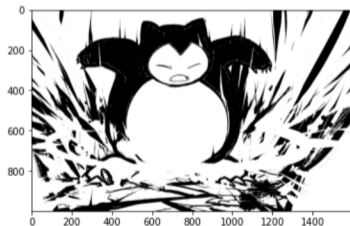
```
# Perform thresholding
```

```
processedImg = grayImgUint > 128
```

```
# Show the image
```

```
plt.figure(); plt.imshow(processedImg, cmap='gray')
```

<matplotlib.image.AxesImage at 0x7fc145092a50>



# Grayscale Thresholding (Otsu's Method)

- We need a way to **automatically determine the threshold value  $T$**  so that the result of thresholding is reproducible.
- A well-known approach is **Otsu's method**
  1. Select an initial estimate of the threshold  $T$ . A good initial value is the average intensity of the image.
  2. Calculate the mean gray values  $\mu_1$  and  $\mu_2$  of the partitions,  $R_1, R_2$ .
  3. Partition the image into two groups,  $R_1, R_2$ , using the threshold  $T$ .
  4. Compute a new threshold

$$T = \frac{1}{2}(\mu_1 + \mu_2)$$

5. Repeat steps 2-4 until the mean values  $\mu_1$  and  $\mu_2$  in successive iterations do not change.



# Grayscale Thresholding (Otsu's Method)

- To perform **Otsu's thresholding**, you need to first import cv2  
`import cv2`
- Then use **threshold()** method of the cv2 module.

## Syntax

```
cv2.threshold(source, thresholdValue, maxVal, thresholdingTechnique)
```

## Parameters:

- source: input image array (must be grayscale)
- thresholdValue: value of threshold below and above which pixel values will change accordingly
- maxVal: Maximum value that can be assigned to a pixel
- thresholdingTechnique: The type of thresholding to be applied  
(For Otsu's, we put `cv2.THRESH_BINARY + cv2.THRESH_OTSU`)
- Return value: Binary image.

```
# Assume Google Drive has been mounted & the path has been added for interpreter to search
```

```
# Import all the required libraries
```

```
import cv2
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
```

```
# Read the image
```

```
img = mpimg.imread('snorlax.png')
```

```
# Convert the color image to gray and show it
```

```
grayImg = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
plt.figure(); plt.imshow(grayImg, cmap='gray', vmin=0, vmax=1)
```

```
# Convert pixel values from [0,1] to [0,255]
```

```
grayImgUint = grayImg*255
grayImgUint = grayImgUint.astype(np.uint8)
```

```
# Perform thresholding using Otsu's method
```

```
thresh, processedImg = cv2.threshold(grayImgUint, 120, 255,
                                     cv2.THRESH_BINARY + cv2.THRESH_OTSU)
```

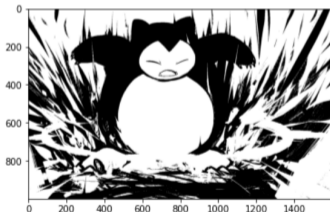
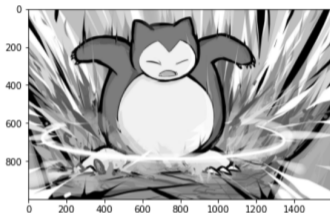
```
print('Optimal threshold:', thresh)
```

```
# Show the image
```

```
plt.figure(); plt.imshow(processedImg, cmap='gray')
```

Optimal threshold: 153.0

<matplotlib.image.AxesImage at 0x7fc14500ef10>



# Histogram Equalization

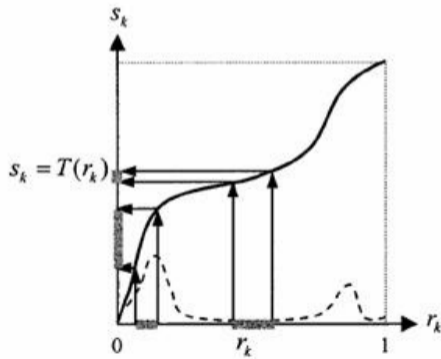
- Histogram equalization is another technique used to improve contrast of images.
- The idea is to spread out the most frequent intensity values.
- Algorithm

