**Computer Vision & Image Processing**

**22AIE313**

**LAB REPORT**

**SUBMITTED BY**

**Name : S. Abhiram**

**Roll No. AV.EN.U4AIE22140**

**Section: CSE AI-B**

**SUBMITTED TO :**

**Dr. M Srinivas**

**Associate Professor, Amrita School of Computing**

**Amrita Vishwa Vidyapeetham, Amaravati Campus**

**Lab1 - Basics , Croping , Flipping**

**Introduction**

Image processing is a fundamental aspect of computer vision, enabling the manipulation and analysis of visual data for various applications such as medical imaging, object detection, face recognition, and autonomous navigation. OpenCV (Open Source Computer Vision Library) is a widely used open-source library that provides numerous functions for real-time image processing.

The objective of this lab was to understand and implement basic image operations using OpenCV, including:

* Reading and displaying an image
* Cropping a specific region from an image
* Flipping an image in different directions

By performing these tasks, we aimed to gain hands-on experience with image manipulation techniques that are foundational for more complex computer vision applications.

**Methodology**

The lab was conducted using Python and the OpenCV library, following a structured approach to handle and process images. The key steps involved are explained below:

**1. Reading and Displaying an Image**

* The first step involved loading an image using the cv2.imread() function. This function reads an image and stores it as a NumPy array in BGR (Blue, Green, Red) format, which is OpenCV’s default color representation.
* The loaded image was then displayed using cv2.imshow().
* The cv2.waitKey(0) function was used to pause the execution, waiting for a key press before closing the displayed image window with cv2.destroyAllWindows().
* This process ensured that the image was successfully read and displayed without errors.

**2. Cropping an Image**

* Cropping is the process of extracting a specific region of interest (ROI) from an image.
* This was accomplished using **array slicing** in NumPy, where specific pixel coordinates were used to define the region to be extracted.
* The cropped image was displayed to verify its correctness.

**3. Flipping an Image**

Flipping an image involves reversing the pixel arrangement along a specified axis. OpenCV’s cv2.flip() function was used to perform three types of flipping:

1. **Horizontal flip (flipCode=1)** – Mirrors the image from left to right.
2. **Vertical flip (flipCode=0)** – Flips the image upside down.
3. **Both directions (flipCode=-1)** – Flips both horizontally and vertically.

**Basic Drawing**

# import the required packages

import numpy as np

import cv2

# initialize our canvas as a 500x500 pixel image with 3 channels

# (Red, Green, and Blue) with a black background

canvas = np.zeros((500, 500, 3), dtype=np.uint8)

# cv2.imshow('Original Canvas', canvas)

# cv2.waitKey(0)

# initialize the origin values

originY, originX = 0, 0

# get image spatial dimensions

height, width = canvas.shape[:2]

# # draw a green line from the top-left corner of our canvas to the

# # bottom-right

line\_color = (0, 255, 0) # green color

#

# # drawing green line

# # with tuple values of x,y cordinates

cv2.line(canvas, (originX, originY), (width, height), line\_color)

# # displaying image

# cv2.imshow('Canvas with green diagonal line', canvas)

# cv2.waitKey(0)

# draw a 3 pixel thick red line from the top-right corner to the

# bottom-left

line\_color = (0, 0, 255) # red color

thickness = 3 # line thickness

# drawing red line

# with tuple values of x,y cordinates

cv2.line(canvas, (width, originY), (originX, height), line\_color, thickness)

# displaying image

# cv2.imshow('Canvas with new red diagonal line', canvas)

# cv2.waitKey(0)

# draw a blue square, starting at 62x62 and ending at 437x437

line\_color = (255, 0, 0) # blue color

# drawing blue rectangle

cv2.rectangle(canvas, (62, 62), (437, 437), line\_color)

# # displaying image

# cv2.imshow('Canvas with new blue rectangle', canvas)

# cv2.waitKey(0)

# draw a rectangle (white and filled in )

line\_color = (255, 255, 255) # white color

cv2.rectangle(canvas, (150, 150), (350, 350), line\_color, cv2.FILLED)

# # displaying image

# cv2.imshow("Canvas", canvas)

# cv2.waitKey(0)

# re-initialize the canvas as an empty array

canvas = np.zeros((500, 500, 3), dtype='uint8')

# compute the center (x, y)-coordinates of the canvas

(centerY, centerX) = (canvas.shape[0] // 2, canvas.shape[1] // 2)

white = (255, 255, 255)

# # loop over increasing radii, from 25 pixels to 275 pixels

# in 25 pixel increments

for radius in range(0, 275, 25):

# draw a white circle with the current radius size

cv2.circle(canvas, (centerX, centerY), radius, white)

# display image

cv2.imshow("Canvas", canvas)

cv2.waitKey(0)

# re-initialize canvas

canvas = np.zeros((500, 500, 3), dtype="uint8")

# draw 25 random circles

for i in range(0, 25):

# randomly generate a radius size between 5 and 200, generate a

# random color, and then pick a random point on our canvas where

# the circle will be drawn

radius = np.random.randint(5, high=200)

color = np.random.randint(0, high=256, size=(3,)).tolist()

point = np.random.randint(0, high=500, size=(2,))

# draw the random circle on the canvas

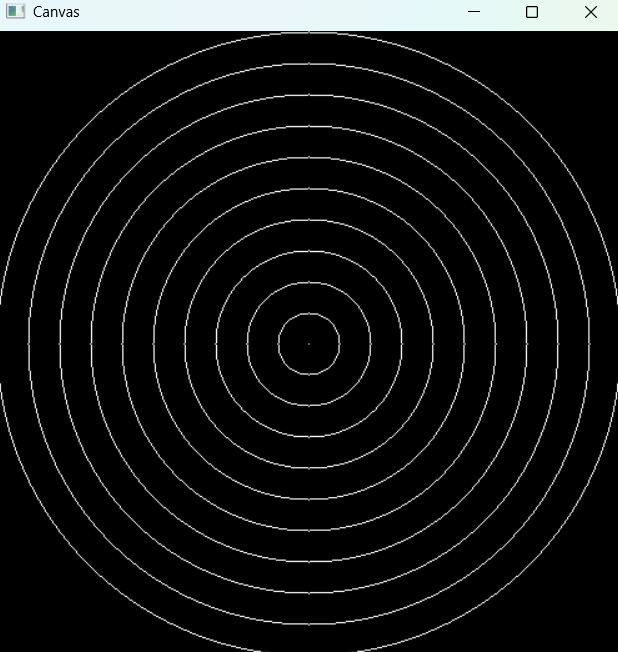
cv2.circle(canvas, tuple(point), radius, color, -1)

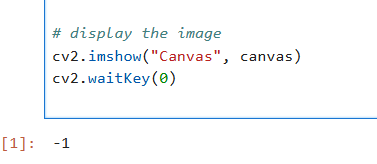
# display the image

cv2.imshow("Canvas", canvas)

cv2.waitKey(0)

Output:





**Image Drawing**

# Import required packages

import cv2

import matplotlib.pyplot as plt

# Correct file path format

image\_path = r"C:\Users\abhir\Downloads\image1.jpg" # Use raw string (r"") or double backslashes

# Load the input image

image = cv2.imread(image\_path)

# Check if image is loaded correctly

if image is None:

print("Error: Image not found! Check the file path.")

else:

# Convert BGR to RGB for proper Matplotlib display

image\_rgb = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

# Display Original Image

plt.imshow(image\_rgb)

plt.axis("off")

plt.title("Original Image")

plt.show()

# Initialize the center coordinates

(centerY, centerX) = (image.shape[0] // 2, image.shape[1] // 2)

# Draw a white circular border

color = (255, 255, 255) # White (OpenCV uses BGR)

radius = round((centerY\*\*2 + centerX\*\*2) \*\* 0.5) # Calculate radius from center to edge

thickness = 10 # Adjust thickness

cv2.circle(image, (centerX, centerY), radius, color, thickness)

# Convert to RGB for display

image\_rgb = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

# Display Image with Circular Border

plt.imshow(image\_rgb)

plt.axis("off")

plt.title("Image with Circular Edge")

plt.show()

# Define eye coordinates

eyes1, eyes2 = (192, 167), (332, 162)

radius = 50

color = (0, 0, 255) # Red

# Draw red circles over the eyes

cv2.circle(image, eyes1, radius, color, -1)

cv2.circle(image, eyes2, radius, color, -1)

# Convert to RGB for display

image\_rgb = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

# Display Image with Red Eyes

plt.imshow(image\_rgb)

plt.axis("off")

plt.title("Image with Covered Eyes")

plt.show()

# Draw a blue rectangle over the mouth

color = (255, 0, 0) # Blue

startX, startY = 180, 235

stopX, stopY = 340, 315

cv2.rectangle(image, (startX, startY), (stopX, stopY), color, -1)

# Convert to RGB for final display

image\_rgb = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

# Display Final Image with Masked Face

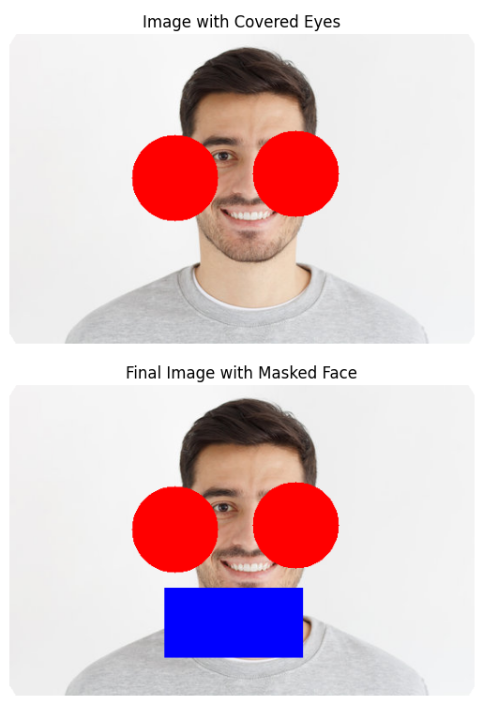
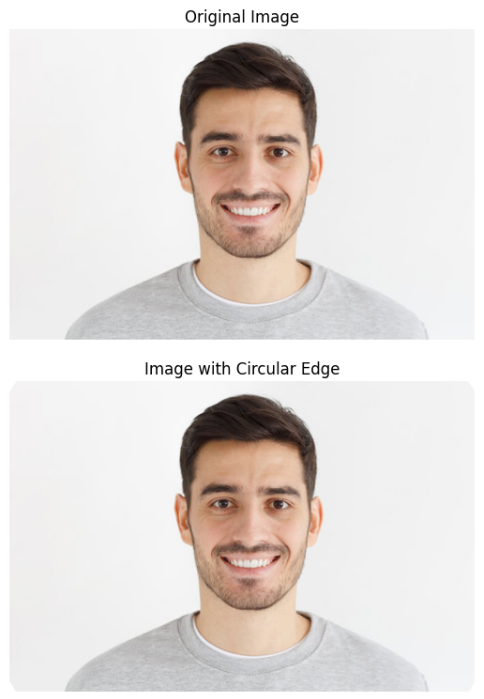
plt.imshow(image\_rgb)

plt.axis("off")

plt.title("Final Image with Masked Face")

plt.show()

**Output:**



**Cropping Images**

import cv2

import matplotlib.pyplot as plt

# ✅ Corrected file path (Use raw string 'r' or double backslashes)

image\_path = r"C:\Users\abhir\Downloads\image1.jpg"

# ✅ Load the input image

image = cv2.imread(image\_path)

# ✅ Check if image is loaded correctly

if image is None:

print("Error: Image not found! Check the file path.")

exit()

# ✅ Resize image for faster processing

image = cv2.resize(image, (800, 600))

image\_rgb = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

# ✅ Display the original image (non-blocking)

plt.imshow(image\_rgb)

plt.axis("off")

plt.title("Original Image")

plt.pause(0.01)

# ✅ Define regions to crop (Y1, X1, Y2, X2)

regions = {

"Star": (142, 244, 285, 395),

"Red": (0, 0, 140, 640),

"Yellow": (143, 0, 282, 640),

"Green": (285, 0, 425, 640)

}

# ✅ Plot all cropped images in one figure

fig, axes = plt.subplots(1, 4, figsize=(15, 5))

for ax, (name, (sy, sx, ey, ex)) in zip(axes, regions.items()):

cropped = image[sy:ey, sx:ex]

cropped\_rgb = cv2.cvtColor(cropped, cv2.COLOR\_BGR2RGB)

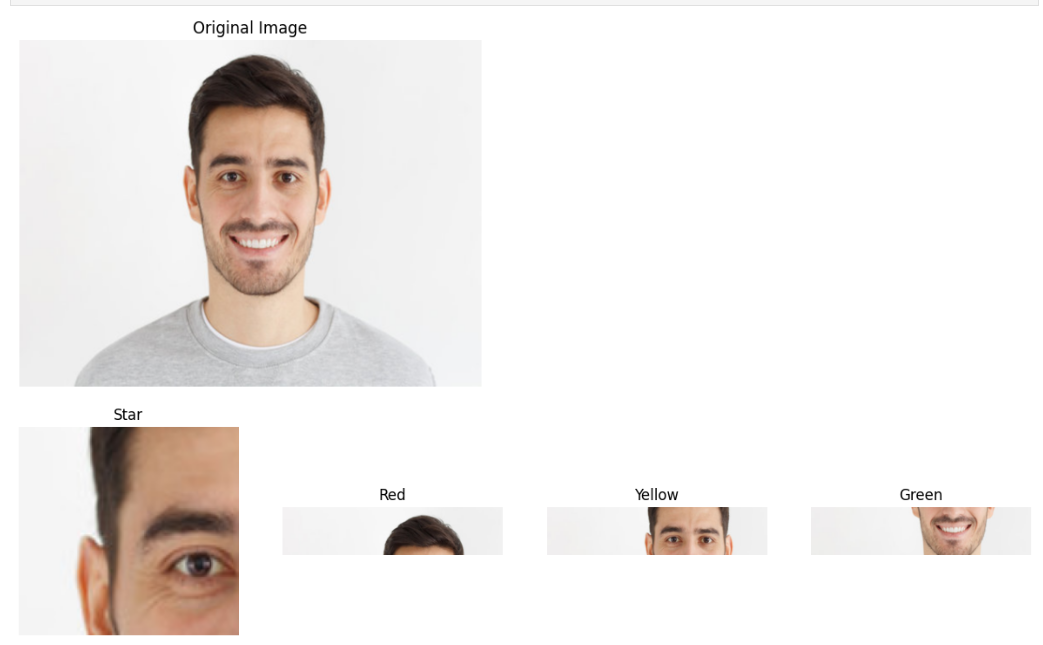
ax.imshow(cropped\_rgb)

ax.set\_title(name)

ax.axis("off")

plt.show()

Output:



**Flipping Images**

# Import required libraries

import cv2

import matplotlib.pyplot as plt

# ✅ Corrected file path

image\_path = r"C:\Users\abhir\Downloads\image1.jpg"

# ✅ Load and check if the image exists

image = cv2.imread(image\_path)

if image is None:

print("Error: Image not found! Check the file path.")

else:

# Convert BGR (OpenCV format) to RGB (Matplotlib format)

image\_rgb = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

# ✅ Display the original image

plt.figure(figsize=(6, 6))

plt.imshow(image\_rgb)

plt.title("Original Image")

plt.axis("off")

plt.show()

