1. **00\_align\_face\_with\_mtcnn.py**

Code:

import os

import json

import dlib

from facenet\_pytorch import MTCNN

import numpy as np

import util\_img as util

import cv2

import torch

os.environ["CUDA\_VISIBLE\_DEVICES"] = '0'

# CUDA (Compute Unified Device Architecture) is a **parallel computing platform and programming model** developed by **NVIDIA**. It enables developers to **leverage the power of GPUs** (Graphics Processing Units) for general-purpose computing tasks beyond graphics rendering, such as deep learning and image processing.

This line ensures that only **GPU 0** is used for processing, which is useful when multiple GPUs are available.

print("Preparing dlib ... ", end='', flush=True)

detector = dlib.get\_frontal\_face\_detector()

Dlib is an open-source machine learning library for Python that can be used for tasks like face detection, facial landmark detection, and facial recognition

Initializes a face detector (get\_frontal\_face\_detector) and loads a facial landmark predictor from a pre-trained model (shape\_predictor\_81\_face\_landmarks.dat).

predictor\_path = '/EX\_STORE/Beauty\_app/DeepRhythm/shape\_predictor\_81\_face\_landmarks.dat'

Link : <https://github.com/codeniko/shape_predictor_81_face_landmarks>

Link : https://github.com/italojs/facial-landmarks-recognition/tree/master

predictor = dlib.shape\_predictor(predictor\_path)

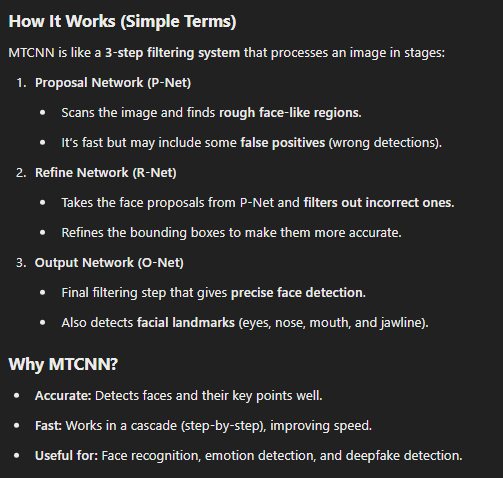
print("Done")

print("Preparing MTCNN ... ", end='', flush=True)

mtcnn = MTCNN(thresholds=[0.3, 0.3, 0.3], margin=20, keep\_all=True, post\_process=False, select\_largest=False, device=torch.device('cuda', 0))

Initializes a Multi-Task Cascaded Convolutional Neural Network (MTCNN) for face detection and alignment. Parameters like thresholds and margin control its behavior.

With Conventional neural networks, it’s hard to train the model on so many parameters, hence we use CNN, which uses a sliding window approach to detect the edges in the image, which are passed on to the next layer, so as to detect features



print("Done")

Provide the path to the i/p

dataset\_root = '/DATASET/FaceForensics/FaceForensics\_dataset/FaceForensics\_dataset/'

video\_dir\_name = "original\_sequences/youtube/c23/"

dataset\_dir = dataset\_root + video\_dir\_name + 'videos/'

Provide the path to o/p

# metadata path

meta\_dir = '/EX\_STORE/Beauty\_app/00\_Face\_data/' + video\_dir\_name

if not os.path.exists(meta\_dir):

os.makedirs(meta\_dir)

'''

Generate aligned face and remove background

'''

def generate\_align\_face(data\_path, save\_path):

if not os.path.exists(save\_path):

os.makedirs(save\_path)

vidlist = os.listdir(data\_path)

vidlist.sort()

sorting is done **lexicographically** (like dictionary order),

Eg "video\_10.mp4" comes before "video\_2.mp4" because "1" (in "video\_10.mp4") appears before "2" in string comparison.

for vidname in vidlist:

vidpath = data\_path + vidname

save\_vid\_path = meta\_dir + vidname[:-4] + '/'

if os.path.exists(save\_vid\_path):

continue

else:

os.mkdir(save\_vid\_path)

try:

align\_face = util.preprocess\_video(vidname[:-4], detector, predictor, mtcnn, vidpath, save\_vid\_path)

except:

print("Error caused !!")

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

If this script is executed, this part is executed (not if this is imported as module)

generate\_align\_face(dataset\_dir, meta\_dir)

1. **Util\_img.py**

generate\_align\_face function calls preprocess\_video present in util\_img.py

Function call :

util.preprocess\_video(vidname[:-4], detector, predictor, mtcnn, vidpath, save\_vid\_path)

Here, predictor = dlib.shape\_predictor(predictor\_path)

detector = dlib.get\_frontal\_face\_detector()

preprocess\_video:

def preprocess\_video(video\_file\_name, detector, predictor, mtcnn, vidpath, facepath):

print("Processing "+video\_file\_name+":")

video\_file\_path = vidpath

Stores the path of the video file.

face\_align\_dir = facepath

Directory where processed face images will be saved.

margin = (20, 20, 20, 20) # (start\_point\_x & y, end\_point\_x & y)

Adds a margin around detected faces to ensure the full face is captured.

batch\_size = 30

Number of frames processed at once to optimize performance

first\_frame\_distance = None

Tracks the distance between eyes in the first frame to maintain consistency across frames.

video = cv2.VideoCapture(video\_file\_path)

Opens the video file for reading

face\_imgs = []

List to store processed face images.

idx = 0

Frame counter

stime = time.time()

Records the start time for performance tracking

frames = []

Stores frames in batches

while True:

# ttime = time.time()

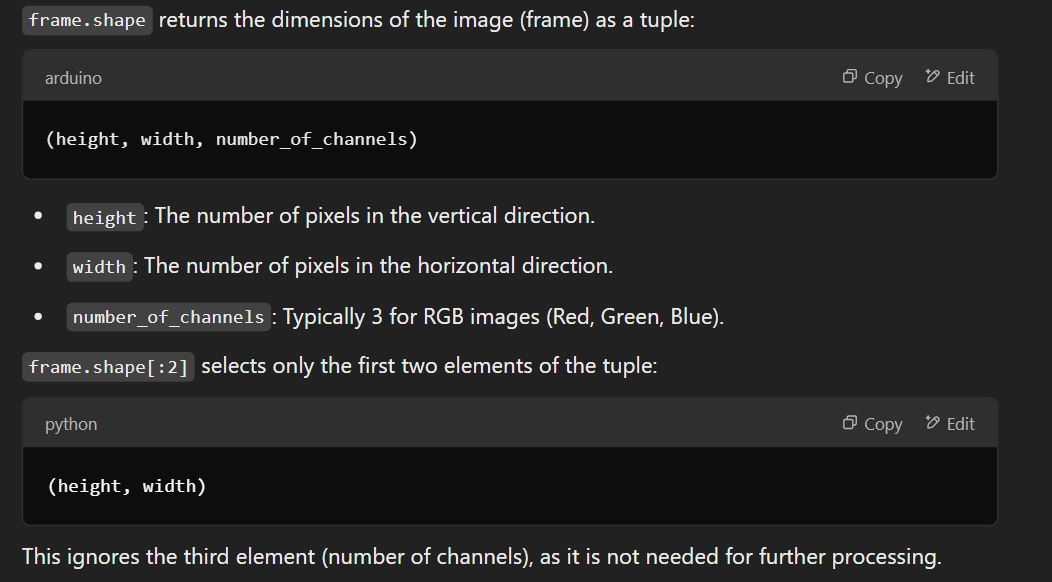
success, frame = video.read()

Reads frames from the video one by one.

if not success:

break

height, width = frame.shape[:2]



frames.append(frame)

idx += 1

if idx > 300:

break

if idx % batch\_size == 0:

first\_frame\_center = None

first\_frame\_center is reset for each batch

batch\_boxes, \_ = mtcnn.detect(frames)

Every batch\_size (30) frames, face detection is performed for all the frames using **MTCNN**.

pos = 0

pos is used to track the position of frames in the batch.

for boxes in batch\_boxes:

Iterates over detected face bounding boxes for each frame

print(" %04d ... " % (idx - batch\_size + pos), end='', flush=True)

# print(box.shape)

if boxes is None:

If no face is detected (boxes is None), it skips the frame.

print("Done, there is no face")

continue

st = [int(boxes[0,0]-margin[0]), int(boxes[0,1]-margin[1])]

Starting **(x, y)** of the expanded bounding box.

ed = [int(boxes[0,2]+margin[2]), int(boxes[0,3]+margin[3])]

Ending **(x, y)** of the expanded bounding box.