1. Compute the histogram,  $H$ , of the image
2. Compute the cumulative histogram,  $C$ , of the image

$$C(i) = \sum_{j=0}^i H(j)$$

3. Map old intensity value to new intensity value as follows:

$$I_{new} = C(I)$$



```

# Assume Google Drive has been mounted & the path has been added for interpreter to search

# Import all the required libraries
import cv2
import matplotlib.image as mpimg
import matplotlib.pyplot as plt

# Read the image
img = mpimg.imread('snorlax-low-contrast.png')

# Prepare subplots
fig = plt.figure(figsize=(12,9))
fig.tight_layout();
fig.subplots_adjust(wspace=0.2, hspace=0.2)

# Add the image to the first row, first col
gImg = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
ax1 = fig.add_subplot(2, 2, 1)
ax1.title.set_text("Original")
plt.imshow(gImg, cmap='gray', vmin=0, vmax=1)

# Convert pixel values from [0,1] to [0,255]
gImgUint = gImg*255
gImgUint = gImgUint.astype(np.uint8)
# Produce histogram
hist1 = cv2.calcHist([gImgUint],
                    [0], None, [256], [0,255])

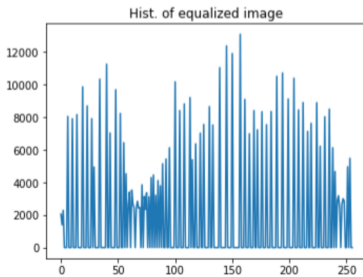
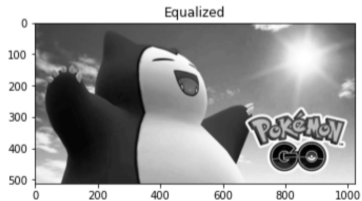
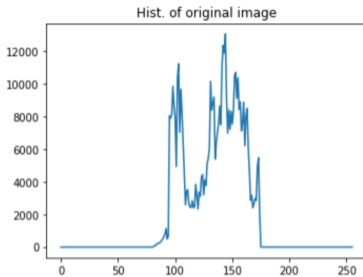
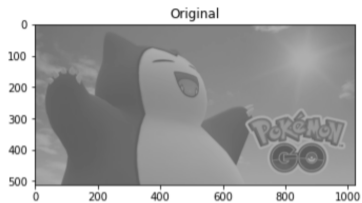
# Add the image to the first row, second col
ax2 = fig.add_subplot(2, 2, 2)
ax2.title.set_text('Hist. of original image')
plt.plot(hist1);

# Produce cumulative histogram
cumHist = np.cumsum(hist1)
cumMax = np.max(cumHist)
table = np.array((cumHist/np.max(cumHist))*255,
                np.uint8)
equalizedImg = cv2.LUT(gImgUint, table)
# Add the image to the second row, first col
ax3 = fig.add_subplot(2, 2, 3)
ax3.title.set_text("Equalized")
plt.imshow(equalizedImg, cmap='gray',
            vmin=0, vmax=255)

# Produce histogram
hist2 = cv2.calcHist([equalizedImg],
                    [0], None, [256], [0,255])
# Add the image to the second row, second col
ax4 = fig.add_subplot(2, 2, 4)
ax4.title.set_text('Hist. of equalized image')
plt.plot(hist2);

```

# Example



# Histogram Equalization

- In fact, **histogram equalization** can be performed using OpenCV function. To do so, you need to first import cv2  
`import cv2`
- Then use **equalizeHist()** method of the cv2 module.

## Syntax

```
equalizeHist(source)
```

## Parameters:

- source: input image array
- Return value: Equalized image

```
# Assume Google Drive has been mounted & the path has been added for interpreter to search
```

```
# Import all the required libraries
```

```
import cv2; import numpy as np  
import matplotlib.image as mpimg; import matplotlib.pyplot as plt
```

```
img = mpimg.imread('snorlax-low-contrast.png') # Read the image
```

```
# Prepare subplots
```

```
fig = plt.figure(figsize=(12,9))  
fig.set_figwidth(6); fig.tight_layout()  
fig.subplots_adjust(wspace=0.2, hspace=0.2)
```

```
# Convert pixel values from [0,1] to [0,255]
```

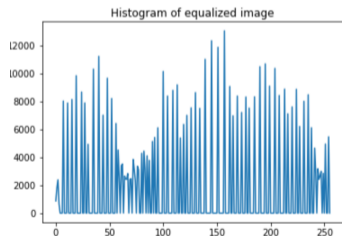
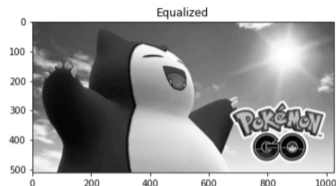
```
grayImg = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)  
grayImgUint = grayImg*255  
grayImgUint = grayImgUint.astype(np.uint8)
```

```
# Perform histogram equalization
```

```
equalizedImg = cv2.equalizeHist(grayImgUint)  
ax1 = fig.add_subplot(2, 1, 1); ax1.title.set_text('Equalized')  
plt.imshow(equalizedImg, cmap='gray', vmin=0, vmax=255)
```

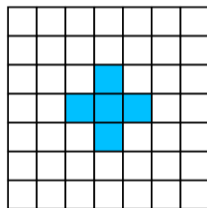
```
# Produce histogram
```

```
hist = cv2.calcHist([equalizedImg], [0], None, [256], [0,255])  
ax2 = fig.add_subplot(2, 1, 2)  
ax2.title.set_text('Histogram of equalized image'); plt.plot(hist)
```

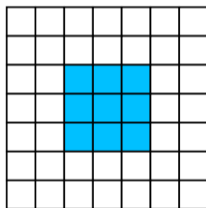


# Local Operations

- Recall, local operations refer to those the **output value at a specific coordinate** is dependent on the **input values in the neighbourhood** of that same coordinate.
- Some of the most **common neighbourhoods** are **4-connected neighbourhood** and the **8-connected neighbourhood**.



