MUHAMMAD KHAIR NOORDIN

COMPUTER VISION ESSENTIALS

Face and Object Detection

0 Getting Started

- Download and install Anaconda
  - https://www.anaconda.com/
- Install it to your computer
- Install the libraries needed for the course that provided in .yaml file.
- Choose your OS, either Windows or MacOS.
- Run Anaconda Prompt by searching it in your OS.
- Make sure that you are in correct directory in which the .yml is located.
- Type this :

```
conda env create -f cvcourse_windows.yml
```

- This command is to create a new virtual environment

What is Virtual Environment?

A virtual environment in Python is a self-contained directory that contains a specific version of Python and a collection of additional packages. Each virtual environment is independent, allowing you to work on multiple projects with different dependencies without causing conflicts between them.

Key Features of Virtual Environments

1. Isolation:
   - Each virtual environment has its own Python interpreter and libraries, separate from the system-wide Python installation.
   - Changes to one environment do not affect other environments or the system Python.

2. Dependency Management:
   - You can install specific versions of packages required for a project, ensuring that dependencies are managed correctly and do not conflict with other projects.

3. Reproducibility:
   - Virtual environments help ensure that a project runs with the same dependencies across different machines, making it easier to reproduce and share your work.

Getting Started

- After creating the environment, you need to activate the environment to use the installed packages:

```
conda activate opencv_FAI
```

- To open the interactive computing environment, type:

```
jupyter-lab
```

What is Juptyer?

Jupyter is an open-source project that provides an interactive computing environment, primarily used for data science, scientific computing, and machine learning. It allows users to create and share documents that contain live code, equations, visualizations, and narrative text. These documents are known as Jupyter Notebooks.

- For first time connection, you need to copy and paste the link given (example

```
http://localhost:8888/?token=abcdeaeafa021312319js
```

Create manually

- You also can create the virtual environment from scratch and manually installed the package that you like.
- You can type this:

```
conda create --name coursename
```

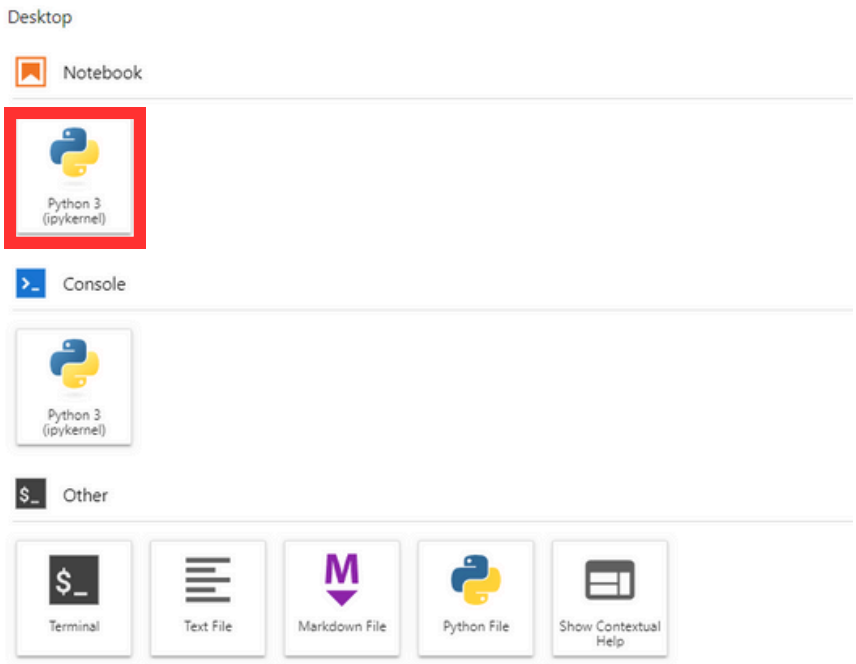
- To install the package, you need to start with the word 'conda install' and followed by package name. For example:

```
conda install numpy
conda install pandas
conda install scikit-learn
conda install matplotlib
conda install opencv
conda install jupyter
```

0 Getting Started

Starting the service

- Click on '+' sign and then Choose Notebook > Python 3



- Now you are ready to write some beginner code for computer vision.

```
print ('hello world')
```

- To run this code, click on the play button or shortcut - Shift + Enter. Check the output.
- You can run this code using as Python file by clicking on the '+' sign again, and click on 'Python file'. Type the same thing.
- Save the Python file. For example the name is 'hello.py'.
- In order to run the Python file that we save, we need to run it via terminal.
- Click again on '+' sign and now pick 'Terminal'.
- To run it, type:

```
python hello.py
```

Numpy & Basic Images

How computer vision works?

It consists 5 steps:

1. **Image acquisition:** the process of capturing images from a camera or retrieving them from a storage device. The images are then converted into a digital format that a computer can process.
2. **Preprocessing:** This step involves preparing the image for further analysis. Preprocessing techniques improve the quality of the image and remove any noise.
    - Grayscale Conversion
    - Noise Reduction
    - Resizing
3. **Feature Extraction:** Identifying and describing important features of the image. These features are used to represent the image in a more compact and informative way.
    - Edge Detection
    - Keypoint Detection
4. **Object Detection:** Locating objects within an image and drawing bounding boxes around them.
    - Using Haar Cascades:
    - Object Recognition
5. Post-processing: Refining the detected objects and interpreting the results for the intended application.

1 Numpy & Basic Images

What is NumPy?
NumPy (Numerical Python) is a fundamental package for scientific computing in Python. It provides support for arrays, which are collections of numbers arranged in a grid of dimensions. NumPy offers a range of mathematical functions and operations to work efficiently with these arrays.

Installation:
To install NumPy, you can open the terminal in Jupyter Lab, and type:

```
conda install numpy
```

Basic Usage:

- You need to import NumPy if you want to use it in your project

```
import numpy as np
```

- Creating arrays

```
# Creating a 1D array
arr = np.array([1, 2, 3, 4, 5])

# Creating a 2D array
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])

# Creating an array of zeros
zeros = np.zeros((3, 3))

# Creating an array of ones
ones = np.ones((2, 4))

# Creating an array with a range of values
range_arr = np.arange(10)  # From 0 to 9

# Creating an array with a range of values with a step
step_arr = np.arange(0, 10, 2)  # From 0 to 8 with a step of 2
```

Numpy & Basic Images

Random Arrays

- np.random.seed is used to seed the random number generator in NumPy, which initializes the random number generation algorithm to produce a reproducible sequence of random numbers.
- When you set a seed, the random numbers generated by functions in np.random will be the same each time you run your code, allowing for reproducibility and consistent results.

```
import numpy as np

# Set the seed to ensure reproducibility
np.random.seed(42)

# Generate a random array
random_array = np.random.rand(3)
print(random_array)  # Output will be the same every time
```

1 Numpy & Basic Images

- Array Attributes:

```
arr = np.array([[1, 2, 3], [4, 5, 6]])

# Shape of the array
print(arr.shape)  # Output: (2, 3)

# Number of dimensions
print(arr.ndim)  # Output: 2

# Data type of elements
print(arr.dtype)  # Output: int64 (may vary)

# Size of the array (number of elements)
print(arr.size)  # Output: 6
```

- Array Indexing and Slicing:

```
arr = np.array([1, 2, 3, 4, 5, 6])

# Accessing elements
print(arr[0])  # Output: 1
print(arr[1:4])  # Output: [2 3 4]

# Accessing elements in a 2D array
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
print(arr_2d[0, 1])  # Output: 2
print(arr_2d[:, 1])  # Output: [2 5]
```

1 Numpy & Basic Images

- Array Operations:

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Element-wise addition
print(arr1 + arr2)  # Output: [5 7 9]

# Element-wise multiplication
print(arr1 * arr2)  # Output: [ 4 10 18]

# Matrix multiplication
arr_2d_1 = np.array([[1, 2], [3, 4]])
arr_2d_2 = np.array([[5, 6], [7, 8]])
print(np.dot(arr_2d_1, arr_2d_2))  # Output: [[19 22] [43 50]]
```

- Statistical Functions:

```
arr = np.array([1, 2, 3, 4, 5])

# Mean of the array
print(np.mean(arr))  # Output: 3.0

# Standard deviation of the array
print(np.std(arr))  # Output: 1.4142135623730951

# Sum of the array
print(np.sum(arr))  # Output: 15

# Median of the array
print(np.median(arr))  # Output: 3.0
```

1 Numpy & Basic Images

- Reshaping and Resizing Arrays:

```
arr = np.array([1, 2, 3, 4, 5, 6])

# Reshaping to a 2x3 array
reshaped = arr.reshape((2, 3))
print(reshaped)  # Output: [[1 2 3] [4 5 6]]

# Flattening a 2D array to 1D
flattened = reshaped.flatten()
print(flattened)  # Output: [1 2 3 4 5 6]
```

- Advanced Indexing:

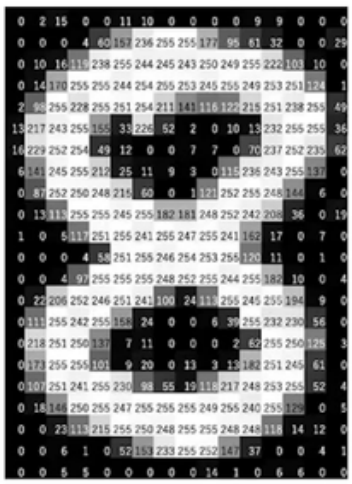
```
    arr = np.array([1, 2, 3, 4, 5])

    # Boolean indexing
    print(arr[arr > 3])  # Output: [4 5]

    # Fancy indexing
    indices = [0, 2, 4]
    print(arr[indices])  # Output: [1 3 5]
```

1 Numpy & Basic Images

How images are handled and stored in computers

- This is the image of the number 8. Now, if we zoom in further and if you look closely, you can see that the images are getting distorted, and you will see some small square boxes on this image.

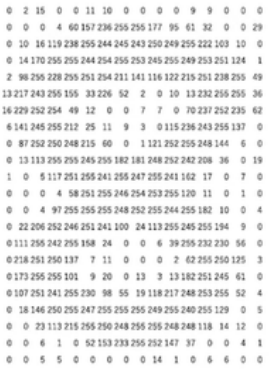
- These small boxes are called Pixels. We often use the term- the dimension of the image is X*Y. The number of pixels across the image's height(x) and width(y). In this case, if you count, it would be 24 pixels across the height and 16 pixels across the width. Although we see an image in this format, the computer store image in the form of numbers.



- Each of these pixels is denoted as a numerical value, and these numbers are called Pixel Values. These pixel values denote the intensity of the pixels. For a grayscale or B&W image, we have pixel values ranging from 0 to 255.

1 Numpy & Basic Images

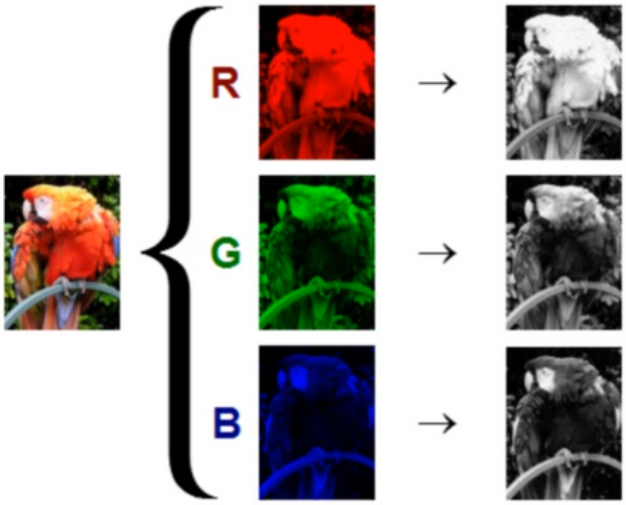
- The smaller numbers closer to zero represent the darker shade while the larger numbers closer to 255 represent the lighter or the white shade.
- So every image in a computer is saved in this form where you have a matrix of numbers, and this matrix is also known as a Channel.



- A channel is a matrix of pixel values, and we have only one channel in the case of a grayscale image.

Colour Images

- This image comprises many different colors. Almost all colors can be generated from the three primary colors – Red, Green, and Blue. Therefore, we can say that each colored image is a unique composition of these three colors or 3 channels – Red, Green, and Blue.



Numpy & Basic Images

- This means that in a colored image, the number of matrices or the number of channels will be more. In this particular example, we have 3 matrices – 1 matrix for red, known as the Red channel,
- Each of these metrics would again have values ranging from 0 to 255, where each of these numbers represents the intensity of the pixels. Or in other words, these values represent different shades of red, green, and blue. All of these channels or matrices superimpose over one another to form the shape of the image when loaded into a computer.
- When we check the shape of images, it will showing something like this:

```
(1280,720,3)
```

- Where: 1280 is the width, 720 is the height and 3 is the channel (3 colours).

Matplotlib

- Matplotlib is a widely-used plotting library in Python that allows you to create static, interactive, and animated visualizations in a variety of formats. It is particularly useful for generating plots, charts, histograms, scatter plots, bar graphs, and more. Matplotlib is highly customizable and integrates well with other libraries like NumPy, Pandas, and OpenCV.
- It plays an important role in visualizing and analyzing the results of computer vision tasks. Here's how Matplotlib is typically used in computer vision:
  - Displaying Images:
    - Matplotlib can be used to display images in Python, especially when working in environments like Jupyter Notebooks.
    - It can display images loaded using OpenCV, but it requires converting the image from BGR (used by OpenCV) to RGB (used by Matplotlib).
  - Visualizing Results of Image Processing:
    - After applying image processing techniques (e.g., edge detection, filtering), Matplotlib can be used to visualize the results.
    - This is useful for comparing the original image with the processed image.

Numpy & Basic Images

- Visualizing Feature Points:
  - In tasks like keypoint detection, Matplotlib can be used to visualize the keypoints detected on an image.
  - This is helpful for debugging or analyzing the performance of feature detection algorithms
  -
- Comparison of Multiple Images:
  - Matplotlib makes it easy to compare multiple images side by side, which is useful in computer vision to visualize different stages of image processing or the output of different algorithms.

PIL (Python Imaging Library)

- To use PIL, you generally need to install and import it in your Python environment. However, PIL itself is now outdated and has been succeeded by Pillow, a modern fork of PIL that is actively maintained and widely used. Most people use Pillow instead of the original PIL.
- Common PIL/Pillow Functions
  - Image.open(): Opens an image file and returns an Image object.
  - Image.save(): Saves an image to a file.
  - Image.resize(): Resizes an image to a given size.
  - Image.crop(): Crops the image to a specified box.
  - Image.rotate(): Rotates the image by a given angle.
  - Image.convert(): Converts the image to a different color mode (e.g., grayscale).
  - ImageFilter: Provides various filters like BLUR, CONTOUR, and SHARPEN.

1 Numpy & Basic Images

```
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np

# Open the image using Pillow
image = Image.open('example.jpg')

# Convert the image to a NumPy array using np.asarray
image_array = np.asarray(image)

# Print the shape of the array to verify
print("Shape of the image array:", image_array.shape)

# Display the image from the array
plt.imshow(image_array)
plt.axis('off')
plt.title("Image from Array")
plt.show()
```

1 Numpy & Basic Images

Image Shape (.shape)

The .shape attribute in NumPy is used to get the dimensions of an array. When dealing with images, this attribute gives you information about the image's height, width, and the number of channels (color layers) it has.

1. Grayscale Image:
    - Shape: (height, width)
    - A grayscale image contains only intensity information, which means each pixel has a single value representing the intensity of light. There is no color information.
    - Example: If an image has a shape of (200, 300), it means the image is 200 pixels tall and 300 pixels wide, with only one channel (grayscale).
2. RGB Image:
    - Shape: (height, width, 3)
    - An RGB image contains color information, with three channels corresponding to Red, Green, and Blue. Each pixel is represented by a triplet of values, one for each color channel.
    - Example: If an image has a shape of (200, 300, 3), it means the image is 200 pixels tall, 300 pixels wide, and has three color channels (RGB).

Channels in Images

- Grayscale:
    - Only one channel, where the intensity values range typically from 0 (black) to 255 (white) for 8-bit images.
    - A grayscale image has no color, just varying shades of gray.

- RGB:
    - Three channels, where each channel corresponds to one of the primary colors: Red, Green, and Blue.
    - Color in the image is created by combining these three channels, with values typically ranging from 0 to 255.

1 Numpy & Basic Images

Manipulate Channel Value

- We can manipulate value of certain channel to turn it off so the channel of the color do not appear in the image

```
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt

# Load the image
image = Image.open('path_to_image.jpg')

# Convert the image to a NumPy array
image_np = np.array(image)

# Turn off the Red channel
image_no_red = image_np.copy()
image_no_red[:, :, 0] = 0  # Set the Red channel to 0

# Turn off the Green channel
image_no_green = image_np.copy()
image_no_green[:, :, 1] = 0  # Set the Green channel to 0

# Turn off the Blue channel
image_no_blue = image_np.copy()
image_no_blue[:, :, 2] = 0  # Set the Blue channel to 0
```

1 Numpy & Basic Images

Manipulate Channel Value

- We can convert it back to the original value of the array, as long as we store the copy of the image value

```
# Convert the NumPy arrays back to images
image_no_red_pil = Image.fromarray(image_no_red)
image_no_green_pil = Image.fromarray(image_no_green)
image_no_blue_pil = Image.fromarray(image_no_blue)

# Display the images with the specific channel turned off
plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
plt.imshow(image_no_red_pil)
plt.title('Red Channel Turned Off')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(image_no_green_pil)
plt.title('Green Channel Turned Off')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(image_no_blue_pil)
plt.title('Blue Channel Turned Off')
plt.axis('off')

plt.show()
```

2 Image Basics with OpenCV

What is OpenCV?

- OpenCV (Open Source Computer Vision Library) is a powerful and widely-used open-source library for computer vision and image processing tasks. It was originally developed by Intel and is now supported by the OpenCV community.
- OpenCV provides tools and functions for various image and video processing operations, making it a popular choice for building real-time computer vision applications.
- When you load an image using OpenCV, the image is loaded in BGR format by default. To correctly display the image using libraries like Matplotlib (which expects images in RGB format), you need to convert the image from BGR to RGB.

```
import cv2
import matplotlib.pyplot as plt

# Load the image using OpenCV
image_bgr = cv2.imread('path_to_image.jpg')

# Convert the image from BGR to RGB
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

# Display the image using Matplotlib
plt.imshow(image_rgb)
plt.title('RGB Image')
plt.axis('off')  # Hide the axis
plt.show()
```

2 Image Basics with OpenCV

Resize Image

- You can resize an image using the cv2.resize() function in OpenCV. The cv2.resize() function allows you to specify the desired size of the output image. Example to resize (1300, 1950, 3) to (1000, 400):

```
import cv2
import matplotlib.pyplot as plt

# Load the image
image = cv2.imread('path_to_image.jpg')

# Check the original size
print("Original size:", image.shape)  # (1300, 1950, 3)

# Desired size (width, height)
new_size = (1000, 400)

# Resize the image
resized_image = cv2.resize(image, new_size)

# Display the resized image
plt.imshow(resized_image)
cv2.waitKey(0)
cv2.destroyAllWindows()

# Save the resized image
cv2.imwrite('resized_image.jpg', resized_image)
```

- Original Size: The original image has dimensions (1300, 1950, 3), which means it's 1300 pixels tall, 1950 pixels wide, with 3 color channels (RGB).
- Desired Size: You specify the new size as (1000, 400) (width, height). Note that resizing may distort the image if the aspect ratio is not maintained.

2 Image Basics with OpenCV

- You also can resize based on ratio

```
import cv2
import matplotlib.pyplot as plt

# Load the image
img = cv2.imread('path_to_image.jpg')

# Define the width and height ratios
width_ratio = 0.8
height_ratio = 0.2

##########################################################
# Resize the image using the scaling factors
new_img = cv2.resize(img, (0, 0), width_ratio, height_ratio)
##########################################################

# Display the resized image
plt.imshow(new_img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

- Or you can do this way by replace the code between the line ##########

```
# Original dimensions
original_height, original_width = image.shape[:2]

# Calculate the new dimensions
new_width = int(original_width * width_ratio)
new_height = int(original_height * height_ratio)

# Resize the image based on the ratios
resized_image = cv2.resize(image, (new_width, new_height))
```

Image Basics with OpenCV

Resize Canvas

- To define the canvas size for an image using Matplotlib's plt.figure(), you can set the figure size, which will control the size of the canvas on which your image is displayed. This is particularly useful when you're displaying images or plotting multiple images in a grid.

```
import matplotlib.pyplot as plt
import cv2

# Load the image using OpenCV
image = cv2.imread('path_to_image.jpg')

# Convert the image from BGR to RGB format
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Define the canvas size (e.g., 10 inches wide, 5 inches
tall)
fig = plt.figure(figsize=(10, 5))

# Add a subplot to the figure
ax = fig.add_subplot(111)  # 1 row, 1 column, 1st subplot

# Display the image on the subplot
ax.imshow(image_rgb)
ax.axis('off')  # Turn off axis labels and ticks
plt.show()
```

Image Basics with OpenCV

Customizing with Multiple Subplots

- If you want to add multiple images or plots in the same figure, you can add more subplots using add_subplot. For example:

```
# Define the canvas size
fig = plt.figure(figsize=(15, 10))

# Add first subplot
ax1 = fig.add_subplot(121)  # 1 row, 2 columns, 1st subplot
ax1.imshow(image_rgb)
ax1.axis('off')

# Add second subplot
ax2 = fig.add_subplot(122)  # 1 row, 2 columns, 2nd subplot
ax2.imshow(image_rgb)
ax2.axis('off')

plt.show()
```

2 Image Basics with OpenCV

Drawing Image

- You can create a blank black image using NumPy's np.zeros() function. This function creates an array filled with zeros, and when used with the appropriate shape and data type, it represents a black image.