Expands the detected face bounding box by adding margins

Ensures the bounding box does not go outside the image boundaries.

if st[0]<0:

If st[0] (x-start) goes negative → Set it to 0

st[0] = 0

if st[1]<0:

If st[1] (y-start) goes negative → Set it to 0

st[1] = 0

if ed[0]>width:

If ed[0] (x-end) exceeds image width → Set it to width

ed[0] = width

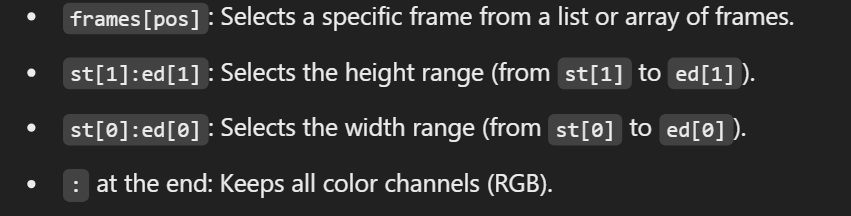
if ed[1]>height:

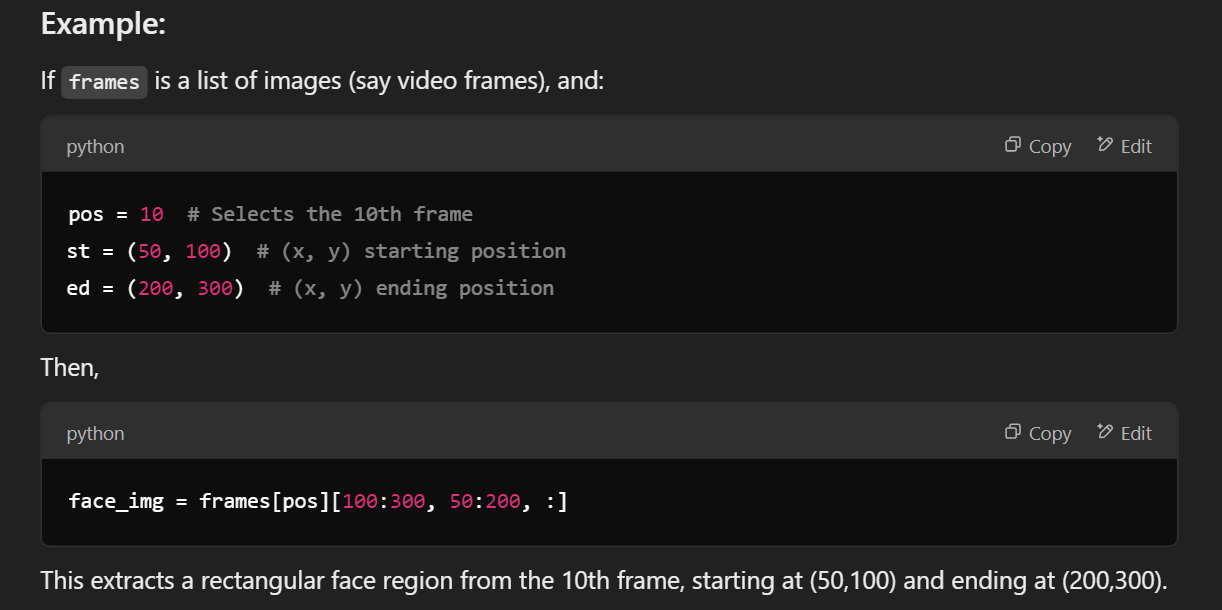
If ed[1] (y-end) exceeds image height → Set it to height.

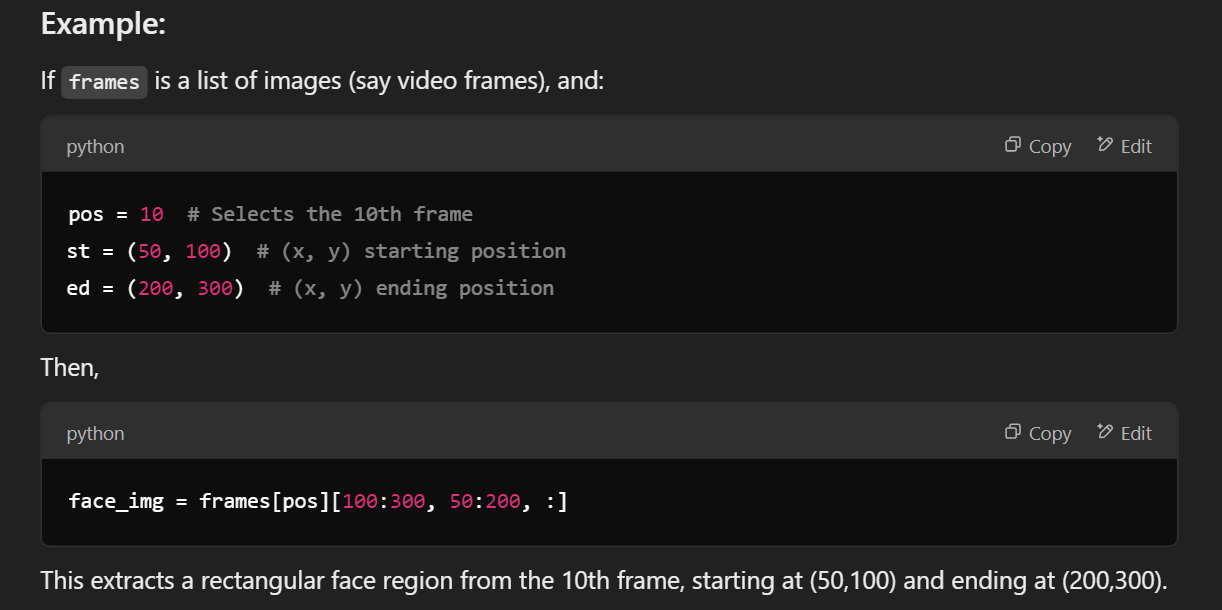
ed[1] = height

box = [st[0], st[1], ed[0], ed[1]]

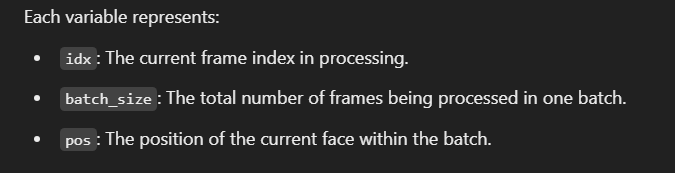
face\_img = frames[pos][st[1]:ed[1], st[0]:ed[0], :]