Rectangular sampling  
4-connected



Rectangular sampling  
8-connected



# Examples

- **Image smoothing:** It removes noise and softens edges and corners of the image. It is also called blurring.
- **Image edge detection:** It detects the boundaries (edges) of objects, or regions within an image.
- **Image sharpening:** It removes blur, enhances details, and dehazes.



# Image Convolution

Image convolution is defined as

$$O(x, y) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} K(m, n)I(x - m, y - n)$$

where  $I$  is the **input image**,  $K$  is the **image kernel**.

- Assume the origin (i.e.,  $(0,0)$ ) of  $I$  is top-left corner, while
- the origin (i.e.,  $(0,0)$ ) of  $K$  is the center of the kernel.

If the image kernel is  $3 \times 3$ , then

$$O(x, y) = \sum_{m=-1}^1 \sum_{n=-1}^1 K(m, n)I(x - m, y - n)$$

Image kernel is also called image filter or image mask.

## Example

Input image I

10	1	3	2	6
4	3	5	8	0
8	7	9	6	5

Image kernel K

-1	0	1
-1	0	1
-1	0	1

$$O(x, y) = \sum_{m=-1}^1 \sum_{n=-1}^1 K(m, n)I(x - m, y - n)$$

$$\begin{aligned} O(1, 1) &= \sum_{m=-1}^1 (K(m, -1)I(1 - m, 1 - (-1)) + K(m, 0)I(1 - m, 1 - 0) + K(m, 1)I(1 - m, 1 - 1)) \\ &= K(-1, -1)I(2, 2) + K(-1, 0)I(2, 1) + K(-1, 1)I(2, 0) + \\ &\quad K(0, -1)I(1, 2) + K(0, 0)I(1, 1) + K(0, 1)I(1, 0) + \\ &\quad K(1, -1)I(0, 2) + K(1, 0)I(0, 1) + K(1, 1)I(0, 0) \\ &= (-1)(9) + (-1)(5) + (-1)(3) + (0)(7) + (0)(3) + (0)(1) + (1)(8) + (1)(4) + (1)(10) \\ &= -9 - 5 - 3 + 8 + 4 + 10 = 5 \end{aligned}$$

Can you calculate all the remaining output image values?

Answer:

5	-5	6
---	----	---

# Very Tedious :( Any Intuitive Way? YES!!!

- Steps

- Inverse the kernel, i.e., flipping the kernel in both horizontal and vertical directions about the center of kernel.

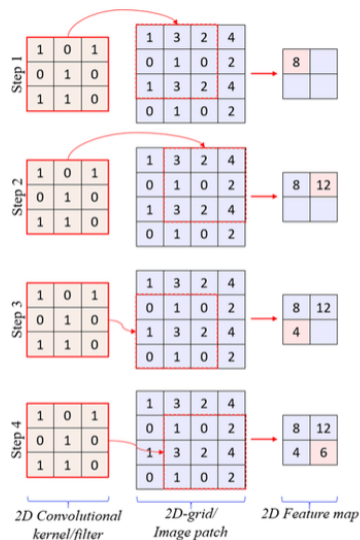
-1	0	1
-1	0	1
-1	0	1

1	0	-1
1	0	-1
1	0	-1

1	0	-1
1	0	-1
1	0	-1

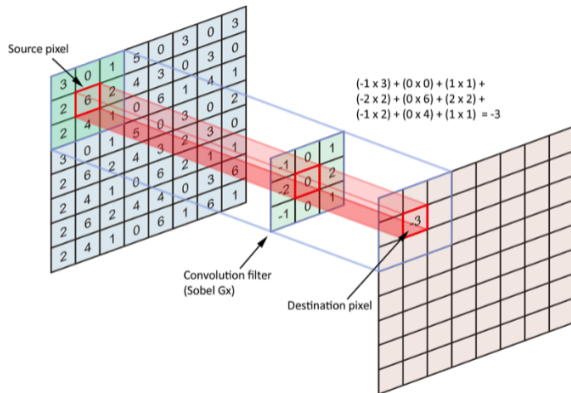
(Left) Original kernel, (Middle) Flipped horizontally, (Right) Flipped vertically

- Slide over the inversed kernel centered at interested point.
- Multiply inversed kernel data with the overlapped area.
- Sum and accumulate the output.



Step 2 to Step 4

# Image Convolution Again



# Problem

- When computing an **output pixel at the boundary of an image**, a portion of the convolution is usually **off the edge of the image**. How to deal with this?
- Solutions:
  1. Just **ignore those boundary pixels**. :P
  2. **Do zero padding** (i.e., add a border of pixels all with value zero around the edges of the input images.)

0	0	0	0	0	0	0
0	10	1	3	2	6	0
0	4	3	5	8	0	0
0	8	7	9	6	5	0
0	0	0	0	0	0	0

3. **Replicating boundary pixels**

10	10	1	3	2	6	6
10	10	1	3	2	6	6
4	4	3	5	8	0	0
8	8	7	9	6	5	5
8	8	7	9	6	5	5

- Solutions:

- 4. Reflecting boundary pixels

10	10	1	3	2	6	6
10	10	1	3	2	6	6
4	4	3	5	8	0	0
8	8	7	9	6	5	5
8	8	7	9	6	5	5

- 5. Mirroring boundary pixels

3	4	3	5	8	0	8
1	10	1	3	2	6	2
3	4	3	5	8	0	8
7	8	7	9	6	5	6
3	4	3	5	8	0	8

# Image Convolution

- To do **image convolution**, you need to first import cv2  
`import cv2`
- Then use **filter2D()** method of the cv2 module.