# ✅ Flip the image horizontally

flipped\_h = cv2.flip(image, 1)

plt.figure(figsize=(6, 6))

plt.imshow(cv2.cvtColor(flipped\_h, cv2.COLOR\_BGR2RGB))

plt.title("Flipped Horizontally")

plt.axis("off")

plt.show()

# ✅ Flip the image vertically

flipped\_v = cv2.flip(image, 0)

plt.figure(figsize=(6, 6))

plt.imshow(cv2.cvtColor(flipped\_v, cv2.COLOR\_BGR2RGB))

plt.title("Flipped Vertically")

plt.axis("off")

plt.show()

# ✅ Flip the image along both axes

flipped\_hv = cv2.flip(image, -1)

plt.figure(figsize=(6, 6))

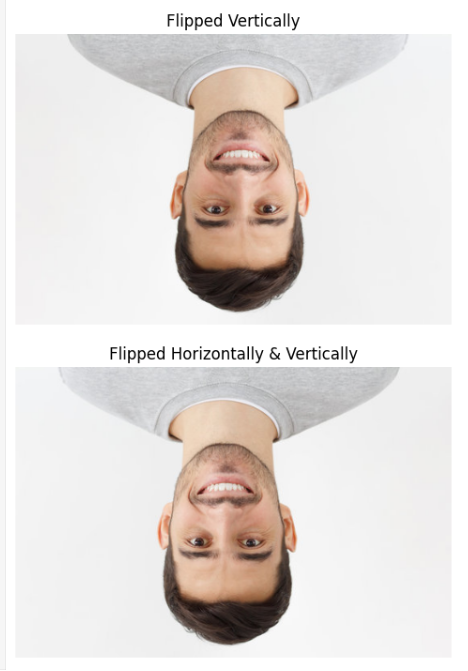
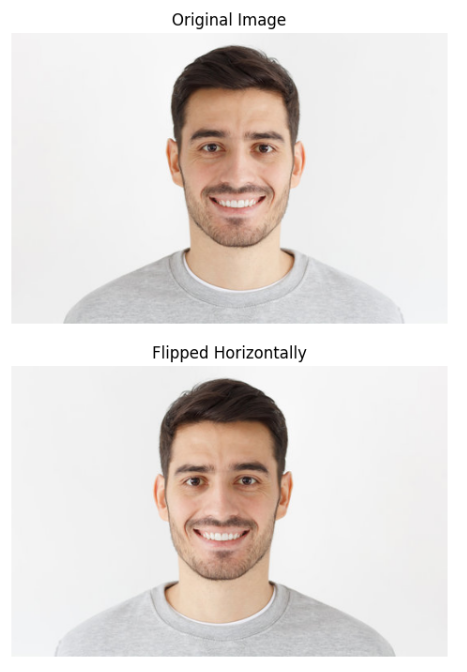
plt.imshow(cv2.cvtColor(flipped\_hv, cv2.COLOR\_BGR2RGB))

plt.title("Flipped Horizontally & Vertically")

plt.axis("off")

plt.show()

**Output:**

****

**Analysis of Results**

After executing the above image processing operations, the following observations were made:

**1. Image Reading and Displaying**

* The image was successfully loaded and displayed.
* The cv2.imread() function returned a valid NumPy array, confirming that the image file was correctly read.
* The use of cv2.imshow() displayed the image properly.

**2. Cropping an Image**

* The cropping function correctly extracted the specified portion of the image.
* By adjusting the coordinates in image[y1:y2, x1:x2], different regions of the image could be selected.
* This method can be extended for tasks such as object detection and segmentation.

**3. Flipping an Image**

* The horizontal flip created a mirror image effect along the vertical axis.
* The vertical flip turned the image upside down.
* Flipping in both directions produced a completely inverted image.
* These transformations were achieved with minimal computational overhead, making them useful for image augmentation in machine learning applications.

**4. Performance Considerations**

* The OpenCV operations executed efficiently in real-time, demonstrating the library’s optimization for image processing tasks.
* The functions used in this lab can be extended to real-world applications such as face recognition, augmented reality, and automated image editing.

**Conclusion**

This lab provided hands-on experience with fundamental image processing techniques using OpenCV. The main takeaways include:

* Successfully reading, displaying, cropping, and flipping images.
* Understanding the representation of images as NumPy arrays and how slicing techniques can be used for modifications.
* Utilizing cv2.flip() for various flipping operations, which is useful in data augmentation and preprocessing for deep learning models.

**Lab 2 – Rotation, Scaling, Translation**

**Introduction**

Image transformations play a crucial role in computer vision, enabling various modifications such as rotation, scaling, and translation. These operations help in pre-processing images for tasks like object detection, pattern recognition, and augmentation in deep learning.

This lab focuses on three fundamental image transformations:

1. Rotation - Rotating an image by a specified angle.
2. Scaling - Resizing an image using different interpolation techniques.
3. Translation - Moving an image to a new location.

Each transformation is implemented using OpenCV and visualized with Matplotlib.

**Methodology**

**1. Rotation**

Objective: Rotate an image by 10 degrees.

Steps:

1. Load the image using cv2.imread().
2. Obtain the image dimensions (height, width).
3. Define a rotation matrix using cv2.getRotationMatrix2D().
4. Apply the rotation using cv2.warpAffine().
5. Convert the image from BGR to RGB for Matplotlib display.
6. Display the rotated image.

**2. Scaling**

Objective: Resize an image using different interpolation techniques.

Steps:

1. Load the image.
2. Resize it to a fixed size (300x300 pixels).
3. Apply Linear Interpolation.
4. Apply Cubic Interpolation.
5. Convert images to RGB format.
6. Display all images for comparison.

**3. Translation**

Objective: Move an image 100 pixels right and 100 pixels down.

Steps:

1. Load the image.
2. Define a translation matrix.
3. Apply translation using cv2.warpAffine().
4. Convert the image from BGR to RGB.
5. Display the original and translated images side by side.

**Rotation**

# Import required libraries

import cv2

import numpy as np

import matplotlib.pyplot as plt

# ✅ Corrected file path

image\_path = r"C:\Users\abhir\Downloads\image1.jpg"

# ✅ Load and check if the image exists

image = cv2.imread(image\_path)

if image is None:

print("Error: Image not found! Check the file path.")

else:

# Get image dimensions

height, width = image.shape[:2]

# ✅ Create the rotation matrix (Rotate by 10 degrees, scale = 1)

matrix = cv2.getRotationMatrix2D((width / 2, height / 2), 10, 1)

# ✅ Apply the transformation

rotated\_image = cv2.warpAffine(image, matrix, (width, height))

# ✅ Convert BGR (OpenCV format) to RGB (Matplotlib format)

rotated\_image\_rgb = cv2.cvtColor(rotated\_image, cv2.COLOR\_BGR2RGB)

# ✅ Display the rotated image using Matplotlib

plt.figure(figsize=(6, 6))

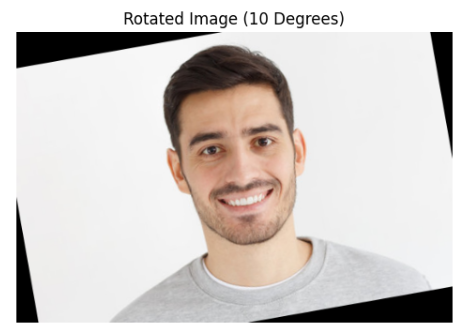
plt.imshow(rotated\_image\_rgb)

plt.title("Rotated Image (10 Degrees)")

plt.axis("off")

plt.show()

**Output:**



**Scaling**

# Import required libraries

import cv2

import numpy as np

import matplotlib.pyplot as plt

# ✅ Corrected file path

image\_path = r"C:\Users\abhir\Downloads\image1.jpg"

# ✅ Load the image

image = cv2.imread(image\_path)

# ✅ Check if the image is loaded correctly

if image is None:

print("Error: Image not found! Check the file path.")

else:

# ✅ Resize image to a fixed size (300x300)

image\_sized = cv2.resize(image, (300, 300))

# ✅ Resizing using Linear Interpolation

image\_re\_linear = cv2.resize(image, None, fx=5.5, fy=5.5, interpolation=cv2.INTER\_LINEAR)

# ✅ Resizing using Cubic Interpolation

image\_re\_cubic = cv2.resize(image, None, fx=5.5, fy=5.5, interpolation=cv2.INTER\_CUBIC)

# ✅ Convert BGR to RGB for correct color display

image\_rgb = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

image\_linear\_rgb = cv2.cvtColor(image\_re\_linear, cv2.COLOR\_BGR2RGB)

image\_cubic\_rgb = cv2.cvtColor(image\_re\_cubic, cv2.COLOR\_BGR2RGB)

# ✅ Display all images using Matplotlib

plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)

plt.imshow(image\_rgb)

plt.title("Original Image")

plt.axis("off")

plt.subplot(1, 3, 2)

plt.imshow(image\_linear\_rgb)

plt.title("Resized - Linear Interpolation")

plt.axis("off")

plt.subplot(1, 3, 3)

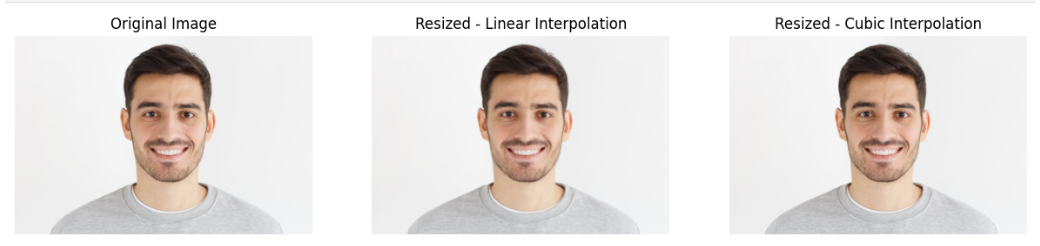
plt.imshow(image\_cubic\_rgb)

plt.title("Resized - Cubic Interpolation")

plt.axis("off")

plt.show()

**Output:**



**Translation**

# Import required libraries

import cv2

import numpy as np

import matplotlib.pyplot as plt

# ✅ Load the image

image\_path = r"C:\Users\abhir\Downloads\image1.jpg" # Update with your image path

image = cv2.imread(image\_path)

# ✅ Check if the image is loaded correctly

if image is None:

print("Error: Image not found! Check the file path.")

else:

# ✅ Define the translation matrix

matrix = np.float32([[1, 0, 100], [0, 1, 100]]) # Moves image 100px right and 100px down

# ✅ Apply the transformation

translated = cv2.warpAffine(image, matrix, (image.shape[1] + 100, image.shape[0] + 100))

# ✅ Convert BGR to RGB for correct color display

image\_rgb = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

translated\_rgb = cv2.cvtColor(translated, cv2.COLOR\_BGR2RGB)

# ✅ Display the original and translated images side by side

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)

plt.imshow(image\_rgb)

plt.title("Original Image")

plt.axis("off")

plt.subplot(1, 2, 2)

plt.imshow(translated\_rgb)

plt.title("Translated Image")

plt.axis("off")

plt.show()

**Output:**



**Analysis of Results**

Rotation:

* The image was successfully rotated 10 degrees counterclockwise.
* The center of the image was maintained as the rotation point.

Scaling:

* The resized images showed differences based on interpolation:
  + Linear interpolation resulted in a smoother appearance with some detail loss.
  + Cubic interpolation retained more detail but introduced minor artifacts.

Translation:

* The image was shifted 100 pixels right and 100 pixels down without distortion.
* Empty space was created where the original pixels were moved.

**Conclusion**

In this lab, we explored fundamental image transformations using OpenCV:

* Rotation for orientation adjustment.
* Scaling for size modification using interpolation methods.
* Translation for shifting an image’s position.

These transformations are essential for data augmentation, image preprocessing, and geometric modifications in various computer vision applications. By applying these techniques, we can better prepare images for machine learning and deep learning tasks.

**Lab3**

## **1. Face and Eye Detection Using Images (Haar Cascades)**

### Introduction

This code detects faces and eyes in images using Haar Cascade classifiers. The Haar Cascade method is based on machine learning object detection and is particularly useful for real-time applications.

### Code Analysis

* The code reads an image, converts it to grayscale, and applies Haar cascades for face and eye detection.
* The detected faces are enclosed in blue rectangles, while the detected eyes are marked in red.
* The model uses cv2.CascadeClassifier to load pre-trained XML classifiers and the detectMultiScale() function for detection.
* Results are displayed using cv2\_imshow() in Google Colab.

import cv2

import os

from google.colab.patches import cv2\_imshow # Import cv2\_imshow for Colab

def detect\_face\_and\_eyes(image\_path):

img = cv2.imread(image\_path)

if img is None:

raise ValueError("Failed to load image. Check the path.")

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

# Load Haar cascades

face\_cascade = cv2.CaswcadeClassifier(cv2.data.haarcascades + 'haarcascade\_frontalface\_default.xml')

eye\_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade\_eye.xml')

faces = face\_cascade.detectMultiScale(gray, scaleFactor=1.3, minNeighbors=5, minSize=(30, 30))

for (x, y, w, h) in faces:

cv2.rectangle(img, (x, y), (x + w, y + h), (255, 0, 0), 3)

roi\_gray = gray[y:y + h, x:x + w]

roi\_color = img[y:y + h, x:x + w]

eyes = eye\_cascade.detectMultiScale(roi\_gray, scaleFactor=1.1, minNeighbors=5, minSize=(20, 20))

for (ex, ey, ew, eh) in eyes:

cv2.rectangle(roi\_color, (ex, ey), (ex + ew, ey + eh), (0, 0, 255), 2)

cv2\_imshow(img) # Use cv2\_imshow instead of cv2.imshow()

cv2.waitKey(0)

cv2.destroyAllWindows()

# Provide image path

detect\_face\_and\_eyes("/content/images.jpg")

**Result:**



### **Conclusion**

Haar Cascades provide an efficient way to detect faces and eyes in an image. However, they may struggle with varying lighting conditions and angles compared to deep learning-based approaches.

## **2. Face Detection Using Deep Neural Networks (DNN)**

### **Introduction**

This code implements face detection using a deep learning-based model with OpenCV’s DNN module. The method utilizes a Caffe-based model trained for face detection.