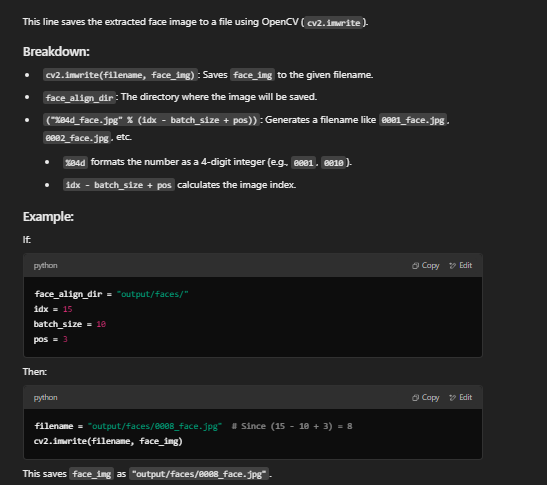






cv2.imwrite(face\_align\_dir + ("%04d\_face.jpg" % (idx - batch\_size + pos)), face\_img)





pre\_img, distance, center, landmark = preprocess\_img(face\_img, detector, predictor, first\_frame\_distance, first\_frame\_center, (idx - batch\_size + pos))

if first\_frame\_distance is None:

first\_frame\_distance = distance

**First frame distance** refers to the distance between the eyes in the first frame of the video. It is used as a reference to maintain consistent scaling of the face across frames.

if first\_frame\_center is None:

first\_frame\_center = center

**First frame center** refers to the center point of the eyes in the first frame of the video. It is used to align the face consistently across frames.

if pre\_img is not None:

print("Done, saving ... ", end='', flush=True)

cv2.imwrite(face\_align\_dir + ("%04d.jpg" % (idx - batch\_size + pos)), pre\_img)

Save the pre\_img as .jpg file and landmark as .npy file

np.save(face\_align\_dir + ("%04d\_landmark.npy" % (idx - batch\_size + pos)), landmark)

print("Done")

else :

print("Done, no face")

pos += 1

frames = []

print("-- Using time:", time.time() - stime)

preprocess\_img function:

def preprocess\_img(img, detector, predictor, first\_frame\_distance, first\_frame\_center, idx):

landmark = get\_landmark(img, detector, predictor)

if landmark is None:

return None, None, None, None

rotate\_img, distance, center, rotate\_land = get\_rotate\_img(img, landmark, first\_frame\_distance, first\_frame\_center, idx)

bound\_img, bounding\_land = get\_bounding\_image(rotate\_img, rotate\_land) # TOUGH!!!!!

no\_bg\_img = remove\_background(bound\_img, bounding\_land)

return no\_bg\_img, distance, center, landmark

get\_landmark function:

def get\_landmark(img, detector, predictor):

height, width = np.shape(img)[:2]

dets = detector(img, 0)

Uses the detector (which is a dlib face detector) to detect faces in the image.

dets stores the detected face regions.

0 **means no upscaling**

landmarks = None

for k, det in enumerate(dets):

shape = predictor(img, det)

For each detected face, the **shape predictor** finds **81 facial landmarks** (x, y coordinates of key points on the face like eyes, nose, mouth, etc.).

landmarks = np.matrix([[p.x, p.y] for p in shape.parts()])

Converts the detected landmarks into a **NumPy matrix**, where:

* Each row represents a landmark point (x, y).
* There are 81 points in total.

for i in range(81):

for j in range(2):

if landmarks[i, j]<0:

landmarks[i, j] = 0

if landmarks[i, 1]>height:

landmarks[i, 1] = height

if landmarks[i, 0]>width:

landmarks[i, 0] = width

return landmarks

remove\_background function:  
  
def remove\_background(img, landmark):

height, width, channels = np.shape(img)

This gets the **height**, **width**, and **number of color channels (RGB = 3)** of the input image.

mask = np.zeros((height, width))

This creates a **black mask** (same size as the image) filled with zeros.

It creates a **grayscale** image (because it has only one channel) where every pixel has a value of 0.

A **grayscale image** is a type of image that only has **shades of gray**, instead of colors like red, green, or blue.

**Simple Explanation:**

* Each pixel in a grayscale image has a **single value** that represents brightness.
* The value ranges from **0 to 255**:
  + 0 → **Black**
  + 255 → **White**
  + Values in between (like 128) are different shades of gray.

A **grayscale image** has only **one channel** because each pixel stores just **one value**—the brightness (how light or dark it is).

In grayscale images, pixel values range from **0 to 255**:

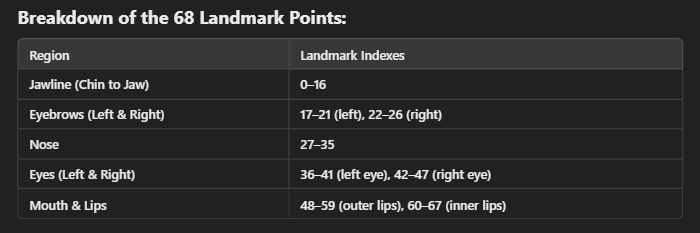
* 0 → **Black**
* 255 → **White**

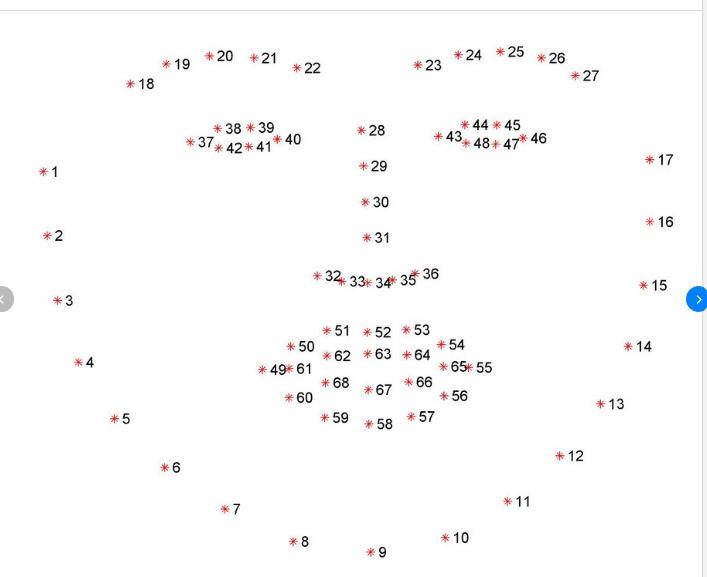
Since np.zeros((height, width)) initializes all values to 0, the entire mask is black.

This mask will later be used to define which areas should be kept or removed.

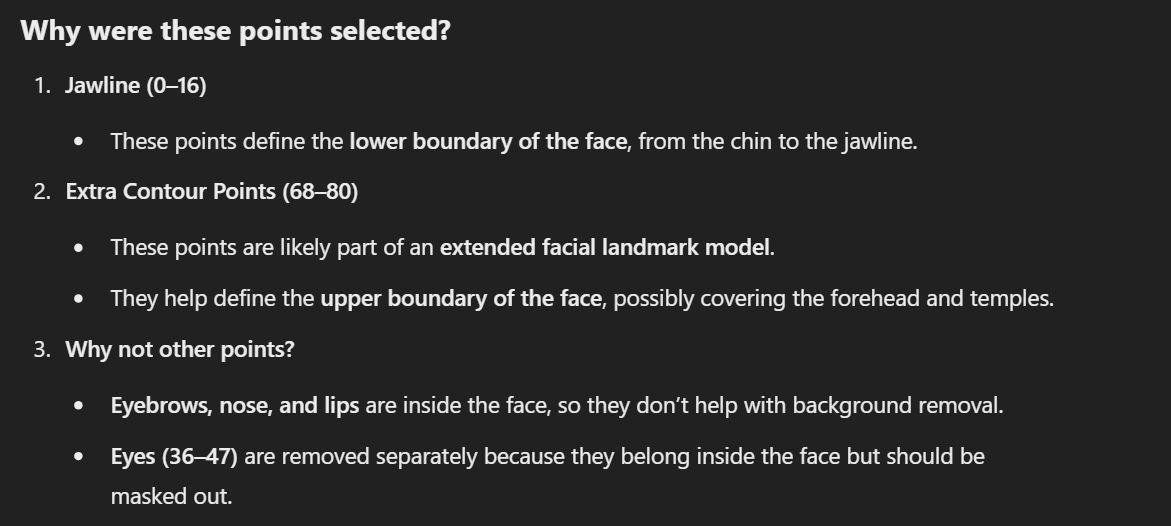
OUT\_POINT = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,78,74,79,73,72,80,71,70,69,68,76,75,77]

The list **OUT\_POINT** contains the landmark indices for the **face outline (jawline, cheeks, forehead, etc.).**





additional 13 landmarks to cover the forehead area



outline = np.array(landmark[OUT\_POINT])

cv2.fillPoly(mask, [outline], (255, 255, 255))

The function cv2.fillPoly(mask, [outline], (255, 255, 255)) **fills** the mask with **white (255)** in this region, indicating the area that should be kept.

