Code

from keras.preprocessing.image import img\_to\_array

import imutils

import cv2

from keras.models import load\_model

import numpy as np

explanation:

1. **keras.preprocessing.image**: This library provides a set of image preprocessing utilities, including functions like **img\_to\_array** that convert an image (PIL image or a numpy array) into a NumPy array.
2. **imutils**: This is a library that provides convenience functions for basic image processing tasks. It includes functions for resizing, rotating, and cropping images, among other operations. It is often used in conjunction with OpenCV for computer vision tasks.
3. **cv2** (OpenCV): OpenCV (Open Source Computer Vision Library) is a popular computer vision library that provides a wide range of functions for image and video processing. It is widely used for tasks such as image manipulation, object detection, and video analysis.
4. **keras.models**: This library is part of Keras, a high-level neural networks API. The **models** module provides functions for creating and manipulating deep learning models, including loading pre-trained models using **load\_model**.
5. **numpy**: This is a fundamental library for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a large collection of mathematical functions to operate on these arrays efficiently.

#parameters for loading data and images

detection\_model\_path = r"C:\Users\PANDU\Dropbox\My PC (LAPTOP-NJRNIHCN)\Downloads\face-classification-main\face-classification-main\faceDetection\models\haarcascade\haarcascade\_frontalface\_default.xml"

emotion\_model\_path = r"C:\Users\PANDU\Dropbox\My PC (LAPTOP-NJRNIHCN)\Downloads\face-classification-main\face-classification-main\models\emotionModel.hdf5"

explanation :

1. **detection\_model\_path**: This parameter specifies the file path to the Haar cascade XML file used for face detection. Haar cascades are machine learning-based classifiers that are commonly used for object detection, particularly for detecting faces. The file **haarcascade\_frontalface\_default.xml** contains pre-trained data for detecting frontal faces. The path specified should point to the location where this XML file is stored on your local machine.
2. **emotion\_model\_path**: This parameter specifies the file path to the emotion model in HDF5 format. The HDF5 format is a data model and file format used to store and manage large amounts of numerical data efficiently. In this case, it refers to a pre-trained model for emotion classification. The path specified should point to the location where the HDF5 file **emotionModel.hdf5** is stored on your local machine.

# hyper-parameters for bounding boxes shape

# loading models

face\_detection = cv2.CascadeClassifier(detection\_model\_path)

emotion\_classifier = load\_model(emotion\_model\_path, compile=False)

EMOTIONS = ["angry","disgust","fear","happy","surprise","sad","gray\_pout"]

Explanation:

1. **# hyper-parameters for bounding boxes shape**: This comment indicates that the following lines define hyper-parameters related to the shape of bounding boxes. Hyper-parameters are variables that are set before the start of the learning process and determine the behavior of the model.
2. **# loading models**: This comment indicates that the following lines are responsible for loading the models required for face detection and emotion classification.
3. **face\_detection = cv2.CascadeClassifier(detection\_model\_path)**: This line creates an instance of the **CascadeClassifier** class from the OpenCV library using the Haar cascade XML file specified by **detection\_model\_path**. This classifier will be used for face detection. The **CascadeClassifier** class is trained to detect objects in images using cascading classifiers, which are a series of simpler classifiers that progressively filter out negative regions in the image.
4. **emotion\_classifier = load\_model(emotion\_model\_path, compile=False)**: This line loads the emotion classification model from the HDF5 file specified by **emotion\_model\_path** using the **load\_model** function from Keras. The **load\_model** function is used to load a pre-trained model saved in the HDF5 format. The resulting model will be assigned to the variable **emotion\_classifier**, which will be used for emotion classification.
5. **EMOTIONS = ["angry","disgust","fear","happy","surprise","sad","gray\_pout"]**: This line creates a list **EMOTIONS** that contains the labels corresponding to different emotions. The list elements represent different emotions such as angry, disgust, fear, happy, surprise, sad, and gray\_pout. These labels are used later in the code for mapping the predicted emotion index to its corresponding label.