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

# Define the dimensions of the image (height, width,
channels)
height = 500  # e.g., 500 pixels
width = 800   # e.g., 800 pixels
channels = 3  # 3 channels for RGB

# Create a black image using np.zeros
black_image = np.zeros((height, width, channels),
dtype=np.uint8)

# Display the black image using Matplotlib
plt.imshow(black_image)
plt.title('Black Image')
plt.axis('off')  # Hide the axis
plt.show()
```

- dtype=np.uint8 specifies that the array elements are 8-bit unsigned integers, which is standard for image data (ranging from 0 to 255). A value of 0 in all channels (R, G, B) represents black.

2 Image Basics with OpenCV

Drawing Image

- You can create a blank black image using NumPy's np.zeros() function. This function creates an array filled with zeros, and when used with the appropriate shape and data type, it represents a black image.

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

# Define the dimensions of the image (height, width,
channels)
height = 500  # e.g., 500 pixels
width = 800   # e.g., 800 pixels
channels = 3  # 3 channels for RGB

# Create a black image using np.zeros
black_image = np.zeros((height, width, channels),
dtype=np.uint8)

# Display the black image using Matplotlib
plt.imshow(black_image)
plt.title('Black Image')
plt.axis('off')  # Hide the axis
plt.show()
```

- dtype=np.uint8 specifies that the array elements are 8-bit unsigned integers, which is standard for image data (ranging from 0 to 255). A value of 0 in all channels (R, G, B) represents black.

2 Image Basics with OpenCV

- To draw shapes like rectangles, circles, and polygons on a black image created with np.zeros(), you can use OpenCV's drawing functions.

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

# Step 1: Create a black image
height, width = 500, 800
black_image = np.zeros((height, width, 3), dtype=np.uint8)

# Step 2: Draw a rectangle on the image
start_point = (100, 100)  # Top-left corner
end_point = (300, 300)    # Bottom-right corner
color = (255, 0, 0)       # Blue color in BGR
thickness = 5             # Line thickness
cv2.rectangle(black_image, start_point, end_point, color,
thickness)

# Step 3: Draw a circle on the image
center = (400, 250)  # Center of the circle
radius = 50          # Radius of the circle
color = (0, 255, 0)  # Green color in BGR
thickness = 3        # Line thickness
cv2.circle(black_image, center, radius, color, thickness)

# Step 4: Draw a polygon on the image
points = np.array([[500, 100], [600, 200], [700, 100], [650,
50]], np.int32)
points = points.reshape((-1, 1, 2))  # Reshape to the format
needed by polylines
cv2.polylines(black_image, [points], isClosed=True, color=(0,
0, 255), thickness=2)
```

2 Image Basics with OpenCV

```
# Step 5: Display the image
plt.imshow(cv2.cvtColor(black_image, cv2.COLOR_BGR2RGB))
plt.title('Shapes on Black Image')
plt.axis('off')  # Hide the axis
plt.show()
```

3 Image Processing

What is Image Processing?

- Image processing involves performing operations on images to enhance them, extract useful information, or prepare them for further analysis. It is used in a variety of applications such as object detection, face recognition, medical imaging, and robotics.

Common Image Processing Operations Using OpenCV

1. Reading and Writing Images
   - Reading an Image: You can read an image from a file using cv2.imread()
   - Writing an Image: You can save an image to a file using cv2.imwrite():

2. Displaying Images
   - Displaying an Image: Use cv2.imshow() to display an image in a window

3. Resizing Images
   - Resizing: You can resize an image using cv2.resize():

4. Image Transformation
   - Rotation: Rotate an image using cv2.getRotationMatrix2D() and cv2.warpAffine()
   - Translation: Translate (shift) an image using a translation matrix:

5. Color Space Conversion
   - Converting Color Spaces: Convert an image from one color space to another (e.g., BGR to Grayscale):

6. Blurring and Smoothing
   - Gaussian Blur: Apply a Gaussian blur to smooth the image
   - Median Blur: Use median filtering to reduce noise:

3 Image Processing

7.Edge Detection
- Canny Edge Detection: Detect edges using the Canny algorithm

8. Morphological Operations
- Erosion: Erode the image to remove noise
- Dilation: Dilate the image to enhance features
- Opening and Closing: Use combinations of erosion and dilation for specific effects

9. Image Thresholding
- Binary Thresholding: Convert an image to a binary image based on a threshold value
- Adaptive Thresholding: Apply adaptive thresholding for uneven lighting

10. Contour Detection
- Finding Contours: Detect contours in an image
- Drawing Contours: Draw contours on the image

11. Face Detection
- Haar Cascade Classifiers: Use pre-trained Haar cascades for face detection

12. Image Filtering
- Custom Filters: Apply custom filters using convolution operations:

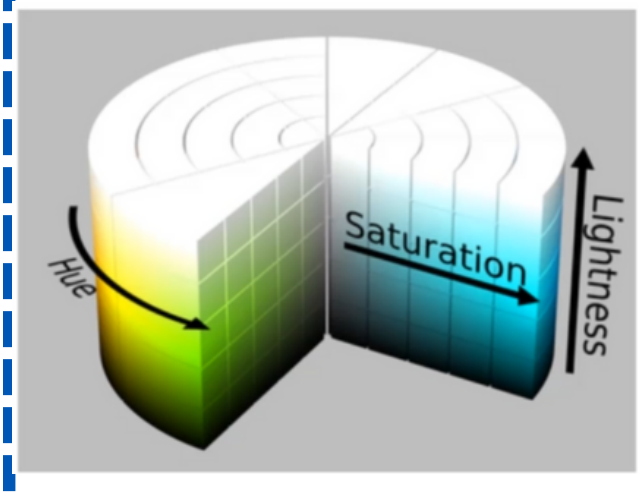
13. Object Detection
- Template Matching: Detect objects in images by comparing a template image with the target image:

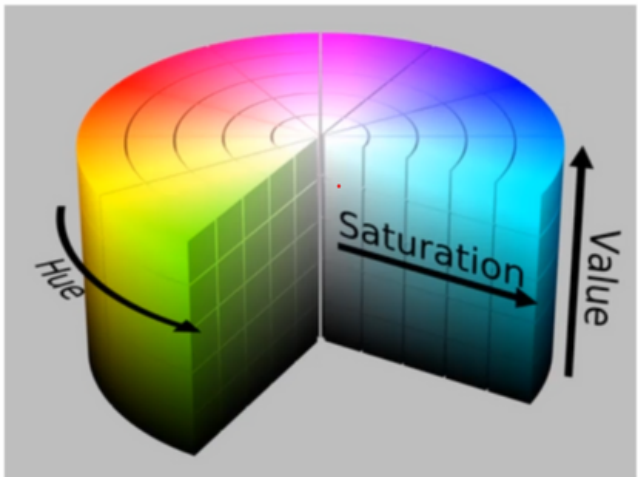
14. Image Segmentation
- Watershed Algorithm: Segment images using markers and the watershed algorithm

3 Image Processing

Colors Space Conversion
HSL and HSV Color Models: An Overview

HSL (Hue, Saturation, Lightness) and HSV (Hue, Saturation, Value) are two related color models that represent colors in a way that is more intuitive for human perception compared to the traditional RGB (Red, Green, Blue) model. Both models are widely used in various fields such as computer graphics, image processing, and color selection tools in design software.





1. Hue (H)

- Hue is a component common to both HSL and HSV and represents the type of color. It's an angular measurement, typically ranging from 0° to 360°, where each angle corresponds to a specific color on the color wheel:
  - 0°: Red
  - 60°: Yellow
  - 120°: Green
  - 180°: Cyan
  - 240°: Blue
  - 300°: Magenta

Image Processing

2. Saturation (S)
   - Saturation represents the intensity or purity of the color.
   - In both HSL and HSV, saturation is measured as a percentage, from 0% to 100%:
     - 0% represents a shade of gray (no color intensity).
     - 100% represents the full intensity of the color.

3. HSL (Hue, Saturation, Lightness)
   - Lightness (L) in HSL represents the brightness of the color.
   - Lightness ranges from 0% (black) to 100% (white):
     - 0% Lightness is completely dark (black).
     - 50% Lightness is the pure color (fully saturated at its original intensity).
     - 100% Lightness is completely light (white).
   - HSL is particularly useful for adjusting the lightness and saturation of a color while keeping the hue constant.
   - HSL Characteristics:
     - HSL is often preferred in design applications because it provides a straightforward way to adjust the lightness of a color independently of its hue and saturation.
     - A common use case is generating tints (lighter shades) and shades (darker tones) of a base color.
   - Example in HSL:
     - Pure red color in HSL: H: 0°, S: 100%, L: 50%
     - Changing lightness to 25% makes the red darker, while changing lightness to 75% makes it lighter.

Image Processing

4. HSV (Hue, Saturation, Value)
- Value (V) in HSV represents the brightness or intensity of the color.
- Value ranges from 0% (black) to 100% (brightest):
  - 0% Value is completely dark (black).
  - 100% Value is the fully bright color.
- HSV is often used in contexts where it's important to understand the color's brightness or how much light is being reflected.
- HSV Characteristics:
  - HSV is widely used in image processing and computer vision because it better aligns with how humans perceive color brightness.
  - In many design tools, HSV is used for selecting colors where you want to keep the saturation and hue constant while adjusting the brightness (value).
- Example in HSV:
  - Pure red color in HSV: H: 0°, S: 100%, V: 100%
  - Red with lower brightness: H: 0°, S: 100%, V: 50%

Comparison Between HSL and HSV
- Lightness vs. Value: In HSL, Lightness adjusts the color by mixing with both white and black, making it lighter or darker. In HSV, Value adjusts the color by mixing only with black, making it brighter or darker.
- Use Cases:
  - HSL: Better for tasks that involve adjusting color tints and shades, such as in graphic design.
  - HSV: Better for tasks that require adjusting brightness or analyzing color, such as in image processing and computer vision.

Image Processing

Image Blending

- Image blending is a technique used to combine two images into one by merging their pixel values in a weighted manner. OpenCV provides a convenient function called cv2.addWeighted that allows you to blend two images together with specific weights, resulting in a smooth transition between the two images.
- Function: cv2.addWeighted
- The cv2.addWeighted function blends two images by applying the following formula to each pixel:

$$\text{Output} = \alpha \times \text{Image1} + \beta \times \text{Image2} + \gamma$$

  - alpha: Weight of the first image.
  - beta: Weight of the second image.
  - gamma: Scalar added to each sum (often set to 0).

```
blended_image = cv2.addWeighted(src1, alpha, src2, beta, gamma)
```

  - src1: The first input image (Image1).
  - alpha: Weight of the first image (0.0 to 1.0).
  - src2: The second input image (Image2).
  - beta: Weight of the second image (0.0 to 1.0).
  - gamma: Scalar added to the weighted sum (usually 0).

```
import cv2
import matplotlib.pyplot as plt

# Load two images of the same size
image1 = cv2.imread('path_to_image1.jpg')
image2 = cv2.imread('path_to_image2.jpg')

# Ensure both images are the same size
image1 = cv2.resize(image1, (600, 400))
image2 = cv2.resize(image2, (600, 400))
```

Image Processing

```
# Set the weights for blending
alpha = 0.7  # Weight for the first image
beta = 0.3   # Weight for the second image
gamma = 0    # Scalar added to the sum (often 0)

# Blend the images
blended_image = cv2.addWeighted(image1, alpha, image2, beta,
gamma)

# Convert the BGR image to RGB for correct color representation
in Matplotlib
blended_image_rgb = cv2.cvtColor(blended_image,
cv2.COLOR_BGR2RGB)

# Display the blended image
plt.imshow(blended_image_rgb)
plt.title('Blended Image')
plt.axis('off')  # Hide axis labels
plt.show()
```

- alpha = 0.7 and beta = 0.3: The first image contributes 70% to the final blended image, and the second image contributes 30%. This creates an image where the first image is more dominant.
- Adjusting Weights: By changing alpha and beta, you can control the dominance of each image. For example, setting alpha = 0.5 and beta = 0.5 would mix the two images equally.
- gamma: This is a constant that is added to every pixel after the weighted sum. It's often set to 0 but can be used to brighten or darken the final image.

Image Processing

```
# Set the weights for blending
alpha = 0.7  # Weight for the first image
beta = 0.3   # Weight for the second image
gamma = 0    # Scalar added to the sum (often 0)

# Blend the images
blended_image = cv2.addWeighted(image1, alpha, image2, beta,
gamma)

# Convert the BGR image to RGB for correct color representation
in Matplotlib
blended_image_rgb = cv2.cvtColor(blended_image,
cv2.COLOR_BGR2RGB)

# Display the blended image
plt.imshow(blended_image_rgb)
plt.title('Blended Image')
plt.axis('off')  # Hide axis labels
plt.show()
```

3 Image Processing

Image Blending (Different Size)

- Blending a smaller image into a larger image, where the images are of different sizes, requires a few additional steps compared to blending two images of the same size. Here's how you can blend a small image into a larger image using OpenCV's cv2.addWeighted function:
- Step-by-Step Guide
  - Load Both Images: Load the larger background image and the smaller image that you want to blend into the larger image.
  - Determine the Region of Interest (ROI): Identify the region in the larger image where the smaller image will be placed. This region should be the same size as the smaller image.
  - Resize the Small Image (Optional): If needed, resize the smaller image to match the size of the desired ROI in the larger image.
  - Blend the Images: Use the cv2.addWeighted function to blend the smaller image with the corresponding region in the larger image.
  - Replace the ROI in the Larger Image: Replace the identified region in the larger image with the blended result.

Step 1: Import Libraries and Load Images

```
import cv2
import numpy as np

# Load the larger image
large_image = cv2.imread('path_to_large_image.jpg')

# Load the smaller image
small_image = cv2.imread('path_to_small_image.jpg')

# Display the original large and small images
cv2.imshow('Large Image', large_image)
cv2.imshow('Small Image', small_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Image Processing

Step 2: Determine the Region of Interest (ROI)
Choose the top-left corner where you want to place the small image within the large image:

```
# Get the dimensions of the small image
small_height, small_width = small_image.shape[:2]

# Define the top-left corner of the region of interest (ROI) in
the large image
x_offset = 50  # X coordinate of the top-left corner in the
large image
y_offset = 100  # Y coordinate of the top-left corner in the
large image

# Define the region of interest (ROI) in the large image
roi = large_image[y_offset:y_offset + small_height,
x_offset:x_offset + small_width]
```

Step 3: Blend the Small Image into the ROI
Use the cv2.addWeighted function to blend the small image with the ROI:

```
# Blend the small image into the ROI
alpha = 0.7  # Weight for the small image
beta = 0.3   # Weight for the ROI in the large image
gamma = 0    # Scalar added to the sum (usually 0)

blended_roi = cv2.addWeighted(small_image, alpha, roi, beta,
gamma)
```

Image Processing

Step 4: Replace the ROI in the Large Image
Replace the original ROI in the large image with the blended result:

```
# Replace the original ROI with the blended ROI in the large
image
large_image[y_offset:y_offset + small_height, x_offset:x_offset
+ small_width] = blended_roi
```

Step 5: Display the Final Image
Finally, display the blended image:

```
# Display the final image with the blended small image
cv2.imshow('Blended Image', large_image)
cv2.waitKey(0)
cv2.destroyAllWindows()

# Save the final blended image if desired
cv2.imwrite('blended_image.jpg', large_image)
```

3 Image Processing

Masking a logo

Step-by-Step Guide to Mask a Logo onto an Image

1. Load the Images: Load both the main (background) image and the logo (foreground) image.
2. Create a Mask of the Logo: Convert the logo image to grayscale and create a binary mask where the logo area is white and the background is black.
3. Create an Inverse Mask: Invert the mask to use for blending.
4. Prepare the Region of Interest (ROI): Define the area on the main image where the logo will be placed.
5. Apply the Mask: Use the bitwise operations to mask the logo onto the ROI.
6. Blend the Logo onto the Image: Combine the ROI and the masked logo to place the logo onto the main image.

Step 1: Load the Logo Image and Main Image

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

# Load the logo image
logo = cv2.imread('logo.png')

# Load the main (background) image
main_image = cv2.imread('background.jpg')
```

Step 2: Get Dimensions and Prepare for Masking

```
# Get dimensions of the logo
logo_height, logo_width = logo.shape[:2]

# Get dimensions of the main image
main_height, main_width = main_image.shape[:2]

# Convert the logo image to grayscale to create a mask
logo_gray = cv2.cvtColor(logo, cv2.COLOR_BGR2GRAY)
```

Image Processing

Step 3: Create a Mask and Its Inverse
Now, we'll create a binary mask from the grayscale logo image. The mask will be used to extract the logo from its background.

```
# Create an inverse mask of the logo (background becomes white,
logo becomes black)
mask_inv = cv2.bitwise_not(logo_gray)
```

Step 4: Create a White Background and Isolate the Logo
Next, we'll create a white background and isolate the logo using the inverse mask.

```
# Create a white background with the same size as the logo
white_background = np.full((logo_height, logo_width, 3), 255,
dtype=np.uint8)

# Apply the inverse mask to the white background to keep only
the background
background_with_logo = cv2.bitwise_or(white_background,
white_background, mask=mask_inv)

# Apply the inverse mask to the logo to isolate the logo
fg_logo = cv2.bitwise_or(logo, logo, mask=mask_inv)
```

Step 5: Determine the Position for the Logo
We'll calculate where to place the logo in the bottom-right corner of the main image.

```
# Calculate the bottom-right corner position for the logo
x_offset = main_width - logo_width
y_offset = main_height - logo_height
```

3 Image Processing

Step 6: Extract the Region of Interest (ROI) from the Main Image
We'll extract the region in the main image where the logo will be placed.

```
# Extract the ROI from the main image where the logo will be
placed
roi = main_image[y_offset:y_offset + logo_height,
x_offset:x_offset + logo_width]
```

Step 7: Combine the Logo with the ROI
We'll combine the isolated logo with the region of the main image (ROI) where it will be placed.

```
# Combine the logo foreground with the ROI
final_roi = cv2.bitwise_or(fg_logo, roi)
```

Step 9: Display the Final Image
We can now display the final image with the logo placed in the bottom-right corner.

```
plt.imshow(final_roi)
```

Issue:

The issue you're encountering arises because of how the masking and bitwise operations are performed. Specifically, the cv2.bitwise_or operation used to blend the logo onto the background doesn't handle transparency well, particularly when the background is bright. This can make the logo appear faint or transparent when placed on light backgrounds.

3 Image Processing

Thresholding in Image Processing

- Thresholding is a simple yet effective technique in image processing used to segment or categorize pixels in an image based on their intensity values. The main idea behind thresholding is to transform a grayscale image into a binary image, where pixels are either black (0) or white (255), depending on whether their intensity is above or below a certain threshold value.

How Thresholding Works

1. Grayscale Image:
   - Before applying thresholding, the image is usually converted to a grayscale image, where each pixel value represents the intensity of light, ranging from 0 (black) to 255 (white).
2. Applying the Threshold:
   - A threshold value, T, is chosen, and each pixel in the grayscale image is compared to this value.
   - If a pixel's intensity is greater than or equal to T, it is set to 255 (white).
   - If a pixel's intensity is less than T, it is set to 0 (black).

The result is a binary image where all pixels are either black or white.

Image Processing

Types of Thresholding

1. Simple Thresholding:

- 1.cv2.THRESH_BINARY: Converts all pixel values above the threshold to the maximum value (e.g., 255) and those below the threshold to 0.
- cv2.THRESH_BINARY_INV: Inverse of THRESH_BINARY. Converts all pixel values above the threshold to 0 and those below the threshold to the maximum value.
- cv2.THRESH_TRUNC: Truncates pixel values above the threshold to the threshold value, leaving other pixel values unchanged.
- cv2.THRESH_TOZERO: Sets pixel values below the threshold to 0, leaving other pixel values unchanged.
- cv2.THRESH_TOZERO_INV: Inverse of THRESH_TOZERO. Sets pixel values above the threshold to 0, leaving other pixel values unchanged.