## Syntax

```
cv2.filter2D(src, ddepth, kernel, dst, anchor, delta, borderType=cv2.BORDER_DEFAULT)
```

## Parameters:

- src: input image that you want to convolve
- ddepth: desired depth of the destination image. If ddepth=-1, the output image will have the same depth as the src
- kernel: convolution kernel, a single-channel floating point matrix; if you want to apply different kernels to different channels, split the image into separate color planes using split and process them individually.
- dst: output image of the same size and the same number of channels as src.
- anchor: anchor of the kernel that indicates the relative position of a filtered point within the kernel; the anchor should lie within the kernel; default value (-1,-1) means that the anchor is at the kernel center.
- delta: optional value added to the filtered pixels before storing them in dst.
- borderType: pixel extrapolation method
  - cv2.BORDER\_CONSTANT: `iiiiii|abcdefgh|iiiiii` with some specified i
  - cv2.BORDER\_REPLICATE: `aaaaaa|abcdefgh|hhhhhh`
  - cv2.BORDER\_REFLECT: `fedcba|abcdefgh|hgfedc`
  - cv2.BORDER\_REFLECT\_101: `gfedcb|abcdefgh|gfedcba`
  - cv2.BORDER\_DEFAULT: Same as BORDER\_REFLECT\_101
- Return value: filtered image.



# Image Convolution

## Note

filter2D **does not mirror** the kernel for you. You will need to flip the kernel before applying cv2.filter2D.

```
# Assume Google Drive has been mounted & the path has been added for interpreter to search

# Import all the required libraries
import cv2; import numpy as np
import matplotlib.image as mpimg
import matplotlib.pyplot as plt

# Read the image
img = mpimg.imread('snorlax-sleep.png')

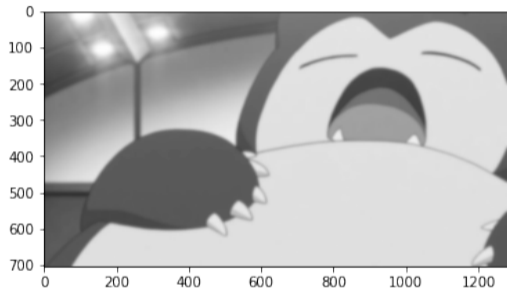
# Convert the color image to gray and show it
grayImg = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
plt.figure(); plt.imshow(grayImg, cmap='gray',
                          vmin=0, vmax=1)

# Prepare a kernel (a sharpening kernel here)
kernel_3x3 = np.array([ [0,-1,0],
                        [-1,5,-1],
                        [0,-1,0] ])

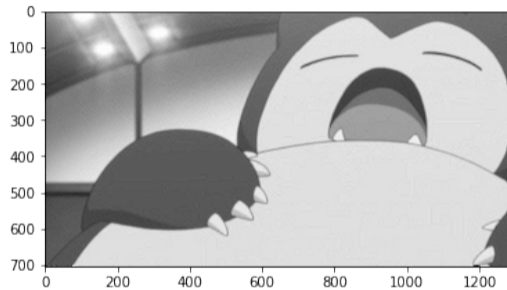
for i in range(5): # Perform filtering 5 times
    grayImg = cv2.filter2D(grayImg, -1,
                           kernel_3x3)

# Show the resulting image
plt.figure(); plt.imshow(grayImg, cmap="gray",
                          vmin=0, vmax=1)
```

# Before and After



Input image



Output image

# Smoothing (Averaging/Blurring) Kernel

$1/9$	$1/9$	$1/9$
$1/9$	$1/9$	$1/9$
$1/9$	$1/9$	$1/9$



Original image

Effects of kernel in different size. (What do you observe?)



3×3 mask



5×5 mask



15×15 mask



25×25 mask

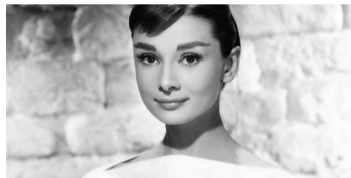
- Blurring an image can be done by averaging pixels
- Analogous to integration, related to sum of pixel intensity values

# Sharpening Kernel

0	0	0
0	1	0
0	0	0

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

-1	-1	-1
-1	9	-1
-1	-1	-1



Original image

+



Detail (Edge)  
[Color flipped for clarity]

=



Sharpened image

- Sharpening has the opposite effect of blurring
- Analogous to differentiation, related to the difference of pixel intensity values

# Edge Kernel - Prewitt

-1	0	1
-1	0	1
-1	0	1

-1	-1	-1
0	0	0
1	1	1

Kernel for detecting vertical edges

Kernel for detecting horizontal edges



Edge image (vertical edges)

$$|G_x|$$



Edge image (horizontal edges)

$$|G_y|$$



Edge image (magnitude)

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

Pixels of the processed images are inverted (i.e. black to white, white to black) for making them more visible.