# starting video streaming

cv2.namedWindow('your\_face')

camera = cv2.VideoCapture(0)

explanation ::

1. **# starting video streaming**: This comment indicates that the following lines are responsible for starting the video streaming process.
2. **cv2.namedWindow('your\_face')**: This line creates a named window with the title "your\_face" using the **namedWindow** function from OpenCV. This window will be used to display the video stream and any subsequent visualizations or overlays.
3. **camera = cv2.VideoCapture(0)**: This line initializes a video capture object named **camera** using the **VideoCapture** function from OpenCV. The argument **0** passed to the function indicates that the default camera device should be used for capturing the video stream. If you have multiple cameras connected to your system, you can specify a different index to select a specific camera. For example, **1** for the second camera, **2** for the third camera, and so on.

while True:

frame = camera.read()[1]

print(type(frame))

#reading the frame

frame = imutils.resize(frame,width=300)

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

explanation :

1. **while True:**: This creates an infinite loop, which is commonly used in video processing to continuously read frames and perform operations until the loop is manually interrupted.
2. **frame = camera.read()[1]**: This line captures a frame from the video stream using the **read** method of the **camera** object. The **read** method returns a tuple, and by accessing the second element **[1]**, we assign the captured frame to the variable **frame**.
3. **print(type(frame))**: This line prints the type of the **frame** variable, which can be useful for debugging or understanding the data type of the captured frame.
4. **frame = imutils.resize(frame,width=300)**: This line resizes the **frame** to a width of 300 pixels using the **resize** function from the **imutils** library. Resizing frames can be helpful for standardizing the input size for further processing or for optimizing performance.
5. **gray = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)**: This line converts the resized **frame** from the previous line to grayscale using the **cvtColor** function from OpenCV. The **cvtColor** function is used to convert an image from one color space to another. In this case, it converts the frame from the default BGR color space to grayscale (represented by **cv2.COLOR\_BGR2GRAY**).

faces = face\_detection.detectMultiScale(gray,scaleFactor=1.1,minNeighbors=5,minSize=(30,30),flags=cv2.CASCADE\_SCALE\_IMAGE)

explanation :

1. **faces = face\_detection.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5, minSize=(30,30), flags=cv2.CASCADE\_SCALE\_IMAGE)**: This line detects faces in the grayscale image **gray** using the **detectMultiScale** method of the **face\_detection** object (which is an instance of **cv2.CascadeClassifier**).
   * **gray**: This is the grayscale image on which the face detection algorithm will be applied.
   * **scaleFactor=1.1**: This parameter specifies how much the image size is reduced at each image scale. A smaller scale factor will result in the detection of smaller faces, but it may increase false positives. Conversely, a larger scale factor will increase the chance of detecting larger faces but may miss smaller faces.
   * **minNeighbors=5**: This parameter specifies the minimum number of neighbors (detections) required for a detected region to be considered as a face. Higher values will result in fewer detections but with higher confidence, while lower values may lead to more detections but with a higher chance of false positives.
   * **minSize=(30,30)**: This parameter specifies the minimum size of the detected face. Any detected region smaller than this size will be ignored. It helps filter out small detections that are likely to be false positives.
   * **flags=cv2.CASCADE\_SCALE\_IMAGE**: This parameter is used to specify the scaling method for the detected regions. In this case, the **cv2.CASCADE\_SCALE\_IMAGE** flag indicates that the image scale is equal to 1. This is the recommended setting for face detection.

The **detectMultiScale** method returns a list of rectangles representing the detected faces. Each rectangle represents the coordinates and dimensions of a detected face region. These rectangles can be used for further processing, such as drawing bounding boxes around the detected faces or extracting the facial regions for further analysis. In this case, the detected face regions are stored in the **faces** variable.

canvas = np.zeros((250, 300, 3), dtype="uint8")

frameClone = frame.copy()

explanation :

1. **canvas = np.zeros((250, 300, 3), dtype="uint8")**: This line creates a NumPy array called **canvas** using the **np.zeros** function. The array has a shape of (250, 300, 3), meaning it has a height of 250 pixels, a width of 300 pixels, and 3 channels (representing the RGB color channels). The **dtype="uint8"** parameter specifies that the array should have 8-bit unsigned integer data type, which is commonly used to represent pixel values in images.
2. **frameClone = frame.copy()**: This line creates a copy of the **frame** variable using the **copy** method. It is important to create a copy because subsequent operations or modifications made to **frameClone** will not affect the original **frame** variable. This is useful if you want to perform operations on the copied frame without altering the original frame, such as drawing bounding boxes or overlays.

