Parallelize Facial Recognition

A PROJECT REPORT

Submitted by

20BCE0563-Boggavarapu Ch N V Shivani
20BCE0885-Richa Kiran
20BCE0908-Arya Dubey

Course Code: CSE 4001

Course Title: Parallel and Distributed Computing

Under the guidance of

Dr. S. AntoAssociate Professor, SCOPE,

VIT , Vellore.

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

April, 2023

TABLE OF CONTENTS

1. INTRODUCTION

Face recognition is a popular and rapidly growing field in computer vision, with numerous applications in security, access control, and user identification. In this project, we have implemented a real-time face recognition system using OpenCV (cv2), Keras, and NumPy libraries in Python.

We have used a pre-trained Convolutional Neural Network (CNN) model in Keras for recognizing faces. Our implementation employs the Haar Cascade Classifier to detect faces in real-time video streams, and a Threadpool Executor for parallelizing the process of face detection and recognition.

The face detection and recognition process is executed in parallel, with each frame of the video stream being processed concurrently by different threads. The recognition process involves cropping the face region from the frame and resizing it to a fixed size. Then, the pre-trained Keras model is used to predict the class label of the face from a set of predefined classes.

The recognition results are displayed in real-time on the video stream with the predicted class label and the probability of the prediction. The code is optimized for performance, and the use of the Threadpool Executor ensures that the process of face detection and recognition is executed efficiently and quickly, even for large datasets.

Overall, this project provides an efficient and scalable solution for face recognition in real-time video streams, with applications in security, access control, and user identification.

2. LITERATURE SURVEY

2.1. Literature Survey

1. A massively parallel algorithm for local binary pattern based face recognition:
   The paper describes the parallelization of the LBP algorithm using the Message Passing Interface (MPI) and OpenMP technologies. The authors demonstrate the efficiency and scalability of their parallel LBP algorithm on multiple datasets and show that their approach can achieve significant speedups over sequential implementations. The paper also highlights the potential of the parallel LBP algorithm to improve the performance of other facial recognition methods that use LBP features. The authors suggest that their parallel algorithm can be used to optimize other LBP-based techniques, such as the Local Derivative Pattern (LDP) method, and to extend the applicability of LBP-based face recognition systems to real-time applications. Overall, the paper presents a novel approach to parallelizing LBP-based face recognition, and demonstrates the potential of parallel computing to improve the performance of face recognition systems.[4]

2. A Parallel Approach for Real-Time Face Recognition from a Large Database:
   The authors propose a parallel approach for real-time face recognition from a large database, where the face images are processed in parallel to achieve high recognition accuracy and processing speed. The proposed approach consists of three main phases: face detection, feature extraction, and classification. The face detection is performed using the Viola-Jones algorithm, while the Local Binary Patterns (LBP) technique is used for feature extraction. Finally, the classification is performed using the k-Nearest Neighbor (k-NN) algorithm. The proposed approach is implemented on a multicore processor using OpenMP, a popular parallel programming framework. The experimental results show that the proposed parallel approach achieves higher recognition accuracy and processing speed compared to the sequential approach. The authors also analyze the impact of varying the number of threads on the recognition accuracy and processing speed and conclude that the performance of the parallel approach is highly dependent on the number of available cores. In summary, the paper "A Parallel Approach for Real-Time Face Recognition from a Large Database" presents a novel parallel approach for face recognition from a large database, which achieves high recognition accuracy and processing speed. The use of OpenMP as a parallel programming framework makes it easy to implement the proposed approach on multicore processors. The future direction of this research could be to investigate the performance of the proposed approach on other parallel architectures such as GPUs and FPGAs and to explore the use of deep learning techniques for face recognition.[3]