# remove eye area

eyeline = np.array(landmark[36:42])

cv2.fillPoly(mask, [eyeline], (0, 0, 0))

eyeline = np.array(landmark[42:48])

cv2.fillPoly(mask, [eyeline], (0, 0, 0))

The eye landmark indices **36 to 41 (left eye)** and **42 to 47 (right eye)** define the eye regions.

cv2.fillPoly(mask, [eyeline], (0, 0, 0)) fills the **eye areas with black (0)**, removing them.

So basically, initially, the whole image is black.

cv2.fillPoly(mask, [outline], (255, 255, 255)) this line fills the jaw and forehead region with white, thus keeping them.

After that, eyes are filled with black, hence only the face shape remains as white, everything else is black, thus removing background

nbg\_img = np.zeros((height, width, channels))

for h in range(height):

for w in range(width):

if mask[h][w]==0:

continue

nbg\_img[h][w] = img[h][w]

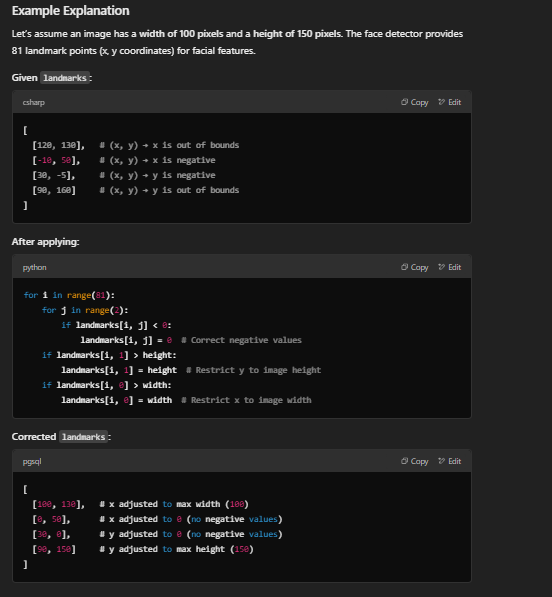
return nbg\_img

Creates an empty image (nbg\_img) with the same shape as the original image.

**Loops through every pixel**:

* If the mask value is **0 (black)**, that pixel is **ignored (background removed).**
* If the mask value is **255 (white)**, that pixel is **copied from the original image (kept).**

**So, black mask is initialized, then face part is kept white, then eyes part is kept black, hence only face remains, eyes and the other background is removed**



1. **01\_1\_resize\_frame.py**

**Code :**

import os

import cv2

import numpy as np

'''

Resize the aligned face so they can be formed into a video, which will

be used into motion magnification.

'''

def resize\_frame(align\_path, resize\_path):

if not os.path.exists(resize\_path):

os.makedirs(align\_path)

vidlist = os.listdir(align\_path)

vidlist.sort()

for vidname in vidlist:

print("{} - {} ... ".format(align\_path, vidname), end='', flush=True)

vidpath = align\_path + vidname + '/'

save\_vid\_path = resize\_path + vidname + '/'

os.mkdir(save\_vid\_path)

imglist = os.listdir(vidpath)

imglist.sort()

f\_h, f\_w = None, None

for imgname in imglist:

if imgname.endswith('.npy'):

continue

if imgname.endswith('\_face.jpg'):

continue

imgpath = vidpath + imgname

img = cv2.imread(imgpath)

if f\_h is None:

f\_h, f\_w = img.shape[:2]

f\_h = ((int(f\_h/10)+1)\*10)

f\_w = ((int(f\_w/10)+1)\*10)

If it's the first image (f\_h is None), it determines its height (f\_h) and width (f\_w).

Rounds f\_h and f\_w **up to the nearest multiple of 10** (to standardize dimensions).

img = cv2.resize(img, (f\_w, f\_h))

Resizes the image to f\_w × f\_h.

save\_img\_path = save\_vid\_path + imgname

cv2.imwrite(save\_img\_path, img)

Saves the resized image in the corresponding directory

idx = None

imglist = os.listdir(save\_vid\_path)

imglist.sort()

imglist.append("0300.jpg")

for imgname in imglist:

if int(imgname[:4]) >= 300:

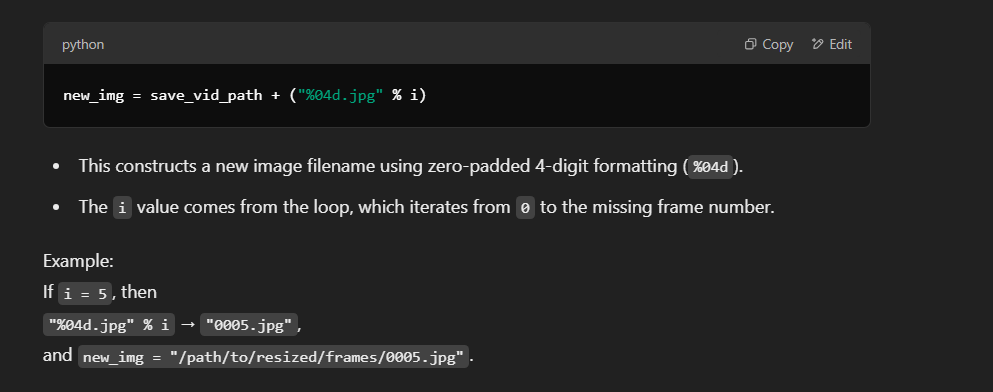
break

if idx is None and int(imgname[:4])!=0:

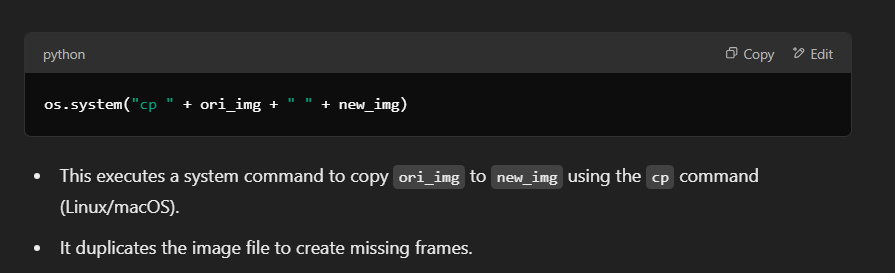
for i in range(0, int(imgname[:4])):