In the given code snippet, **canvas** is an empty image (all black) with a specific shape, and **frameClone** is a copy of the original frame. These variables can be used for drawing or overlaying visual elements on the frame, such as bounding boxes or text, without modifying the original frame.

if len(faces) > 0:

faces = sorted(faces, reverse=True,

key=lambda x: (x[2] - x[0]) \* (x[3] - x[1]))[0]

(fX, fY, fW, fH) = faces

#

1. **if len(faces) > 0:**: This line checks if there are any detected faces in the **faces** variable. If the length of the **faces** list is greater than 0, it means that at least one face has been detected.
2. **faces = sorted(faces, reverse=True, key=lambda x: (x[2] - x[0]) \* (x[3] - x[1]))[0]**: This line sorts the **faces** list in descending order based on the area of each face's bounding box. The **key=lambda x: (x[2] - x[0]) \* (x[3] - x[1])** part defines a lambda function that calculates the area of each bounding box **(x[2] - x[0]) \* (x[3] - x[1])**. The **sorted** function then sorts the list based on this area value in reverse order. Finally, **[0]** is used to select the first (largest) face from the sorted list. So, after this line, the **faces** variable will contain the coordinates and dimensions of the largest face.
3. **(fX, fY, fW, fH) = faces**: This line unpacks the values from the **faces** variable into separate variables **fX**, **fY**, **fW**, and **fH**. These variables represent the x-coordinate, y-coordinate, width, and height of the bounding box of the largest face, respectively. So, **fX** is the x-coordinate of the top-left corner of the bounding box, **fY** is the y-coordinate of the top-left corner, **fW** is the width of the bounding box, and **fH** is the height of the bounding box.

roi = gray[fY:fY + fH, fX:fX + fW]

roi = cv2.resize(roi, (64,64))

roi = roi.astype("float") / 255.0

roi = img\_to\_array(roi)

roi = np.expand\_dims(roi, axis=0)

preds = emotion\_classifier.predict(roi)[0]

emotion\_probability = np.max(preds)

label = EMOTIONS[preds.argmax()]

Explanation :

1. **roi = gray[fY:fY + fH, fX:fX + fW]**: This line extracts a region of interest (ROI) from the grayscale image **gray**. The ROI corresponds to the bounding box of the detected face. It is defined by the coordinates **fX**, **fY** (top-left corner) and the dimensions **fW**, **fH** (width and height) of the bounding box.
2. **roi = cv2.resize(roi, (64,64))**: This line resizes the ROI image to a fixed size of 64x64 pixels using the **resize** function from OpenCV. Resizing the ROI image to a consistent size is often necessary to ensure compatibility with the input size expected by the emotion classification model.
3. **roi = roi.astype("float") / 255.0**: This line converts the pixel values in the ROI image from integers to floats and scales them to the range [0, 1]. This normalization step is commonly performed to ensure that the input values are within a certain range, which can improve the performance of the model.
4. **roi = img\_to\_array(roi)**: This line converts the ROI image from a NumPy array to a Keras-compatible array. It converts the image to an array representation, preserving its spatial structure.
5. **roi = np.expand\_dims(roi, axis=0)**: This line adds an extra dimension to the ROI array. It reshapes the array to have shape **(1, height, width, channels)**, where **height**, **width**, and **channels** correspond to the dimensions of the ROI image.
6. **preds = emotion\_classifier.predict(roi)[0]**: This line feeds the ROI to the emotion classification model (**emotion\_classifier**) and makes a prediction. The **predict** method returns a probability distribution over the different emotion classes. The **[0]** indexing is used to access the first (and only) prediction from the resulting array.
7. **emotion\_probability = np.max(preds)**: This line calculates the maximum probability value from the predicted probability distribution. It represents the predicted probability of the most likely emotion class.
8. **label = EMOTIONS[preds.argmax()]**: This line determines the predicted emotion label by selecting the emotion class with the highest probability. The **argmax** function returns the index of the maximum value in the **preds** array, which corresponds to the index of the most likely emotion class. The **label** variable is then assigned the corresponding emotion label from the **EMOTIONS** list.

for (i, (emotion, prob)) in enumerate(zip(EMOTIONS, preds)):

text = "{}: {:.2f}%".format(emotion, prob \* 100)

w = int(prob \* 300)

cv2.rectangle(canvas, (5, (i \* 35) + 5),

(w, (i \* 35) + 35), (0, 0, 255), -1)

cv2.putText(canvas, text, (10, (i \* 35) + 23),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.45,