3. Research on Parallelization of Face Recognition :

The paper "Research on Parallelization of Face Recognition" proposes a parallelized face recognition algorithm that can improve the efficiency of traditional face recognition methods. The authors use the popular OpenCV library along with the MPI (Message Passing Interface) protocol for parallelization.
The paper first discusses the preprocessing steps involved in face recognition, such as face detection and feature extraction. The authors use the Haar-like feature-based cascade classifier for face detection and the PCA (Principal Component Analysis) method for feature extraction. The parallelization is achieved using the MPI protocol, which is a popular message-passing standard used in high-performance computing. The authors use MPI to distribute the workload across multiple processors and achieve parallelism. They also use OpenCV to implement the face recognition algorithm. The authors test their parallelized face recognition algorithm on the ORL face database, which contains 400 images of 40 different subjects. They compare the performance of their parallelized algorithm with the traditional sequential algorithm and show that the parallelized algorithm can achieve significant speedup. The paper concludes that the proposed parallelized face recognition algorithm can significantly improve the efficiency of traditional face recognition methods.Overall, "Research on Parallelization of Face Recognition" provides valuable insights into the use of parallelization techniques to improve the efficiency of face recognition algorithms. The paper also highlights the importance of using popular libraries and protocols such as OpenCV and MPI for implementing parallelized algorithms. [2]

4. <u>Parallelizing Image Processing Algorithms for Face Recognition on Multicore Platforms:</u>
   The paper by Mia and Assaduzzaman focuses on the parallelization of image processing algorithms for face recognition on multicore platforms. The authors start by discussing the importance of face recognition and the need for efficient and accurate algorithms to perform this task. They then move on to discuss the challenges associated with parallelizing image processing algorithms and the techniques that can be used to overcome these challenges. The authors propose a parallel implementation of the Viola-Jones algorithm for face detection and the Eigenface algorithm for face recognition using OpenMP and MPI. They then evaluate the performance of their implementation on a multicore platform and compare it with a sequential implementation. The results of the evaluation show that the parallel implementation is able to achieve significant speedup over the sequential implementation for both face detection and face recognition. The authors also discuss the limitations of their approach and suggest areas for further research. Overall, the paper provides a valuable contribution to the field of face recognition by demonstrating the potential benefits of parallelization on multicore

platforms. The study also highlights the need for further research in this area to develop more efficient and accurate algorithms for face recognition. [1]

2.2. <u>Problem Definition</u>

The project aims to solve the problem of identifying individuals in real-time video streams, with applications in security, access control, and user identification. Traditional methods for face recognition involve manual detection and recognition, which can be time-consuming and error-prone. The proposed solution aims to automate this process using computer vision techniques and machine learning algorithms.

The face recognition system should be accurate, efficient, and scalable, capable of recognizing faces in real-time video streams with high precision and recall rates. The system should also be able to handle large datasets of faces, with the ability to recognize faces from a predefined set of classes or labels.

Overall, the problem definition for the face recognition project is to develop an accurate and efficient system for real-time face recognition, with applications in security, access control, and user identification.

3. OVERVIEW OF THE WORK

3.1. <u>Objectives of the Project</u>

The largest problem with current facial recognition systems, in addition to inadequate robustness, is a lack of acceptable performance. Conventional facial recognition techniques require extensive computationally intensive calculations to provide moderately reliable results.In real-time image processing, processing should be done with high speed on rapid sequence of images or on a single image. Such high-speed image processing can be handled using parallel or distributed image processing. The objective of our project was to implement a parallel version of LBP Face Recognition, using OpenMP pragmas to parallelize the codes to increase the robustness of the systems against different factors.

3.2. Software Requirements

- Python
- OpenCV
- Keras
- Tensorflow

3.3. Hardware Requirements

- PC, Mac or laptop with x86-64 (64-bit) compatible processors.

4. SYSTEM DESIGN

4.1. Algorithm

Our project uses **haar cascade algorithm** for face detection. The haarcascade_frontalface_default.xml file is a pre-trained Haar Cascade Classifier file used for face detection in OpenCV, a popular computer vision library. Haar Cascade is a machine learning-based approach for object detection. It was proposed by Viola and Jones in their seminal 2001 paper, "Rapid Object Detection using a Boosted Cascade of Simple Features."
The Haar Cascade classifier is trained to identify an object of interest by analyzing features of positive and negative samples of the object. In the case of face detection, the classifier is trained on thousands of positive and negative images of faces to learn the features that are unique to a face. Once trained, the classifier can be used to detect faces in new images or video streams.
The haarcascade_frontalface_default.xml file is the default pre-trained classifier provided by OpenCV for detecting faces in images or video streams.
The algorithm is as follows:

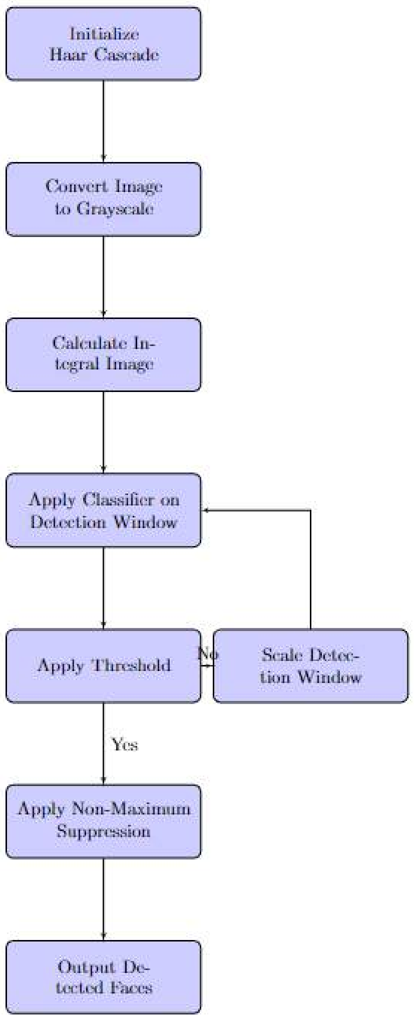
1. Convert the image to grayscale: The algorithm starts by converting the input image to grayscale, which makes the computation faster and easier.
2. Detection of Haar-like features: The next step is to detect Haar-like features in the image. Haar-like features are patterns in the image that can be used to identify objects. These features are rectangular in shape and can be either black or white.

They are identified by calculating the difference between the sum of the pixels inside the rectangle and the sum of the pixels outside the rectangle.

3. Integral Image: The integral image is calculated from the grayscale image, which is used to speed up the calculation of the Haar-like features. The integral image is a two-dimensional array that stores the sum of all pixels above and to the left of a given pixel.
4. Adaboost: Adaboost is a machine learning algorithm that is used to train a classifier to recognize objects in the image. It is used to select the most relevant Haar-like features from the set of all possible features.
5. Cascading Classifiers: To improve the detection speed and accuracy, multiple classifiers are combined in a cascading structure. The output of one classifier is used as the input to the next classifier. If an object is not detected in one stage, it is discarded and the next stage is executed. This helps to reduce the number of false positives and speed up the detection process.
6. Detection and Output: After the image has been processed by the cascading classifiers, the final output is obtained. The location of the object is determined by the bounding box around the detected object.