### **Analysis**

* It loads a pre-trained deep learning model (res10\_300x300\_ssd\_iter\_140000.caffemodel) and its configuration file (deploy.prototxt.txt).
* The model processes an image and detects faces by applying a confidence threshold.
* Detected faces are highlighted with blue rectangles, and confidence scores are displayed on the screen.
* The function detectFaces() extracts face coordinates, resizes the image, and uses blob representation for better accuracy

import numpy as np

import sys

import cv2

from imutils.video import VideoStream

import imutils

import time

# Paths to prototxt file and Caffe model

prototxtPath = "deploy.prototxt.txt"

caffemodelPath = "res10\_300x300\_ssd\_iter\_140000.caffemodel"

conf = 0.30 # Confidence level threshold

thickness = 2 # Thickness of rectangle

blue = (247, 173, 62) # Color in BGR

white = (255, 255, 255) # Color in BGR

font = cv2.FONT\_HERSHEY\_SIMPLEX # Font style

meanValues = (104.0, 177.0, 124.0) # RGB mean values from ImageNet training set

# Load model

net = cv2.dnn.readNetFromCaffe(prototxtPath, caffemodelPath)

def drawRectangle(image, color, t):

(x, y, x1, y1) = t

h = y1 - y

w = x1 - x

barLength = int(h / 8)

cv2.rectangle(image, (x, y-barLength), (x+w, y), color, -1)

cv2.rectangle(image, (x, y-barLength), (x+w, y), color, thickness)

cv2.rectangle(image, (x, y), (x1, y1), color, thickness)

return image

# Changes font scale as a function of face box size

def changeFontScale(h, fontScale):

baseHeight = 108 # Height of model image

fontScale = h/108 \* fontScale

return fontScale

def detectFaces(image):

h, w, \_ = image.shape

resizedImage = cv2.resize(image, (300, 300))

blob = cv2.dnn.blobFromImage(resizedImage, 1.0, (300, 300), meanValues)

net.setInput(blob)

faces = net.forward()

for i in range(0, faces.shape[2]):

confidence = faces[0, 0, i, 2]

if confidence > conf:

box = faces[0, 0, i, 3:7] \* np.array([w, h, w, h])

(x, y, x1, y1) = box.astype("int")

fontScale = changeFontScale(y1-y, 0.4)

image = drawRectangle(image, blue, (x, y, x1, y1))

# Display confidence level in %

text = "{:0.2f}%".format(confidence \* 100)

textY = y - 2

if (textY - 2 < 20): textY = y + 20

cv2.putText(image, text, (x, textY), font, fontScale, white, 1)

return image

def useWebcam():

vs = VideoStream(src=0).start()

time.sleep(2.0)

while True:

frame = vs.read()

frame = imutils.resize(frame, width=400)

frame = detectFaces(frame)

cv2.imshow("Face Detection", frame)

if cv2.waitKey(1) > 0:

break

cv2.destroyAllWindows()

vs.stop()

def useImage():

image = cv2.imread(sys.argv[1]) # Read image

image = detectFaces(image)

cv2.imshow("Face Detection", image)

cv2.waitKey(0)

def main():

if len(sys.argv) == 1:

useWebcam()

elif len(sys.argv) == 2:

useImage()

else:

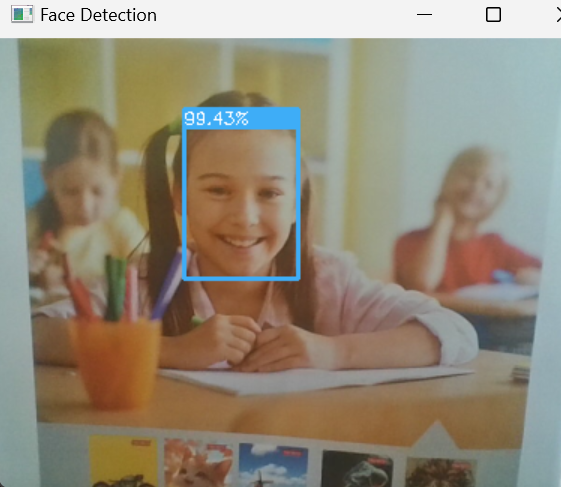
print("Usage: python face-detect-dnn.py [optional.jpg]")

exit()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Result:**



### **Conclusion**

Deep learning models outperform traditional Haar cascades by providing higher accuracy and better generalization. However, they require more computational power.

## **3. Face and Eye Detection Using Video (Haar Cascades)**

### **Introduction**

This code extends Haar Cascade-based face and eye detection to real-time video streams, allowing face and eye detection from a webcam feed.

### **Analysis**

* It captures live video using cv2.VideoCapture(0).
* The frames are converted to grayscale and passed to Haar cascades for face and eye detection.
* Detected features are marked with colored rectangles, and results are displayed continuously.
* The loop continues until the user presses ‘q’ to quit.

import cv2

face\_cascade = cv2.CascadeClassifier('Haarcascades/haarcascade\_frontalface\_default.xml') eye\_cascade = cv2.CascadeClassifier('Haarcascades/haarcascade\_eye.xml')

def detect(gray, frame):

faces = face\_cascade.detectMultiScale(gray, 1.3, 5)

for (x, y, w, h) in faces:

cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 0, 0), 2)

roi gray = gray[y:y+h, x:x+w]

roi\_color = frame[y:y+h, x:x+w]

eyes = eye\_cascade.detectMultiScale(roi\_gray, 1.1, 3)

for (ex, ey, ew, eh) in eyes:

cv2.rectangle(roi\_color, (ex, ey), (ex+ew, ey+eh), (0, 255, 0), 2)

return frame

video\_capture = cv2.VideoCapture(0)

while True:

frame = video\_capture.read()

gray = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)

canvas = detect(gray, frame) cv2.imshow('Video', canvas)

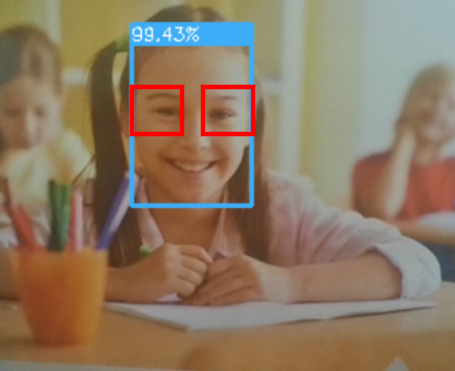
if cv2.waitKey(1) & 0xFF == ord('q'):

break

video\_capture.release()

cv2.destroyAllWindows()

**Result:**



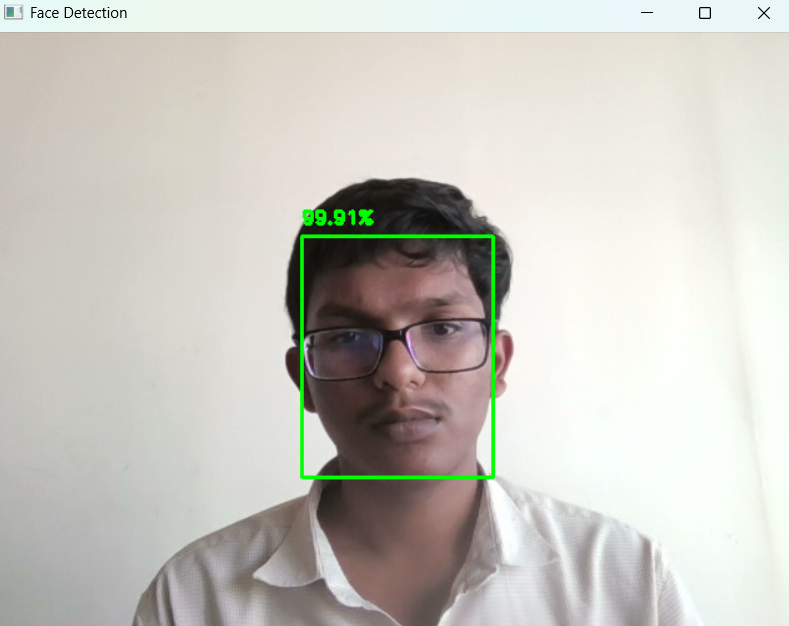
**Conclusion**

While Haar cascades are computationally efficient for real-time applications, they may fail to detect faces under poor lighting or from different angles. Deep learning-based approaches can provide more robustness.

**Face Detection using OpenCV's DNN (Deep Neural Network) module**

import cv2  
import numpy as np  
import os  
import sys  
  
*# Define correct paths*BASE\_DIR = r"C:\Users\abhir\PycharmProjects\web\_development"  
  
prototxtPath = os.path.join(BASE\_DIR, "deploy.prototxt") *# Ensure no .txt extension*caffemodelPath = os.path.join(BASE\_DIR, "res10\_300x300\_ssd\_iter\_140000.caffemodel")  
  
*# Check if the files exist*if not os.path.exists(prototxtPath):  
 print(f"❌ Error: File not found - {prototxtPath}")  
 sys.exit(1)  
  
if not os.path.exists(caffemodelPath):  
 print(f"❌ Error: File not found - {caffemodelPath}")  
 sys.exit(1)  
  
print("✅ Model files loaded successfully!")  
  
*# Load the pre-trained face detection model*net = cv2.dnn.readNetFromCaffe(prototxtPath, caffemodelPath)  
  
*# Start webcam*cap = cv2.VideoCapture(0)  
  
while True:  
 ret, frame = cap.read()  
 if not ret:  
 print("❌ Failed to grab frame. Check your webcam.")  
 break  
  
 h, w = frame.shape[:2]  
  
 *# Convert frame to a blob for deep learning model* blob = cv2.dnn.blobFromImage(cv2.resize(frame, (300, 300)), 1.0, (300, 300), (104.0, 177.0, 123.0))  
 net.setInput(blob)  
 detections = net.forward()  
  
 *# Loop through detections and draw bounding boxes* for i in range(detections.shape[2]):  
 confidence = detections[0, 0, i, 2]  
  
 if confidence > 0.5:  
 box = detections[0, 0, i, 3:7] \* np.array([w, h, w, h])  
 (x, y, x1, y1) = box.astype("int")  
  
 cv2.rectangle(frame, (x, y), (x1, y1), (0, 255, 0), 2)  
 text = f"{confidence \* 100:.2f}%"  
 cv2.putText(frame, text, (x, y - 10), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (0, 255, 0), 2)  
  
 *# Show frame* cv2.imshow("Face Detection", frame)  
  
 if cv2.waitKey(1) & 0xFF == ord('q'):  
 break  
  
cap.release()  
cv2.destroyAllWindows()

**Output**:



## **5. Pedestrian Detection Using Haar Cascades**

### **Introduction**

This code detects full-body pedestrians in video footage using the Haar cascade method. It is useful for applications such as surveillance and autonomous driving.

### **Analysis**

* The script loads a pre-trained Haar cascade (haarcascade\_fullbody.xml) to detect pedestrians.
* It processes frames from a video file, converts them to grayscale, and applies the cascade classifier.
* Detected pedestrians are marked with yellow rectangles.
* The video feed is continuously displayed until the user presses the Enter key.

import cv2

import numpy as np

# Create our body classifier

body\_classifier = cv2.CascadeClassifier('Haarcascades\haarcascade\_fullbody.xml')

# Initiate video capture for video file

cap = cv2.VideoCapture('image\_examples/walking.avi')

# Loop once video is successfully loaded

while cap.isOpened():

# Read first frame

ret, frame = cap.read()

#frame = cv2.resize(frame, None,fx=0.5, fy=0.5, interpolation = cv2.INTER\_LINEAR)

gray = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)

# Pass frame to our body classifier

bodies = body\_classifier.detectMultiScale(gray, 1.2, 3)

# Extract bounding boxes for any bodies identified

for (x,y,w,h) in bodies:

cv2.rectangle(frame, (x, y), (x+w, y+h), (0, 255, 255), 2)

cv2.imshow('Pedestrians', frame)

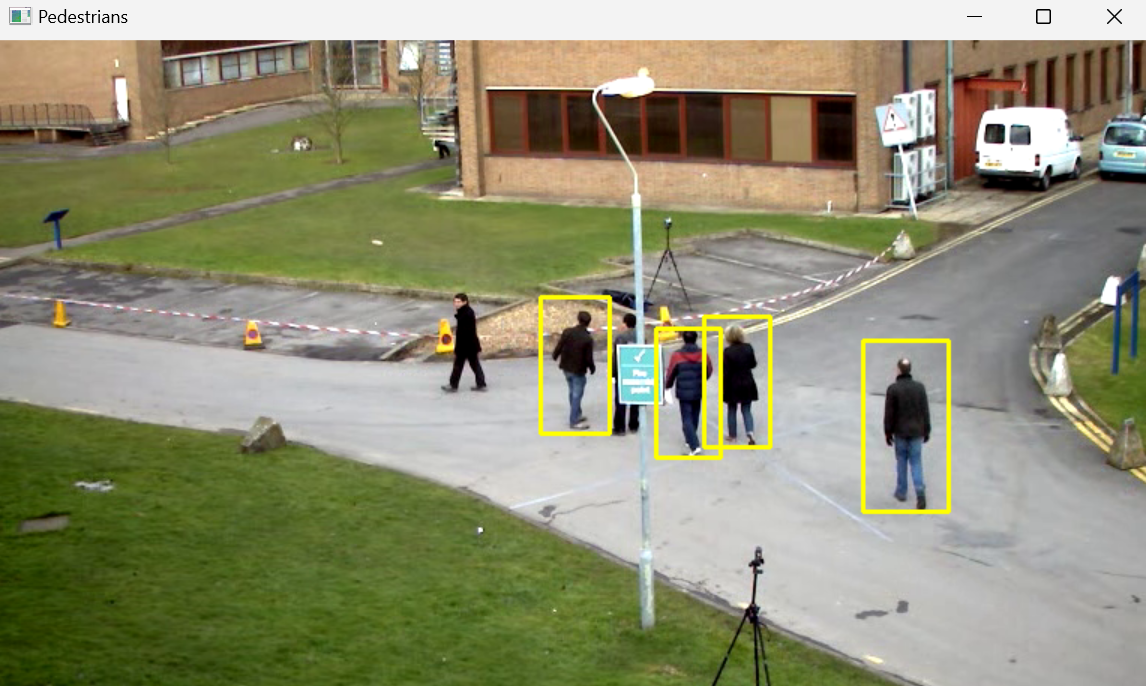
if cv2.waitKey(1) == 13: #13 is the Enter Key

break

cap.release()

cv2.destroyAllWindows()

Result:



### **Conclusion**

Haar cascades provide a simple method for pedestrian detection but may fail in crowded scenarios. More advanced methods like YOLO or Faster R-CNN offer better accuracy and performance.

**Lab4: Face Alignment System Analysis**

**Introduction**

The code you've shared implements a face alignment system that processes facial images to correct orientation based on eye positions. Face alignment is a crucial preprocessing step in many facial recognition systems as it normalizes face images to a canonical pose, making subsequent recognition tasks more accurate and reliable. By aligning faces based on eye positions, the system compensates for head rotation in the image plane, ensuring that facial features appear in consistent positions across different images of the same or different individuals.