# Edge Kernel - Sobel

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

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

Kernel for detecting vertical edges

Kernel for detecting horizontal edges



Edge image (vertical edges)

$$|G_x|$$



Edge image (horizontal edges)

$$|G_y|$$



Edge image (magnitude)

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

Pixels of the processed images are inverted (i.e. black to white, white to black) for making them more visible.

## Interesting Kernels



Original image

0	0	0
0	1	0
0	0	0

Identity kernel



Resulting image

Convolve the original image with the kernel 5 times, we still get back the identical image.



Original image

0	0	0
1	0	0
0	0	0

Shifted identity kernel



Resulting image

Convolve the original image with the kernel 5 times, we get back an image shifted by 5 pixels.

# Image Convolutions

- Clearly, **image convolution** is **powerful in finding the features of an image** if we already know the right kernel to use.
- Kernel design is an art and has been refined over the last few decades to do some pretty amazing things with images. But the important question is, what if we don't know the features we are looking for? Or what if we do know, but we don't know what kernel should look like?





# Non-linear Filtering

- **Non-linear filters** are typically more powerful than linear filters
  - Suppression of spikes
  - Edge preserving properties
- Examples:
  - Median filter
  - Morphological filters



# Median Filter

- The **median filter** is often used to **remove noise** from an image.
- It **preserves edges** while removing noise. Also, **no new gray value is introduced**.
- Idea:
  - Consider each pixel in the image in turn and looks at its nearby neighbors to decide whether or not it is representative of its surroundings.
  - It replaces the pixel with the median of those values.
  - The median is calculated by first sorting all the pixel values from the surrounding neighborhood into numerical order and then replacing the pixel being considered with the middle pixel value.
- Example:

123	125	126	130	140
122	124	126	127	135
118	120	150	125	134
119	115	119	123	133
111	116	110	120	130

- Neighborhood values: 124, 126, 127, 120, 150, 125, 115, 119, 123
- Sort the values: 115, 119, 120, 123, 124, 125, 126, 127, 150
- Pick the median value: 124

# Median Filter

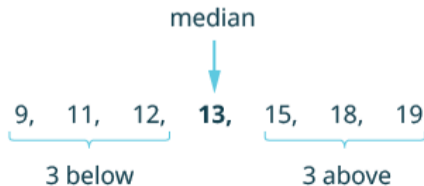
- To perform **median filtering**, you need to first import cv2  
`import cv2`
- Then use **medianBlur()** method of the cv2 module.

## Syntax

```
cv2.medianBlur(src, kernelSize)
```

## Parameters:

- src: input image that you want to process
- kernelSize: The size of the kernel
- Return value: filtered image.



```
# Assume Google Drive has been mounted & the path has been added for interpreter to search

# Import all the required libraries
import cv2; import numpy as np
import matplotlib.image as mpimg
import matplotlib.pyplot as plt

# Read the image
grayImg = mpimg.imread('snorlax-noisy-result.png')
# Convert the pixel values from [0,1] to [0,255]
grayImgUint = grayImg*255
grayImgUint = grayImgUint.astype(np.uint8)
plt.figure(); plt.imshow(grayImgUint, cmap='gray',
                          vmin=0, vmax=255)

# Perform median filtering
resultImg = cv2.medianBlur(grayImgUint, 5)

# Show the resulting image
plt.figure();
plt.imshow(resultImg, cmap="gray", vmin=0, vmax=255)
```



## Part III

### Reference Materials



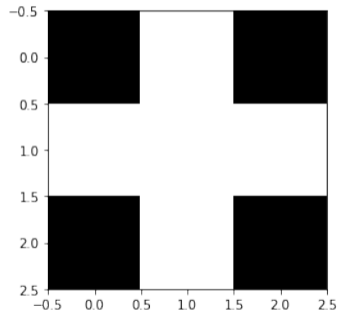
# Morphological Filter

- Morphological filters are a set of image processing operations where the shapes of the image's object are manipulated.
- Similar to convolutional kernels, morphological operations utilize a structuring element to transform each pixel of an image to a value based on its neighbors' value.
- An example of structuring element:

```
import numpy as np

structuring_element = np.array([ [0,1,0],
                                [1,1,1],
                                [0,1,0] ], np.uint8)

plt.imshow(structuring_element, cmap='gray');
```

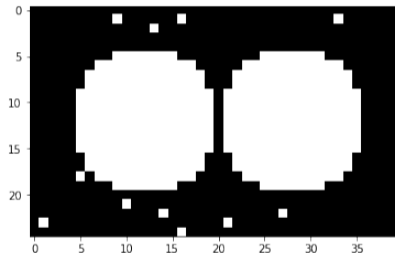


# Morphological Filter

- To demonstrate how morphological filter work, let us create two adjacent circles with random noise on its background.

```
from skimage.draw import disk
import numpy as np

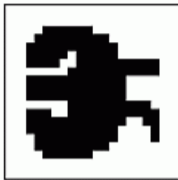
circle_image = np.zeros((25, 40))
circle_image[disk((12, 12), 8)] = 1
circle_image[disk((12, 28), 8)] = 1
for x in range(20):
    circle_image[np.random.randint(25),
                 np.random.randint(40)] = 1
plt.imshow(circle_image, cmap='gray');
```