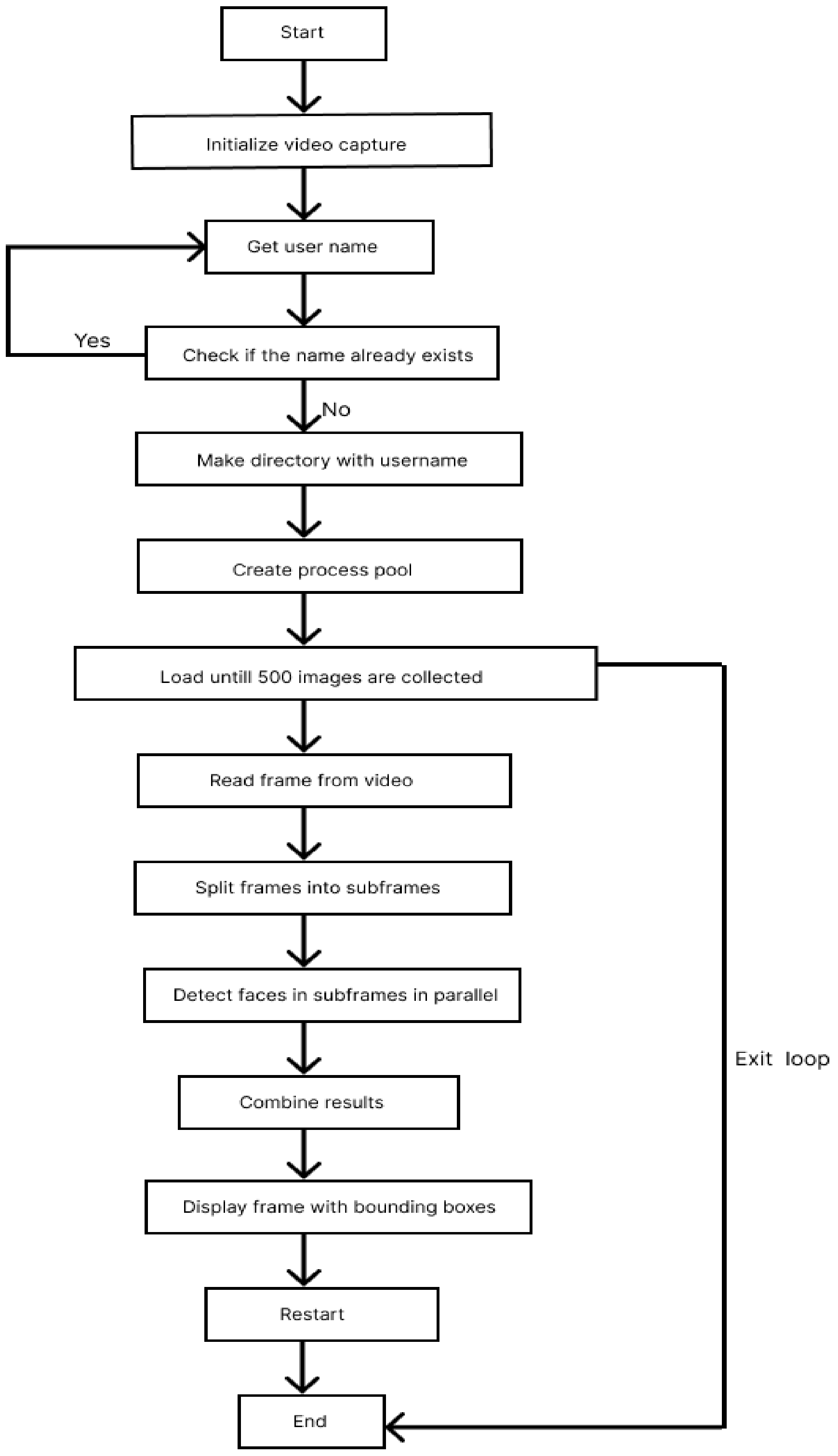
4.2. Block Diagrams

- Haar Cascade algorithm:

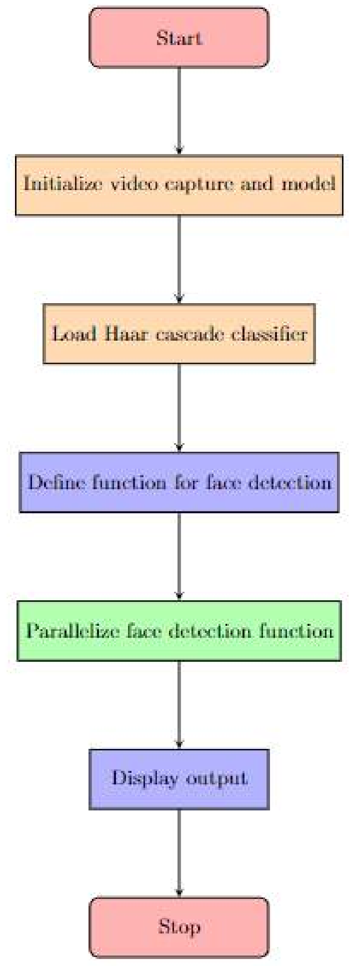


Flowchart of the parallelized code:

DataCollect.py


Start
Initialize video capture
Get user name
Yes
Check if the name already exists
No
Make directory with username
Create process pool
Load untill 500 images are collected
Read frame from video
Split frames into subframes
Detect faces in subframes in parallel
Exit loop
Combine results
Display frame with bounding boxes
Restart
End

Test.py


Start
Initialize video capture and model
Load Haar cascade classifier
Define function for face detection
Parallelize face detection function
Display output
Stop

5. Implementation Details:

5.1. Code Description and Explanation:

Datacollect.py

1. Import the required modules: cv2, os, and multiprocessing as mp.
2. Initialize a video capture object using cv2.VideoCapture() and set its index to 0.
3. Load the haarcascade_frontalface_default.xml file using cv2.CascadeClassifier().
4. Set a counter variable count to 0.
5. Take user input for name and convert it to lowercase using str(input("Enter Your Name: ")).lower().
6. Check if a directory with the given name exists in the "images" folder using

os.path.exists().