```
# Apply THRESH_BINARY thresholding _, thresh_binary =
cv2.threshold(img, 127, 255, cv2.THRESH_BINARY)

# Apply THRESH_BINARY_INV thresholding _, thresh_binary_inv =
cv2.threshold(img, 127, 255, cv2.THRESH_BINARY_INV)

# Apply THRESH_TRUNC thresholding _, thresh_trunc =
cv2.threshold(img, 127, 255, cv2.THRESH_TRUNC)

# Apply THRESH_TOZERO thresholding _, thresh_tozero =
cv2.threshold(img, 127, 255, cv2.THRESH_TOZERO)

# Apply THRESH_TOZERO_INV thresholding _, thresh_tozero_inv =
cv2.threshold(img, 127, 255, cv2.THRESH_TOZERO_INV)
```

Image Processing

2. Adaptive Thresholding:

- cv2.adaptiveThreshold: Applies local thresholding based on the mean or weighted sum of the neighborhood values, which is useful for images with varying lighting conditions. This is a more advanced method compared to simple global thresholding.

2.1 Adaptive Thresholding with Mean

```
# Apply adaptive thresholding using mean
adaptive_thresh_mean = cv2.adaptiveThreshold(img, 255,
cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 11, 2)
```

2.2 Adaptive Thresholding with Gaussian

```
# Apply adaptive thresholding using Gaussian
adaptive_thresh_gaussian = cv2.adaptiveThreshold(img, 255,
cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2)
```

3. Otsu's Binarization:

- cv2.THRESH_OTSU: Automatically calculates the optimal threshold value by minimizing the intra-class variance in a bimodal histogram. It can be combined with THRESH_BINARY or THRESH_BINARY_INV.

```
# Apply Otsu's thresholding
_, otsu_thresh = cv2.threshold(img, 0, 255, cv2.THRESH_BINARY +
cv2.THRESH_OTSU)
```

3 Image Processing

Gamma Correction

- Gamma correction is a non-linear operation used to encode and decode luminance or color values in images. The formula for gamma correction is:

$$\text{Output} = \left(\frac{\text{Input}}{255}\right)^{\gamma} \times 255$$

Where:

- Input is the pixel value of the original image.
- Gamma is the gamma value that you want to apply.
- Output is the new pixel value after gamma correction.
- If gamma <1, the image will appear brighter.
- If gamma >1, the image will appear darker.

Step 1: Import Libraries and Load the Image

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

# Load an image
img = cv2.imread('image.jpg')

# Convert the image to float32 (normalize)
img = img.astype(np.float32) / 255.0
```

Image Processing

Step 2: Apply Gamma Correction Using np.power

```
# Define the gamma value
gamma = 2.0  # Example: gamma > 1 will darken the image

# Apply gamma correction
gamma_corrected = np.power(img, gamma)
```

Step 3: Display the Result

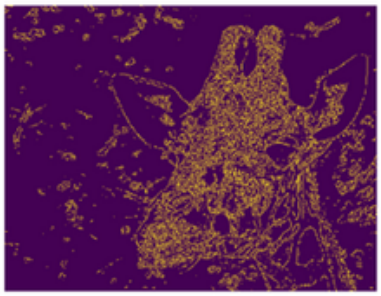
```
plt.imshow(gamma_corrected)
```

3 Image Processing

Smoothing and Blurring

- Smoothing and blurring are fundamental operations in image processing that are used to reduce noise, detail, and other high-frequency components in an image. These techniques are often used as a pre-processing step in various computer vision applications, such as edge detection, object recognition, and image segmentation.

Original Image



Edges without Blur

Edges with Blur

1. Averaging Blur (Box Filter)

- Averaging Blur is one of the simplest blurring techniques, where each pixel's value is replaced by the average value of its neighboring pixels. This method is also known as a box filter because it considers all pixels within a rectangular (box) region around the pixel.
- How It Works:
  - A kernel (typically a square matrix of size k x k) is moved across the image.
  - For each pixel, the kernel computes the average of all pixel values under it.
  - The center pixel is then replaced with this average value.
- Effect:
  - This method effectively reduces noise but can result in a loss of detail and sharpness in the image.
  - All pixels within the kernel's area contribute equally to the output pixel value.

3 Image Processing

```
import cv2
import matplotlib.pyplot as plt

# Load an image
img = cv2.imread('image.jpg')

# Apply Averaging Blur
blurred_img = cv2.blur(img, (5, 5))  # 5x5 kernel

# Display the result
plt.imshow(blurred_img)
```

2. Gaussian Blur

- Gaussian Blur is a more advanced blurring technique where the image is blurred using a Gaussian function. Unlike the averaging blur, the Gaussian blur applies different weights to the pixels under the kernel, giving more importance to the central pixels and less to those farther away.
- How It Works:
  - A Gaussian kernel is applied to the image. The kernel values are determined by the Gaussian function, which creates a smooth gradient from the center of the kernel outward.
  - The standard deviation (sigma) of the Gaussian distribution controls the amount of blur. A larger sigma results in more blur.
- Effect:
  - This method is effective in reducing Gaussian noise and is widely used in computer vision tasks.
  - It preserves edges better than averaging blur but still smooths out noise and detail.

Image Processing

```
# Apply Gaussian Blur
blurred_img = cv2.GaussianBlur(img, (5, 5), 0)  # 5x5 kernel,
sigma=0 (auto)

# Display the result
plt.imshow(blurred_img)
```

3. Median Blur

- Median Blur is a non-linear filtering technique where the median of all the pixels under the kernel window is calculated and the central pixel is replaced with this median value. This method is particularly effective at removing salt-and-pepper noise.
- How It Works:
  - The kernel slides over the image.
  - For each pixel, the median of the pixel values under the kernel is computed.
  - The central pixel is then replaced with this median value.
- Effect:
  - It is excellent for removing salt-and-pepper noise (isolated noise pixels) while preserving edges better than the averaging or Gaussian blur.
  - However, it may result in slight blurring of the image.

```
# Apply Median Blur
blurred_img = cv2.medianBlur(img, 5)  # 5x5 kernel (size must
be odd)

# Display the result
plt.imshow(blurred_img)
```

Image Processing

3. Bilateral Filter

- powerful image processing function in OpenCV used for edge-preserving and noise-reducing smoothing. Unlike other blurring techniques like Gaussian blur, which can blur edges along with noise, the bilateral filter can smooth the image while keeping the edges sharp.

```
dst = cv2.bilateralFilter(src, d, sigmaColor, sigmaSpace,
borderType=cv2.BORDER_DEFAULT)
```

- src: The source image on which the filter is applied.
- d: Diameter of each pixel neighborhood. It represents the size of the filter's window around each pixel. A larger value means more surrounding pixels will be considered.
- If d is negative, it is computed from sigmaSpace.
- sigmaColor: Filter sigma in the color space. A larger sigmaColor means that more colors in the neighborhood will be mixed together. It controls how much the filter considers the intensity difference between pixels.
- sigmaSpace: Filter sigma in the coordinate space. A larger sigmaSpace means that pixels farther from the center pixel (in the spatial domain) will influence the result more. It controls how much the filter considers the distance between pixels.
- borderType: Optional parameter that defines how the borders are handled. The default is cv2.BORDER_DEFAULT.

```
dst = cv2.bilateralFilter(img, 9, 75, 75)
```

Image Processing

Morphological operators

- Morphological operators are a set of operations that process images based on their shapes. They are primarily used in the analysis of binary images, though they can be adapted for grayscale images as well. The fundamental operations of morphological image processing are Erosion and Dilation, which can be combined to perform more complex operations like Opening, Closing, Gradient, Top Hat, and Black Hat.
- These operations are particularly useful in tasks such as noise removal, image segmentation, and feature extraction.

1. Erosion

Erosion is an operation that shrinks the boundaries of the foreground (usually white) regions in a binary image. It works by sliding a structuring element (kernel) across the image and eroding away the boundaries of objects.

- How It Works:
  - The structuring element is placed over the image, and if all the pixels under the structuring element (kernel) are 1, the output pixel is set to 1; otherwise, it is set to 0.
  - Essentially, it removes pixels on object boundaries, making objects smaller and removing small noise.
- Use Cases:
  - Removing small white noise.
  - Detaching two connected objects in an image.

```
# Define a 5x5 kernel
kernel = np.ones((5, 5), np.uint8)

# Apply Erosion
eroded_img = cv2.erode(img, kernel, iterations=1)
```

Image Processing

2. Dilation

Dilation is the opposite of erosion. It expands the boundaries of the foreground regions in a binary image. The operation is useful for closing small holes and connecting disjointed elements in an image.

- How It Works:
  - The structuring element is placed over the image, and if at least one pixel under the structuring element is 1, the output pixel is set to 1.
  - This operation adds pixels to the boundaries of objects, making objects larger.
- Use Cases:
  - Filling small holes in an image.
  - Connecting adjacent objects.

```
# Apply Dilation
dilated_img = cv2.dilate(img, kernel, iterations=1)
```

3. Opening

Opening is an operation that combines erosion followed by dilation. It is used to remove small objects from an image (like noise) while preserving the shape and size of larger objects.

- How It Works:
  - First, erosion is applied to remove small objects and noise.
  - Then, dilation is applied to restore the size of the remaining objects.
- Use Cases:
  - Removing small objects or noise from the foreground.
  - Smoothing object contours.

```
# Apply Opening
opened_img = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel)
```

Image Processing

4. Closing

Closing is the reverse of opening; it is dilation followed by erosion. This operation is used to close small holes or gaps within objects in an image.

- How It Works:
  - First, dilation is applied to fill small holes and gaps.
  - Then, erosion is applied to restore the original size of the objects.
- Use Cases:
  - Closing small holes in the foreground.
  - Connecting disjointed elements in an image.

```
# Apply Closing
closed_img = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel)
```

5. Morphological Gradient

Morphological Gradient is the difference between the dilation and erosion of an image. It highlights the edges of objects.

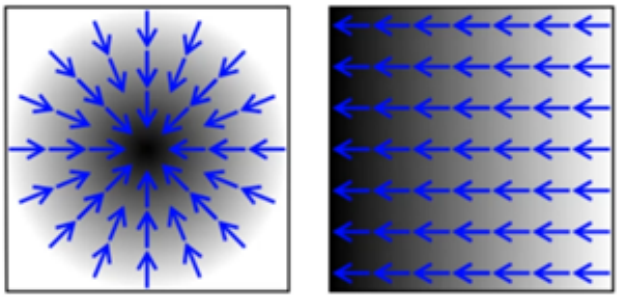
- How It Works:
  - The gradient is computed as the difference between the dilated image and the eroded image.
- Use Cases:
  - Extracting object outlines.
  - Highlighting boundaries in an image.

```
# Apply Morphological Gradient
gradient_img = cv2.morphologyEx(img, cv2.MORPH_GRADIENT,
kernel)
```

3 Image Processing

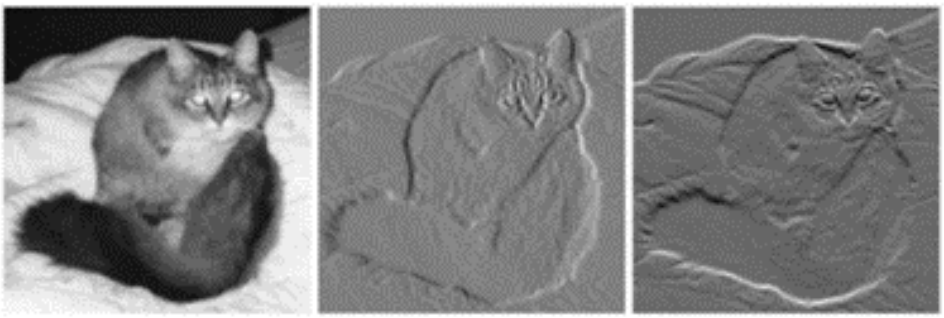
Gradient in Image Processing

- In image processing, the gradient refers to the directional change in the intensity or color in an image. The gradient is a vector that points in the direction of the greatest rate of increase of the function (image intensity), and its magnitude corresponds to how steep the increase is.
- In practice, gradients are used for edge detection. An edge is characterized by a significant change in intensity, so the gradient at the edges of an object in an image will be large.



Sobel Operator (Sobel-Feldman Operator)

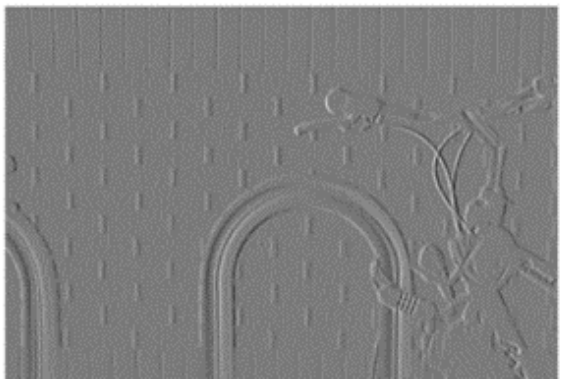
- The Sobel Operator, also known as the Sobel-Feldman Operator, is a widely used gradient-based edge detection method. It computes an approximation of the gradient of the image intensity function.
- Gradients can be calculated in a specific direction



3 Image Processing

Example:

- When normalized x-gradient from Sobel



- When normalized y-gradient from Sobel

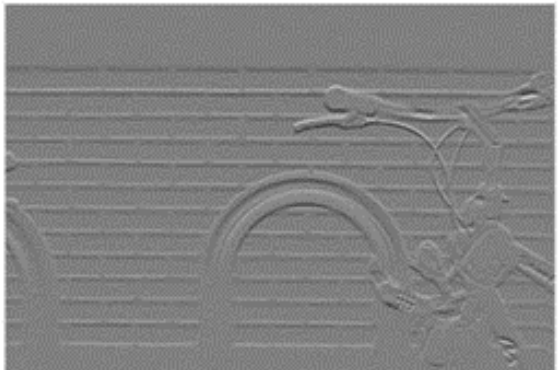


Image Processing

- Convolution with Kernels:
  - The Sobel operator uses two 3x3 convolution kernels, one for detecting changes in the horizontal direction (Gx) and one for the vertical direction (Gy).
- Kernels:
  - The horizontal kernel Gx is used to detect vertical edges, and the vertical kernel Gy is used to detect horizontal edges.

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

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

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

# Apply Sobel operator in the x direction (horizontal edges)
sobelx = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=3)  # Gx

# Apply Sobel operator in the y direction (vertical edges)
sobely = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=3)  # Gy
```

Image Processing

Laplacian Operator

- The Laplacian operator is a second-order derivative operator used in image processing to detect areas of rapid intensity change, often highlighting regions that correspond to edges. Unlike first-order derivative operators like Sobel, which detect edges in specific directions (horizontal, vertical), the Laplacian operator is isotropic, meaning it detects edges regardless of their orientation.

$$\nabla^2 f(x,y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

In digital image processing, the Laplacian operator is approximated using a convolution kernel. A common kernel used for the Laplacian operator is:

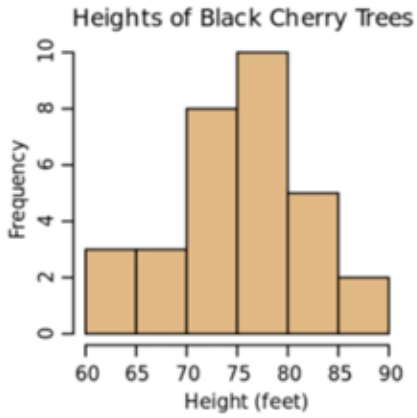
$$L = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \qquad L = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

```
# Apply the Laplacian operator
 laplacian = cv2.Laplacian(img, cv2.CV_64F)
```

3 Image Processing

Histogram in Image Processing

- A histogram in image processing is a graphical representation of the distribution of pixel intensity values in an image. It plots the number of pixels (frequency) for each intensity value. Histograms are a fundamental tool in image analysis and enhancement, providing insights into the brightness, contrast, and overall distribution of pixel values within an image.



- Key Concepts
  a. Pixel Intensity:
     - In a grayscale image, pixel intensity values range from 0 (black) to 255 (white). The histogram counts how many pixels have each intensity value.
     - In a color image, histograms can be created for each color channel (Red, Green, and Blue) separately.

  b. Histogram Plot:
     - The x-axis represents the possible intensity values (0 to 255 for an 8-bit image).
     - The y-axis represents the frequency or count of pixels at each intensity level.

3 Image Processing

Grayscale Histogram

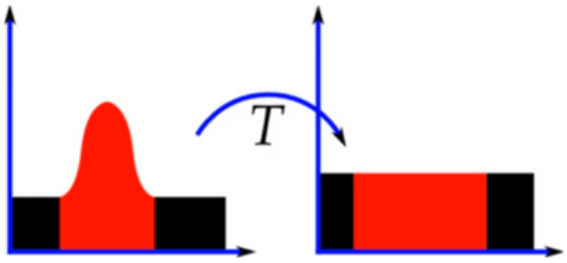
```
# Calculate the histogram
hist = cv2.calcHist([img], [0], None, [256], [0, 256])
```

Color Histogram

```
# Split the image into its color channels
colors = ('b', 'g', 'r')

# Calculate and plot the histogram for each color channel
for i, color in enumerate(colors):
    hist = cv2.calcHist([img_color], [i], None, [256], [0,
    256])
    plt.plot(hist, color=color)
    plt.xlim([0, 256])

plt.title('Image Histogram')
```

Histogram Equalization

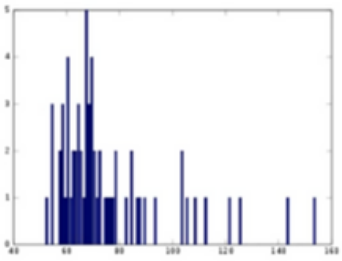
- Histogram Equalization is a technique used to enhance the contrast of an image by spreading out the most frequent intensity values. It transforms the intensity distribution of an image to be more uniform, which can improve the visual quality, particularly in images with low contrast.



3 Image Processing

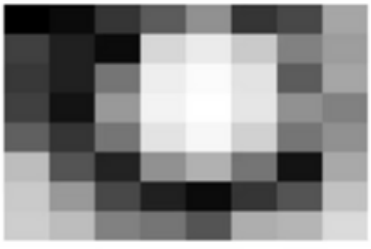
- How It Works
  - Input Histogram: The histogram of the original image may have peaks and valleys, with pixel values concentrated in certain intensity ranges, leading to poor contrast.
  - Equalization Process:
    - The histogram equalization process redistributes the pixel intensities so that the histogram of the output image is approximately flat (uniform distribution).
    - This increases the contrast in the image, making features more visible.

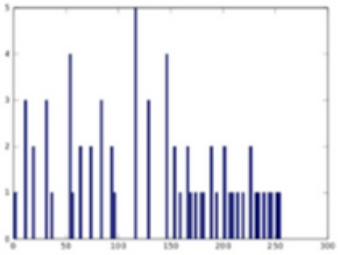
52	55	61	59	70	61	76	61
62	59	55	104	94	85	59	71
63	65	66	113	144	104	63	72
64	70	70	126	154	109	71	69
67	73	68	106	122	88	68	68
68	79	60	79	77	66	58	75
69	85	64	58	55	61	65	83
70	87	69	68	65	73	78	90



Original Image

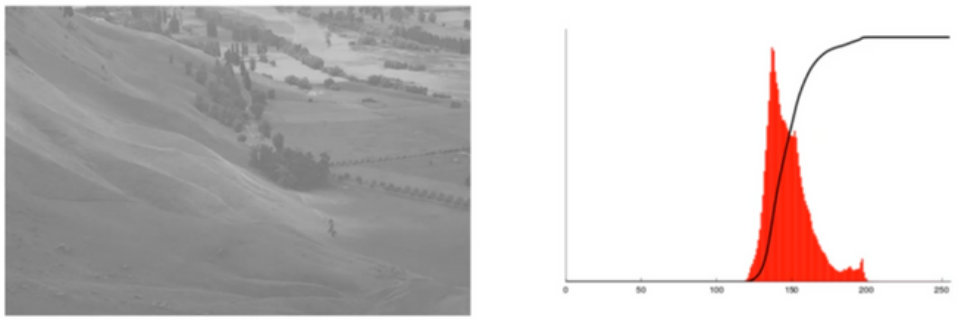
0	12	53	32	146	53	174	53
57	32	12	227	219	202	32	154
65	85	93	239	251	227	65	158
73	146	146	247	255	235	154	130
97	166	117	231	243	210	117	117
117	190	36	190	178	93	20	170
130	202	73	20	12	53	85	194
146	206	130	117	85	166	182	215



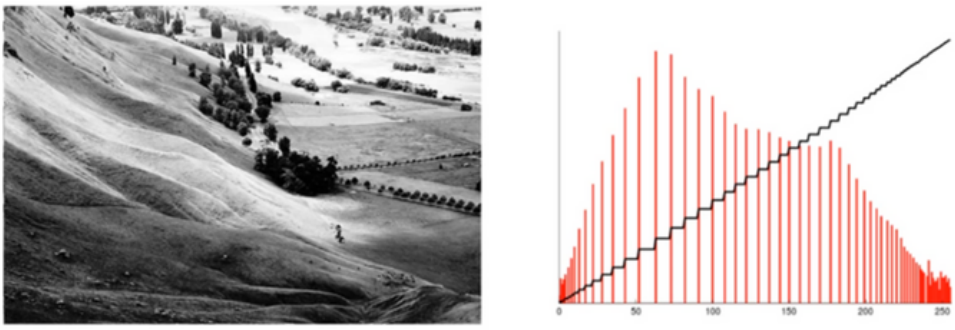


After applying histogram equalization

3 Image Processing



Original Image



After histogram equalization