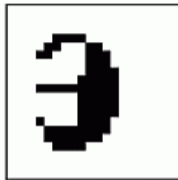
# Examples

- Erosion
- Dilation
- Opening
- Closing

a. Original



b. Erosion



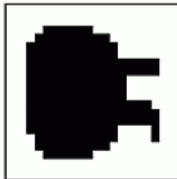
c. Dilation



d. Opening



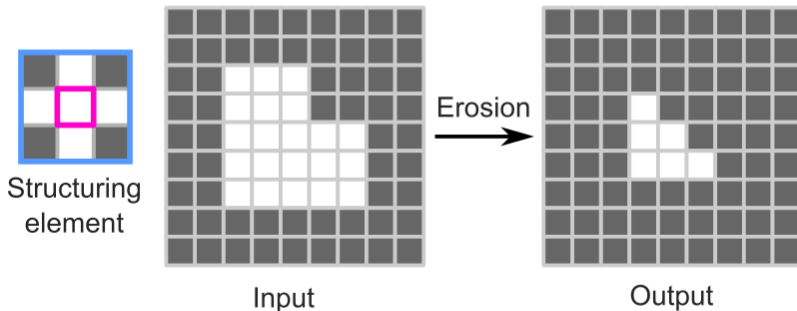
e. Closing





# Erosion Filter

- **Erosion** is used for **shrinking of element in input image** by using the structuring element.
- The **pixel values are retained** only when the **structuring element is completely contained inside input image**. Otherwise, it gets deleted or eroded.



# Erosion Filter

- To perform **erosion**, you need to first import cv2  
`import cv2`
- Then use **erode()** method of the cv2 module.

## Syntax

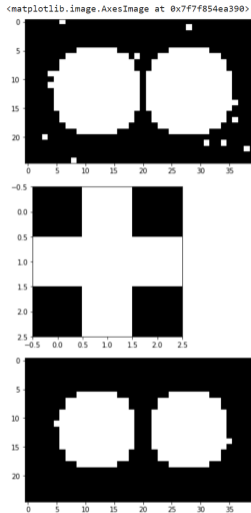
```
cv2.erode(src, kernel, dst, anchor, iterations, borderType, borderValue)
```

## Parameters:

- src: input image that you want to erode
- kernel: A structuring element used for erosion
- dst: Output image
- anchor: Integer representing anchor point and its default value Point is (-1,-1) which means that the anchor is at the kernel center.
- borderType: cv2.BORDER\_CONSTANT, cv2.BORDER\_REFLECT, etc.
- iterations: Number of times erosion is applied.
- borderValue: It is border value in case of a constant border.
- Return value: filtered image.

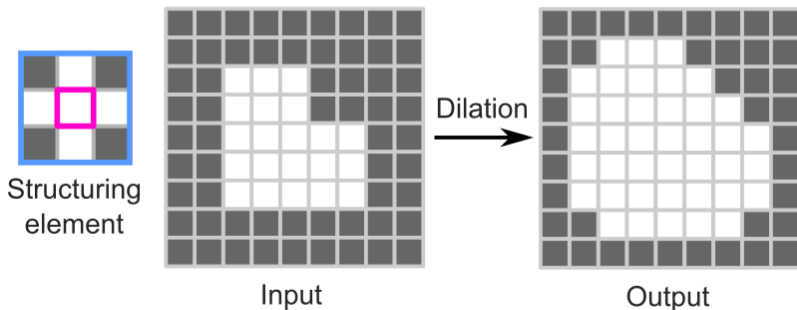
```
import cv2
import matplotlib.pyplot as plt

# Show circle_image
plt.figure(); plt.imshow(circle_image, cmap='gray')
# Show structuring element
plt.figure(); plt.imshow(structuring_element, cmap='gray');
# Perform erosion filter
eroded_img = cv2.erode(circle_image, structuring_element)
# Show the resulting image
plt.figure(); plt.imshow(eroded_img, cmap='gray')
```



# Dilation Filter

- **Dilation** is used for **expanding of element in input image** by using the structuring element.
- The **pixel values are “on”** only when the **structuring element has overlapped with the input image**. Otherwise, the pixel values are “off”.



# Dilation Filter

- To perform **dilation**, you need to first import cv2  
`import cv2`
- Then use **dilate()** method of the cv2 module.

## Syntax

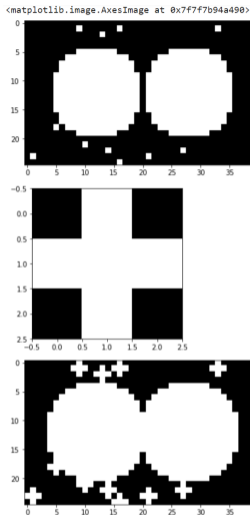
```
cv2.dilate(src, kernel, dst, anchor, iterations, borderType, borderValue)
```

## Parameters:

- src: input image that you want to dilate
- kernel: A structuring element used for dilation
- dst: Output image
- anchor: Integer representing anchor point and its default value Point is (-1,-1) which means that the anchor is at the kernel center.
- borderType: cv2.BORDER\_CONSTANT, cv2.BORDER\_REFLECT, etc.
- iterations: Number of times dilation is applied.
- borderValue: It is border value in case of a constant border.
- Return value: filtered image.