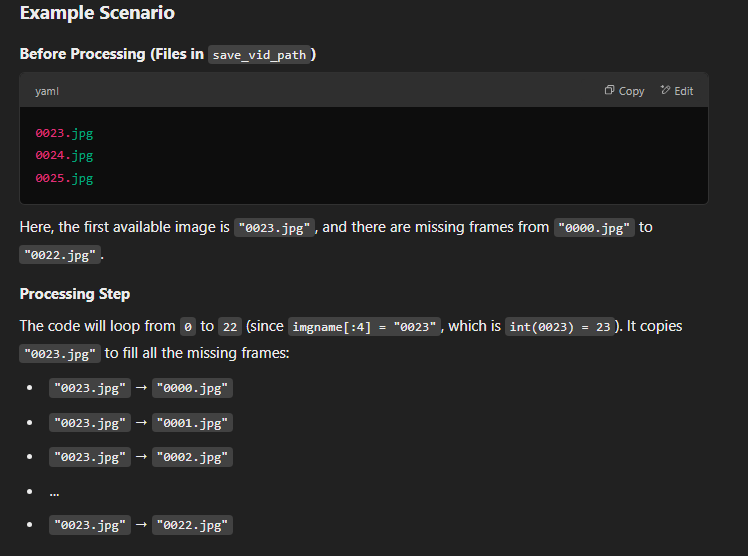
ori\_img = save\_vid\_path + imgname

new\_img = save\_vid\_path + ("%04d.jpg"%i)



os.system("cp " + ori\_img + " " + new\_img)





if idx is not None and idx < int(imgname[:4]):

for i in range(idx, int(imgname[:4])):

ori\_img = save\_vid\_path + ("%04d.jpg"%(idx-1))

new\_img = save\_vid\_path + ("%04d.jpg"%i)

os.system("cp " + ori\_img + " " + new\_img)

This checks if:

idx has been assigned a value (idx is not None).

idx is smaller than the current image number (idx < int(imgname[:4])).

It ensures that there is a gap between the last processed frame (idx-1) and the current image (imgname).

Loops from idx (the next expected frame) to the current frame number (imgname[:4]).

This fills all missing frames between the last known frame and the new frame.

Selects the last available frame (idx-1) to use as a filler.

If idx = 5, then "0004.jpg" is used as ori\_img

Creates the new filename for the missing frame.

If i = 6, it will generate "0006.jpg".

Executes a system command to copy the last available frame (ori\_img) to the missing frame (new\_img).

idx = int(imgname[:4]) + 1

print("Done")

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

video\_dir\_name = "original\_sequences/youtube/c23/"

data\_root\_dir = "/EX\_STORE/Beauty\_app/00\_Face\_data/"

datadir = data\_root\_dir + video\_dir\_name

resize\_root\_dir = "/EX\_STORE/Beauty\_app/01\_1\_resize\_original\_frame/"

resizedir = resize\_root\_dir + video\_dir\_name

if not os.path.exists(resizedir):

os.makedirs(resizedir)

resize\_frame(datadir, resizedir)

1. **01\_2\_align\_video.py**

import os

import cv2

import numpy as np

'''

Generate video with aligned face. These videos will be used in motion magnificaiton.

'''

def generate\_align\_video(video\_frame\_path, video\_store\_path):

if not os.path.exists(video\_store\_path):

os.makedirs(video\_store\_path)

vidlist = os.listdir(video\_frame\_path)

vidlist.sort()

for vidname in vidlist:

print("{} - {} ... ".format(video\_frame\_path, vidname), end='', flush=True)

vidpath = video\_frame\_path + vidname +'/'

save\_vid\_path = video\_store\_path + vidname + '.avi'

os.system("ffmpeg -i {}%04d\_face.jpg {}".format(vidpath, save\_vid\_path))

Command : ffmpeg -i Frames/Video1/%04d\_face.jpg Output/Video1.avi

**%04d** means the images are named sequentially like 0001\_face.jpg, 0002\_face.jpg, etc.

FFmpeg reads all the **\_face images** in order and treats them as frames of a video. It stitches them together into a .avi file

Output/Video1.avi: This is the final video file that FFmpeg creates from those images.

Why .avi?

**Less Compression** – AVI files use less compression than MP4, preserving more quality.

**Lossless Editing** – Easier to edit without quality loss.

**FFmpeg Default Support** – AVI works well with FFmpeg without needing extra codecs.

**Research & Processing** – Often used in AI/ML projects where raw video data is needed for analysis.

print("Done")

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

video\_dir\_name = "original\_sequences/youtube/c23/"

data\_root\_dir = "/EX\_STORE/Beauty\_app/01\_0\_resize\_original\_frame/"

datadir = data\_root\_dir + video\_dir\_name

new\_vid\_root\_dir = "/EX\_STORE/Beauty\_app/01\_0\_align\_original\_video/"

newviddir = new\_vid\_root\_dir + video\_dir\_name

if not os.path.exists(newviddir):

os.makedirs(newviddir)

generate\_align\_video(datadir, newviddir)

1. **02\_gen\_motion\_mag\_video.py**

Code :

import os

from python\_eulerian\_video\_magnification.magnifycolor import MagnifyColor

from python\_eulerian\_video\_magnification.metadata import MetaData

from python\_eulerian\_video\_magnification.mode import Mode

import cv2

import numpy as np

'''

Magnification video

'''

def generate\_mag\_video(vid\_path, mag\_path):

if not os.path.exists(mag\_path):

os.makedirs(mag\_path)

vidlist = os.listdir(vid\_path)

vidlist.sort()

for vidname in vidlist:

print("{} - {} ... ".format(video\_dir\_name, vidname), end='', flush=True)

vidpath = vid\_path + vidname

save\_vid\_path = mag\_path + vidname

MagnifyColor(MetaData(file\_name=vidpath, low=0.833, high=2, levels=1,

amplification=10, target\_path=save\_vid\_path, output\_folder='', mode=Mode.COLOR, suffix='color')).do\_magnify()

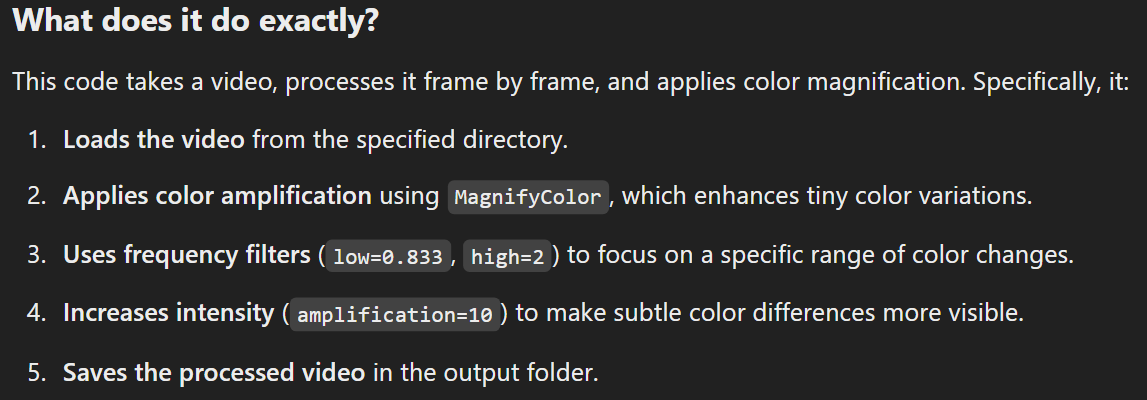
The main goal is to highlight tiny changes in color or motion that are otherwise hard to notice.

This executes the do\_magnify() function, which applies **Eulerian Video Magnification** to the input video.