(255, 255, 255), 2)

cv2.putText(frameClone, label, (fX, fY - 10),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.45, (0, 0, 255), 2)

cv2.rectangle(frameClone, (fX, fY), (fX + fW, fY + fH),

(0, 0, 255), 2)

Explanation:

1. **for (i, (emotion, prob)) in enumerate(zip(EMOTIONS, preds)):**: This line iterates over the **EMOTIONS** list and the **preds** array simultaneously using the **zip** function. The **enumerate** function is used to obtain the index **i** and the corresponding values **emotion** and **prob** from the iteration.
2. **text = "{}: {:.2f}%".format(emotion, prob \* 100)**: This line creates a formatted text string that combines the **emotion** and **prob** values. The emotion label and the corresponding probability are displayed in the format "emotion: probability%".
3. **w = int(prob \* 300)**: This line calculates the width **w** of the colored rectangle that represents the probability of the emotion. The width is calculated by multiplying the probability by 300 (which represents the maximum width) and converting it to an integer value.
4. **cv2.rectangle(canvas, (5, (i \* 35) + 5), (w, (i \* 35) + 35), (0, 0, 255), -1)**: This line draws a filled rectangle on the **canvas** image to represent the probability of the emotion. The rectangle is positioned at **(5, (i \* 35) + 5)** and has a width of **w** and a height of 30 pixels. The color of the rectangle is specified as **(0, 0, 255)**, which corresponds to blue, and the parameter **-1** indicates that the rectangle should be filled.
5. **cv2.putText(canvas, text, (10, (i \* 35) + 23), cv2.FONT\_HERSHEY\_SIMPLEX, 0.45, (255, 255, 255), 2)**: This line overlays the text on the **canvas** image. It displays the formatted **text** string at the position **(10, (i \* 35) + 23)** using the **cv2.putText** function. The font face is specified as **cv2.FONT\_HERSHEY\_SIMPLEX**, the font scale is set to **0.45**, and the color is **(255, 255, 255)** (white). The parameter **2** specifies the thickness of the text.
6. **cv2.putText(frameClone, label, (fX, fY - 10), cv2.FONT\_HERSHEY\_SIMPLEX, 0.45, (0, 0, 255), 2)**: This line overlays the predicted emotion label on the **frameClone** image. It displays the **label** text at the position **(fX, fY - 10)** (just above the bounding box of the detected face). The font face, font scale, and color are set similar to the previous line.
7. **cv2.rectangle(frameClone, (fX, fY), (fX + fW, fY + fH), (0, 0, 255), 2)**: This line draws a rectangle on the **frameClone** image to represent the bounding box of the detected face. The rectangle is positioned at **(fX, fY)** (top-left corner) and has a width of **fW** and a height of **fH**. The color of the rectangle is specified as **(0, 0, 255)**, which corresponds to red, and the parameter **2** specifies the thickness of the rectangle.

cv2.imshow('your\_face', frameClone)

cv2.imshow("Probabilities", canvas)

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

break

camera.release()

cv2.destroyAllWindows()

explanation :

1. **cv2.imshow('your\_face', frameClone)**: This line displays the **frameClone** image in a window with the title "your\_face" using the **cv2.imshow** function. It shows the frame with the detected face and the predicted emotion label.
2. **cv2.imshow("Probabilities", canvas)**: This line displays the **canvas** image in a separate window with the title "Probabilities". The **canvas** image shows colored rectangles representing the probabilities of different emotions.
3. **if cv2.waitKey(1) & 0xFF == ord('q'):**: This line waits for a key press using the **cv2.waitKey** function. It checks if the pressed key is 'q' by comparing the return value with **ord('q')**. If the 'q' key is pressed, the condition evaluates to **True**, and the loop is exited using the **break** statement.
4. **camera.release()**: This line releases the camera resources by calling the **release** method on the **camera** object. It is necessary to release the camera before the program exits to free up system resources.
5. **cv2.destroyAllWindows()**: This line closes all the OpenCV windows that were opened during the execution of the program using the **cv2.destroyAllWindows** function. It ensures that all the windows are closed when the program finishes