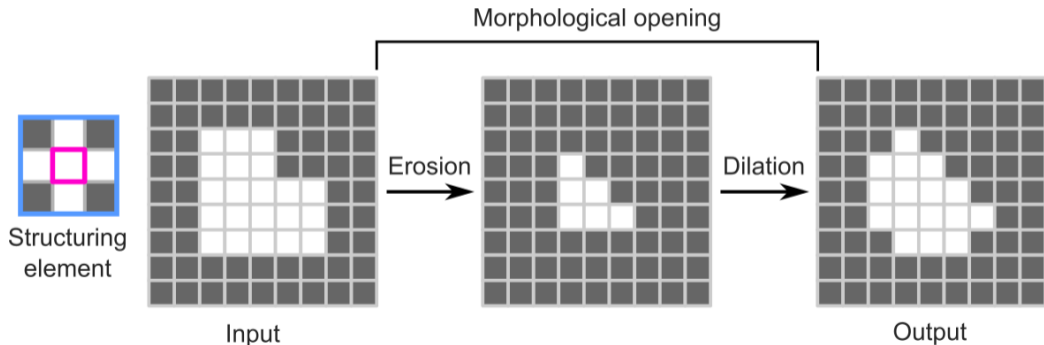
```
import cv2
import matplotlib.pyplot as plt

# Show circle_image
plt.figure(); plt.imshow(circle_image, cmap='gray')
# Show structuring element
plt.figure(); plt.imshow(structuring_element, cmap='gray');
# Perform erosion filter
dilated_img = cv2.dilate(circle_image, structuring_element)
# Show the resulting image
plt.figure(); plt.imshow(dilated_img, cmap='gray')
```



# Opening Filter

- Opening filter removed small objects while also maintaining the original shape of the object.
- Opening is done by applying the erosion first, and then applying dilation.



# Opening Filter

- To perform **opening**, you need to first import cv2  
`import cv2`
- Then use **morphologyEx()** method of the cv2 module.

## Syntax

```
cv2.morphologyEx(src, op, kernel, dst=None, anchor=None,  
                 iterations=None, borderType=None, borderValue=None)
```

## Parameters:

- src: input image that you want to process
- op: Operations (cv2.MORPH\_ERODE, cv2.MORPH\_DILATE, cv2.MORPH\_OPEN, cv2.MORPH\_CLOSE)
- kernel: A structuring element used for dilation
- dst: Output image
- anchor: Integer representing anchor point and its default value Point is (-1,-1) which means that the anchor is at the kernel center.
- iterations: Number of times dilation is applied (e.g. iterations = 2, erode×2, dilate×2).
- borderType: cv2.BORDER\_CONSTANT, cv2.BORDER\_REFLECT, etc.
- borderValue: It is border value in case of a constant border.
- Return value: filtered image.



```

import cv2
import matplotlib.pyplot as plt

img = plt.imread('input-open.png')

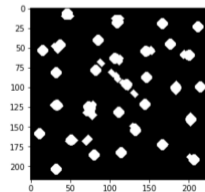
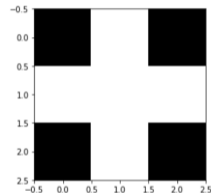
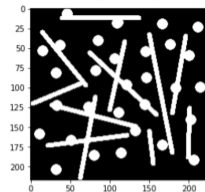
# Show input image
plt.figure(); plt.imshow(img, cmap='gray')
# Show structuring element
plt.figure();
plt.imshow(structuring_element, cmap='gray');

# Perform opening filter
opened_img = cv2.morphologyEx(img, cv2.MORPH_OPEN,
                               structuring_element,
                               iterations=4)

# Show the resulting image
plt.figure();
plt.imshow(opened_img, cmap='gray')

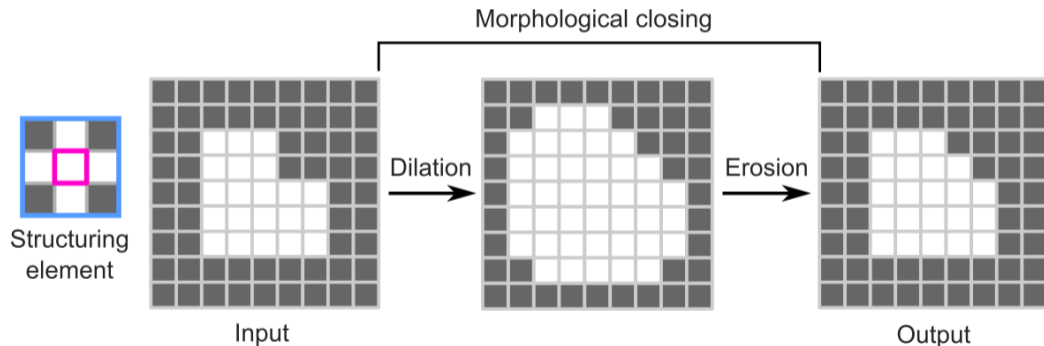
```

matplotlib.image.AxesImage at 0x7f



# Closing Filter

- Closing filter removes small holes while also maintaining the original shape of the object.
- Closing is done by applying the dilation first, and then applying erosion.