**Parameters explained:**

* file\_name=vidpath: Path to the input video file.
* low=0.833, high=2: Defines the frequency band (in Hz) for filtering.
* levels=1: Number of pyramid levels used for processing.
* amplification=10: Magnification factor for color variations.
* target\_path=save\_vid\_path: Path where the magnified video will be saved.
* output\_folder='': No specific output folder is defined.
* mode=Mode.COLOR: Specifies that this is a color magnification mode.
* suffix='color': Suffix added to the output filename.

print("Done")



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

video\_dir\_name = "manipulated\_sequences/Deepfakes/c23/"

data\_root\_dir = "/EX\_STORE/Beauty\_app/01\_2\_align\_video/"

datadir = data\_root\_dir + video\_dir\_name

new\_vid\_root\_dir = "/EX\_STORE/Beauty\_app/02\_mag\_video/"

newviddir = new\_vid\_root\_dir + video\_dir\_name

if not os.path.exists(newviddir):

os.makedirs(newviddir)

generate\_mag\_video(datadir, newviddir)

1. **03\_gen\_map.py**

Code :

import os

import cv2

import util\_mit as util

import numpy as np

datadir = "/DATASET/\_\_saika\_data/dfdc/03\_mag\_video/"

newviddir = "/DATASET/\_\_saika\_data/dfdc/stmap/"

'''

Use magnified video to produce mmst-map

'''

def generate\_mmst\_map(mag\_path, map\_path):

if not os.path.exists(map\_path):

os.makedirs(map\_path)

ROI\_h, ROI\_w = 5, 5

These define the height and width of the region of interest

vidlist = os.listdir(mag\_path)

vidlist.sort()

for vidname in vidlist:

print("{} - {} ... ".format(typename, vidname), end='', flush=True)

vidpath = mag\_path + vidname

vid = cv2.VideoCapture(vidpath)

Opens the video file so frames can be read

full\_st\_map = np.zeros((300, 25, 3))

np.zeros((300, 25, 3)): Creates an empty NumPy array (size 300x25x3) to store processed frame data.

* 300: Number of frames (time dimension).
* 25: Width of the processed data.
* 3: Likely represents color channels (RGB).

idx = 0

frame index 0

while idx < 300:

success, frame = vid.read()

Reads the next frame from the video

if not success:

If no more frames are available, exit the loop.

break

frame\_seg = util.get\_frame\_seg(frame, ROI\_h, ROI\_w)

extracts a small segment of the frame

full\_st\_map[idx] = frame\_seg

Stores the processed frame data in full\_st\_map

idx += 1

# print("Done")

save\_vid\_path = map\_path + vidname

np.save(save\_vid\_path + ".npy", full\_st\_map)

Saves the processed MMST map as a .npy file

print("Done")

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

generate\_mmst\_map(datadir, newviddir)

get\_frame\_seg function :

def get\_frame\_seg(img, ROI\_h, ROI\_w):

ROI\_seg = calculate\_ROI(img, ROI\_h, ROI\_w)

# frame\_seg = reshape\_ROI\_SEG(ROI\_seg)

return ROI\_seg

calculate\_ROI function :

def calculate\_ROI(img, block\_h, block\_w):

height, width, channels = np.shape(img)

blk\_h, blk\_w = int(height/block\_h), int(width/block\_w)

roi\_seg = np.zeros((block\_h\*block\_w, channels))

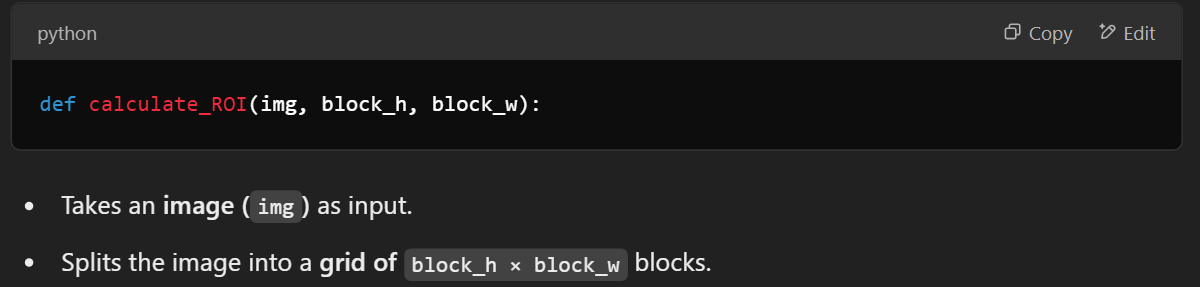
for bh in range(block\_h):

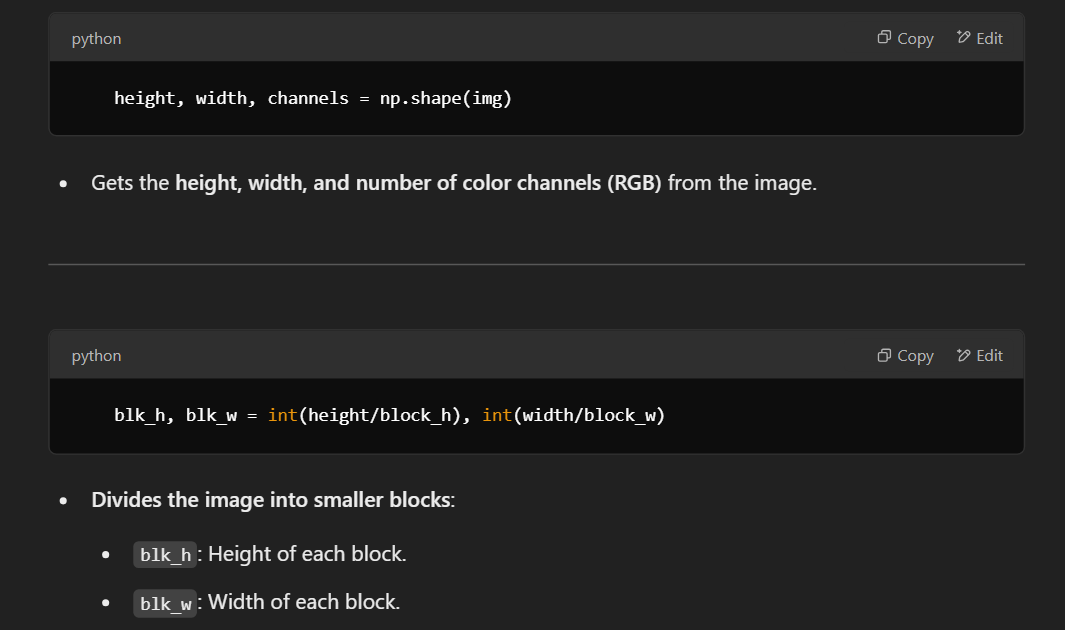
for bw in range(block\_w):

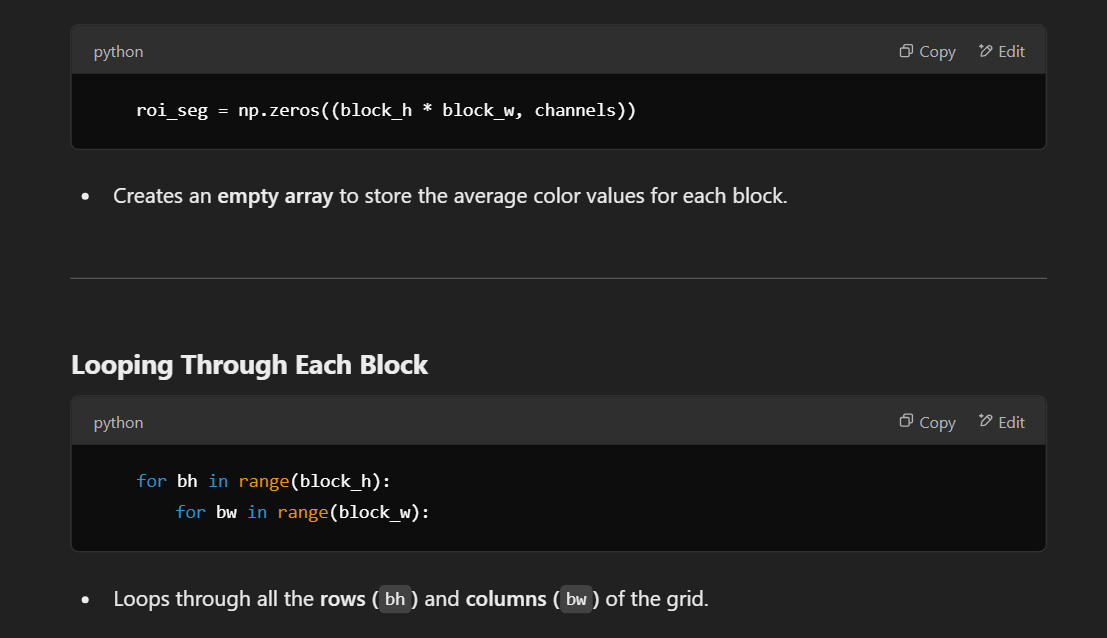
roi\_seg[bh\*block\_w+bw] = [np.average(img[blk\_h\*bh:blk\_h\*(bh+1),blk\_w\*bw:blk\_w\*(bw+1),i]) for i in range(3)]