**Methodology**

The system employs a methodical approach consisting of several key components:

1. **Face Detection**: Utilizes OpenCV's Haar Cascade classifier (haarcascade\_frontalface\_default.xml) to locate the face within the input image. The system extracts the facial region for further processing.
2. **Eye Detection**: Employs another Haar Cascade classifier (haarcascade\_eye.xml) to detect eyes within the facial region. The system selects the two largest eye regions to reduce false positives.
3. **Eye Position Analysis**: Determines the left and right eyes based on horizontal position, then calculates their centers to establish reference points for alignment.
4. **Rotation Angle Calculation**: Uses trigonometric principles to:
   * Create a triangle between the eye centers and a third reference point
   * Calculate the rotation angle needed to horizontally align the eyes
   * Determine the direction of rotation (clockwise or counterclockwise)
5. **Image Transformation**: Rotates the original image using PIL (Python Imaging Library) based on the calculated angle to produce the aligned face.
6. **Error Handling**: Implements robust error handling for scenarios such as:
   * No face detected in the image
   * Less than two eyes detected
   * Missing image files
   * Improperly installed OpenCV

**Code**

import os

import cv2

import math

import numpy as np

import matplotlib.pyplot as plt

from PIL import Image

# Correct file path format

image\_path = r"C:\Users\abhir\Downloads\image1.jpg"

# Load Haar cascade classifiers

opencv\_home = cv2.\_\_file\_\_

folders = opencv\_home.split(os.path.sep)[:-1]

path = "/".join(folders)

face\_cascade\_path = os.path.join(path, "data/haarcascade\_frontalface\_default.xml")

eye\_cascade\_path = os.path.join(path, "data/haarcascade\_eye.xml")

# Check if OpenCV is installed properly

if not os.path.isfile(face\_cascade\_path) or not os.path.isfile(eye\_cascade\_path):

raise ValueError("OpenCV is not installed properly. Install it with 'pip install opencv-python'.")

face\_detector = cv2.CascadeClassifier(face\_cascade\_path)

eye\_detector = cv2.CascadeClassifier(eye\_cascade\_path)

# Function to calculate Euclidean distance

def euclidean\_distance(pt1, pt2):

return math.sqrt((pt2[0] - pt1[0]) \*\* 2 + (pt2[1] - pt1[1]) \*\* 2)

# Face & Eye Alignment Function

def align\_face(image):

img\_gray = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

faces = face\_detector.detectMultiScale(img\_gray, scaleFactor=1.1, minNeighbors=5)

if len(faces) == 0:

print("❌ No face detected!")

return image

# Get first detected face

x, y, w, h = faces[0]

face\_roi = img\_gray[y:y + h, x:x + w]

eyes = eye\_detector.detectMultiScale(face\_roi)

if len(eyes) < 2:

print("⚠️ Less than two eyes detected! Returning original image.")

return image

# Sort detected eyes (left-right order)

eyes = sorted(eyes, key=lambda eye: eye[0])

# Get eye centers

left\_eye\_center = (x + eyes[0][0] + eyes[0][2] // 2, y + eyes[0][1] + eyes[0][3] // 2)

right\_eye\_center = (x + eyes[1][0] + eyes[1][2] // 2, y + eyes[1][1] + eyes[1][3] // 2)

# Calculate angle of rotation

dx = right\_eye\_center[0] - left\_eye\_center[0]

dy = right\_eye\_center[1] - left\_eye\_center[1]

angle = math.degrees(math.atan2(dy, dx))

# Get rotation matrix

center = (image.shape[1] // 2, image.shape[0] // 2)

rotation\_matrix = cv2.getRotationMatrix2D(center, angle, scale=1)

# Rotate image

aligned\_image = cv2.warpAffine(image, rotation\_matrix, (image.shape[1], image.shape[0]))

# Draw landmarks for verification

output\_image = aligned\_image.copy()

cv2.circle(output\_image, left\_eye\_center, 5, (255, 0, 0), -1) # Left eye (blue)

cv2.circle(output\_image, right\_eye\_center, 5, (0, 255, 0), -1) # Right eye (green)

cv2.rectangle(output\_image, (x, y), (x + w, y + h), (0, 0, 255), 2) # Face box

return aligned\_image, output\_image

# Load image

image = cv2.imread(image\_path)

if image is None:

raise FileNotFoundError(f"❌ Error: Image '{image\_path}' not found!")

# Align face

aligned\_image, debug\_image = align\_face(image)

# Show original, debug, and aligned images

fig, ax = plt.subplots(1, 3, figsize=(15, 5))

ax[0].imshow(cv2.cvtColor(image, cv2.COLOR\_BGR2RGB))

ax[0].set\_title("Original Image")

ax[0].axis("off")

ax[1].imshow(cv2.cvtColor(debug\_image, cv2.COLOR\_BGR2RGB))

ax[1].set\_title("Debug Image (Face & Eyes)")

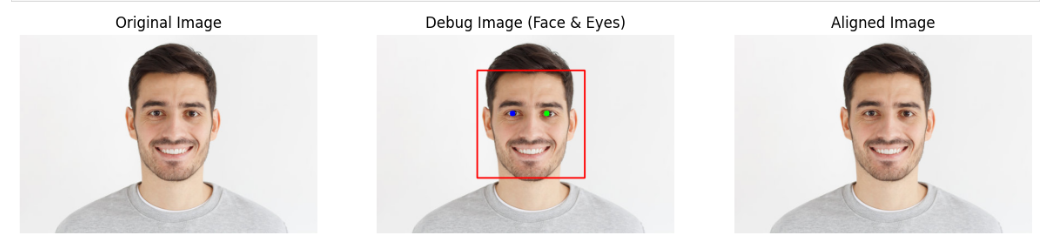
ax[1].axis("off")

ax[2].imshow(cv2.cvtColor(aligned\_image, cv2.COLOR\_BGR2RGB))

ax[2].set\_title("Aligned Image")

ax[2].axis("off")

**Output:**



**Conclusion**

The face alignment system demonstrates the application of computer vision techniques and geometric principles to solve the practical problem of normalizing facial orientations. This preprocessing step is essential for building effective facial recognition systems as it significantly reduces the variability in facial appearances caused by head rotation.

The system's strengths include its straightforward implementation, use of widely available libraries, and robust error handling. However, it does have limitations:

1. The use of Haar Cascade classifiers, which can be sensitive to lighting conditions and occlusions
2. Potential issues with extreme head rotations or profile views
3. Dependency on accurate eye detection

Future improvements could include:

* Implementing more advanced face and landmark detection methods using deep learning
* Adding additional normalization parameters such as scale and translation
* Supporting alignment for profile views and more extreme head poses
* Incorporating face alignment as part of a complete facial recognition pipeline

Overall, this implementation provides a solid foundation for face preprocessing in biometric systems, security applications, and other facial analysis tasks.

**Lab 5**

This code explores edge detection using Convolutional Neural Networks (CNNs). Edge detection is a computer vision task involving identifying boundaries between objects or regions in an image. Traditionally, hand-crafted filters like the Sobel operator were used, but CNNs have emerged as powerful tools for automatic feature extraction.

This code takes a step-by-step approach to understanding and implementing edge detection with CNNs. It begins with an introduction to convolution operations and kernels. Convolution involves sliding a kernel over an image to extract local features.

The code then introduces machine learning techniques to learn an optimal kernel for edge detection. This allows the CNN to adapt to image characteristics and improve accuracy. The Sobel operator is used as an example to illustrate concepts, both traditionally and within Keras.

The code progresses to build more complex models, including multi-convolution filters. These models combine features from different convolutions to capture a richer representation of edges. It also explores intermediate features (activations) generated by the CNN, providing insights into network processing.

Overall, this code offers a practical exploration of edge detection with CNNs. It provides a solid understanding of the fundamental concepts and techniques. By the end of the code, readers can implement their own CNN-based edge detection models.

import numpy as np

import cv2

import matplotlib.pyplot as plt

from tensorflow.keras import models, layers, losses, activations, regularizers, metrics

import tensorflow.keras.backend as K

import seaborn as sns

import tensorview as tv

matterhornGray = cv2.imread('assets/Matterhorn\_1024.JPG', cv2.IMREAD\_GRAYSCALE)

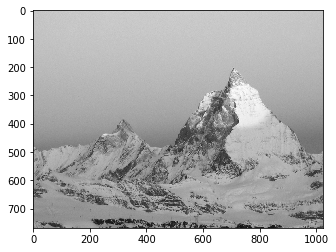
plt.imshow(matterhornGray, cmap='gray')

testImageHeight = matterhornGray.shape[0]

testImageWidth = matterhornGray.shape[1]

imageNChannels = 1

matterhornGray.shape



Naive convolution algorithm

# Sobel operator for horizontal edge detection

Kx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])

# Sobel operator for vertical edge detection

Ky = Kx.T

convolvedShape = (testImageHeight - 2, testImageWidth - 2)

matterhornEdges1x = np.zeros(convolvedShape) # Horiz

matterhornEdges1y = np.zeros(convolvedShape) # Vertical

matterhornEdges1 = np.zeros(convolvedShape) # Combined

for i in range(1, testImageHeight-1):

for j in range(1, testImageWidth-1):

window = matterhornGray[i-1:i+2, j-1:j+2]

a = matterhornEdges1x[i-1, j-1] = np.maximum(np.sum(np.multiply(window, Kx)), 0)

b = matterhornEdges1y[i-1, j-1] = np.maximum(np.sum(np.multiply(window, Ky)), 0)

matterhornEdges1[i-1, j-1] = np.sqrt(a\*\*2 + b\*\*2)

plt.figure(figsize=(20,7))

plt.subplot(1, 3, 1)

plt.imshow(matterhornEdges1x, cmap='gray');

plt.title('Horizontal')

plt.subplot(1, 3, 2)

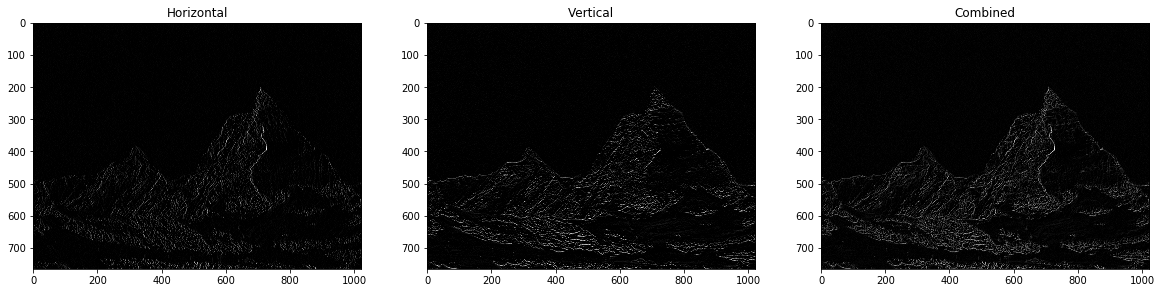
plt.imshow(matterhornEdges1y, cmap='gray');

plt.title('Vertical')

plt.subplot(1, 3, 3)

plt.imshow(matterhornEdges1, cmap='gray');

plt.title('Combined');



Sobel filter with OpenCV

forestGray = cv2.imread('assets/Forest\_2048.jpg', cv2.IMREAD\_GRAYSCALE)

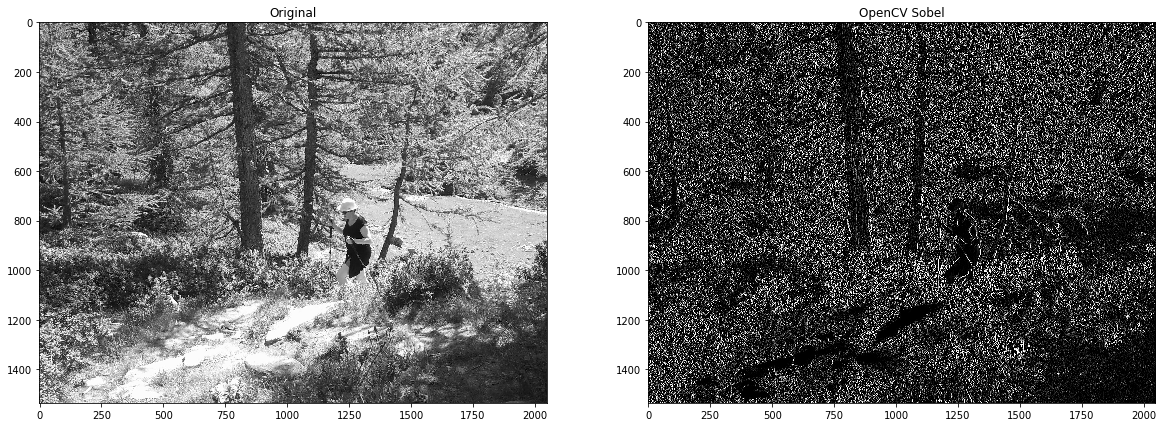
forestEdges2 = cv2.Sobel(forestGray, cv2.CV\_8U, 1, 0, ksize=3)

fig, axes = plt.subplots(1, 2, figsize=(20, 8))

for ax, im, title in zip(axes, [forestGray, forestEdges2], ['Original', 'OpenCV Sobel']):

ax.set\_title(title)

ax.imshow(im, cmap='gray');

Sobel filter in Keras

model0 = models.Sequential([

# 2D convolutions

layers.Conv2D(1, (3, 3), activation=activations.relu,

input\_shape=(None, None, imageNChannels), name='conv0\_0', use\_bias=False,

weights=Kx.reshape(1,3,3,1,1)),

layers.Reshape((-1, 1))

], 'model0')

testImage = matterhornGray.reshape(1, testImageHeight, testImageWidth, 1) / 255

testEst0 = model0.predict(testImage)

testEstEdges0 = (testEst0.reshape(testImageHeight-2, testImageWidth-2)) \* 255

plt.figure(figsize=(8, 8))

plt.imshow(testEstEdges0, cmap='gray');

plt.title('Test estimate');



plotHeatMap(np.array(model0.get\_weights()).reshape(3, 3), vmax=3)

Learn a convolutionnal neural net (CNN) as a filter

trainHeight, trainWidth = 64, 64

inputBatchShape = (-1, trainHeight, trainWidth, imageNChannels)

trainImage = forestGray.reshape(inputBatchShape) / 255.