```
import cv2
import matplotlib.pyplot as plt

# Load the gorilla image in color
img = cv2.imread('gorilla.jpg')

# Convert the image from BGR to HSV color space
hsv_img = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

# Apply histogram equalization to the V channel
hsv_img[:, :, 2] = cv2.equalizeHist(hsv_img[:, :, 2])

# Convert the image to RGB color space for viewing
img_equalized_hsv = cv2.cvtColor(hsv_img, cv2.COLOR_HSV2RGB)
```

3 Image Processing

Brightening a dark image, such as a gorilla image, can be effectively achieved using histogram equalization. You can perform histogram equalization directly on the grayscale version of the image using cv2.equalizeHist() or apply equalization in the HSV color space to better control the brightness while preserving the color balance.

Histogram with Mask

- A histogram with a mask allows you to calculate the histogram of a specific region of interest (ROI) within an image rather than the entire image. This is useful when you are only interested in analyzing or processing a particular part of the image.
- How It Works
  - Mask: A mask is a binary image (same size as the original image) where the region of interest is marked with white pixels (255), and the area outside the region of interest is marked with black pixels (0). The histogram is then computed only for the pixels corresponding to the white areas in the mask.
  - Application: Useful in situations where the analysis or enhancement should be restricted to a particular part of the image, such as focusing on a specific object in a scene.

Image Processing

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

# Load a image
img = cv2.imread('image.jpg')
img= cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# Create a mask with the same dimensions as the image
mask = np.zeros(img.shape[:2], dtype="uint8")

mask [300:400,100:400] = 255
show_masked_img = cv2.bitwise_and(img,img, mask=mask)

# Display the original image and the mask
plt.figure(show_masked_img)

# Calculate the histogram only for the masked region channel R
masked_hist = cv2.calcHist([img], [0], mask, [256], [0, 256])

plt.plot(masked_hist)
```

4 Video Basics with Python and OpenCV

Connecting Webcam

- Connecting a webcam to OpenCV is a common task in computer vision applications. OpenCV provides an easy way to access your webcam and capture video frames for processing.

```
import cv2

# Initialize the webcam
cap = cv2.VideoCapture(0)

# Get the width and height of the video frames
frame_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
frame_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))

while True:
    # Capture frame-by-frame
    ret, frame = cap.read()

    # If frame is read correctly, ret is True
    if not ret:
        print("Failed to grab frame")
        break

    # Convert the frame to grayscale
    gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # Display the resulting grayscale frame
    cv2.imshow('Webcam (Grayscale)', gray_frame)

    # Press 'q' on the keyboard to exit the loop
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
```

Video Basics with Python and OpenCV

```
# Release the webcam and close windows
cap.release()
cv2.destroyAllWindows()
```

Saving Video

- To save the video, we need to use the correct codec. If you are using Windows, use DIVX codec and if you are using MacOS/Linux, use XVID codec

```
import cv2

# Initialize the webcam
cap = cv2.VideoCapture(0)

# Check if the webcam is opened correctly
if not cap.isOpened():
    print("Error: Could not open webcam")
    exit()

# Get the width and height of the video frames
frame_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
frame_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))


# Define the codec and create VideoWriter object
output_file = 'output.mp4'
fourcc = cv2.VideoWriter_fourcc(*'mp4v')  # Codec for MP4
format
fps = 20.0  # Frame rate
out = cv2.VideoWriter(output_file, fourcc, fps, (frame_width,
frame_height))
```

4 Video Basics with Python and OpenCV

```
while True:
    # Capture frame-by-frame
    ret, frame = cap.read()

    # If frame is read correctly, ret is True
    if not ret:
        print("Failed to grab frame")
        break

    # Write the frame to the video file
    out.write(frame)

    # Display the resulting frame
    cv2.imshow('Webcam', frame)

    # Press 'q' on the keyboard to exit the loop
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
```

Opening Video

- To open the video, write as follow

```
import cv2

# Open the video file
cap = cv2.VideoCapture('your_video.mp4')

# Check if the video was opened successfully
if not cap.isOpened():
    print("Error: Could not open video file")
    exit()
```

4 Video Basics with Python and OpenCV

Drawing during live webcam

- Drawing a rectangle on a live webcam feed using a mouse click in OpenCV involves several steps. You need to capture the webcam feed, detect mouse events, and update the display with the drawn rectangle.

Step 1: Import Libraries

```
import cv2
import numpy as np
```

Step 2: Initialize Global Variables

You'll need some global variables to store the starting and ending points of the rectangle, as well as a flag to indicate whether the drawing is in progress.

```
# Global variables
drawing = False  # True if the mouse is pressed
ix, iy = -1, -1  # Initial mouse coordinates
```

Step 3: Define the Mouse Callback Function

This function will handle the mouse events. It will track when the mouse button is pressed and released and update the coordinates for drawing the rectangle.

```
def draw_rectangle(event, x, y, flags, param):
    global ix, iy, drawing, img

# When the left mouse button is pressed, record the starting
coordinates and set drawing to True
    if event == cv2.EVENT_LBUTTONDOWN:
        drawing = True
        ix, iy = x, y
```

4 Video Basics with Python and OpenCV

```
# When the mouse is moved and the button is pressed, draw the
rectangle
    elif event == cv2.EVENT_MOUSEMOVE:
        if drawing == True:
            img_copy = img.copy()  # Copy the image to avoid
drawing over the previous rectangles
            cv2.rectangle(img_copy, (ix, iy), (x, y), (0, 255,
0), 2)
            cv2.imshow('Webcam', img_copy)

    # When the left mouse button is released, finalize the
rectangle
    elif event == cv2.EVENT_LBUTTONUP:
        drawing = False
        cv2.rectangle(img, (ix, iy), (x, y), (0, 255, 0), 2)
```

Step 4: Capture Webcam Feed and Set Up Mouse Callback
Now, you can capture the webcam feed and set up the mouse callback function to handle the drawing.

```
# Initialize the webcam
cap = cv2.VideoCapture(0)

# Set up the mouse callback function
cv2.namedWindow('Webcam')
cv2.setMouseCallback('Webcam', draw_rectangle)

while True:
    # Capture frame-by-frame
    ret, img = cap.read()
```

Video Basics with Python and OpenCV

```
    # If the frame was not read correctly, exit the loop
    if not ret:
        print("Failed to grab frame")
        break

    # Display the resulting frame
    cv2.imshow('Webcam', img)

    # Press 'q' on the keyboard to exit the loop
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Release the webcam and close windows
cap.release()
cv2.destroyAllWindows()
```

Step 4: Capture Webcam Feed and Set Up Mouse Callback
Now, you can capture the webcam feed and set up the mouse callback function to handle the drawing.

Object Detection with OpenCV

```
    # If the frame was not read correctly, exit the loop
    if not ret:
        print("Failed to grab frame")
        break

    # Display the resulting frame
    cv2.imshow('Webcam', img)

    # Press 'q' on the keyboard to exit the loop
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Release the webcam and close windows
cap.release()
cv2.destroyAllWindows()
```

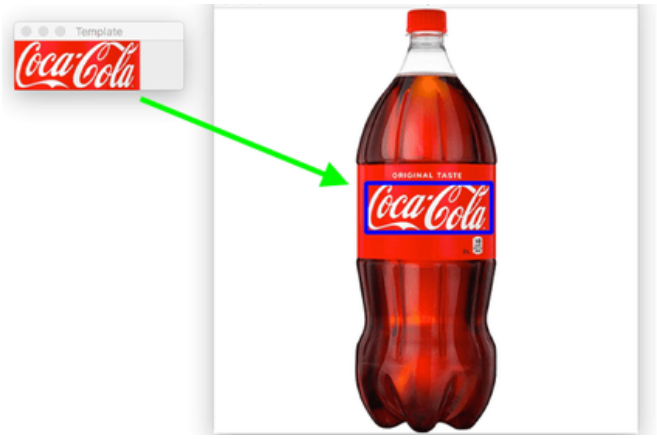
Fundamental of object detection

- Object detection is a crucial task in computer vision that involves identifying and locating objects within an image or video. OpenCV, a popular open-source computer vision library, provides several methods for object detection. Before we learn about more complex methods, we need to know about the fundamentals such as template matching, corner detection, edge detection, grid detection, contour detection, watershed algorithm.
- These are fundamental techniques in image processing and computer vision that serve various purposes, from detecting specific patterns in images to segmenting regions of interest.

5 Object Detection with OpenCV

Template matching

- Template Matching is a technique used to find a portion of an image that matches a template image. It's commonly used for tasks like object detection, where the goal is to locate the template within the larger image.
- How It Works:
  - Sliding Window: The template is slid over the input image (as a sub-image), and a comparison method (like correlation) is used to determine how well the template matches the image at each position.
  - Matching Metrics: Common metrics include the squared difference, normalized cross-correlation, and correlation coefficient.



- Common methods include:
  - cv2.TM_CCOEFF: Correlation coefficient, measures the similarity between the template and the image region.
  - cv2.TM_CCOEFF_NORMED: Normalized version of correlation coefficient.
  - cv2.TM_CCORR: Cross-correlation, another way of measuring similarity.
  - cv2.TM_CCORR_NORMED: Normalized cross-correlation.
  - cv2.TM_SQDIFF: Sum of squared differences, measures the difference between the template and the image region.
  - cv2.TM_SQDIFF_NORMED: Normalized sum of squared differences.

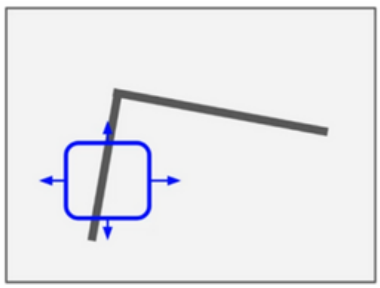
5 Object Detection with OpenCV

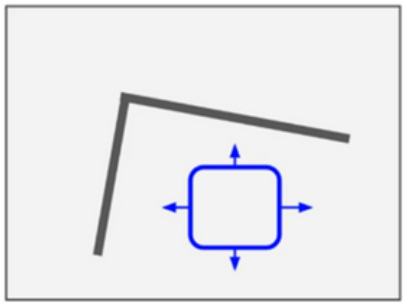
```
import cv2
import numpy as np

# Load the main image and the template
img = cv2.imread('main_image.jpg', 0)
template = cv2.imread('template_image.jpg', 0)
w, h = template.shape[::-1]

# Perform template matching
res = cv2.matchTemplate(img, template, cv2.TM_CCOEFF_NORMED)
min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(res)

# Draw a rectangle around the matched region
top_left = max_loc
bottom_right = (top_left[0] + w, top_left[1] + h)
cv2.rectangle(img, top_left, bottom_right, 255, 2)

# Display the result
cv2.imshow('Detected Template', img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

5 Object Detection with OpenCV

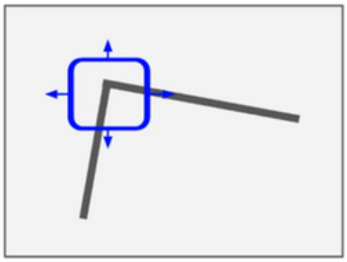
2. Corner Detection

- Corner detection is a technique used to identify points in an image where the intensity changes significantly in multiple directions. Corners are key features used in various computer vision tasks, including object recognition, image matching, and tracking.
- Harris Corner Detector
  - The Harris Corner Detector is a popular algorithm used to detect corners in an image. Corners are points in an image where the intensity changes significantly in all directions. The Harris detector identifies these points by analyzing the image gradients.
  - However, it can sometimes detect too many corners, including those that are not very distinct.
  - Harris Corner Detection algorithm is to compute the gradient of the image in both the x and y directions. The gradient represents how the intensity of the image changes with respect to each pixel.





$$f(x,y) = \sum_{(x_k,y_k)\in W} (I(x_k,y_k) - I(x_k+\Delta x, y_k+\Delta y))^2$$



5 Object Detection with OpenCV

Step 1: Load the Image

Load the image on which you want to perform corner detection. Convert it to grayscale since the Harris Corner Detector operates on single-channel images.

```
import cv2
import numpy as np

# Load the image
img = cv2.imread('image.jpg')

# Convert the image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Convert the grayscale image to float32
gray = np.float32(gray)
```

Step 2: Apply Harris Corner Detection

Now, apply the Harris Corner Detector using the cv2.cornerHarris() function. This function requires the grayscale image, block size, Sobel kernel size, and the Harris detector free parameter kkk.

- blockSize: The size of the neighborhood considered for corner detection.
- ksize: Aperture parameter for the Sobel operator (usually 3).
- k: Harris detector free parameter, typically in the range [0.04, 0.06].

```
# Apply Harris corner detection
dst = cv2.cornerHarris(gray, blockSize=2, ksize=3, k=0.04)

# Result is dilated for marking the corners
dst = cv2.dilate(dst, None)
```

5 Object Detection with OpenCV

Step 3: Threshold and Mark the Corners

You can threshold the image to mark the corners. The corners will be marked in the original image.

```
# Threshold for an optimal value, marking the corners in red
img[dst > 0.01 * dst.max()] = [0, 0, 255]

# Display the result
cv2.imshow('Harris Corners', img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

2. Shi-Tomasi Corner Detector

- The Shi-Tomasi Corner Detection method, also known as the Good Features to Track algorithm, is an improvement over the Harris Corner Detection method. It simplifies the corner detection process by selecting corners based on the minimum eigenvalue of the structure tensor, making it more reliable and less sensitive to noise.

Step 1: Load and Prepare the Image

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

# Load the image
img = cv2.imread('image.jpg')

# Convert the image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Object Detection with OpenCV

Step 2: Apply Shi-Tomasi Corner Detection

Use OpenCV's cv2.goodFeaturesToTrack() function to detect the corners. This function is specifically designed for the Shi-Tomasi method.

- maxCorners: The maximum number of corners you want to detect.
- qualityLevel: A parameter that determines the minimum quality of the corners. A value of 0.01 means that only corners with a cornerness response greater than 1% of the best corner are considered.
- minDistance: The minimum Euclidean distance between the detected corners.

```
# Detect corners using the Shi-Tomasi method
corners = cv2.goodFeaturesToTrack(gray, maxCorners=100,
qualityLevel=0.01, minDistance=10)
corners = np.int0(corners)  # Convert to integer
```

Step 3: Draw the Corners on the Image

```
# Draw circles around detected corners
for i in corners:
    x, y = i.ravel()  # Flatten the array
    cv2.circle(img, (x, y), 3, (0, 255, 0), -1)

# Convert BGR to RGB for display with matplotlib
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# Display the image with corners
plt.imshow(img_rgb)
```

5 Object Detection with OpenCV

Step 2: Apply Shi-Tomasi Corner Detection

Use OpenCV's cv2.goodFeaturesToTrack() function to detect the corners. This function is specifically designed for the Shi-Tomasi method.

- maxCorners: The maximum number of corners you want to detect.
- qualityLevel: A parameter that determines the minimum quality of the corners. A value of 0.01 means that only corners with a cornerness response greater than 1% of the best corner are considered.
- minDistance: The minimum Euclidean distance between the detected corners.

```
# Detect corners using the Shi-Tomasi method
corners = cv2.goodFeaturesToTrack(gray, maxCorners=100,
qualityLevel=0.01, minDistance=10)
corners = np.int0(corners)  # Convert to integer
```

Step 3: Draw the Corners on the Image

```
# Draw circles around detected corners
for i in corners:
    x, y = i.ravel()  # Flatten the array
    cv2.circle(img, (x, y), 3, (0, 255, 0), -1)

# Convert BGR to RGB for display with matplotlib
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# Display the image with corners
plt.imshow(img_rgb)
```

Object Detection with OpenCV

Edge Detection

- Canny Edge Detection is one of the most popular and widely used edge detection algorithms in computer vision. Developed by John F. Canny in 1986, this algorithm is known for its ability to detect a wide range of edges in images, while reducing noise and minimizing false detections.
- Key Features of Canny Edge Detection:
  - Multi-stage Algorithm: The Canny edge detector is a multi-stage process, ensuring that the edges detected are accurate and continuous.
  - Gradient-based: It is a gradient-based edge detector, meaning it identifies edges by looking for areas of the image where the intensity of pixels changes sharply.

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

# Load the image
img = cv2.imread('image.jpg')

# Apply Canny edge detection
edges = cv2.Canny(img, threshold1=100, threshold2=200)

# Display the result using matplotlib
plt.imshow(edges)
```

These are the lower and upper thresholds for the hysteresis procedure. threshold1 is the lower threshold for detecting weak edges, and threshold2 is the higher threshold for detecting strong edges.

Object Detection with OpenCV

Contour Detection

- Contour detection is a fundamental technique in image processing and computer vision. Contours represent the boundaries of objects within an image, and they are particularly useful for shape analysis, object detection, and image segmentation.

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

# Load the image
img = cv2.imread('image.jpg')

# Convert the image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Apply thresholding to get a binary image
ret, thresh = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY)

# Find contours
contours, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

# Draw all contours on the original image
cv2.drawContours(img, contours, -1, (0, 255, 0), 3)

# Convert the image from BGR to RGB for displaying with
matplotlib
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# Display the image with contours
plt.imshow(img_rgb)
```

5 Object Detection with OpenCV

Contour retrieval mode

- There are several contour retrieval modes used by the cv2.findContours() function to retrieve contours from a binary image. These modes determine how the contours are stored and how they relate to one another, particularly in cases where contours are nested within each other (e.g., a contour within another contour).

Mode	OpenCV Constant	Description	Use Case
External Contours	cv2.RETR_EXTERNAL	Retrieves only the outermost contours. All child contours (nested within another contour) are ignored.	Use this when you are only interested in the outer boundaries of objects, ignoring any internal contours.
All Contours (List)	cv2.RETR_LIST	Retrieves all the contours without establishing any hierarchical relationships. Each contour is treated as an independent entity.	Use this when you want to detect all contours without considering any hierarchy, such as when analyzing individual shapes.
Two-Level Hierarchy	cv2.RETR_CCOMP	Retrieves all the contours and organizes them into a two-level hierarchy. The first level contains the outer contours, and the second level contains the contours of holes.	Use this when you need to separate the contours into outer boundaries and their corresponding inner boundaries (holes).
Full Hierarchy	cv2.RETR_TREE	Retrieves all the contours and reconstructs the entire hierarchy of nested contours, providing full parent-child relationships.	Use this when the hierarchical relationship between contours (e.g., nested objects) is important for your analysis.

5 Object Detection with OpenCV

Feature Matching

- Feature matching is a fundamental concept in computer vision that involves finding correspondences between key points or features in different images. This is useful in various applications, such as image stitching, object recognition, and 3D reconstruction. There are several techniques to match features between images, including Brute-Force Matching with ORB and SIFT, and FLANN-based Matching.

1. Brute-Force Matching with ORB

- ORB (Oriented FAST and Rotated BRIEF):
- ORB is a fast and efficient feature detector and descriptor extractor. It is an alternative to SIFT and SURF, designed to be computationally inexpensive while maintaining good performance.
- ORB uses the FAST algorithm to detect key points and BRIEF descriptors, with added rotation invariance.
- Brute-Force Matcher:
    - The Brute-Force Matcher compares each descriptor in the first set to every descriptor in the second set and finds the closest match based on a chosen distance metric (usually Euclidean distance or Hamming distance).