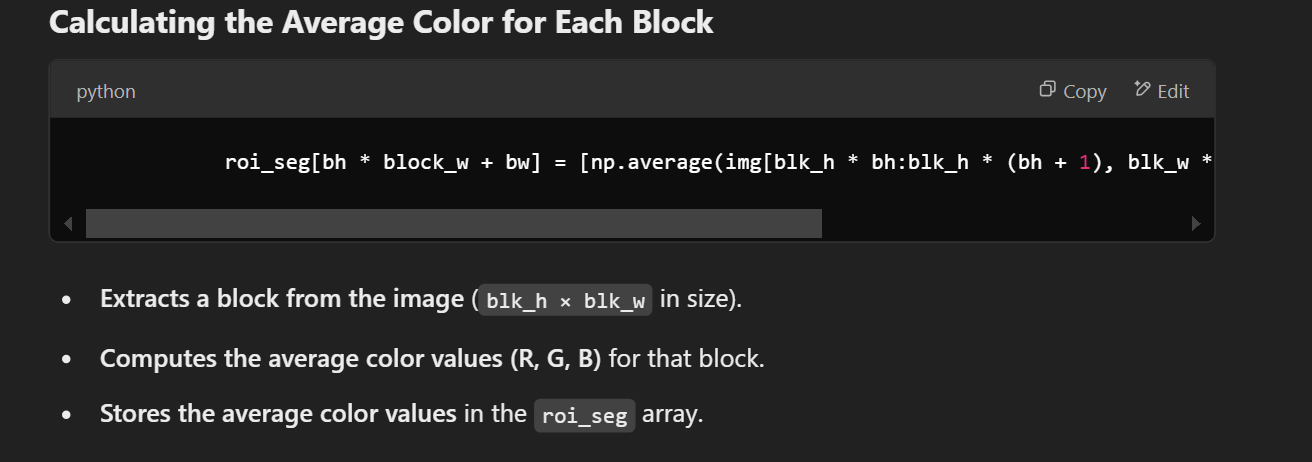
# roi\_seg[bh][bw] = [np.average(img[blk\_h\*bh:blk\_h\*(bh+1),blk\_w\*bw:blk\_w\*(bw+1),i]) for i in range(3)]

return roi\_seg









Returns roi\_seg, which contains the **average color values of all blocks**.

1. **04\_gen\_training\_data.py**

Code:

import numpy as np

from classifiers import \*

from pipeline import \*

from keras.preprocessing.image import ImageDataGenerator

from keras.models import load\_model

import os

import cv2

os.environ["CUDA\_VISIBLE\_DEVICES"] = '1'

Specifies **which GPU** (device 1) should be used for deep learning model execution

# ---------------------------- data\_Meso ------------------------------

'''

collect meso data

'''

# Load the model and its pretrained weights

classifier = MesoInception4()

MesoInception4 (a deep learning model for detecting DeepFakes).

classifier.load('/WORKSPACE/ff++/final\_data/model/FaceForensics/faceforensics++\_models\_subset/face\_detection/Meso/c23/Deepfakes/weights.h5')

Loads pre-trained weights from the given path

real\_video\_dir = "/WORKSPACE/ff++/original/youtube/c23/align\_original/"

real\_vid\_list = os.listdir(real\_video\_dir)

real\_vid\_list.sort()

real\_vid\_dict = {x:real\_video\_dir + x for x in real\_vid\_list}

Creates a dictionary {filename: filepath}

fake\_video\_dir = "/WORKSPACE/ff++/manipulate/DeepFakes/c23/align\_original/"

fake\_vid\_list = os.listdir(fake\_video\_dir)

fake\_vid\_list.sort()

fake\_vid\_dict = {x:fake\_video\_dir + x for x in fake\_vid\_list}

video\_dict = real\_vid\_dict

video\_dict.update(fake\_vid\_dict)

data\_name\_list = real\_vid\_list + fake\_vid\_list

# ---------------------------- data\_mit ------------------------------

'''

collect mmst map data

'''

Reads pre-processed .npy files (NumPy format) from the fake MIT dataset directory.

fake\_mit\_dir = "/WORKSPACE/ff++/manipulate/DeepFakes/c23/data\_mit\_rgb/"

fake\_mit\_list = os.listdir(fake\_mit\_dir)

fake\_mit\_list.sort()

fake\_mit\_dict = {x:fake\_mit\_dir + x + '/' + x + '.npy' for x in fake\_mit\_list}

real\_mit\_dir = "/WORKSPACE/ff++/original/youtube/c23/data\_mit\_rgb/"

real\_mit\_list = os.listdir(real\_mit\_dir)

real\_mit\_list.sort()

real\_mit\_dict = {x:real\_mit\_dir + x + '/' + x + '.npy' for x in real\_mit\_list}

mit\_dict = fake\_mit\_dict

mit\_dict.update(real\_mit\_dict)

# ---------------------------- save ------------------------------

'''

save meso data and mmst map data into a form that can be used in training

'''

save\_path = "/WORKSPACE/ff++/final\_data/data\_new/df\_ytb\_c23/"

if not os.path.exists(save\_path):

os.mkdir(save\_path)

data\_Meso\_set = []

Model predictions for frames.

data\_mit\_set = []

MIT data

data\_y\_set = []

Labels (1 = fake, 0 = real).

data\_name\_set = []

Video filenames.

for vid\_name in data\_name\_list:

print("video {}:".format(vid\_name), end='', flush=True)

if vid\_name+".avi" in mit\_dict:

vid\_path = video\_dict[vid\_name] + '/'

img\_list = os.listdir(vid\_path)

if img\_list==[]:

continue

* Gets the **path of the video’s frame folder** and lists images inside it.
* **Skips empty folders**.

img\_list.sort()

data\_Meso = np.ones(300) \* 0.5

Creates an array of **300 values initialized to 0.5** (default confidence).

try:

for img\_name in img\_list:

if int(img\_name[:-4]) >= 300:

continue

img\_path = vid\_path + img\_name

img = cv2.resize(cv2.imread(img\_path), (256, 256))

pred = classifier.predict(np.array([img]))

data\_Meso[int(img\_name[:-4])] = pred

**Reads each image** from the video folder.

**Resizes it to 256x256 pixels**.

**Runs it through the classifier** (MesoInception4 model).

**Stores prediction in data\_Meso array**.

except Exception:

continue

data\_Meso\_set.append(data\_Meso)

data\_mit = np.load(mit\_dict[vid\_name+".avi"])

Loads the **MIT .npy file** for the video.

data\_mit\_set.append(data\_mit)

if vid\_name.find('\_')!=-1:

data\_y\_set.append(1)

print(1)

else:

data\_y\_set.append(0)

print(0)

If **video name contains \_**, it is **fake (1)**.