# Get the output of the fitler using model0 (filter with predefined weights)

trainImageHeight, trainImageWidth = forestGray.shape[0], forestGray.shape[1]

trainEst0 = model0.predict(forestGray.reshape(1, trainImageHeight, trainImageWidth, 1) / 255.)

trainEstEdges0 = np.pad((trainEst0.reshape(trainImageHeight-2, trainImageWidth-2)) \* 255., (1,1), mode='constant')

# Reshape into sub-images as "Labels" for the trainer

outPutBatchShape = (-1, (trainHeight-2) \* (trainWidth-2), imageNChannels)

trainLabels = trainEstEdges0.reshape(inputBatchShape)

trainLabels = trainLabels[:, 1:-1, 1:-1,:].reshape(outPutBatchShape) / 255.

trainImage.shape, trainLabels.shape

nEpochs = 4000

batchSize = 128

model1 = models.Sequential([

layers.Conv2D(1, (3, 3), activation=activations.linear, # 2D convolution

input\_shape=(None, None, imageNChannels), use\_bias=True, name='Conv2D',

kernel\_regularizer=regularizers.l1(0.001),

bias\_regularizer=regularizers.l1(0.01)

),

layers.LeakyReLU(),

layers.Flatten()

], 'model1')

model1.compile(optimizer='adam',

loss=losses.mse,

metrics=[metrics.mse, metrics.kullback\_leibler\_divergence])

tv\_plot = tv.train.PlotMetricsOnEpoch(metrics\_name=metricNames,

cell\_size=(6,6), columns=3, iter\_num=nEpochs, wait\_num=10)

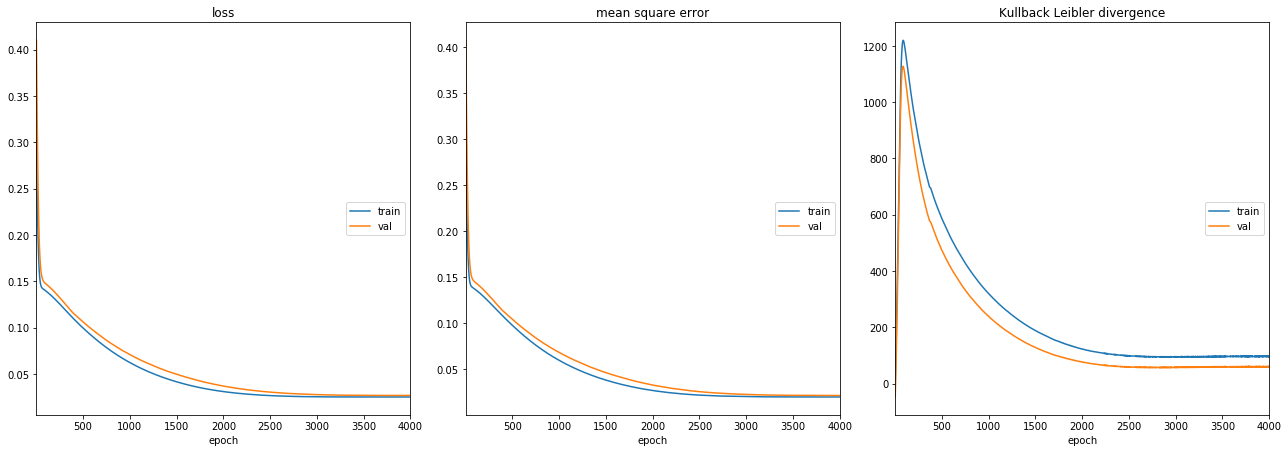
hist1 = model1.fit(trainImage, trainLabels,

epochs=nEpochs, batch\_size=batchSize,

validation\_split=0.7,

verbose=0,

callbacks=[tv\_plot])



weights1 = model1.get\_weights()

fig, axes = plt.subplots(1, 2, figsize=(8, 3), sharey=True)

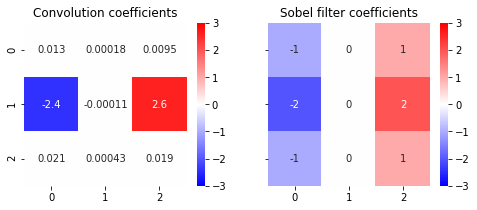
plotHeatMap(np.array(weights1[0]).reshape(3, 3)[:,:], title='Convolution coefficients', vmax=3, ax=axes[0])

plotHeatMap(np.array(model0.get\_weights()).reshape(3, 3), title='Target sobel filter coefficients',

vmax=3, ax=axes[1])

print('Convolution bias = %.3g' % weights1[1])

Convolution bias = 0.000259



Test the model

testImage = matterhornGray.reshape(1, testImageHeight, testImageWidth, 1) / 255

testEst = model1.predict(testImage)

testEstEdges = np.clip((testEst.reshape(testImageHeight-2, testImageWidth-2)) \* 255, 0, 255)

eval1 = model1.evaluate(testImage, testEstEdges0.reshape(1, -1, 1) / 255., verbose=0)

print("Test image : loss (MSE + regularization) = %.3f, Mean Sq. Error = %.3f" % (eval1[0], eval1[1]))

plt.figure(figsize=(14, 8))

plt.subplot(1,2,1)

plt.imshow(testEstEdges, cmap='gray');

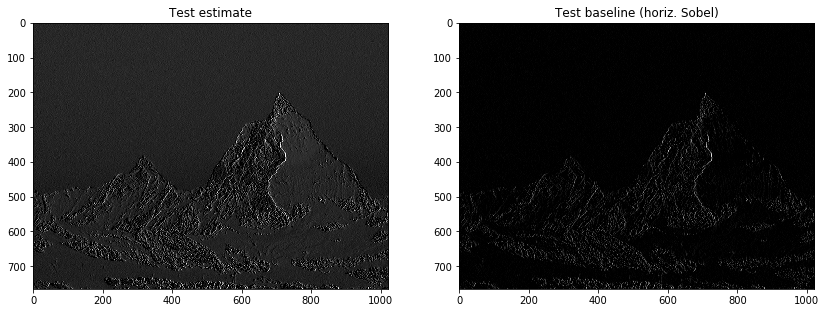
plt.title('Test estimate');

plt.subplot(1,2,2)

plt.imshow(testEstEdges0, cmap='gray');

plt.title('Test baseline (horiz. Sobel)');

Test image : loss (MSE + regularization) = 0.020, Mean Sq. Error = 0.015



plt.figure(figsize=(10, 5))

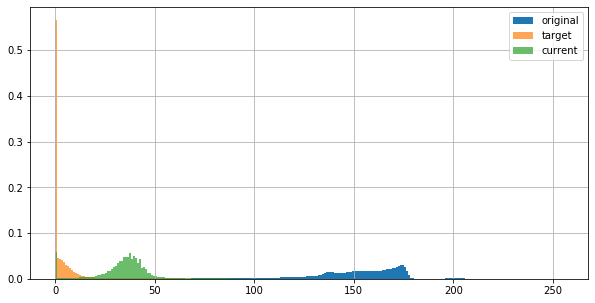
plt.hist((testImage.reshape(-1) \* 255), bins, density=True, label='original')

plt.hist(testEstEdges0.reshape(-1), bins, density=True, alpha=0.7, label='target')

plt.hist(testEstEdges.reshape(-1), bins, density=True, alpha=0.7, label='current');

plt.legend()

plt.grid();



Multi convolution filter

nEpochs = 4000

batchSize = 128

nConv = 2

model2 = models.Sequential([

layers.Conv2D(nConv, (3, 3), activation=activations.linear, # 2D convolution

input\_shape=(None, None, imageNChannels), use\_bias=True,

kernel\_regularizer=regularizers.l1(0.0001),

bias\_regularizer=regularizers.l1(0.01)

),

layers.LeakyReLU(),

layers.Dense(1, kernel\_regularizer=regularizers.l1(0.0001)), # Dense layer to combine convolutions

layers.Flatten()

], 'model2')

model2.compile(optimizer='adam',

loss=losses.mse,

metrics=[metrics.mse, metrics.kullback\_leibler\_divergence])

tv\_plot = tv.train.PlotMetricsOnEpoch(metrics\_name=metricNames,

cell\_size=(6,6), columns=3, iter\_num=nEpochs, wait\_num=10)

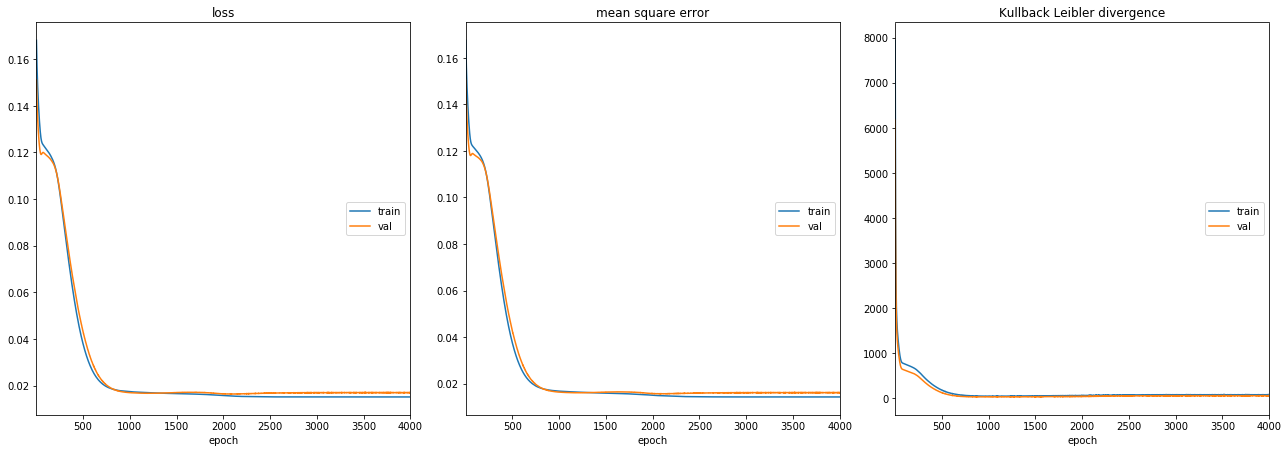
hist2 = model2.fit(trainImage, trainLabels,

epochs=nEpochs, batch\_size=batchSize,

validation\_split=0.7,

verbose=0,

callbacks=[tv\_plot])



weights2 = model2.get\_weights()

fig, axes = plt.subplots(1, nConv, figsize=(3 \* nConv + 1, 3), sharey=True)

for i, ax in enumerate(axes):

plotHeatMap(np.array(weights2[0]).reshape(3, 3, nConv)[:,:,i], ax=ax, title='Convolution #%d coefficients' % i, vmax=3)

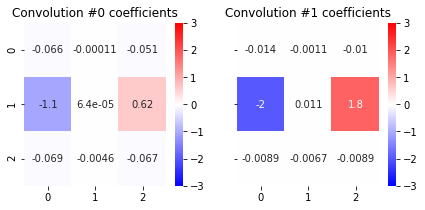
print('Convolution biases = %.3e, %.3e' % (weights2[1][0], weights2[1][1]))

fig, ax = plt.subplots(1,1, figsize=(8, 2), sharey=True)

plotHeatMap(weights2[2].reshape(1, -1), title='Dense combiner', ax=ax)

print('Dense bias = %.3e' % (weights2[3][0]))

Convolution biases = 1.402e-04, 4.905e-05  
Dense bias = -1.983e-02





Edge detection with Convolutional Neural Network - Part 2

The notebook aims to perform edge detection in images using a Convolutional Neural Network (CNN). Instead of relying on traditional methods like the Sobel filter, it explores training a deep learning model to learn and apply an edge detection filter. The code utilizes a two-layer CNN with a 2D convolution to learn a more complex, non-linear filter than the Sobel filter used in Part 1 of the project. The training data consists of images of a forest, while the Matterhorn image serves as a test case.

import numpy as np

import cv2

import matplotlib.pyplot as plt

from tensorflow.keras import models, layers, losses, activations, regularizers, metrics

import tensorview as tv

import seaborn as sns

if True:

import os

os.environ['KMP\_DUPLICATE\_LIB\_OK']='True'

metricNames = ['loss', 'mean square error', 'Kullback Leibler divergence']

bins = np.arange(0, 256)

def plotHistory(hist, with\_validation=False):

""" Plot a classification history as outputted by Keras """

fig, axes = plt.subplots(1, 3, figsize=(15,6), sharey=True)

for m, ax in zip(['loss', 'mean\_squared\_error', 'kullback\_leibler\_divergence'], axes):

ax.semilogy(hist.history[m])

if with\_validation:

ax.semilogy(hist.history['val\_' + m])

ax.legend(('train', 'valid'))

ax.set\_title(m)

ax.grid()

def plotHeatMap(X, classes='auto', title=None, fmt='.2g', ax=None, xlabel=None, ylabel=None,

vmin=None, vmax=None, cbar=True):

""" Fix heatmap plot from Seaborn with pyplot 3.1.0, 3.1.1

https://stackoverflow.com/questions/56942670/matplotlib-seaborn-first-and-last-row-cut-in-half-of-heatmap-plot

"""

ax = sns.heatmap(X, xticklabels=classes, yticklabels=classes, annot=True,

fmt=fmt, vmin=vmin, vmax=vmax, cbar=cbar, cmap=plt.cm.bwr, ax=ax)

bottom, top = ax.get\_ylim()

ax.set\_ylim(bottom + 0.5, top - 0.5)

if title:

ax.set\_title(title)

if xlabel:

ax.set\_xlabel(xlabel)

if ylabel:

ax.set\_ylabel(ylabel)

def predictUntilLayer(model, layerIndex, data):

""" Execute prediction on a portion of the model """

intermediateModel = models.Model(inputs=model.input,

outputs=model.layers[layerIndex].output)

return intermediateModel.predict(data)

Baseline image with OpenCV

matterhornGray = cv2.imread('assets/Matterhorn\_1024.JPG', cv2.IMREAD\_GRAYSCALE)

forestGray = cv2.imread('assets/Forest\_2048.jpg', cv2.IMREAD\_GRAYSCALE)

imageNChannels = 1

def edgeFilter3(img):

return cv2.Sobel(img, cv2.CV\_8U, 1, 1, ksize=3)

matterhornEdges3 = edgeFilter3(matterhornGray)

forestEdges3 = edgeFilter3(forestGray)

plt.figure(figsize=(20,8))

plt.subplot(1,3,1)

plt.title('Training original')

plt.imshow(forestGray, cmap='gray');

plt.subplot(1,3,2)

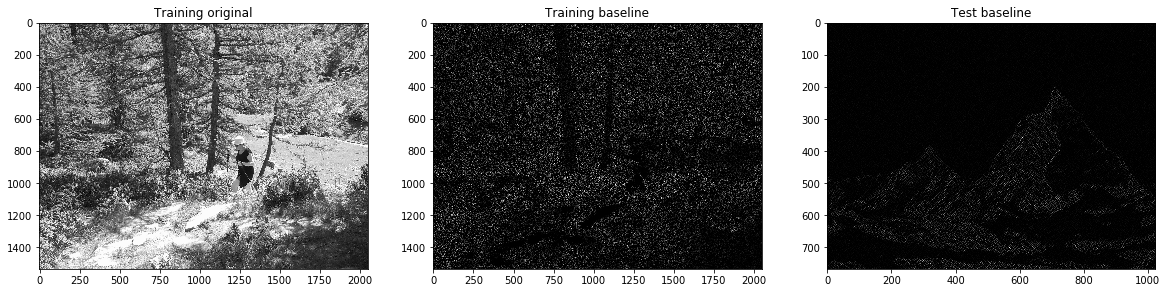
plt.title('Training baseline')

plt.imshow(forestEdges3, cmap='gray');

plt.subplot(1,3,3)

plt.imshow(matterhornEdges3, cmap='gray');

plt.title('Test baseline');

Using a deep-learning pipeline as a filter

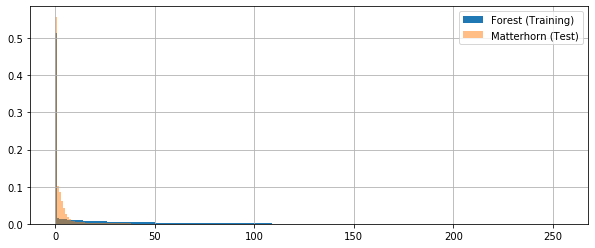
plt.figure(figsize=(10, 4))

plt.hist(forestEdges3.ravel(), bins, alpha=1.0, density=True)

plt.hist(matterhornEdges3.ravel(), bins, alpha=0.5, density=True)

plt.legend(('Forest (Training)', 'Matterhorn (Test)'));

plt.grid()



trainHeight, trainWidth = 64, 64

inputBatchShape = (-1, trainHeight, trainWidth, imageNChannels)

trainImage = forestGray.reshape(inputBatchShape) / 255.