5 Object Detection with OpenCV

```
import cv2
import matplotlib.pyplot as plt

# Load the images
img1 = cv2.imread('image1.jpg', cv2.IMREAD_GRAYSCALE)
img2 = cv2.imread('image2.jpg', cv2.IMREAD_GRAYSCALE)

# Initialize the ORB detector
orb = cv2.ORB_create()

# Detect key points and descriptors
kp1, des1 = orb.detectAndCompute(img1, None)
kp2, des2 = orb.detectAndCompute(img2, None)

# Initialize the Brute-Force matcher
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

# Match descriptors
matches = bf.match(des1, des2)

# Sort matches based on distance (best matches first)
matches = sorted(matches, key=lambda x: x.distance)

# Draw the top 10 matches
img_matches = cv2.drawMatches(img1, kp1, img2, kp2,
matches[:10], None,
flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# Display the matches
plt.imshow(img_matches)
```

5 Object Detection with OpenCV

Flag Name	Integer Value	Description
cv2.DRAW_MATCHES_FLAGS_DEFAULT	0	Draws matched keypoints with default settings. Keypoints are drawn as small circles, and lines are drawn between matching keypoints.
cv2.DRAW_MATCHES_FLAGS_DRAW_OVER_OUTIMG	1	Draws matches on the provided output image (the image passed as an argument). The original content of the output image is overwritten by the matches.
cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS	2	Draws only the matched keypoints. Unmatched keypoints (those without a corresponding match in the other image) are not drawn.
cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS	4	Draws keypoints with additional information such as size and orientation. Keypoints are drawn with circles indicating their scale and direction.

5 Object Detection with OpenCV

2. Brute force with SIFT (Scale-Invariant Feature Transform)

- One of the most popular and robust feature detection and description algorithms in computer vision. It was developed by David Lowe in 1999 and is widely used for tasks like object recognition, image stitching, and 3D reconstruction.
- Key Concepts in SIFT:
  - Scale-Invariant: SIFT is designed to detect features that are invariant to scale. This means that it can detect the same features in an object even if the object appears larger or smaller in the image.
  - Rotation-Invariant: SIFT is also invariant to rotation. It can detect features regardless of the object's orientation in the image.
  - Illumination-Invariant: SIFT is robust to changes in lighting conditions, making it effective even when the image's brightness varies.

```
import cv2
import matplotlib.pyplot as plt

# Load the images
img1 = cv2.imread('image1.jpg', cv2.IMREAD_GRAYSCALE)
img2 = cv2.imread('image2.jpg', cv2.IMREAD_GRAYSCALE)

# Initialize the SIFT detector
sift = cv2.SIFT_create()

# Detect key points and descriptors
kp1, des1 = sift.detectAndCompute(img1, None)
kp2, des2 = sift.detectAndCompute(img2, None)

# Initialize the Brute-Force matcher
bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)

# Match descriptors
matches = bf.match(des1, des2)
```

Object Detection with OpenCV

```
# Sort matches based on distance (best matches first)
matches = sorted(matches, key=lambda x: x.distance)

# Draw the top 10 matches
img_matches = cv2.drawMatches(img1, kp1, img2, kp2,
matches[:10], None,
flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# Display the matches
plt.imshow(img_matches)
```

3. FLANN (Fast Library for Approximate Nearest Neighbors)

- FLANN is a library designed for fast nearest neighbor searches in high-dimensional spaces. It's particularly useful for feature matching in computer vision tasks, where you need to find correspondences between features (e.g., SIFT or SURF descriptors) in different images. FLANN is optimized to handle large datasets efficiently, making it much faster than brute-force matching, especially for large numbers of features.
- Key Concepts
  - Approximate Nearest Neighbors:
    - Unlike brute-force matching, which exhaustively compares each feature with every other feature, FLANN uses approximate nearest neighbor algorithms. These algorithms find a "good enough" match quickly, which is typically sufficient for most practical applications.
    - FLANN uses various algorithms such as KD-trees and K-means trees to speed up the search process.
  - High-Dimensional Data:
    - FLANN is designed to work well with high-dimensional data, such as the 128-dimensional SIFT descriptors or 64-dimensional ORB descriptors.
  - Speed vs. Accuracy:
    - FLANN trades off a small amount of accuracy for a significant gain in speed. This trade-off is adjustable based on the application's needs.

Object Detection with OpenCV

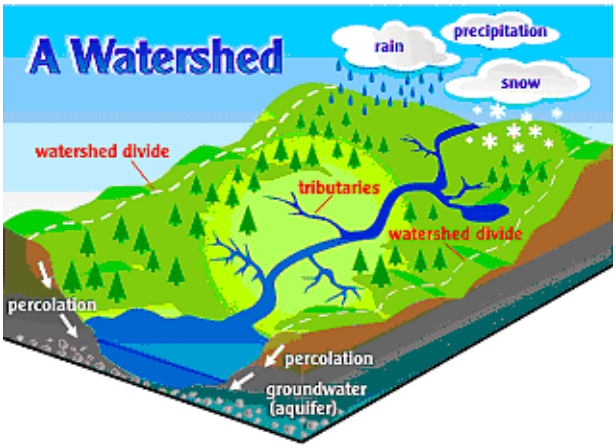
```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the images
img1 = cv2.imread('image1.jpg', cv2.IMREAD_GRAYSCALE)
img2 = cv2.imread('image2.jpg', cv2.IMREAD_GRAYSCALE)

# Initialize the SIFT detector
sift = cv2.SIFT_create()

# Detect key points and descriptors
kp1, des1 = sift.detectAndCompute(img1, None)
kp2, des2 = sift.detectAndCompute(img2, None)

# Define FLANN parameters
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)

# Initialize the FLANN matcher
flann = cv2.FlannBasedMatcher(index_params, search_params)

# Match descriptors using KNN
matches = flann.knnMatch(des1, des2, k=2)

# Apply the ratio test to filter good matches
good_matches = []
for m, n in matches:
    if m.distance < 0.7 * n.distance:
        good_matches.append(m)
```

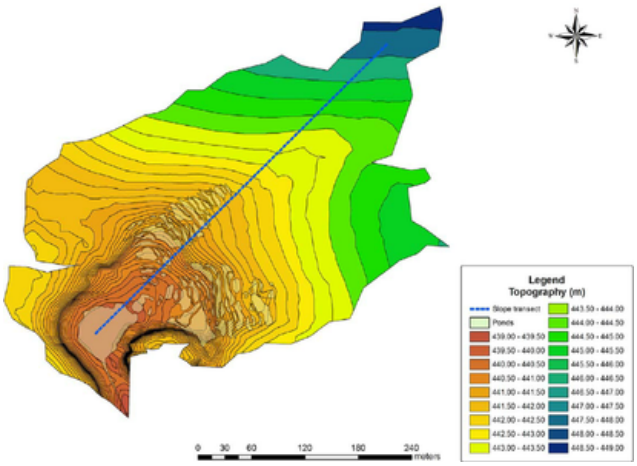
5 Object Detection with OpenCV

```
# Draw the good matches
img_matches = cv2.drawMatches(img1, kp1, img2, kp2,
good_matches, None,
flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# Display the matches
plt.imshow(img_matches)
```

Watershed Algorithm in Image Processing

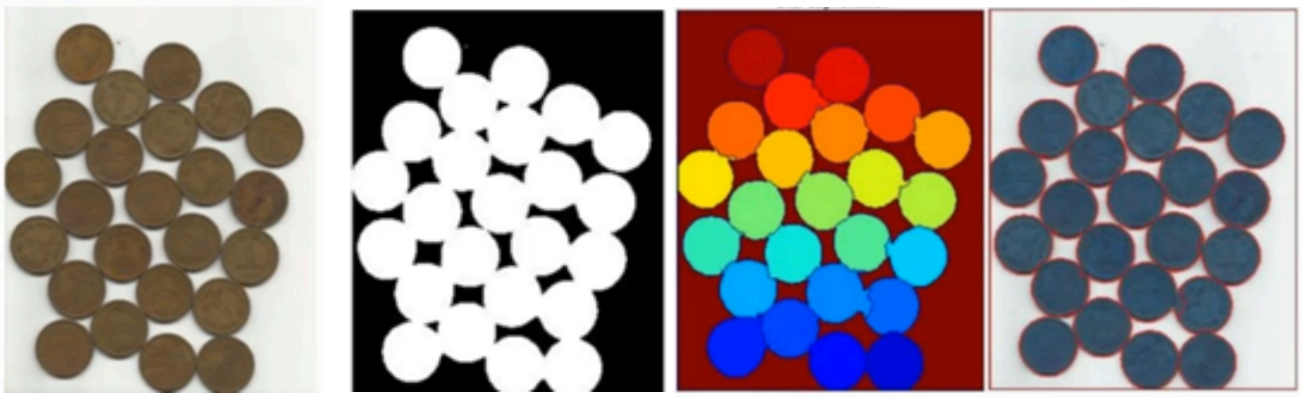
- The Watershed Algorithm is a powerful tool in image processing for segmenting images into different regions. The concept behind the watershed algorithm is derived from the idea of topographic maps, where the algorithm treats pixel intensity values as elevations in a terrain. The algorithm is often used for separating different objects in an image, particularly when the objects are touching or overlapping.





5 Object Detection with OpenCV

- Key Concepts
  - Topographic Interpretation:
    - The image is interpreted as a topographic surface, where higher pixel values (intensities) represent peaks (mountains), and lower values represent valleys.
    - The goal is to "flood" the valleys with water, and the lines where different water sources meet are considered as the boundaries between regions. These boundaries are called watershed lines.
  - Markers:
    - Markers are used to define the initial points of different regions (objects) in the image. These can be set manually or automatically based on certain criteria (e.g., local minima).
    - The algorithm floods the image from these marker points, growing regions until they meet the boundaries of other regions.
  - Segmentation:
    - The result of the watershed algorithm is a segmented image where different regions are identified and separated by the watershed lines.



Segmentation Process

5 Object Detection with OpenCV

- Key Concepts
  - Topographic Interpretation:
    - The image is interpreted as a topographic surface, where higher pixel values (intensities) represent peaks (mountains), and lower values represent valleys.
    - The goal is to "flood" the valleys with water, and the lines where different water sources meet are considered as the boundaries between regions. These boundaries are called watershed lines.
  - Markers:
    - Markers are used to define the initial points of different regions (objects) in the image. These can be set manually or automatically based on certain criteria (e.g., local minima).
    - The algorithm floods the image from these marker points, growing regions until they meet the boundaries of other regions.
  - Segmentation:
    - The result of the watershed algorithm is a segmented image where different regions are identified and separated by the watershed lines.

Draw custom seeds

Segmentation

Object Detection with OpenCV

1. Import Libraries

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

2. Define a Helper Function to Display Images

```
def display(img, cmap='gray'):
    fig = plt.figure(figsize=(12, 10))
    ax = fig.add_subplot(111)
    ax.imshow(img, cmap=cmap)
    plt.axis('off')  # Hide the axis
```

3. Load and Preprocess the Image

```
duit = cv2.imread('duit.jpg')

# Apply median blur to reduce noise
blur = cv2.medianBlur(duit, 25)

# Convert to grayscale
gray = cv2.cvtColor(blur, cv2.COLOR_BGR2GRAY)
```

4. Apply Thresholding to Create a Binary Image

```
ret, thresh = cv2.threshold(gray, 130, 255,
cv2.THRESH_BINARY_INV)
```

5. Find Contours

```
contours, hier = cv2.findContours(thresh.copy(),
cv2.RETR_CCOMP, cv2.CHAIN_APPROX_SIMPLE)
```

Object Detection with OpenCV

6. Draw External Contours on the Original Image

```
for i in range(len(contours)):
    if hier[0][i][3] == -1:
        cv2.drawContours(duit, contours, i, (255, 0, 0), 10)
```

7. Display the Image with External Contours

```
display(duit, cmap='gray')
```

8. Reload the Original Image for Watershed Segmentation

```
myimg = cv2.imread('duit.jpg')

# Apply median blur to reduce noise
myblur = cv2.medianBlur(myimg, 35)
```

9. Apply Otsu's Thresholding

```
ret, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV
+ cv2.THRESH_OTSU)
```

10. Distance Transform to Identify Sure Foreground

```
dist_transform = cv2.distanceTransform(thresh, cv2.DIST_L2, 5)
ret, sure_fg = cv2.threshold(dist_transform, 0.7 *
dist_transform.max(), 255, 0)

# Convert sure_fg to uint8
sure_fg = np.uint8(sure_fg)
```

Object Detection with OpenCV

11. Determine Sure Background

```
kernel = np.ones((3, 3), np.uint8)
sure_bg = cv2.dilate(thresh, kernel, iterations=3)
```

12. Identify Unknown Regions

```
unknown = cv2.subtract(sure_bg, sure_fg)
```

13. Marker Labelling for Watershed

```
ret, markers = cv2.connectedComponents(sure_fg)

# Add 1 to all labels so that sure background is not 0 but 1
markers = markers + 1

# Mark the region of unknown with zero
markers[unknown == 255] = 0
```

14. Apply the Watershed Algorithm

```
watershed = cv2.watershed(myimg, markers)
```

15. Mark Watershed Boundaries

```
myimg[watershed == -1] = [255, 0, 0]
```

16. Find and Draw Contours on the Markers

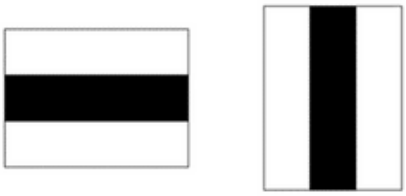
```
contours, hier = cv2.findContours(markers.copy(),
cv2.RETR_CCOMP, cv2.CHAIN_APPROX_SIMPLE)

for i in range(len(contours)):
    if hier[0][i][3] == -1:
        cv2.drawContours(myimg, contours, i, (255, 0, 0), 10)
```

5 Object Detection with OpenCV

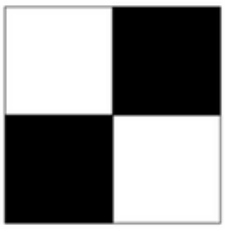
Face Detection Using Haar Cascades

- Haar Cascade is a machine learning-based approach where a cascade function is trained from a lot of positive and negative images. It is one of the most popular methods for face detection and was introduced by Paul Viola and Michael Jones in their 2001 paper "Rapid Object Detection using a Boosted Cascade of Simple Features." This method is efficient, allowing for real-time face detection.
- Haar-like features are simple rectangular features that resemble Haar wavelets. They are used to detect the presence of particular patterns in an image, such as edges, lines, and corners.
- Types of Features:
  - Edge features: Consist of adjacent rectangles, where one is light, and the other is dark. They detect transitions between light and dark regions.

  - Line features: Consist of three rectangles, where the middle one is light, and the two outer ones are dark, or vice versa.

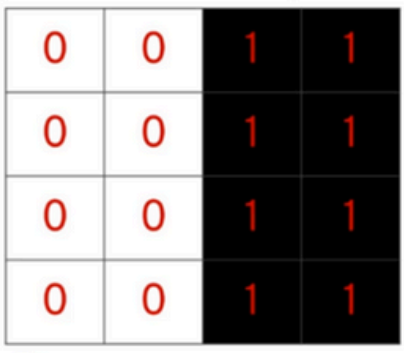


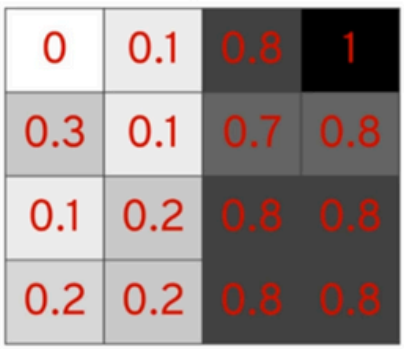
  - Four-sided features: Consist of four rectangles that form a checkerboard pattern.



5 Object Detection with OpenCV

- Calculation of a Haar-like Feature
  - Consider an edge feature composed of two adjacent rectangles:
    - Dark Region: This is typically the region where we expect lower pixel values (e.g., the shadow under the eyes).
    - Light Region: This is the region with higher pixel values (e.g., the forehead above the eyes).
  - The value of a Haar-like feature is calculated as the difference between the sum of pixel intensities in the dark region and the sum of pixel intensities in the light region.





  - The closer the value to 1, the better the feature is
  - Example:
    - Mean dark region = (0.8+0.7+0.8+0.8+1+0.8+0.8+0.8)/8 = 0.8125
    - Mean light region = (0+0.3+0.1+0.2+0.1+0.1+0.2+0.2)/8 = 0.15
    - Delta = 0.8125 - 0.15 = 0.6625
- Calculation of a Haar-like Feature
  - This type of calculation will be very computationally expensive
  - Need to use Viola-Jones algorithm using integral image

5 Object Detection with OpenCV

- The Viola-Jones algorithm is a robust and efficient method for object detection, particularly for face detection. A key innovation of this algorithm is the use of integral images (also known as summed area tables) to compute features rapidly. This speed, combined with the cascade of classifiers, allows the algorithm to perform real-time face detection.

31	2	4	33	5	36
12	26	9	10	29	25
13	17	21	22	20	18
24	23	15	16	14	19
30	8	28	27	11	7
1	35	34	3	32	6

31	33	37	70	75	111
43	71	84	127	161	222
56	101	135	200	254	333
80	148	197	278	346	444
110	186	263	371	450	555
111	222	333	444	555	666

15 + 16 + 14 + 28 + 27 + 11 =
101 + 450 − 254 − 186 = 111

- The key idea behind the cascade is that not all image regions need to be examined in detail.
- When scanning an image, each region (e.g., a 24x24 pixel window) is first passed through the first stage of the cascade.
- If the region fails this first classifier (i.e., it is unlikely to contain a face), it is immediately rejected, and no further computation is performed on that region.

Original Image

Convert to grayscale

Haar Cascade Feature - search for edge feature indicating eyes and cheeks

5 Object Detection with OpenCV

Haar Cascade Feature - search for edge feature indicating nose

Haar Cascade Feature - search for edge feature indicating mouth, etc.