## Closing Filter

- To perform **closing**, you need to first import cv2  
`import cv2`
- Then use **morphologyEx()** method of the cv2 module.

### Syntax

```
cv2.morphologyEx(src, op, kernel, dst=None, anchor=None,  
                 iterations=None, borderType=None, borderValue=None)
```

### Parameters:

- src: input image that you want to process
- op: Operations (cv2.MORPH\_ERODE, cv2.MORPH\_DILATE, cv2.MORPH\_OPEN, cv2.MORPH\_CLOSE)
- kernel: A structuring element used for dilation
- dst: Output image
- anchor: Integer representing anchor point and its default value Point is (-1,-1) which means that the anchor is at the kernel center.
- iterations: Number of times dilation is applied (e.g. iterations = 2, erode×2, dilate×2).
- borderType: cv2.BORDER\_CONSTANT, cv2.BORDER\_REFLECT, etc.
- borderValue: It is border value in case of a constant border.
- Return value: filtered image.

```
import cv2
import matplotlib.pyplot as plt

img = plt.imread('input-close.png')

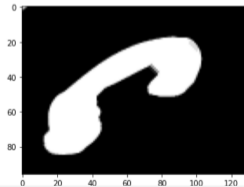
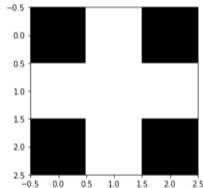
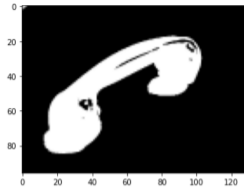
# Show input image
plt.figure(); plt.imshow(img, cmap='gray')

# Show structuring element
plt.figure();
plt.imshow(structuring_element, cmap='gray',
           vmin=0, vmax=1);

# Perform closing filter
closed_img = cv2.morphologyEx(img, cv2.MORPH_CLOSE,
                              structuring_element,
                              iterations=4)

# Show the resulting image
plt.figure();
plt.imshow(closed_img, cmap='gray')
```

<matplotlib.image.AxesImage at 0x7f7f71a5bb>



# Useful Links

- `imread()`: [https://matplotlib.org/stable/api/\\_as\\_gen/matplotlib.pyplot.imread.html](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.imread.html)
- `imshow()`: [https://matplotlib.org/stable/api/\\_as\\_gen/matplotlib.pyplot.imshow.html](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.imshow.html)
- `imsave()`: [https://matplotlib.org/stable/api/\\_as\\_gen/matplotlib.pyplot.imsave.html](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.imsave.html)
- `cvtColor()`: [https://docs.opencv.org/3.4/df/d9d/tutorial\\_py\\_colorspaces.html](https://docs.opencv.org/3.4/df/d9d/tutorial_py_colorspaces.html)
- `warpAffine()`, `getRotationMatrix2D()`, `resize()`:  
[https://docs.opencv.org/3.4/da/d6e/tutorial\\_py\\_geometric\\_transformations.html](https://docs.opencv.org/3.4/da/d6e/tutorial_py_geometric_transformations.html)
- `copyMakeBorder()`: [https://docs.opencv.org/3.4/dc/da3/tutorial\\_copyMakeBorder.html](https://docs.opencv.org/3.4/dc/da3/tutorial_copyMakeBorder.html)
- `calcHist()`: [https://docs.opencv.org/3.4/dd/d0d/tutorial\\_py\\_2d\\_histogram.html](https://docs.opencv.org/3.4/dd/d0d/tutorial_py_2d_histogram.html)
- `convertScaleAbs()`:  
[https://docs.opencv.org/3.4/d2/de8/group\\_\\_core\\_\\_array.html#ga3460e9c9f37b563ab9dd550c4d8c4e7d](https://docs.opencv.org/3.4/d2/de8/group__core__array.html#ga3460e9c9f37b563ab9dd550c4d8c4e7d)
- `threshold()`: [https://docs.opencv.org/3.4/d7/d4d/tutorial\\_py\\_thresholding.html](https://docs.opencv.org/3.4/d7/d4d/tutorial_py_thresholding.html)
- `equalizeHist()`: [https://docs.opencv.org/3.4/d5/daf/tutorial\\_py\\_histogram\\_equalization.html](https://docs.opencv.org/3.4/d5/daf/tutorial_py_histogram_equalization.html)
- `filter2D()`, `medianBlur()`: [https://docs.opencv.org/3.4/d4/d13/tutorial\\_py\\_filtering.html](https://docs.opencv.org/3.4/d4/d13/tutorial_py_filtering.html)
- `erode()`, `dilate()`, `morphologyEx()`,  
[https://docs.opencv.org/3.4/d9/d61/tutorial\\_py\\_morphological\\_ops.html](https://docs.opencv.org/3.4/d9/d61/tutorial_py_morphological_ops.html)

That's all!

Any questions?