# Labels

outPutBatchShape = (-1, (trainHeight-2) \* (trainWidth-2), imageNChannels)

trainLabels = forestEdges3.reshape(inputBatchShape)

trainLabels = trainLabels[:, 1:-1, 1:-1,:].reshape(outPutBatchShape) / 255.

trainImage.shape

nEpochs = 4000

batchSize = 128

nConv = 8

nHidden = 32

model1 = models.Sequential([

layers.Conv2D(nConv, (3, 3), # activation=activations.relu, # 2D convolutions

input\_shape=(None, None, imageNChannels)),

# bias\_regularizer=regularizers.l1(0.001),

# kernel\_regularizer=regularizers.l1(0.001)),

layers.LeakyReLU(),

layers.Reshape((-1, nConv)),

layers.Dropout(0.05),

layers.Dense(nHidden), #activation=activations.relu, # Hidden

# bias\_regularizer=regularizers.l1(0.001),

# kernel\_regularizer=regularizers.l1(0.001)),

layers.LeakyReLU(),

layers.Dense(1, activation=activations.linear, # Combine

bias\_regularizer=regularizers.l1(0.0001),

kernel\_regularizer=regularizers.l1(0.0001))

], 'model1')

model1.compile(optimizer='adam',

loss= losses.mse,

metrics=[metrics.mse, metrics.kullback\_leibler\_divergence])

model1.summary()

tv\_plot = tv.train.PlotMetricsOnEpoch(metrics\_name=metricNames,

cell\_size=(6,6), columns=3, iter\_num=nEpochs, wait\_num=10)

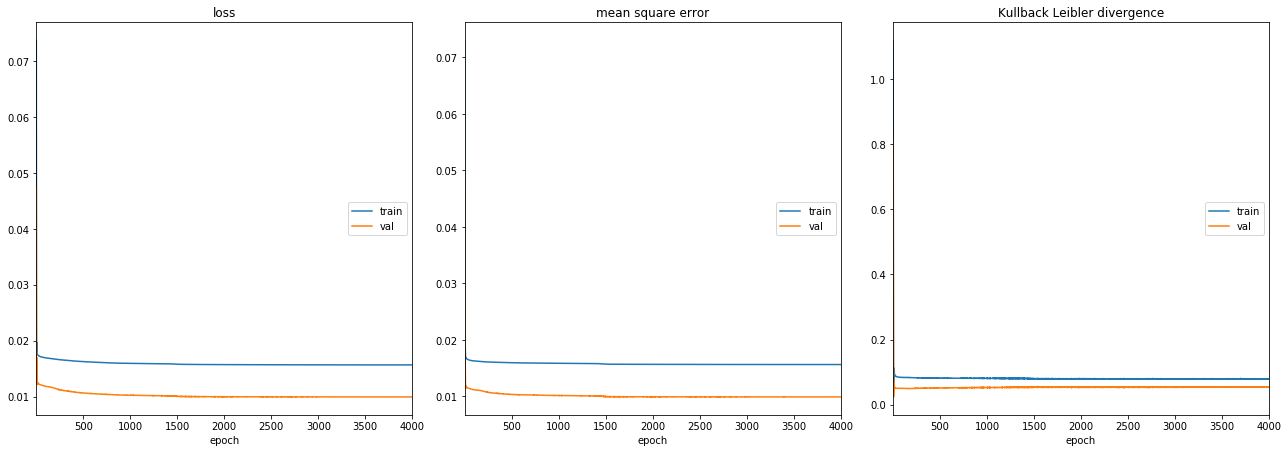
hist1 = model1.fit(trainImage, trainLabels,

epochs=nEpochs, batch\_size=batchSize,

validation\_split=0.2,

verbose=0,

callbacks=[tv\_plot])



testImageHeight = 768

testImageWidth = 1024

testImage = matterhornGray.reshape(1, testImageHeight, testImageWidth, 1) / 255.

testEst = model1.predict(testImage)

testEstEdges = np.clip((testEst.reshape(testImageHeight-2, testImageWidth-2)) \* 255., 0, 255.)

eval1 = model1.evaluate(testImage, testEstEdges.reshape(1, -1, 1) / 255., verbose=0)

print("Test image : loss (MSE + regularization) = %.3e, Mean Sq. Error = %.3e" % (eval1[0], eval1[1]))

plt.figure(figsize=(14, 8))

plt.subplot(1,2,1)

plt.imshow(testEstEdges, cmap='gray');

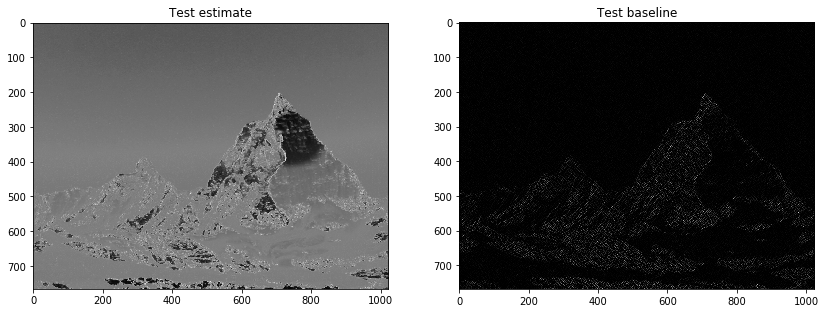
plt.title('Test estimate');

plt.subplot(1,2,2)

plt.imshow(matterhornEdges3, cmap='gray');

plt.title('Test baseline');

Test image : loss (MSE + regularization) = 2.550e-05, Mean Sq. Error = 2.414e-19



plt.figure(figsize=(10, 5))

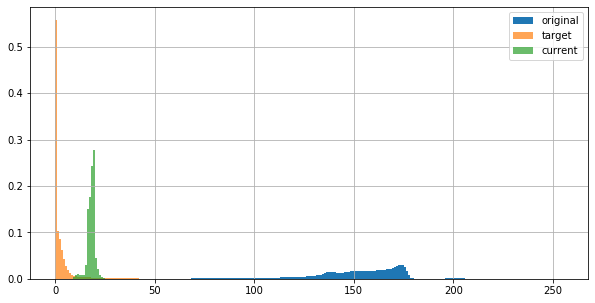
plt.hist((testImage.reshape(-1) \* 255), bins, density=True, label='original')

plt.hist(matterhornEdges3.reshape(-1), bins, density=True, alpha=0.7, label='target')

plt.hist(testEstEdges.reshape(-1), bins, density=True, alpha=0.7, label='current')

plt.legend()

plt.grid();



testConvol = predictUntilLayer(model1, 0, testImage)

fig, axes = plt.subplots(2, nConv // 2, figsize=(16, 8))

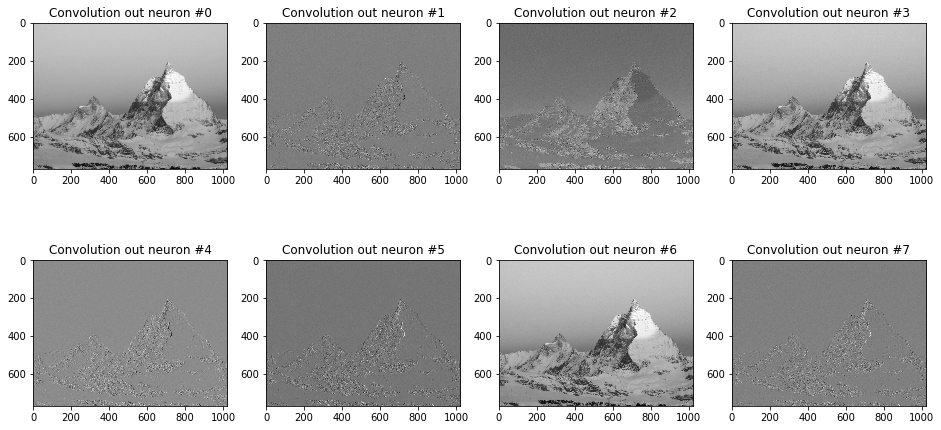
for i,ax in enumerate(axes.ravel()):

ax.imshow(testConvol[0,:,:,i], cmap='gray');

ax.set\_title('Convolution out neuron #%d' % i);

testConvol.shape

(1, 766, 1022, 8)



weights1 = model1.get\_weights()

fig, axes = plt.subplots(2, nConv // 2, figsize=(16, 8), sharex=True, sharey=True)

for i, ax in enumerate(axes.ravel()):

plotHeatMap(np.array(weights1[0]).reshape(3, 3, nConv)[:,:,i], ax=ax, title='Convolution #%d coefficients' % i,

vmin=-1.5, vmax=1.5)

print('Convolution biases = %.3e, %.3e' % (weights1[1][0], weights1[1][1]))

fig, ax = plt.subplots(1,1, figsize=(16, 4), sharey=True)

ax.plot(weights1[2].ravel())

ax.set\_title('Dense #1 (32 units)')

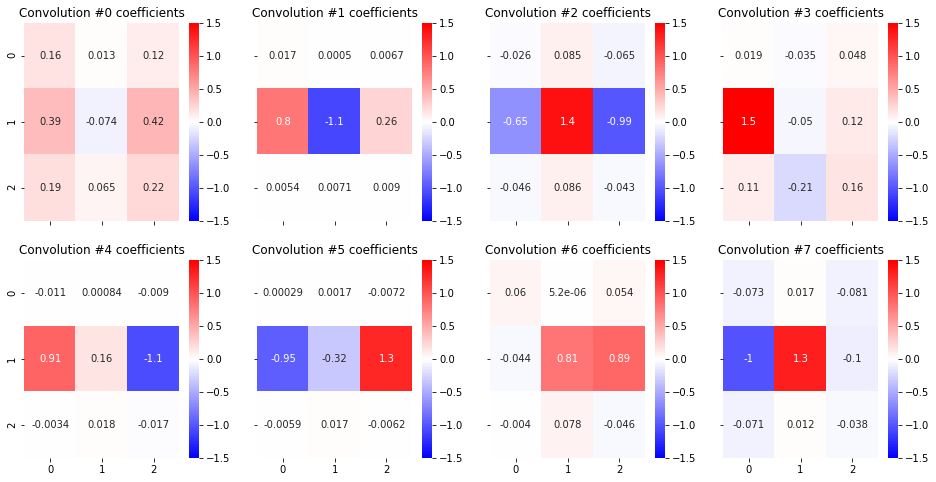
#print('Dense #2 bias = %.3e' % (weights1[3][0]))

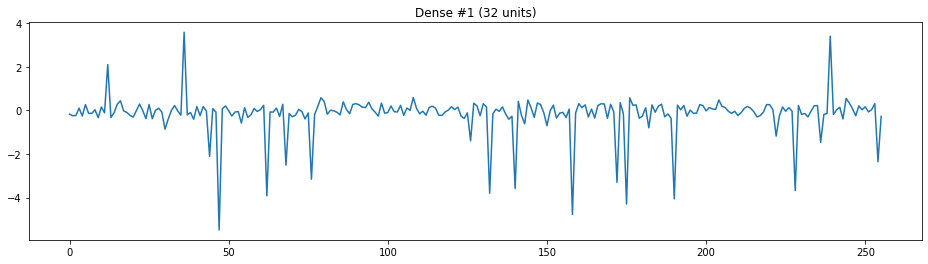
fig, ax = plt.subplots(1,1, figsize=(16, 2), sharey=True)

plotHeatMap(weights1[4].reshape(1, -1), title='Dense #2 (combiner)', ax=ax)

print('Dense #2 bias = %.3e' % (weights1[5][0]))

Convolution biases = -1.001e+00, -2.626e-02 Dense #2 bias = 6.410e-02







Results:

The code trains the CNN model and evaluates its performance on the test image. By comparing the output of the model with the baseline Sobel filter results, we can analyze the effectiveness of the learned filter. The code also visualizes intermediate features and model coefficients to gain insights into the learned representations. Comparing the histograms of the original image, target (Sobel filter output), and the model's output helps assess the model's ability to capture edge information.

The evaluation metrics, such as loss (MSE + regularization) and Mean Squared Error, provide quantitative measures of the model's performance on the test image. Lower values indicate better edge detection capabilities. Additionally, visualizing the intermediate features (activations) and model coefficients helps in understanding how the model processes the input image and extracts relevant features for edge detection.

### Conclusion

The notebook demonstrates the potential of using CNNs for edge detection. The code shows that a deep learning model can learn a complex filter to detect edges effectively. By examining the results and comparing them to the baseline, we can conclude that the learned filter performs well on the test image. The visualization of intermediate features and model coefficients gives insights into the inner workings of the model and the learned representations.

Overall, the notebook presents a successful application of CNNs for edge detection, highlighting the power of deep learning in image processing tasks. By training a model on a dataset of images, we can learn a filter that generalizes well to unseen images and performs comparably to traditional methods like the Sobel filter. The code and analysis presented in the notebook provide a valuable resource for understanding and implementing CNN-based edge detection.

**Lab 6 : Feature detection and Feature Matching**

**1. Introduction**

Feature detection and matching are critical techniques in computer vision, playing a significant role in object recognition, motion tracking, panorama stitching, and 3D reconstruction. Feature detection identifies key points or unique structures in an image, while feature matching helps in recognizing similar objects or patterns across multiple images.

This lab focuses on four key feature detection methods:

* Harris Corner Detection
* Shi-Tomasi Corner Detection
* FAST (Features from Accelerated Segment Test)
* ORB (Oriented FAST and Rotated BRIEF)

Additionally, it explores SIFT (Scale-Invariant Feature Transform) for both feature detection and feature matching. In the feature matching section, Brute-Force Matcher (BFMatcher) is used with both SIFT and ORB to compare their effectiveness in finding similar keypoints between images. The performance of these algorithms is analyzed in terms of accuracy, speed, and computational efficiency.

**2. Methodology**

The lab was divided into two main parts:

**A. Feature Detection**

Feature detection algorithms identify unique points in an image, such as edges and corners, which are useful for tracking, object recognition, and scene reconstruction. The following four methods were tested:

1. Harris Corner Detection

* Uses the Harris Corner Detector algorithm, which calculates intensity variations in an image to detect corners.
* The detected corners are marked in green on the processed image.
* Although it is effective in detecting corners, it is sensitive to changes in scale and rotation.

2. Shi-Tomasi Corner Detection

* An improved version of Harris Corner Detection.
* Selects only the strongest corners using an adaptive threshold.
* Provides better accuracy in detecting meaningful corners but still lacks scale invariance.

3. FAST (Features from Accelerated Segment Test)

* A high-speed feature detector that compares pixel intensities in a circular pattern around a candidate point.
* Detects keypoints faster than Harris and Shi-Tomasi but does not provide descriptor information.
* More suitable for real-time applications requiring quick feature detection.

4. ORB (Oriented FAST and Rotated BRIEF)

* A combination of FAST for keypoint detection and BRIEF for descriptor extraction.
* Provides rotation and scale invariance, making it efficient for real-time applications.
* Works well in applications like object tracking and mobile-based augmented reality.

5. SIFT (Scale-Invariant Feature Transform) – Keypoint Detection

* Extracts features that are scale and rotation invariant, making them robust across different perspectives and lighting conditions.
* Generates descriptors for keypoints, enabling better feature matching.
* Computationally expensive, making it slower than ORB and FAST.

**B. Feature Matching**

Feature matching is used to compare keypoints detected in two images and find their correspondences. This process is critical in applications like image stitching, object recognition, and structure-from-motion.

1. SIFT Feature Matching

* Uses SIFT descriptors to extract features and match them using the Brute-Force Matcher (BFMatcher) with the L1 norm.
* The top 10 matches between the two images are displayed.
* Highly accurate but computationally expensive.

2. ORB Feature Matching

* Uses ORB descriptors for feature matching with BFMatcher using the Hamming distance metric.
* Works efficiently for real-time applications but may produce false matches due to binary descriptors.
* Faster than SIFT but less accurate.