Face detected

- However, this algorithm required a very large datasets to perform this operation of detecting features. But, there are a lot of pre-trained sets of features already exist.
- Download Haar Cascade pre trained .xml file at:
  - https://github.com/mdkhair/CV/archive/refs/heads/main.zip

```
import cv2
import matplotlib.pyplot as plt

# Load the pre-trained Haar Cascade classifier for face
detection
face_cascade =
cv2.CascadeClassifier('haarcascade_frontalface_default.xml')

# Load the image
img = cv2.imread('person.jpg')

# Convert to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Object Detection with OpenCV

```
# Detect faces in the image
faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1,
minNeighbors=5, minSize=(30, 30))

# Draw rectangles around detected faces
for (x, y, w, h) in faces:
 cv2.rectangle(img, (x, y), (x+w, y+h), (255, 0, 0), 2)

# Convert BGR to RGB for displaying with matplotlib
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# Display the output
plt.imshow(img_rgb)
plt.axis('off') # Hide axis ticks and labels
plt.show()
```

6 Object Tracking

Object Tracking

- Object tracking is a critical task in computer vision that involves following an object or multiple objects over time across a sequence of images or video frames. Unlike object detection, which identifies objects in individual frames, object tracking aims to maintain the identity of objects as they move across different frames. This is essential in many applications, such as video surveillance, autonomous driving, human-computer interaction, and sports analysis.
- Key Concepts in Object Tracking
  - Initialization:
    - Detection: The process typically starts with detecting the object of interest in the first frame.
    - Initialization: Once detected, the object is initialized for tracking in subsequent frames.
  - Tracking:
    - Tracking Algorithms: The object is tracked across subsequent frames using various algorithms that predict the object's location and update its position based on movement, appearance changes, and environmental factors.
  - Challenges:
    - Occlusion: Objects may be partially or fully obscured by other objects or by the background.
    - Illumination Changes: Varying lighting conditions can affect the appearance of the object.
    - Scale Variation: Objects can move closer or farther from the camera, leading to changes in size.
    - Object Motion: Objects can move in unpredictable ways, making tracking difficult.
    - Background Clutter: A complex background can make it challenging to distinguish the object from the surroundings.

6 Object Tracking

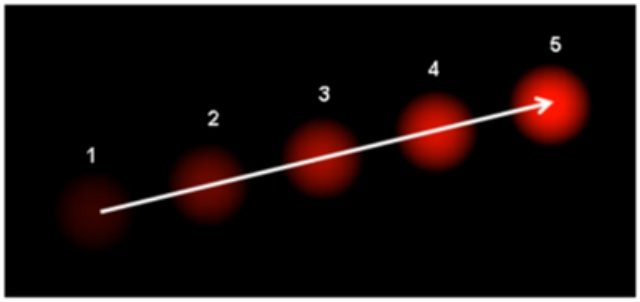
Object Tracking

- Object tracking is a critical task in computer vision that involves following an object or multiple objects over time across a sequence of images or video frames. Unlike object detection, which identifies objects in individual frames, object tracking aims to maintain the identity of objects as they move across different frames. This is essential in many applications, such as video surveillance, autonomous driving, human-computer interaction, and sports analysis.
- Key Concepts in Object Tracking
  - Initialization:
    - Detection: The process typically starts with detecting the object of interest in the first frame.
    - Initialization: Once detected, the object is initialized for tracking in subsequent frames.
  - Tracking:
    - Tracking Algorithms: The object is tracked across subsequent frames using various algorithms that predict the object's location and update its position based on movement, appearance changes, and environmental factors.
  - Challenges:
    - Occlusion: Objects may be partially or fully obscured by other objects or by the background.
    - Illumination Changes: Varying lighting conditions can affect the appearance of the object.
    - Scale Variation: Objects can move closer or farther from the camera, leading to changes in size.
    - Object Motion: Objects can move in unpredictable ways, making tracking difficult.
    - Background Clutter: A complex background can make it challenging to distinguish the object from the surroundings.

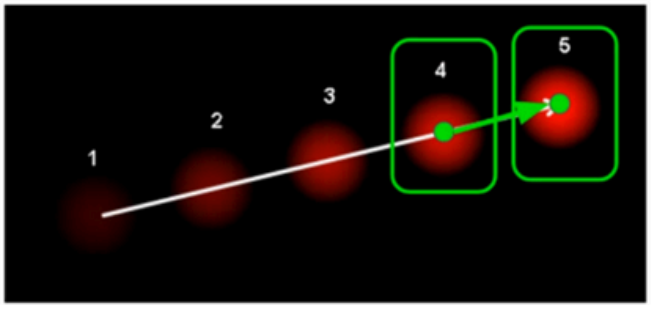
6 Object Tracking

Optical Flow

- Optical flow is a concept in computer vision that refers to the pattern of apparent motion of objects, surfaces, and edges in a visual scene, caused by the relative movement between an observer (such as a camera) and the scene. It provides a way to estimate the motion of every pixel in an image sequence (video) between two consecutive frames.
- Key Concepts of Optical Flow
  - Motion Estimation:
    - Optical flow estimates how much and in which direction each pixel in an image has moved between two consecutive frames.



  - Vector Field:
    - The result of optical flow estimation is a vector field, where each vector represents the direction and magnitude of motion for a specific pixel.



  - Applications:
    - Optical flow is used in various applications such as video compression, object tracking, motion detection, and autonomous navigation.

6 Object Tracking

- Assumptions in Optical Flow
  - To compute optical flow, certain assumptions are generally made:
  - Brightness Constancy:
    - The intensity of a pixel remains constant between consecutive frames, even as it moves. This means that if a pixel with intensity at time moves to a new location.
  - Small Motion:
    - The change in position of pixels between frames is assumed to be small, allowing for the use of linear approximations.
  - Spatial and Temporal Coherence:
    - Neighboring pixels in an image tend to have similar motion (spatial coherence), and the motion is smooth over time (temporal coherence).

Lucas-Kanade Method

- The Lucas-Kanade method is a widely used algorithm for estimating optical flow, which is the motion of objects or the camera between consecutive frames in a video sequence. Named after Bruce D. Lucas and Takeo Kanade, who proposed it in 1981, this method is particularly popular due to its balance between accuracy and computational efficiency.

```
import cv2
import numpy as np

# Initialize the webcam
cap = cv2.VideoCapture(0)

# Set parameters for Lucas-Kanade optical flow
lk_params = dict(winSize=(15, 15),
                 maxLevel=2,
                 criteria=(cv2.TERM_CRITERIA_EPS |
cv2.TERM_CRITERIA_COUNT, 10, 0.03))
```

Object Tracking

```
# Parameters for Shi-Tomasi corner detection
feature_params = dict(maxCorners=100,
 qualityLevel=0.3,
 minDistance=7,
 blockSize=7)

# Capture the first frame and detect corners
ret, old_frame = cap.read()
if not ret:
 print("Failed to capture image")
 cap.release()
 cv2.destroyAllWindows()

# Convert frame to grayscale
old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)

# Detect initial points to track
p0 = cv2.goodFeaturesToTrack(old_gray, mask=None,
**feature_params)

# Create a mask image for drawing purposes
mask = np.zeros_like(old_frame)

while True:
 # Capture a new frame
 ret, frame = cap.read()
 if not ret:
 break

 # Convert to grayscale
 frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

Object Tracking

```
# Calculate optical flow
 p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, frame_gray,
p0, None, **lk_params)

 # Select good points
 good_new = p1[st == 1]
 good_old = p0[st == 1]

 # Draw the tracks
 for i, (new, old) in enumerate(zip(good_new, good_old)):
 a, b = new.ravel()
 c, d = old.ravel()
 mask = cv2.line(mask, (a, b), (c, d), (0, 255, 0), 2)
 frame = cv2.circle(frame, (a, b), 5, (0, 0, 255), -1)

 # Overlay the original frame with the drawing
 img = cv2.add(frame, mask)

 # Display the frame with the tracking lines
 cv2.imshow('Lucas-Kanade Optical Flow Tracking', img)

 # Exit when 'q' is pressed
 if cv2.waitKey(1) & 0xFF == ord('q'):
 break

 # Update the previous frame and points
 old_gray = frame_gray.copy()
 p0 = good_new.reshape(-1, 1, 2)

# Release the webcam and close windows
cap.release()
cv2.destroyAllWindows()
```

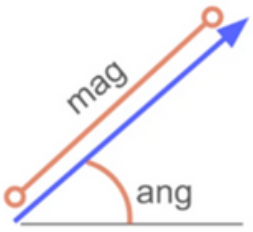
Object Tracking

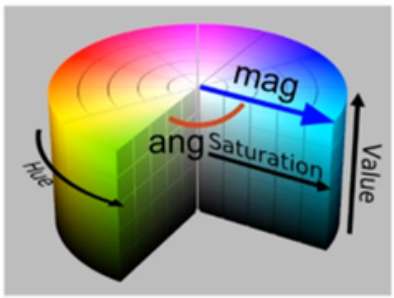
```
# Calculate optical flow
 p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, frame_gray,
p0, None, **lk_params)

 # Select good points
 good_new = p1[st == 1]
 good_old = p0[st == 1]

 # Draw the tracks
 for i, (new, old) in enumerate(zip(good_new, good_old)):
 a, b = new.ravel()
 c, d = old.ravel()
 mask = cv2.line(mask, (a, b), (c, d), (0, 255, 0), 2)
 frame = cv2.circle(frame, (a, b), 5, (0, 0, 255), -1)

 # Overlay the original frame with the drawing
 img = cv2.add(frame, mask)

 # Display the frame with the tracking lines
 cv2.imshow('Lucas-Kanade Optical Flow Tracking', img)

 # Exit when 'q' is pressed
 if cv2.waitKey(1) & 0xFF == ord('q'):
 break

 # Update the previous frame and points
 old_gray = frame_gray.copy()
 p0 = good_new.reshape(-1, 1, 2)

# Release the webcam and close windows
cap.release()
cv2.destroyAllWindows()
```

6 Object Tracking

Dense Optical Flow

- Dense Optical Flow is a technique in computer vision used to estimate the motion of every pixel in a video frame between two consecutive frames. Unlike sparse optical flow methods, such as the Lucas-Kanade method, which track only specific points (like corners), dense optical flow attempts to compute the flow vector for every pixel in the image. This results in a complete motion field that describes the movement across the entire image.
- Key Concepts of Dense Optical Flow
  - Motion Estimation for Every Pixel:
    - Dense optical flow provides a flow vector (displacement vector) for every pixel, indicating how that pixel has moved between two consecutive frames.
  - Vector Field:
    - The output of dense optical flow is a vector field where each vector represents the motion (magnitude and direction) of a pixel.
  - Applications:
    - Dense optical flow is used in motion detection, object tracking, video compression, video stabilization, and understanding the dynamic behavior of scenes in videos.





Object Tracking

Farneback Method

- The Farneback method estimates the motion of each pixel by modeling the neighborhood of each pixel with a quadratic polynomial. It then uses these models to compute the displacement between two consecutive frames.

```
import cv2
import numpy as np

# Initialize the webcam
cap = cv2.VideoCapture(0)

# Capture the first frame
ret, frame1 = cap.read()
prev_gray = cv2.cvtColor(frame1, cv2.COLOR_BGR2GRAY)

while True:
    # Capture the next frame
    ret, frame2 = cap.read()
    if not ret:
        break

    # Convert to grayscale
    gray = cv2.cvtColor(frame2, cv2.COLOR_BGR2GRAY)

    # Calculate dense optical flow using the Farneback method
    flow = cv2.calcOpticalFlowFarneback(prev_gray, gray, None,
0.5, 3, 15, 3, 5, 1.2, 0)

    # Convert flow vectors to polar coordinates (magnitude and
angle)
    mag, ang = cv2.cartToPolar(flow[..., 0], flow[..., 1])
```

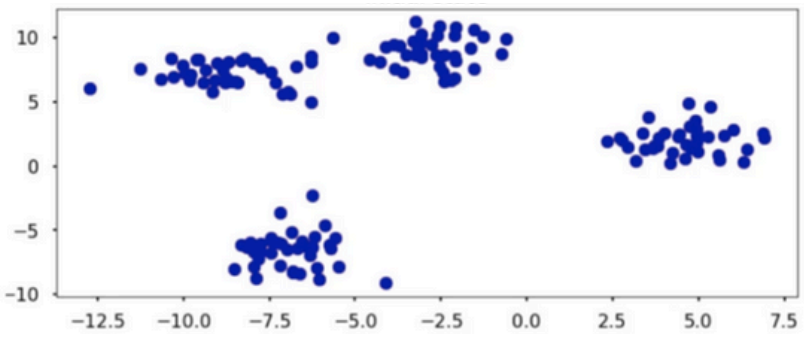
Object Tracking

```
 # Use the angle to set the hue and the magnitude to set the
value in the HSV color space
 hsv = np.zeros_like(frame1)
 hsv[..., 1] = 255
 hsv[..., 0] = ang * 180 / np.pi / 2
 hsv[..., 2] = cv2.normalize(mag, None, 0, 255,
cv2.NORM_MINMAX)

 # Convert HSV image to BGR color space for display
 bgr = cv2.cvtColor(hsv, cv2.COLOR_HSV2BGR)

 # Display the result
 cv2.imshow('Dense Optical Flow', bgr)

 # Exit the loop if 'q' is pressed
 if cv2.waitKey(1) & 0xFF == ord('q'):
 break

 # Update the previous frame
 prev_gray = gray.copy()

# Release the webcam and close windows
cap.release()
cv2.destroyAllWindows()
```

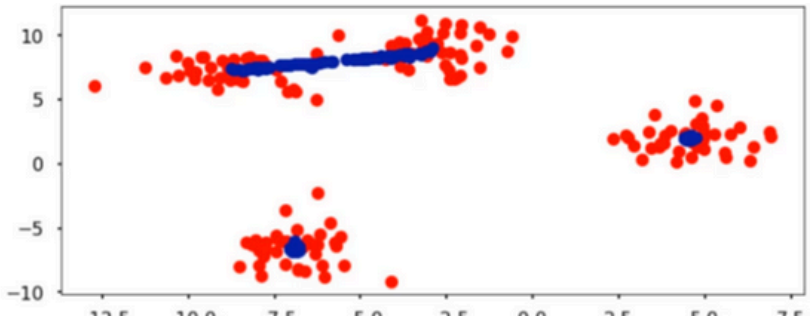
6 Object Tracking

Mean Shift

- Mean Shift is a non-parametric feature-space analysis technique widely used for clustering and mode-seeking. In computer vision, it is particularly popular for object tracking and image segmentation. The algorithm is based on shifting data points towards the mode (peak) of the data distribution, effectively finding the densest areas in a feature space.
- Imagine you have a 2D space with several points scattered across it. These points might represent pixels in an image with specific colors or other features. As the Mean Shift algorithm runs:
  - Each point is examined to see where the highest density of nearby points lies.

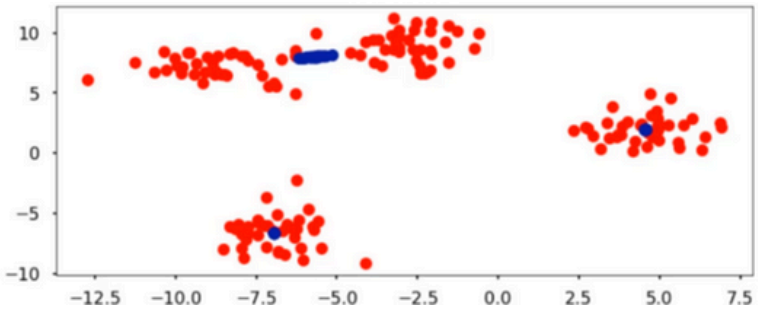


- The point is then shifted towards this denser area.

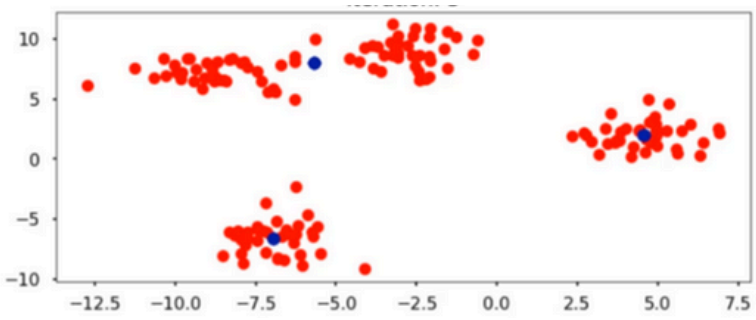


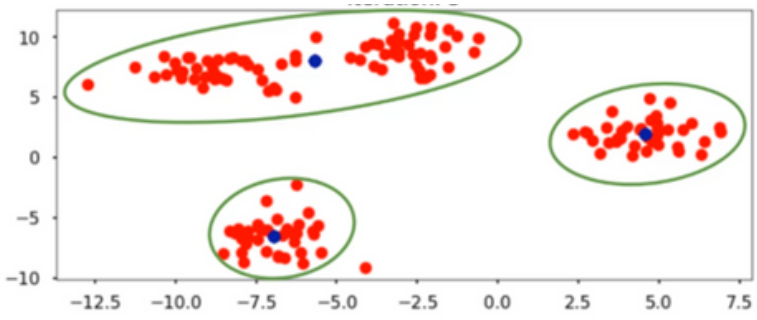
6 Object Tracking

- This process repeats until all points have moved to their respective modes.



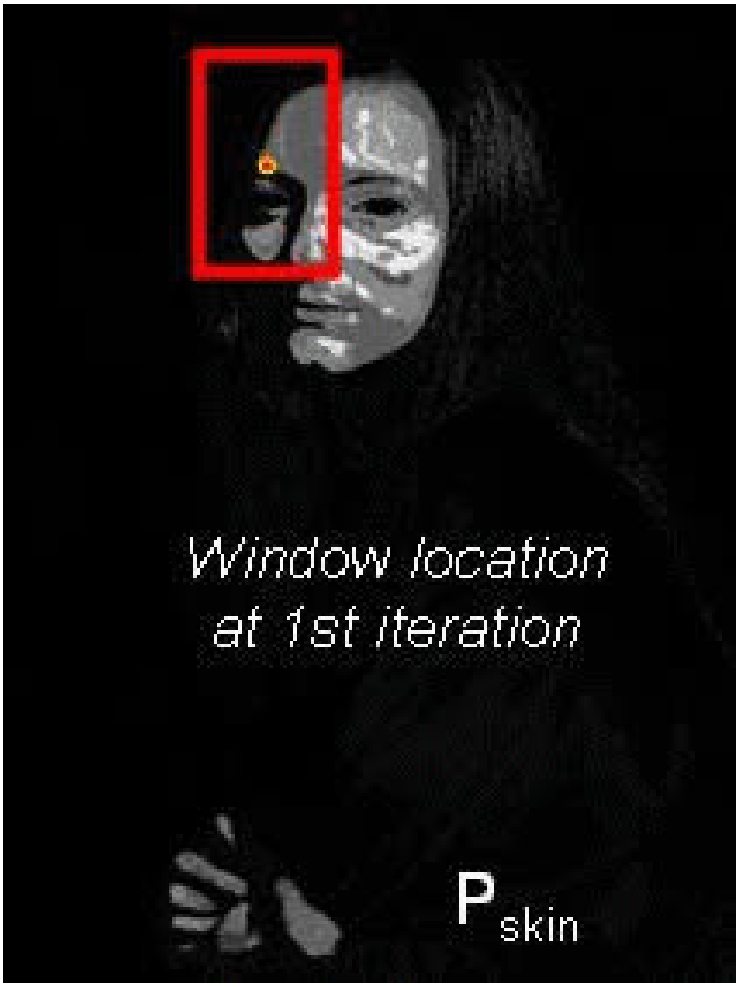
- Points that end up at the same mode are considered to belong to the same cluster, meaning they share similar characteristics or are closely located in the feature space.





6 Object Tracking

- Example:



Object Tracking

```
import cv2

# Load the pre-trained Haar Cascade classifier for face
detection
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_frontalface_default.xml')

# Initialize the webcam
cap = cv2.VideoCapture(0)

# Take the first frame of the video
ret, frame = cap.read()

# Convert the frame to grayscale (Haar cascades work better
with grayscale images)
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

# Detect faces in the frame
face_rects = face_cascade.detectMultiScale(gray,
scaleFactor=1.1, minNeighbors=5, minSize=(30, 30))

# Check if any face is detected
if len(face_rects) > 0:
    # Get the coordinates of the first detected face
    x, y, w, h = tuple(face_rects[0])
    track_window = (x, y, w, h)

    # Set up the ROI for tracking
    roi = frame[y:y+h, x:x+w]
    hsv_roi = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)
```

Object Tracking

```
 # Create a mask to filter out low light values
 mask = cv2.inRange(hsv_roi, (0, 60, 32), (180, 255, 255))
 roi_hist = cv2.calcHist([hsv_roi], [0], mask, [180], [0, 180])

 # Normalize the histogram
 cv2.normalize(roi_hist, roi_hist, 0, 255, cv2.NORM_MINMAX)

 # Set up the termination criteria, either 10 iterations or
move by at least 1 pt
 term_crit = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT,
10, 1)

 while True:
 ret, frame = cap.read()
 if not ret:
 break

 hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

 # Back-projection to get the probability distribution
 dst = cv2.calcBackProject([hsv], [0], roi_hist, [0, 180], 1)

##########################################################
 # Apply mean shift to get the new location
 ret, track_window = cv2.meanShift(dst, track_window,
term_crit)

 # Draw the tracked window on the frame
 x, y, w, h = track_window
 img2 = cv2.rectangle(frame, (x, y), (x+w, y+h), 255, 2)
##########################################################
```

Object Tracking

```
 # Display the result
 cv2.imshow('Mean Shift Tracking', img2)

# Exit if 'q' is pressed
 if cv2.waitKey(1) & 0xFF == ord('q'):
 break

 cap.release()
 cv2.destroyAllWindows()
else:
 print("No face detected in the first frame.")
 cap.release()
 cv2.destroyAllWindows()
```

- CamShift is an advanced version of Mean Shift that adapts the size and orientation of the tracking window as the object moves. Just replace the the coding that between ########################## with this coding

```
        # Apply CamShift to get the new location and size
        ret, track_window = cv2.CamShift(dst, track_window,
 term_crit)

        # Draw the tracked window on the frame (as a rotated
 rectangle)
        pts = cv2.boxPoints(ret)
        pts = np.int0(pts)
        img2 = cv2.polylines(frame, [pts], True, 255, 2)
```

6 Object Tracking

Tracking API

- When it comes to object tracking using OpenCV, the library provides several APIs (Application Programming Interfaces) that implement various tracking algorithms. These tracking APIs allow you to follow an object across video frames by maintaining its position and size as the object moves, changes scale, or even rotates.
- OpenCV offers a unified interface to several different tracking algorithms, each with its own strengths and weaknesses. These algorithms are implemented as part of the cv2.Tracker class in OpenCV.

BOOSTING Tracker (cv2.TrackerBoosting_create):

- Based on the same algorithm used in the Haar Cascade detector, this tracker uses multiple weak classifiers and is somewhat outdated compared to other methods. It tends to perform poorly in terms of accuracy and robustness.

MIL Tracker (cv2.TrackerMIL_create):

- MIL stands for Multiple Instance Learning. This tracker improves on BOOSTING by considering not just the best guess but a neighborhood of possible locations. This makes it more robust to changes in appearance but can still struggle with fast motion or significant scale changes.