Otherwise, it is **real (0)**. (Based on the naming convention of videos in the dataset)

data\_name\_set.append(vid\_name)

print(" ", np.shape(data\_Meso\_set))

print(" ", np.shape(data\_mit\_set))

print(" ", np.shape(data\_y\_set))

print(" ", np.shape(data\_name\_set))

Prints the **size of each dataset** being collected.

print("SAVING .... ")

np.save(save\_path+"Meso.npy", data\_Meso\_set)

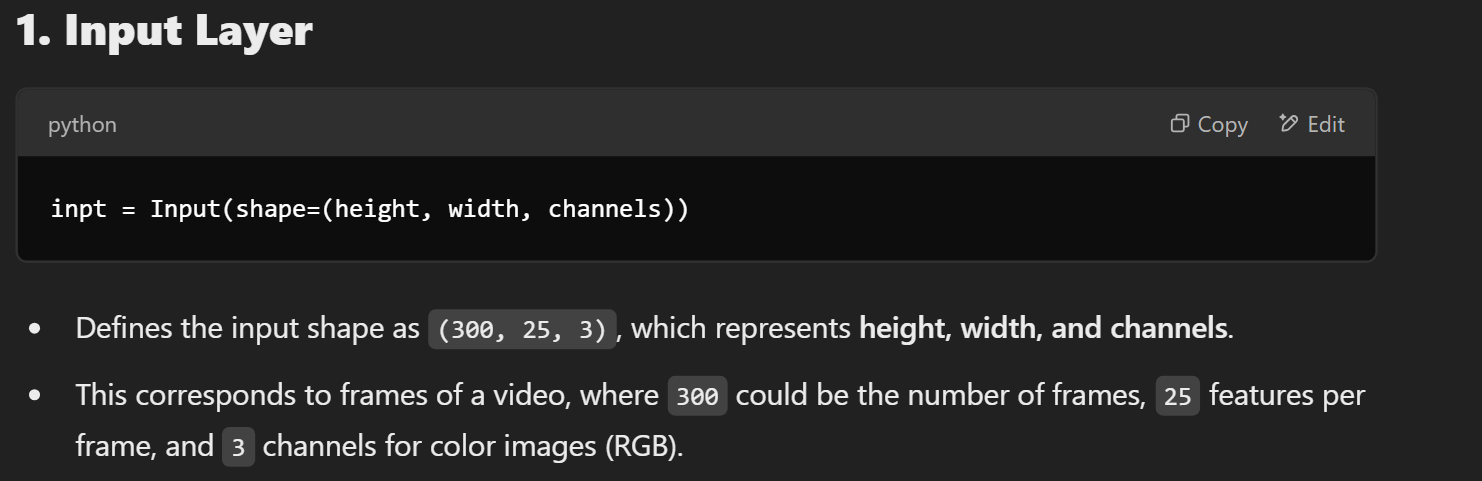
np.save(save\_path+"mit.npy", data\_mit\_set)

np.save(save\_path+"y.npy", data\_y\_set)

np.save(save\_path+"name.npy", data\_y\_set)

**Saves everything as .npy files** for later use

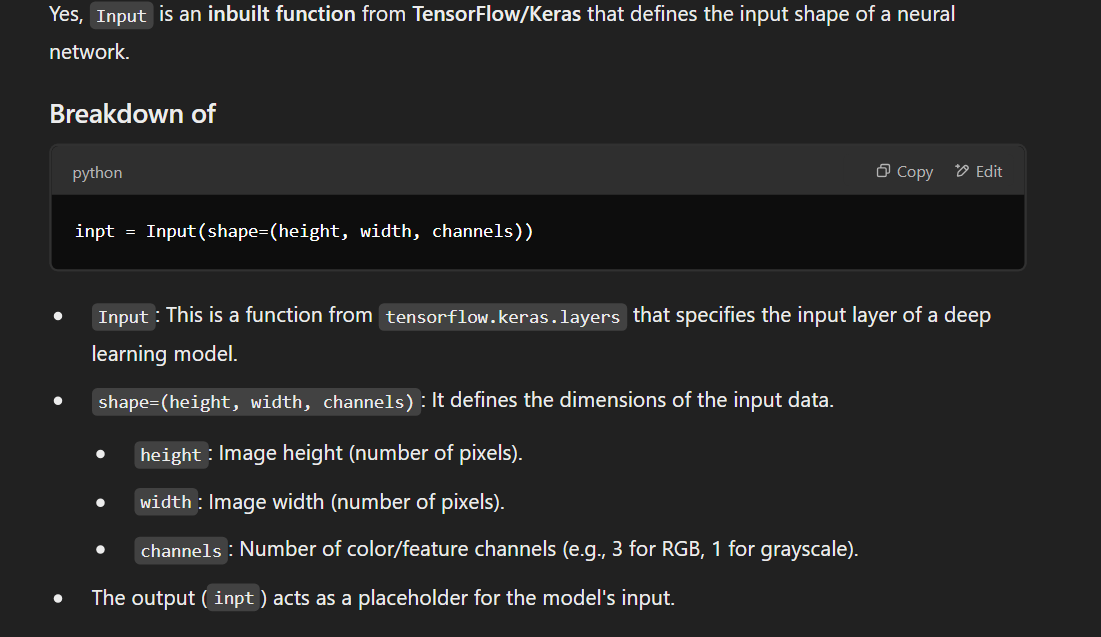
Layers used in training:

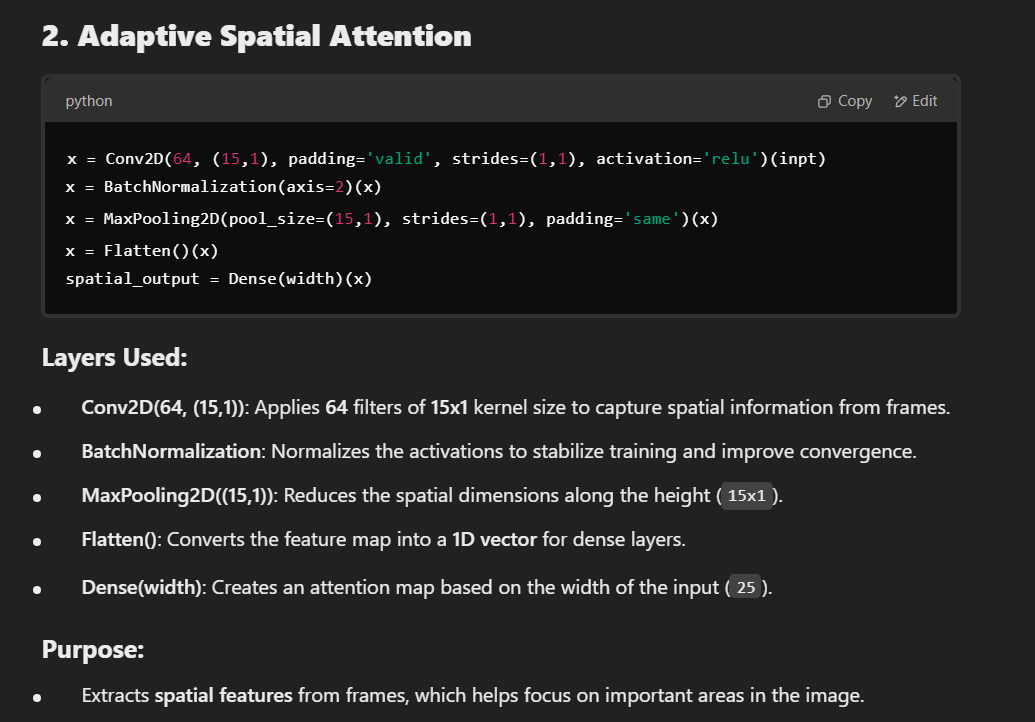


In simple terms, **channels** refers to the number of color components or feature maps in the input image.

For example:

* A **grayscale** image has **1 channel** (only intensity values).
* A **color (RGB)** image has **3 channels** (Red, Green, and Blue).
* If you're working with **feature maps** from a neural network, channels represent the depth of the feature representation at that layer.





(inpt) at the end means the convolution operation is applied to the input tensor inpt.