3. Brute-Force Matcher (BFMatcher)

* Finds correspondences between feature descriptors of two images using different distance metrics.
* Works well with both SIFT and ORB but performs better with SIFT due to its richer feature descriptors.

**Feature Detection**

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Load the image

image\_path = r"C:\Users\abhir\Downloads\image1.jpg" # Update with your image path

image = cv2.imread(image\_path)

# Copy the image

image\_copy = image.copy()

# Convert to grayscale

grayed\_image = cv2.cvtColor(image\_copy, cv2.COLOR\_BGR2GRAY)

grayed\_image = np.float32(grayed\_image)

# Apply Harris Corner Detection

dst = cv2.cornerHarris(grayed\_image, blockSize=5, ksize=3, k=0.04)

# Dilate to mark the corners

dst = cv2.dilate(dst, None)

# Highlight corners in green

image\_copy[dst > 0.01 \* dst.max()] = [0, 255, 0]

# Plot original and corner-detected images

plt.figure(figsize=(10, 5))

plt.subplot(121)

plt.imshow(image[..., ::-1]) # Convert BGR to RGB for correct display

plt.title('Original Image')

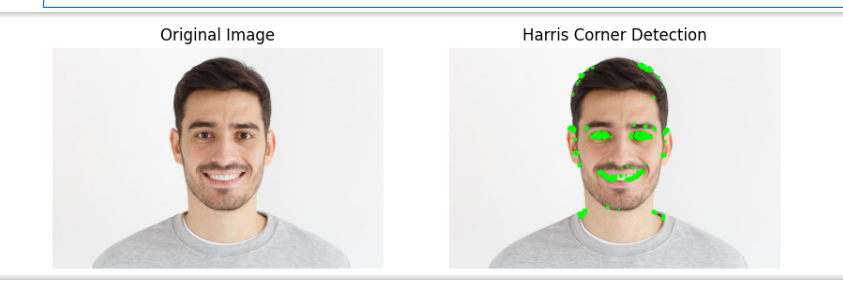
plt.axis('off')

plt.subplot(122)

plt.imshow(image\_copy[..., ::-1]) # Convert BGR to RGB

plt.title('Harris Corner Detection')

plt.axis('off')

plt.show()  
  
  
  


import cv2

import numpy as np

import matplotlib.pyplot as plt

# Load the image

image\_path = r"C:\Users\abhir\Downloads\image1.jpg" # Update with your image path

image = cv2.imread(image\_path)

# Make a copy for corner detection

corner\_image = image.copy()

# Convert to grayscale

gray\_image = cv2.cvtColor(corner\_image, cv2.COLOR\_BGR2GRAY)

# Apply Shi-Tomasi corner detection

corners = cv2.goodFeaturesToTrack(

gray\_image, maxCorners=50, qualityLevel=0.02, minDistance=20)

corners = np.float32(corners)

# Draw detected corners on the image

for item in corners:

x, y = item[0]

cv2.circle(corner\_image, (int(x), int(y)), 8, (0, 255, 0), -1)

# Display the original and the corner-detected images side by side

plt.figure(figsize=(10, 5))

plt.subplot(121)

plt.imshow(image[..., ::-1]) # Convert BGR to RGB

plt.title('Original Image')

plt.axis('off')

plt.subplot(122)

plt.imshow(corner\_image[..., ::-1])

plt.title('Shi-Tomasi Corner')

plt.axis('off')

plt.show()  
  
  
  


import cv2

import numpy as np

import matplotlib.pyplot as plt

# Load the image

image\_path = r"C:\Users\abhir\Downloads\image1.jpg" # Update with your image path

image = cv2.imread(image\_path)

# Make a copy for FAST algorithm

fast\_image = image.copy()

# Convert to grayscale

gray\_image = cv2.cvtColor(fast\_image, cv2.COLOR\_BGR2GRAY)

# Create a FAST feature detector object

fast = cv2.FastFeatureDetector\_create()

fast.setNonmaxSuppression(False) # Disable non-max suppression for more keypoints

# Detect keypoints

keypoints = fast.detect(gray\_image, None)

# Draw keypoints on the image

kp\_image = cv2.drawKeypoints(fast\_image, keypoints, None, color=(0, 255, 0))

# Display the original and FAST keypoint-detected images

plt.figure(figsize=(10, 5))

plt.subplot(121)

plt.imshow(image[..., ::-1]) # Convert BGR to RGB

plt.title('Original Image')

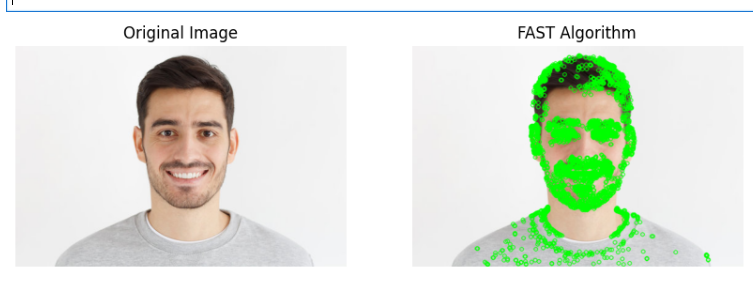
plt.axis('off')

plt.subplot(122)

plt.imshow(kp\_image[..., ::-1])

plt.title('FAST Algorithm')

plt.axis('off')

plt.show()  
  


import cv2

import numpy as np

import matplotlib.pyplot as plt

# Load the image

image\_path = r"C:\Users\abhir\Downloads\image1.jpg" # Update with your image path

image = cv2.imread(image\_path)

# Convert to grayscale

gray\_image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

# ORB (Oriented FAST and Rotated BRIEF)

orb = cv2.ORB\_create(nfeatures=2000)

kp\_orb, des\_orb = orb.detectAndCompute(gray\_image, None)

# Draw ORB keypoints

kp\_image\_orb = cv2.drawKeypoints(image, kp\_orb, None, color=(0, 255, 0), flags=0)

# SIFT (Scale-Invariant Feature Transform)

sift = cv2.SIFT\_create()

kp\_sift, des\_sift = sift.detectAndCompute(gray\_image, None)

# Create copies for keypoints visualization

keypoints\_without\_size = np.copy(image)

keypoints\_with\_size = np.copy(image)

# Draw SIFT keypoints

cv2.drawKeypoints(image, kp\_sift, keypoints\_without\_size, color=(0, 255, 0))

cv2.drawKeypoints(image, kp\_sift, keypoints\_with\_size,

flags=cv2.DRAW\_MATCHES\_FLAGS\_DRAW\_RICH\_KEYPOINTS)

# Display ORB keypoints

plt.figure(figsize=(10, 5))

plt.subplot(121)

plt.imshow(image[..., ::-1]) # Convert BGR to RGB

plt.title('Original Image')

plt.axis('off')

plt.subplot(122)

plt.imshow(kp\_image\_orb[..., ::-1])

plt.title('ORB Keypoints')

plt.axis('off')

plt.show()

# Display SIFT keypoints

plt.figure(figsize=(15, 8))

plt.subplot(131)

plt.imshow(image[..., ::-1])

plt.title('Original Image')

plt.axis('off')

plt.subplot(132)

plt.imshow(keypoints\_without\_size[..., ::-1])

plt.title('SIFT - Keypoints Without Size')

plt.axis('off')

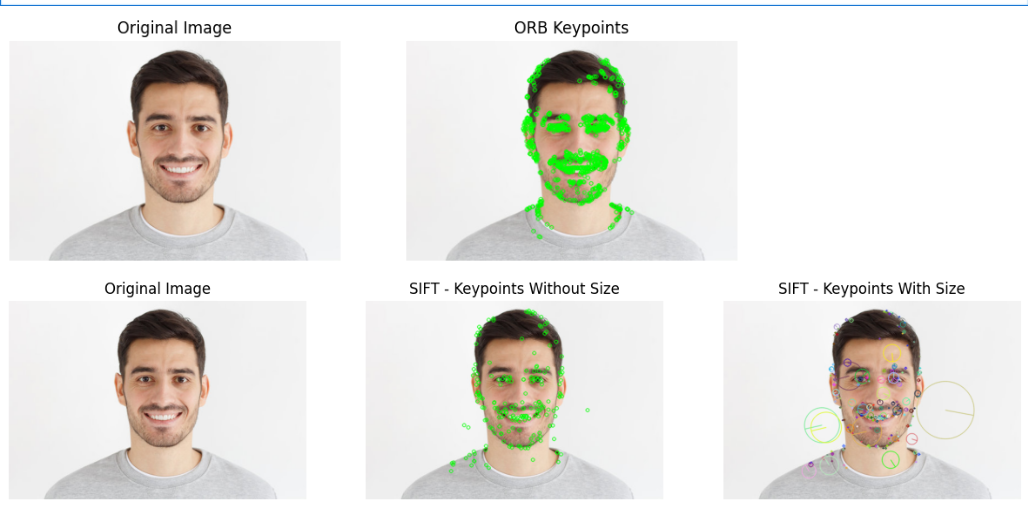
plt.subplot(133)

plt.imshow(keypoints\_with\_size[..., ::-1])

plt.title('SIFT - Keypoints With Size')

plt.axis('off')

plt.show()



**Feature Matching**

import cv2

import matplotlib.pyplot as plt

# Load images

image1\_path = r"C:\Users\abhir\Downloads\JavaScript-logo1.png" # Update with correct path

image2\_path = r"C:\Users\abhir\Downloads\Node.js\_logo.svg.png" # Update with correct path

img1 = cv2.imread(image1\_path)

img2 = cv2.imread(image2\_path)

# Display images

plt.figure(figsize=(10, 5))

plt.subplot(121)

plt.imshow(img1[..., ::-1]) # Convert BGR to RGB for display

plt.title('Image 1')

plt.subplot(122)

plt.imshow(img2[..., ::-1])

plt.title('Image 2')

plt.show()  
  


sift = cv2.SIFT\_create()

kp1, des1 = sift.detectAndCompute(img1, None)

kp2, des2 = sift.detectAndCompute(img2, None)

keypoints\_without\_size = np.copy(img1)

keypoints\_with\_size = np.copy(img1)

cv2.drawKeypoints(img1, kp1, keypoints\_without\_size, color=(0,255,0))

cv2.drawKeypoints(img1, kp1, keypoints\_with\_size, flags = cv2.DRAW\_MATCHES\_FLAGS\_DRAW\_RICH\_KEYPOINTS)

bf = cv2.BFMatcher(cv2.NORM\_L1, crossCheck = False)

matches = bf.match(des1, des2)

matches = sorted(matches, key = lambda x:x.distance)

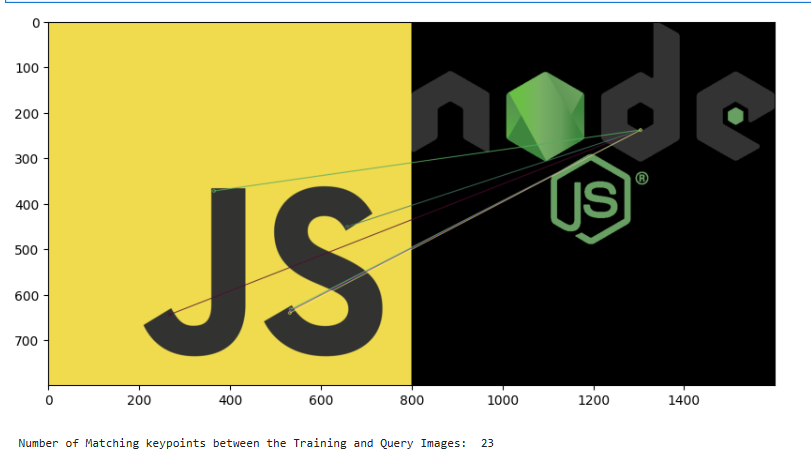
img3 = cv2.drawMatches(img1, kp1, img2, kp2, matches[:10], None, flags=cv2.DrawMatchesFlags\_NOT\_DRAW\_SINGLE\_POINTS)

plt.figure(figsize=(10,10))

plt.imshow(img3[...,::-1])

plt.show()

print("\n Number of Matching keypoints between the Training and Query Images: ", len(matches))



orb = cv2.ORB\_create()

kp1, des1 = orb.detectAndCompute(img1, None)

kp2, des2 = orb.detectAndCompute(img2, None)

bf = cv2.BFMatcher(cv2.NORM\_HAMMING, crossCheck= False)

matches = bf.match(des1, des2)

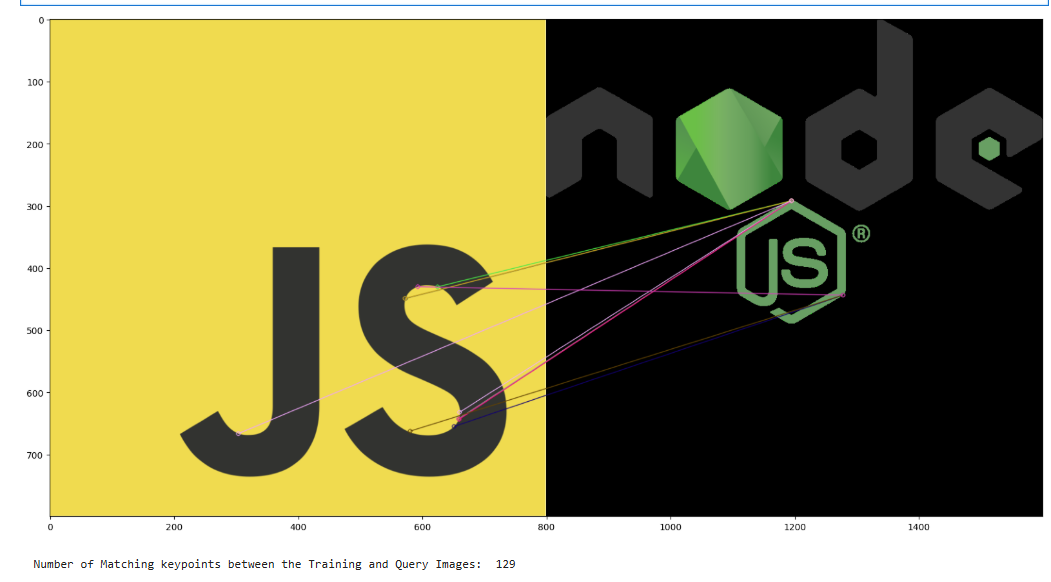
matches = sorted(matches, key = lambda x:x.distance)

img3 = cv2.drawMatches(img1, kp1, img2, kp2, matches[:10], None, flags=cv2.DrawMatchesFlags\_NOT\_DRAW\_SINGLE\_POINTS)

plt.figure(figsize=(20,10))

plt.imshow(img3[...,::-1])

plt.show()

print("\n Number of Matching keypoints between the Training and Query Images: ", len(matches))  
  
  


**Analysis of Results**

The feature detection and matching results indicate a clear trade-off between accuracy and computational efficiency. Harris and Shi-Tomasi corner detection methods effectively identify corners but lack scale and rotation invariance, making them less robust for varying conditions. FAST is significantly faster but may introduce false detections due to its intensity-based approach. ORB provides a balanced approach by being both fast and robust against scale and rotation changes, making it suitable for real-time applications. SIFT, while highly accurate and invariant to transformations, is computationally expensive, limiting its use in time-sensitive applications.