KCF Tracker (cv2.TrackerKCF_create):

- KCF stands for Kernelized Correlation Filters. This tracker is faster than MIL and works well when the object is not changing much in scale or orientation. It uses a kernelized approach to detect and track objects efficiently.

TLD Tracker (cv2.TrackerTLD_create):

- TLD stands for Tracking, Learning, and Detection. This tracker not only tracks the object but also learns its appearance, allowing it to re-detect the object if tracking is lost. However, it can be prone to false positives.

6 Object Tracking

MedianFlow Tracker (cv2.TrackerMedianFlow_create):

- This tracker performs well when the object's movement is predictable and relatively slow. It is especially good at handling issues like occlusions or tracking failures by correcting them. However, it can fail with fast or erratic motion.

GOTURN Tracker (cv2.TrackerGOTURN_create):

- GOTURN stands for Generic Object Tracking Using Regression Networks. It uses a deep learning-based approach, which makes it more accurate in complex scenarios. However, it requires pre-trained models and is not included by default in OpenCV.

MOSSE Tracker (cv2.TrackerMOSSE_create):

- MOSSE (Minimum Output Sum of Squared Error) tracker is extremely fast and performs well when the object has minimal changes in scale and rotation. It is particularly useful when computational speed is critical.

CSRT Tracker (cv2.TrackerCSRT_create):

- CSRT stands for Discriminative Correlation Filter with Channel and Spatial Reliability. This tracker is more accurate than KCF but slower. It is designed to be robust to changes in scale and appearance, making it one of the best-performing trackers in OpenCV for general-purpose use.

Object Tracking

```
import cv2

# Dictionary mapping user input to OpenCV tracker creation
functions
trackers = {
    "1": ("BOOSTING", cv2.legacy.TrackerBoosting_create),
    "2": ("MIL", cv2.legacy.TrackerMIL_create),
    "3": ("KCF", cv2.legacy.TrackerKCF_create),
    "4": ("TLD", cv2.legacy.TrackerTLD_create),
    "5": ("MedianFlow", cv2.legacy.TrackerMedianFlow_create),
    "6": ("GOTURN", cv2.TrackerGOTURN_create),
    "7": ("MOSSE", cv2.legacy.TrackerMOSSE_create),
    "8": ("CSRT", cv2.TrackerCSRT_create)
}

# Function to display the available tracking methods and get
user input
def choose_tracker():
    print("Choose a tracking method:")
    for key, (name, _) in trackers.items():
        print(f"{key}. {name}")

    choice = input("Enter the number of the tracking method you
want to use: ")
    return trackers.get(choice)

# Initialize the webcam
cap = cv2.VideoCapture(0)

# Read the first frame from the video
ret, frame = cap.read()
```

Object Tracking

```
# Let the user select a tracking algorithm
tracker_info = choose_tracker()

if tracker_info:
 tracker_name, tracker_func = tracker_info
 print(f"Using {tracker_name} tracker")

 # Define an initial bounding box
 bbox = cv2.selectROI("Select ROI", frame, False)

 # Initialize the selected tracker
 tracker = tracker_func()
 tracker.init(frame, bbox)

 while True:
 # Capture a new frame
 ret, frame = cap.read()
 if not ret:
 break

 # Update the tracker
 success, bbox = tracker.update(frame)

 # Draw the bounding box
 if success:
 x, y, w, h = int(bbox[0]), int(bbox[1]), int(bbox[2]),
int(bbox[3])
 cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 0, 0), 2)
 cv2.putText(frame, f"Tracking: {tracker_name}", (10, 30),
cv2.FONT_HERSHEY_SIMPLEX, 0.75, (50, 170, 50), 2)
 else:
 cv2.putText(frame, "Tracking failed", (100, 80),
cv2.FONT_HERSHEY_SIMPLEX, 0.75, (0, 0, 255), 2)
```

6 Object Tracking

```
    # Display the result
    cv2.imshow(f"{tracker_name} Tracker", frame)

    # Exit on 'q' key
    if cv2.waitKey(1) & 0xFF == ord('q'):
    break

    cap.release()
    cv2.destroyAllWindows()
else:
    print("Invalid selection. Exiting...")
    cap.release()
    cv2.destroyAllWindows()
```

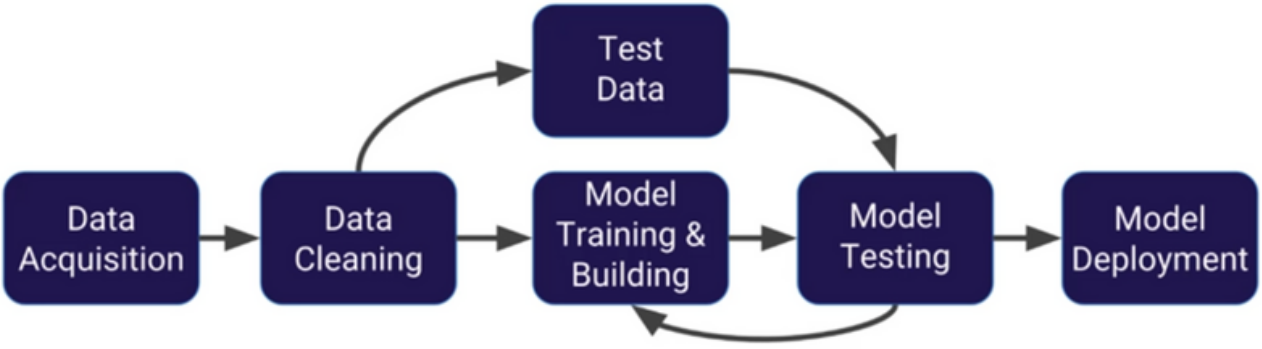
7 Deep Learning for CV

Machine Learning

- Machine Learning (ML) is a branch of artificial intelligence (AI) that focuses on developing algorithms and models that allow computers to learn from and make decisions based on data. Unlike traditional programming, where specific rules are hardcoded, machine learning systems can improve their performance over time by learning from experience.

Supervised Learning::

- In supervised learning, the model is trained on a labeled dataset, which means that each training example comes with an associated output or label.
- The goal is for the model to learn the mapping from inputs to outputs so that it can predict the labels of new, unseen data.
- How It Works:
  - The model learns from the labeled data by minimizing the error between its predictions and the actual labels. This process often involves adjusting the model's parameters using algorithms like gradient descent.
- Examples:
  - Classification: Assigning labels to inputs, such as determining if an email is spam or not (binary classification) or identifying the species of a flower (multi-class classification).
  - Regression: Predicting a continuous value, such as forecasting stock prices or predicting house prices based on features like size and location.



7 Deep Learning for CV

Classification Metrics

- Evaluating a machine learning model, especially in classification tasks, is crucial to understanding its performance. Various metrics are used to assess how well a model performs in predicting the correct classes. Here's an explanation of the key classification metrics:

1. Accuracy

- Definition: Accuracy is the ratio of correctly predicted instances to the total instances.

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Instances}}$$

- When to Use: Accuracy is a useful metric when the classes are balanced (i.e., the number of instances in each class is roughly equal).
- Limitations: In cases of imbalanced datasets (where one class significantly outnumbers the others), accuracy can be misleading. For example, if 95% of the data belongs to one class, a model that predicts the majority class all the time will have high accuracy but poor predictive power.

2. Precision

- Definition: Precision (also known as Positive Predictive Value) is the ratio of correctly predicted positive observations to the total predicted positives.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

- When to Use: Precision is important when the cost of a false positive is high. For instance, in email spam detection, a false positive (classifying a legitimate email as spam) can lead to important emails being missed.
- Limitations: Precision alone does not consider how many actual positives were missed (which is captured by recall).

3. Recall (Sensitivity or True Positive Rate)

- Definition: Recall is the ratio of correctly predicted positive observations to all observations in the actual class.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

- When to Use: Recall is important when the cost of false negatives is high. For example, in medical diagnostics, missing a positive case (false negative) can be very dangerous.
- Limitations: Recall alone does not take into account the number of false positives.

4. F1 Score

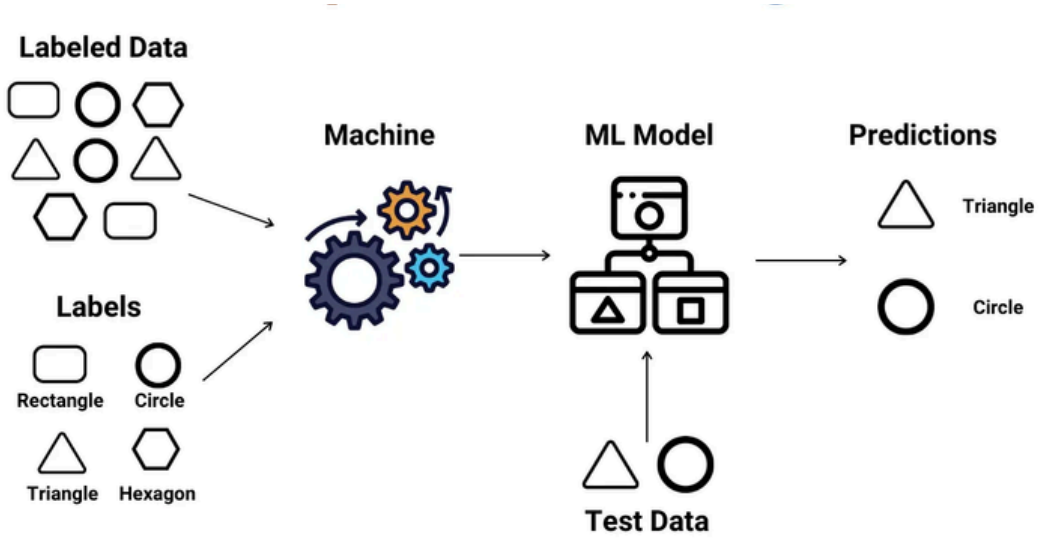
- Definition: The F1 score is the harmonic mean of Precision and Recall. It provides a balance between the two metrics and is particularly useful when the classes are imbalanced.

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

- When to Use: The F1 score is a good choice when you need to find a balance between precision and recall, and when the dataset has an uneven class distribution.
- Limitations: The F1 score may not be intuitive to interpret because it combines two metrics into one. It's best used when you want a single measure that balances precision and recall.

7 Deep Learning for CV

Prediction process



1. Data Collection
   - Collect Labeled Images:
     - You need a dataset of images where each image is labeled as either "triangle" or "circle." This dataset will be used to train the model.
     - For example, you might have 1,000 images of triangles and 1,000 images of circles, all correctly labeled.
2. Data Preprocessing
   - Resize Images:
     - Resize all images to a uniform size (e.g., 64x64 pixels) to ensure consistency when feeding them into the model.
   - Convert to Grayscale (Optional):
     - If the shape recognition task doesn't rely on color, convert images to grayscale to reduce complexity.
   - Normalization:
     - Normalize pixel values to a range of 0 to 1 (by dividing pixel values by 255). This helps the model learn more effectively.

7 Deep Learning for CV

- Data Augmentation:
  - Optionally, apply transformations such as rotations, flips, or scaling to increase the diversity of your training data, which can improve model robustness.
- Label Encoding:
  - Convert the labels ("triangle" and "circle") into numerical format, such as 0 for triangle and 1 for circle.
- Train-Test Split:
  - Split the dataset into a training set (e.g., 80%) and a test set (e.g., 20%). The training set is used to train the model, and the test set is used to evaluate its performance.

3. Feature Extraction (Optional)

- Manual Feature Extraction:
  - If using traditional machine learning models (like SVM, Decision Trees, etc.), you might extract features manually, such as edges, corners, or shapes using techniques like edge detection (Canny), contour detection, or Hough transform.
- Automatic Feature Extraction:
  - If using deep learning models (like Convolutional Neural Networks or CNNs), feature extraction is handled automatically during the training process.

4. Model Selection

- Choose a Model:
  - For image classification, Convolutional Neural Networks (CNNs) are commonly used due to their ability to automatically learn spatial hierarchies of features from input images.
  - Alternatively, for simpler tasks or smaller datasets, traditional machine learning models like Support Vector Machines (SVM) or Random Forests might also be effective.

7 Deep Learning for CV

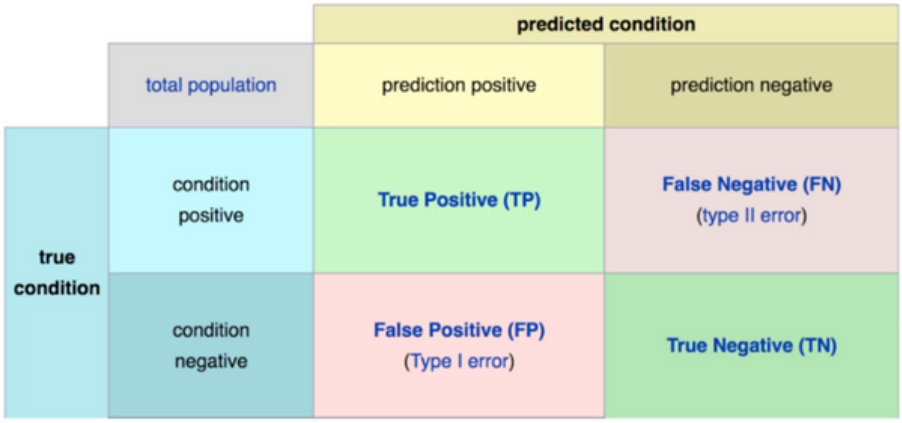
5. Model Training
- Feed Data into the Model:
  - Use the training data (images and their corresponding labels) to train the model. The model will learn to map the input images to the correct labels.
- Loss Function:
  - Use a loss function like Cross-Entropy Loss to measure how well the model's predictions match the actual labels.
- Optimization:
  - Use an optimization algorithm like Stochastic Gradient Descent (SGD) or Adam to adjust the model's parameters to minimize the loss function.
- Epochs and Batch Size:
  - Train the model over multiple epochs (iterations over the entire training dataset) and use a suitable batch size (number of images processed before the model's parameters are updated).

6. Model Evaluation
- Evaluate on Test Data:
  - After training, evaluate the model on the test dataset to see how well it generalizes to new, unseen data.
- Classification Metrics:
  - Use metrics like Accuracy, Precision, Recall, F1 Score, and Confusion Matrix to assess the model's performance.

7 Deep Learning for CV

Confusion Matrix

- A Confusion Matrix is a powerful tool for evaluating the performance of a classification model. It provides a detailed breakdown of the model's predictions compared to the actual labels, allowing you to see not only how many predictions were correct but also where the model is making mistakes.



1. True Positive (TP):
   - The number of instances where the model correctly predicted the positive class.
   - Example: The model correctly predicts that an image is a circle.
2. True Negative (TN):
   - The number of instances where the model correctly predicted the negative class.
   - Example: The model correctly predicts that an image is not a circle (e.g., a triangle).
3. False Positive (FP):
   - The number of instances where the model incorrectly predicted the positive class (also known as a "Type I error").
   - Example: The model incorrectly predicts that a triangle image is a circle.
4. False Negative (FN):
   - The number of instances where the model incorrectly predicted the negative class (also known as a "Type II error").
   - Example: The model incorrectly predicts that a circle image is not a circle (e.g., labels it as a triangle).

7 Deep Learning for CV

- Example of a Confusion Matrix
  - Let's consider a binary classification task where a model is trying to distinguish between circles and triangles. Suppose you have a test dataset with 100 images: 60 circles and 40 triangles.

	Predicted Circle	Predicted Triangle
Actual Circle	50 (TP)	10 (FN)
Actual Triangle	5 (FP)	35 (TN)

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

$$\text{Accuracy} = \frac{50 + 35}{50 + 35 + 5 + 10} = \frac{85}{100} = 0.85 \text{ or } 85\%$$

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Precision} = \frac{50}{50 + 5} = \frac{50}{55} \approx 0.91 \text{ or } 91\%$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

$$\text{Recall} = \frac{50}{50 + 10} = \frac{50}{60} \approx 0.83 \text{ or } 83\%$$

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

$$\text{F1 Score} = 2 \times \frac{0.91 \times 0.83}{0.91 + 0.83} \approx 0.87 \text{ or } 87\%$$

7 Deep Learning for CV

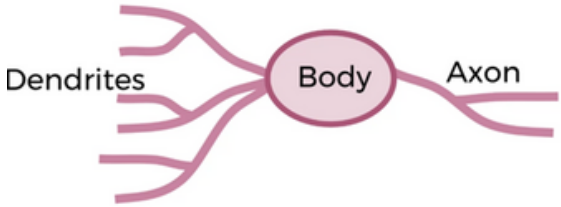
Deep Learning

- Deep Learning is a subfield of machine learning that focuses on algorithms inspired by the structure and function of the brain called artificial neural networks. It is a key technology behind many modern artificial intelligence (AI) systems, including those for image and speech recognition, natural language processing, autonomous vehicles, and more.

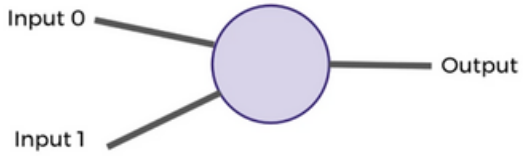
Artificial Neural Networks (ANNs):

- Neural networks are the foundation of deep learning. They consist of layers of interconnected nodes, called neurons, which are loosely modeled after the neurons in the human brain.
- Each neuron receives one or more inputs, processes them using a weighted sum, and then passes the result through an activation function to produce an output.

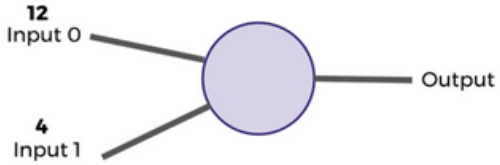
- The biological neuron:



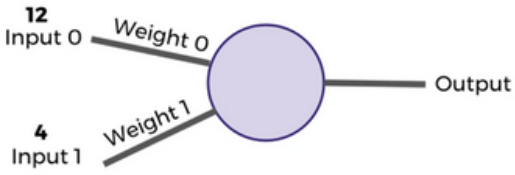
This simple model is known as a perceptron.



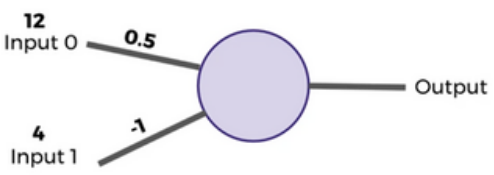
Inputs will be values of features



Inputs are multiplied by a weight

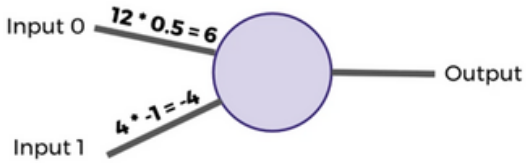


Weights initially start off as random

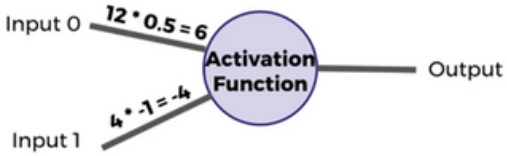


7 Deep Learning for CV

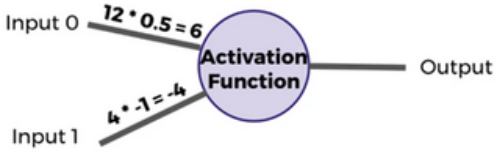
Inputs are now multiplied by weights



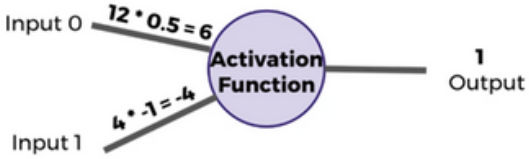
Many activation functions to choose from, we'll cover this in more detail later!



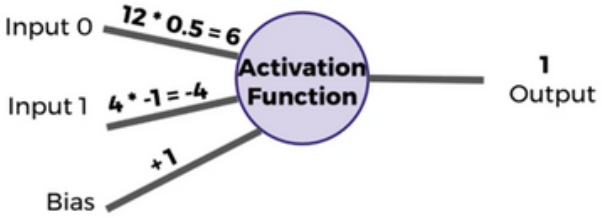
If sum of inputs is positive return 1, if sum is negative output 0.



In this case 6-4=2 so the activation function returns 1.



The problem occurs when the original input is 0. When it is times by the weighted, it will remain 0. So, a bias need to be introduced to prevent the problem.