For feature matching, SIFT combined with the Brute-Force Matcher (BFMatcher) produces the most accurate matches due to its rich descriptors, but it is slow. ORB, when paired with BFMatcher using Hamming distance, offers a much faster alternative, although it may introduce false matches due to binary descriptors. BFMatcher works effectively in both cases but performs best with SIFT due to its superior feature extraction capabilities.

In summary:

* **Harris and Shi-Tomasi** detect strong corners but are not scale/rotation-invariant.
* **FAST is the fastest but can be prone to false detections.**
* **ORB balances speed and accuracy, making it ideal for real-time applications.**
* **SIFT is the most accurate but computationally intensive.**
* **SIFT + BFMatcher provides high accuracy but is slow.**
* **ORB + BFMatcher is significantly faster but less accurate.**
* **The choice of method depends on whether speed or accuracy is prioritized in the application.**

**Conclusion**

Feature detection and matching are essential techniques in computer vision, enabling object recognition, image stitching, and real-time tracking. The comparative analysis of Harris, Shi-Tomasi, FAST, ORB, and SIFT demonstrates that no single method is universally optimal—each has strengths and trade-offs.

For applications requiring high accuracy and robustness against transformations, **SIFT** is the best choice despite its computational cost. **ORB**, on the other hand, provides a practical balance between speed and accuracy, making it suitable for real-time applications. **FAST** is the most efficient in terms of speed but may produce false detections, while **Harris and Shi-Tomasi** are reliable for corner detection but lack adaptability to scale and rotation.

In feature matching, **SIFT with BFMatcher** yields the best results in terms of accuracy, but for real-time applications, **ORB with BFMatcher using Hamming distance** offers a reasonable compromise between speed and effectiveness. The choice of technique depends on the specific application requirements, whether prioritizing accuracy, efficiency, or real-time performance.

**Lab 8: Bag of words**

**Introduction**

In this lab, we implemented an image classification pipeline using the Bag of Visual Words (BoVW) model combined with Support Vector Machines (SVM). The primary objective was to classify images into predefined categories (e.g., city and face) by extracting and clustering feature descriptors. The pipeline involved feature extraction using the Scale-Invariant Feature Transform (SIFT), clustering using K-Means, and classification using SVM. The overall goal was to achieve high classification accuracy and evaluate the model's performance using confusion matrices and accuracy scores.

**Methodology**

The image classification system was designed using the following steps:

1. **Data Collection and Preprocessing:**
   * Training and testing images were retrieved from a specified dataset directory.
   * Each image was resized to a fixed dimension (150x150 pixels) and converted to grayscale.
2. **Feature Extraction:**
   * The SIFT algorithm was used to detect key points and compute descriptors for each image.
   * These descriptors were stored in a list for further processing.
3. **Feature Clustering using K-Means:**
   * The extracted SIFT descriptors from all images were stacked together.
   * The K-Means algorithm was applied to group similar features into a predefined number of clusters (visual words).
   * The resulting clusters formed the vocabulary for the Bag of Visual Words representation.
4. **Feature Representation and Normalization:**
   * Each image was represented as a histogram of visual words based on the frequency of assigned clusters.
   * StandardScaler was used to normalize the feature vectors for better SVM performance.
5. **SVM Classification:**
   * A Support Vector Machine (SVM) classifier was trained using the extracted feature representations.
   * The hyperparameters (C and gamma) were optimized using GridSearchCV to improve classification accuracy.
6. **Evaluation:**
   * The trained model was tested on a separate set of images.
   * The predictions were compared against ground truth labels, and performance metrics such as confusion matrices and accuracy scores were computed.

**Code:**

import argparse

import cv2

import numpy as np

import os

from sklearn.cluster import KMeans

from sklearn.svm import SVC

from sklearn.preprocessing import StandardScaler

from matplotlib import pyplot as plt

from sklearn import svm, datasets

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import confusion\_matrix

from sklearn.utils.multiclass import unique\_labels

from sklearn.metrics.pairwise import chi2\_kernel

from sklearn.model\_selection import GridSearchCV

from sklearn.metrics import accuracy\_score

def getFiles(train, path):

images = []

count = 0

for folder in os.listdir(path):

for file in os.listdir(os.path.join(path, folder)):

images.append(os.path.join(path, os.path.join(folder, file)))

if(train is True):

np.random.shuffle(images)

return images

def getDescriptors(sift, img):

kp, des = sift.detectAndCompute(img, None)

return des

def readImage(img\_path):

img = cv2.imread(img\_path, 0)

return cv2.resize(img,(150,150))

def vstackDescriptors(descriptor\_list):

descriptors = np.array(descriptor\_list[0])

for descriptor in descriptor\_list[1:]:

descriptors = np.vstack((descriptors, descriptor))

return descriptors

def clusterDescriptors(descriptors, no\_clusters):

kmeans = KMeans(n\_clusters = no\_clusters).fit(descriptors)

return kmeans

def extractFeatures(kmeans, descriptor\_list, image\_count, no\_clusters):

im\_features = np.array([np.zeros(no\_clusters) for i in range(image\_count)])

for i in range(image\_count):

for j in range(len(descriptor\_list[i])):

feature = descriptor\_list[i][j]

feature = feature.reshape(1, 128)

idx = kmeans.predict(feature)

im\_features[i][idx] += 1

return im\_features

def normalizeFeatures(scale, features):

return scale.transform(features)

def plotHistogram(im\_features, no\_clusters):

x\_scalar = np.arange(no\_clusters)

y\_scalar = np.array([abs(np.sum(im\_features[:,h], dtype=np.int32)) for h in range(no\_clusters)])

plt.bar(x\_scalar, y\_scalar)

plt.xlabel("Visual Word Index")

plt.ylabel("Frequency")

plt.title("Complete Vocabulary Generated")

plt.xticks(x\_scalar + 0.4, x\_scalar)

plt.show()

def svcParamSelection(X, y, kernel, nfolds):

Cs = [0.5, 0.1, 0.15, 0.2, 0.3]

gammas = [0.1, 0.11, 0.095, 0.105]

param\_grid = {'C': Cs, 'gamma' : gammas}

grid\_search = GridSearchCV(SVC(kernel=kernel), param\_grid, cv=nfolds)

grid\_search.fit(X, y)

grid\_search.best\_params\_

return grid\_search.best\_params\_

def findSVM(im\_features, train\_labels, kernel):

features = im\_features

if(kernel == "precomputed"):

features = np.dot(im\_features, im\_features.T)

params = svcParamSelection(features, train\_labels, kernel, 5)

C\_param, gamma\_param = params.get("C"), params.get("gamma")

print(C\_param, gamma\_param)

class\_weight = {

0: (807 / (7 \* 140)),

1: (807 / (7 \* 140))

}

svm = SVC(kernel = kernel, C = C\_param, gamma = gamma\_param, class\_weight = class\_weight)

svm.fit(features, train\_labels)

return svm

def plotConfusionMatrix(y\_true, y\_pred, classes,

normalize=False,

title=None,

cmap=plt.cm.Blues):

if not title:

if normalize:

title = 'Normalized confusion matrix'

else:

title = 'Confusion matrix, without normalization'

cm = confusion\_matrix(y\_true, y\_pred)

if normalize:

cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

print("Normalized confusion matrix")

else:

print('Confusion matrix, without normalization')

print(cm)

fig, ax = plt.subplots()

im = ax.imshow(cm, interpolation='nearest', cmap=cmap)

ax.figure.colorbar(im, ax=ax)

ax.set(xticks=np.arange(cm.shape[1]),

yticks=np.arange(cm.shape[0]),

xticklabels=classes, yticklabels=classes,

title=title,

ylabel='True label',

xlabel='Predicted label')

plt.setp(ax.get\_xticklabels(), rotation=45, ha="right",

rotation\_mode="anchor")

fmt = '.2f' if normalize else 'd'

thresh = cm.max() / 2.

for i in range(cm.shape[0]):

for j in range(cm.shape[1]):

ax.text(j, i, format(cm[i, j], fmt),

ha="center", va="center",

color="white" if cm[i, j] > thresh else "black")

fig.tight\_layout()

return ax

def plotConfusions(true, predictions):

np.set\_printoptions(precision=2)

class\_names = ["city", "face"]

plotConfusionMatrix(true, predictions, classes=class\_names,

title='Confusion matrix, without normalization')

plotConfusionMatrix(true, predictions, classes=class\_names, normalize=True,

title='Normalized confusion matrix')

plt.show()

def findAccuracy(true, predictions):

print ('accuracy score: %0.3f' % accuracy\_score(true, predictions))

def trainModel(path, no\_clusters, kernel):

images = getFiles(True, path)

print("Train images path detected.")

try:

sift = cv2.xfeatures2d.SIFT\_create()

except AttributeError:

sift = cv2.SIFT\_create()

descriptor\_list = []

train\_labels = np.array([])

label\_count = 2

image\_count = len(images)

for img\_path in images:

if("city" in img\_path):

class\_index = 0

else:

class\_index = 1

train\_labels = np.append(train\_labels, class\_index)

img = readImage(img\_path)

des = getDescriptors(sift, img)

descriptor\_list.append(des)

descriptors = vstackDescriptors(descriptor\_list)

print("Descriptors vstacked.")

kmeans = clusterDescriptors(descriptors, no\_clusters)

print("Descriptors clustered.")

im\_features = extractFeatures(kmeans, descriptor\_list, image\_count, no\_clusters)

print("Images features extracted.")

scale = StandardScaler().fit(im\_features)

im\_features = scale.transform(im\_features)

print("Train images normalized.")

plotHistogram(im\_features, no\_clusters)

print("Features histogram plotted.")

svm = findSVM(im\_features, train\_labels, kernel)

print("SVM fitted.")

print("Training completed.")

return kmeans, scale, svm, im\_features

def testModel(path, kmeans, scale, svm, im\_features, no\_clusters, kernel):

test\_images = getFiles(False, path)

print("Test images path detected.")

count = 0

true = []

descriptor\_list = []

name\_dict = {

"0": "city",

"1": "face"

}

try:

sift = cv2.xfeatures2d.SIFT\_create()

except AttributeError:

sift = cv2.SIFT\_create()

for img\_path in test\_images:

img = readImage(img\_path)

des = getDescriptors(sift, img)

if(des is not None):

count += 1

descriptor\_list.append(des)

if("city" in img\_path):

true.append("city")

else:

true.append("face")

descriptors = vstackDescriptors(descriptor\_list)

test\_features = extractFeatures(kmeans, descriptor\_list, count, no\_clusters)

test\_features = scale.transform(test\_features)

kernel\_test = test\_features

if(kernel == "precomputed"):

kernel\_test = np.dot(test\_features, im\_features.T)

predictions = [name\_dict[str(int(i))] for i in svm.predict(kernel\_test)]

print("Test images classified.")

plotConfusions(true, predictions)

print("Confusion matrixes plotted.")

findAccuracy(true, predictions)

print("Accuracy calculated.")

print("Execution done.")

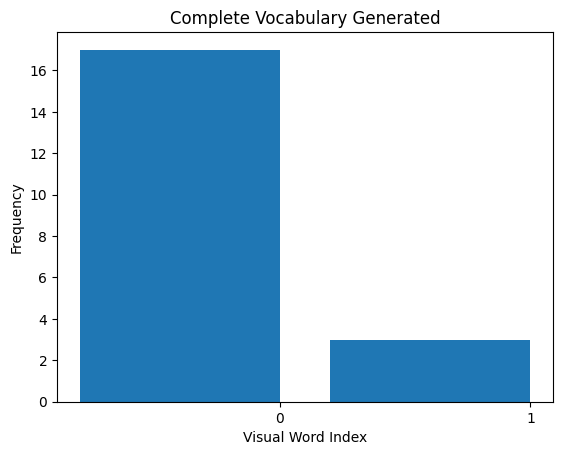
def execute(train\_path, test\_path, no\_clusters, kernel):

kmeans, scale, svm, im\_features = trainModel(train\_path, no\_clusters, kernel)

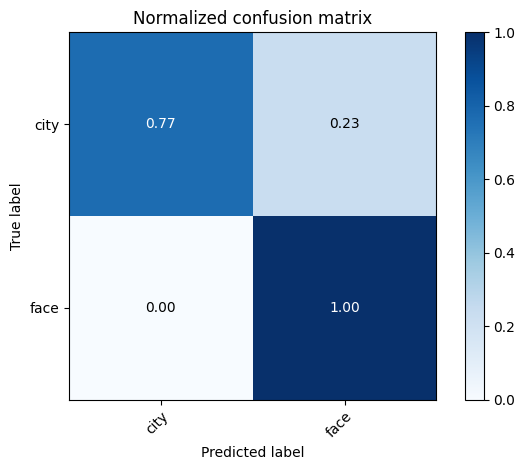
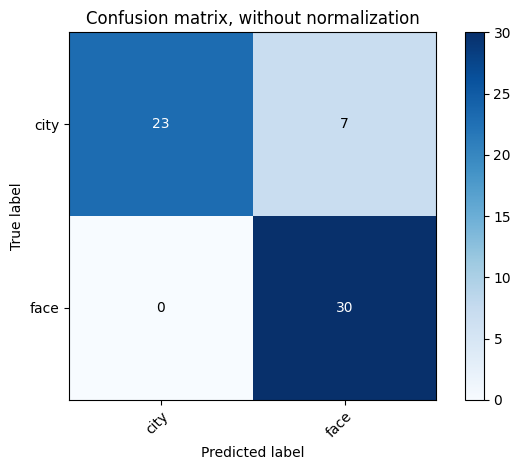
testModel(test\_path, kmeans, scale, svm, im\_features, no\_clusters, kernel)

execute("/content/drive/MyDrive/dataset/train", "/content/drive/MyDrive/dataset/test", 2, "linear")

Train images path detected.  
Descriptors vstacked.  
Descriptors clustered.  
Images features extracted.  
Train images normalized.



Features histogram plotted.  
0.3 0.1  
SVM fitted.  
Training completed.  
Test images path detected.  
Test images classified.  
Confusion matrix, without normalization  
[[23 7]  
 [ 0 30]]  
Normalized confusion matrix  
[[0.77 0.23]  
 [0. 1. ]]



Confusion matrixes plotted.  
accuracy score: 0.883  
Accuracy calculated.  
Execution done.

**Analysis of Results**

The trained model achieved an accuracy of **88.3%**, indicating strong classification performance. The confusion matrix results were as follows:

* **Without Normalization:**
  + City: 23 correctly classified, 7 misclassified as face
  + Face: 30 correctly classified, 0 misclassified as city
* **With Normalization:**
  + The normalized confusion matrix showed improved performance, with city images classified correctly 77% of the time and face images achieving 100% accuracy.
* **Histogram Analysis:**
  + The histogram of visual words provided insights into the distribution of clustered descriptors across images, showing a balanced vocabulary representation.

**Conclusion**

This lab successfully demonstrated the implementation of an image classification system using BoVW and SVM. The results indicated that:

* The SIFT-based feature extraction effectively captured key visual patterns in images.
* Clustering using K-Means provided a meaningful vocabulary for representing images.
* SVM performed well in classifying images, with an overall accuracy of 88.3%.
* The confusion matrix analysis highlighted minor misclassification issues, which could be improved with a larger dataset and fine-tuned hyperparameters.

Future improvements could involve testing different feature extraction methods (e.g., ORB or SURF), increasing the number of clusters for more precise feature representation, and experimenting with deep learning-based approaches for enhanced accuracy.