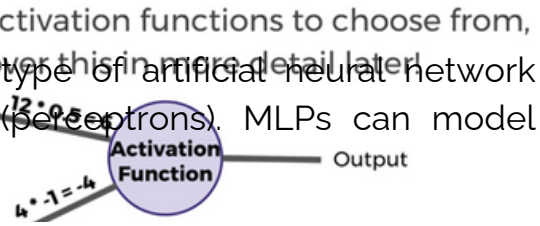
Mathematical model to represent perceptron model:

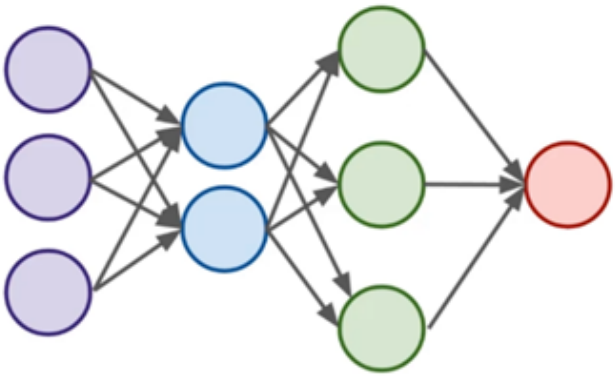
$$y = f\left(\sum_{i=1}^{n} w_i \cdot x_i + b\right)$$

7 Deep Learning for CV

Multi-Layer Perceptron Neural Network

- Multi-Layer Perceptron (MLP), which is a type of artificial neural network consisting of multiple layers of neurons (perceptrons). MLPs can model complex, non-linear relationships in data.



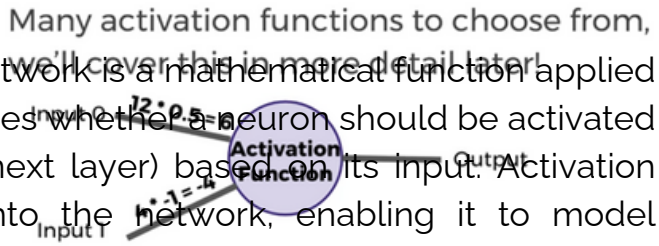


Structure of a Multi-Layer Perceptron

1. Input Layer:
   - The input layer consists of neurons that directly receive the features from the input data. Each neuron corresponds to one input feature.
2. Hidden Layers:
   - Between the input and output layers are one or more hidden layers. Each hidden layer consists of multiple neurons, and each neuron in a hidden layer is connected to all neurons in the previous layer.
   - The hidden layers are where the network learns to capture complex patterns and representations in the data.
   - A "deep" network refers to a neural network with many layers (typically more than three). These layers enable the network to learn complex features and representations from data.
3. Output Layer:
   - The output layer produces the final output of the network. In binary classification, this layer typically consists of a single neuron with a sigmoid activation function. In multi-class classification, the output layer might have multiple neurons with a softmax activation function.

7 Deep Learning for CV

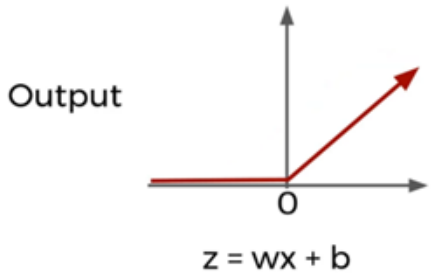
Activation Function



- An Activation Function in a neural network is a mathematical function applied to the output of a neuron. It determines whether a neuron should be activated (i.e., pass its signal forward to the next layer) based on its input. Activation functions introduce non-linearity into the network, enabling it to model complex patterns and relationships in data.
- Without activation functions, a neural network would behave like a simple linear model, no matter how many layers it has. Linear models can only learn linear relationships, which limits their ability to solve complex tasks. By introducing non-linear activation functions, neural networks can learn and approximate almost any function, making them powerful and versatile.

ReLU (Rectified Linear Unit)

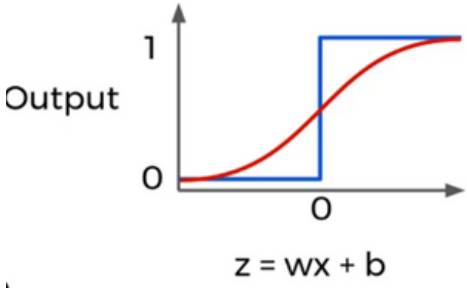
- If the input x is positive, ReLU outputs x.
- If the input x is negative or zero, ReLU outputs 0.



$$\mathrm{ReLU}(x) = \max(0, x)$$

Sigmoid

- The sigmoid function maps any real-valued input to a value between 0 and 1.

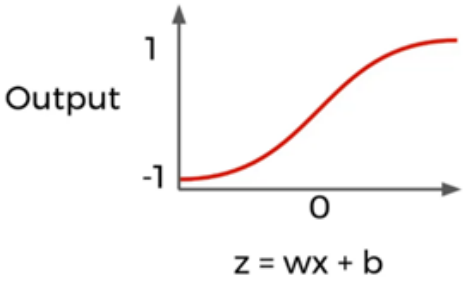


$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

7 Deep Learning for CV

Tanh (Hyperbolic Tangent)

- The tanh function maps any real-valued input to a value between -1 and 1.



Many activation functions to choose from, we'll cover this in more detail later!

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

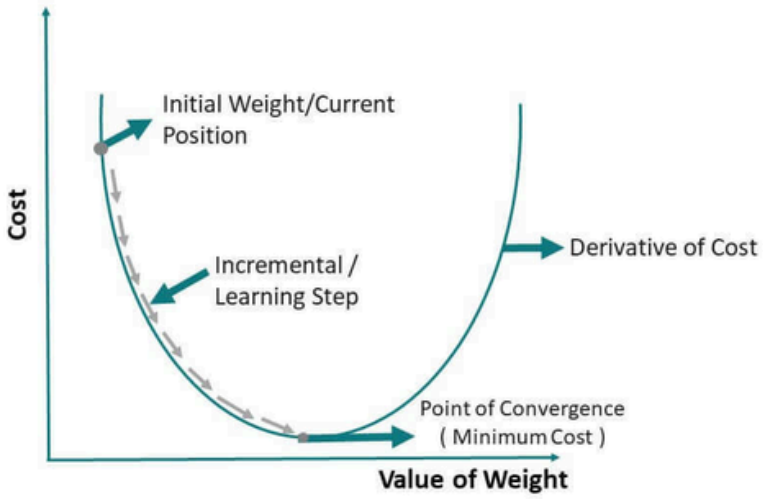
Cost Function / Loss Function

- A Loss Function (also known as a cost function or objective function) is a crucial component in training a machine learning model, particularly in supervised learning. It quantifies how well the model's predictions align with the actual target values. The primary goal during training is to minimize the loss function, thereby improving the model's predictions.
- Two common cost function:
  - Cross-Entropy Loss
    - Cross-Entropy Loss, also known as Log Loss or Negative Log-Likelihood, is a commonly used loss function for classification tasks, particularly when the output is a probability distribution over different classes.
    - Cross-Entropy Loss is used when you're trying to classify something into one or more categories.
  - Mean Squared Error (MSE)
    - Mean Squared Error (MSE) is a loss function commonly used for regression tasks, where the goal is to predict a continuous output, such as house prices, temperatures, or stock prices.

7 Deep Learning for CV

Gradient Descent

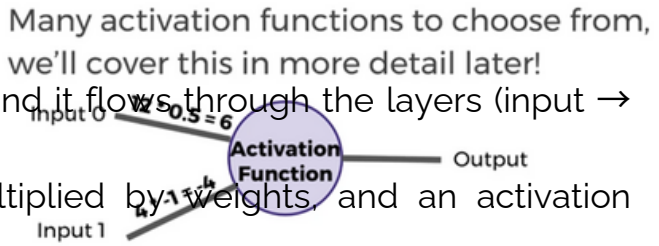
- Gradient Descent is an optimization algorithm used to minimize the cost function (also known as the loss function) in machine learning and deep learning models. The cost function measures how well the model's predictions match the actual data. Gradient Descent adjusts the model's parameters (such as weights and biases) to minimize the cost function, which in turn improves the model's accuracy.



- How quick gradient descent to adjust the optimal parameters across the entire network? So, we need a good algorithm to do it fast. It is the job for backpropagation.
- Backpropagation is the mechanism for computing the gradients of the loss function with respect to the network's weights. It works by propagating the error backward through the network, layer by layer.

7 Deep Learning for CV

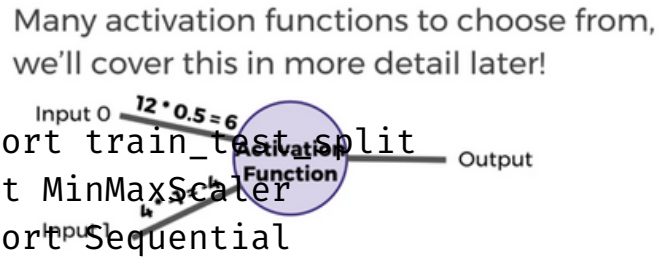
Steps of Backpropagation



1. Forward Pass:
   - You input data into the network, and it flows through the layers (input → hidden → output).
   - At each neuron, the data is multiplied by weights, and an activation function is applied.
   - The network produces an output (a prediction).
2. Calculate the Error (Loss):
   - Compare the network's prediction to the actual target (e.g., did it correctly identify the image as a cat?).
   - Use a loss function to measure how far off the prediction is from the actual answer. This gives you the error or "loss."
3. Backward Pass (Backpropagation):
   - The goal is to adjust the weights to reduce this error. But how?
   - Backpropagation works backward from the output layer to the input layer, figuring out how much each weight contributed to the error.
4. Key Idea:
   - If a weight contributed to a large error, it needs to be adjusted more.
   - If it contributed to a small error, it needs less adjustment.
5. Update the Weights:
   - After determining how much each weight contributed to the error, backpropagation gives us a way to update these weights.
   - We tweak the weights slightly to reduce the error. This is done using an algorithm called Gradient Descent.
   - Gradient Descent adjusts the weights in the direction that reduces the error, like a ball rolling downhill to reach the lowest point.
6. Repeat:
   - This process of forward pass, calculating error, backpropagation, and updating weights is repeated many times (over many examples) until the network learns to make accurate predictions.

7 Deep Learning for CV

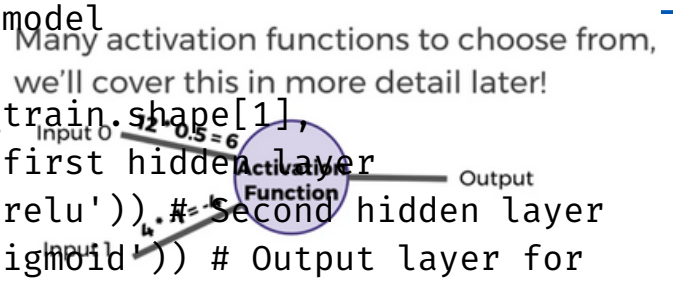
Predicting features from CSV file




```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.models import load_model

# Load the data using numpy.genfromtxt
data = np.genfromtxt('your_data.csv', delimiter=',',
skip_header=1)

# Separate features (X) and labels (y)
X = data[:, :-1]  # All columns except the last one are
features
y = data[:, -1]   # The last column is the label

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Initialize MinMaxScaler to scale features between 0 and 1
scaler = MinMaxScaler()

# Fit the scaler on the training data and transform both
training and testing data
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

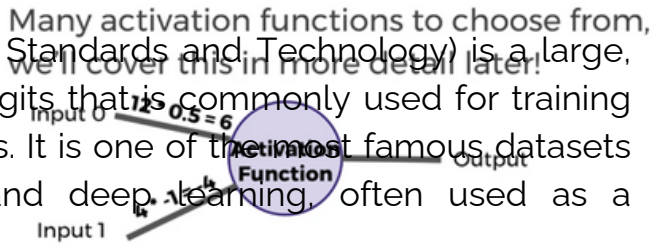
7 Deep Learning for CV

```
# Build a simple neural network model
model = Sequential()
model.add(Dense(32, input_dim=X_train.shape[1],
activation='relu')) # Input and first hidden layer
model.add(Dense(16, activation='relu')) # Second hidden layer
model.add(Dense(1, activation='sigmoid')) # Output layer for
binary classification

# Compile the model with appropriate loss function and
optimizer
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

# Train the model on the training data
model.fit(
 X_train,
 y_train,
 epochs=20,
 batch_size=32,
 validation_split=0.2,
 verbose=1
)

# Evaluate the model on the testing data
loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {accuracy:.4f}")

# Save the trained model to a file
model.save('my_model.h5')
print("Model saved to 'my_model.h5'")
```



7 Deep Learning for CV

MNIST

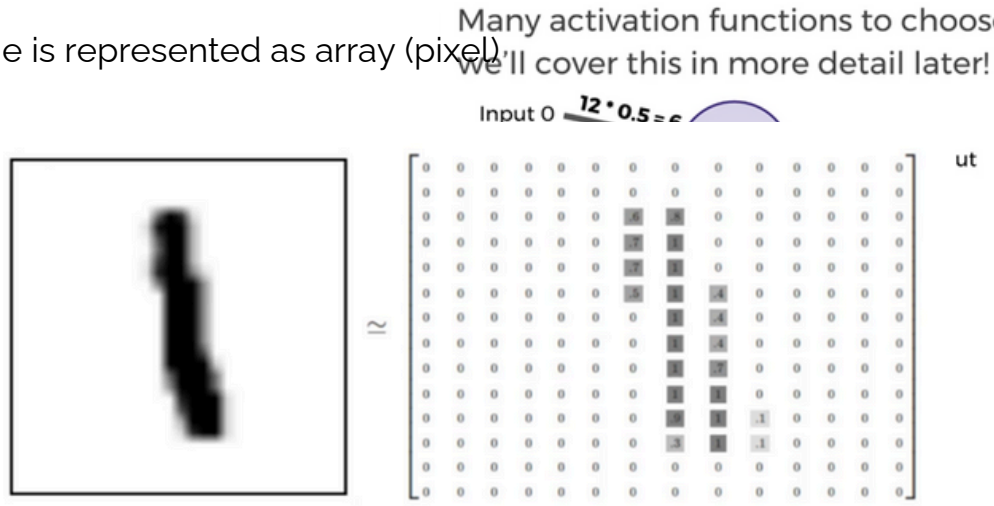


- MNIST (Modified National Institute of Standards and Technology) is a large, well-known dataset of handwritten digits that is commonly used for training and testing image processing systems. It is one of the most famous datasets in the field of machine learning and deep learning, often used as a benchmark for evaluating algorithms.
- Key Characteristics of the MNIST Dataset
  - Dataset Size:
    - The MNIST dataset consists of 70,000 images in total.
  - Training Set: 60,000 images.
    - Test Set: 10,000 images.
  - Image Dimensions:
    - Each image in the MNIST dataset is a grayscale image.
    - The images are 28x28 pixels in size, with each pixel having a value between 0 and 255, where 0 represents black, and 255 represents white.
  - Classes:
    - The dataset includes 10 classes, corresponding to the digits 0 through 9.
  - Labels:
    - Each image is labeled with the digit it represents, so this is a supervised learning problem where the task is to classify the images into one of the 10 digit classes.

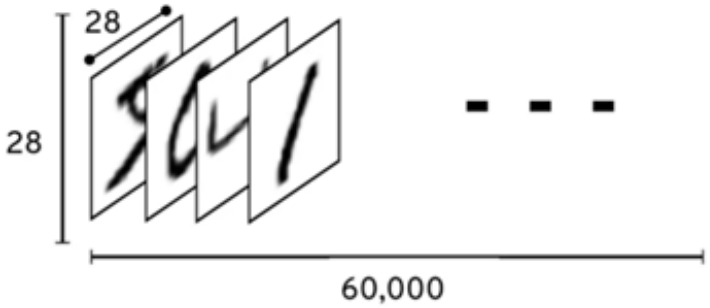
7 Deep Learning for CV

MNIST

- Each image is represented as array (pixel)



- The entire group of image is having 4 dimension - (60000,28,28,1)



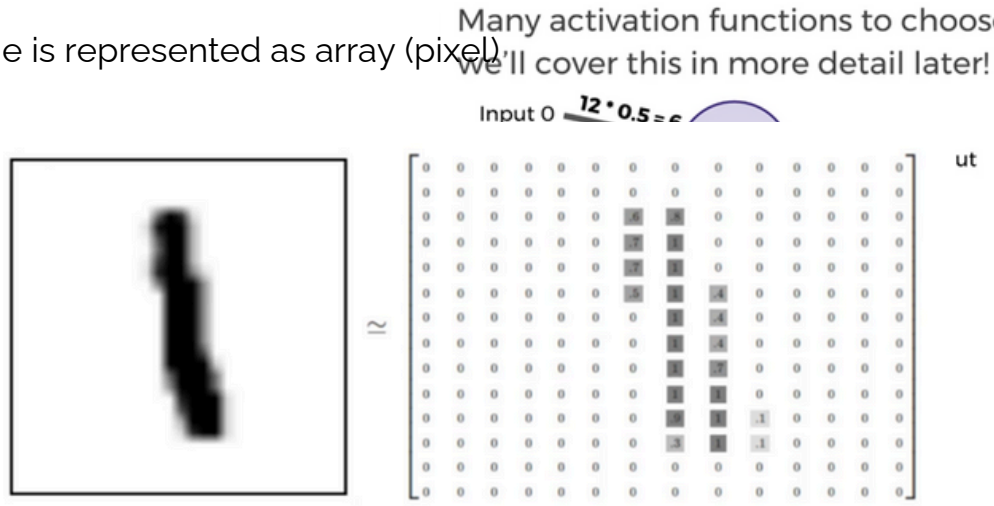
One-Hot Encoding

- One-Hot Encoding is a technique used to convert categorical data into a numerical format that machine learning models can understand. It is particularly useful when dealing with categorical variables, where the data can take on one of a limited number of possible values (categories).
- How One-Hot Encoding Works
  - One-hot encoding transforms each categorical value into a new binary column. Each unique category becomes a new column, and a value of 1 is placed in the column corresponding to the category for that observation, while 0s are placed in all other columns.

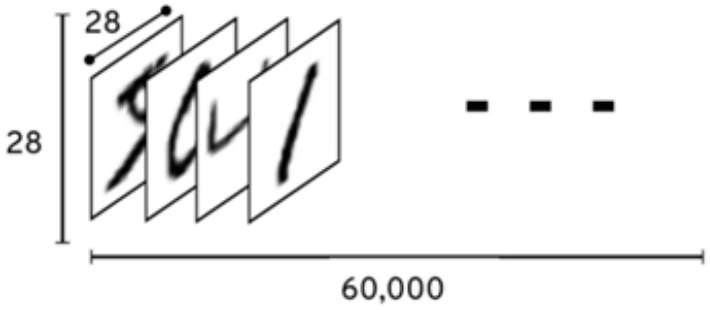
7 Deep Learning for CV

MNIST

- Each image is represented as array (pixel)



- The entire group of image is having 4 dimension - (60000,28,28,1)

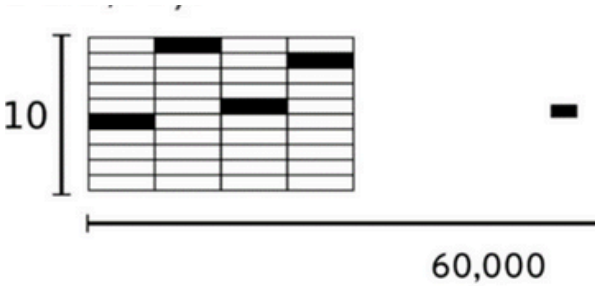


One-Hot Encoding

- One-Hot Encoding is a technique used to convert categorical data into a numerical format that machine learning models can understand. It is particularly useful when dealing with categorical variables, where the data can take on one of a limited number of possible values (categories).
- How One-Hot Encoding Works
  - One-hot encoding transforms each categorical value into a new binary column. Each unique category becomes a new column, and a value of 1 is placed in the column corresponding to the category for that observation, while 0s are placed in all other columns.

7 Deep Learning for CV

- The MNIST dataset contains images of handwritten digits (0-9) and their corresponding labels. Each image is 28x28 pixels, and the label is a digit between 0 and 9.
- The to_categorical function converts each label into a one-hot encoded vector. For example, if the label is 3, it becomes [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]. This vector has 10 elements, where the index corresponding to the label is set to 1 and all other elements are 0.
- As a result, the labels for the training data ends up to be a large 2d array which is (60000,10)




```
array([[0., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 1., 0.]],
```

A Convolutional Neural Network (CNN)

- A Convolutional Neural Network (CNN) is a Deep Learning algorithm that can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image, and be able to differentiate one from the other. The pre-processing required in a CNN is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, CNN have the ability to learn these filters/characteristics.
- The architecture of a CNN is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area.

7 Deep Learning for CV