7. If the directory exists, prompt the user to enter a different name until a unique name is entered.
8. If the directory does not exist, create the directory using os.makedirs().
9. Start an infinite loop to read frames from the video capture object.
10. Within the loop, read the next frame using video.read().
11. Detect faces in the frame using facedetect.detectMultiScale() and pass the frame, scaling factor 1.3, and minimum neighbor count 5 as parameters.
12. For each detected face, increment the counter variable count by 1.
13. Construct a filename using './images/'+nameID+'/'+ str(count) + '.jpg'.
14. Save the detected face as an image using cv2.imwrite() and pass the filename and a cropped image of the face using frame[y:y+h,x:x+w] as parameters.
15. Draw a green rectangle around the detected face using cv2.rectangle() and pass the frame, the top-left corner coordinates (x,y), the bottom-right corner coordinates (x+w, y+h), and the RGB color (0,255,0) as parameters.
16. Display the frame in a window named "WindowFrame" using cv2.imshow().
17. Wait for a key press event for 1 millisecond using cv2.waitKey(1).
18. If the counter variable count is greater than 500, break the loop.
19. Release the video capture object using video.release() and destroy all windows using cv2.destroyAllWindows().

- **EXPLANATION:**
  The code uses the Haar Cascade Classifier to detect faces in a live video stream captured by a webcam. The user is prompted to enter their name, and a directory with their name is created to save the detected face images. The code uses multiprocessing to speed up the face detection process by dividing the video frames into multiple parts and processing them in parallel. The detect_faces function takes a frame, nameID, and count as input and uses the cascade classifier to detect faces in the frame. It saves the detected faces as images in the user's directory and draws rectangles around the detected faces in the frame. The while loop continuously reads frames from the video stream and passes them to the detect_faces function to detect faces in parallel. The program exits after detecting 500 faces or when the user presses the 'q' key to quit.

Test.py

1. Import required libraries:
   - cv2
   - keras.models.load_model
   - numpy

    - concurrent.futures.ThreadPoolExecutor
2. Load the pre-trained face detection classifier from 'haarcascade_frontalface_default.xml' file using cv2.CascadeClassifier() and store it in 'facedetect' variable.
3. Create a video capture object using cv2.VideoCapture(0) and set the frame width and height to 640 and 480 respectively using cap.set(3, 640) and cap.set(4, 480).
4. Set the font style to cv2.FONT_HERSHEY_COMPLEX.
5. Set the numpy print options to suppress scientific notation using np.set_printoptions(suppress=True).
6. Load the pre-trained keras model from 'keras_Model.h5' file using keras.models.load_model() and store it in 'model' variable.
7. Define a function 'get_className()' that takes class number as input and returns the corresponding class name.
8. Define a function 'process_frame()' that takes a frame as input and performs the following steps:
    - Detect faces in the input frame using 'facedetect.detectMultiScale()' and store the coordinates in 'faces'.
    - For each face, crop the face region from the input frame using 'frame[y:y+h,x:x+h]' and resize it to 224x224 using 'cv2.resize()'.
    - Reshape the image to (1, 224, 224, 3) using 'img.reshape()'.
    - Predict the class of the cropped face image using 'model.predict()' and store the class index in 'classIndex'.
    - Calculate the probability of the predicted class using 'np.amax()'.
    - Draw a green rectangle around the detected face using 'cv2.rectangle()'.
    - Display the predicted class name and probability on top of the rectangle using 'cv2.putText()'.
    - Return the processed frame.
9. Create a 'ThreadPoolExecutor' object with max_workers set to 4 using 'ThreadPoolExecutor(max_workers=4)' This is the parallelized part of the program where the task is divided among multiple threads.
10. Start an infinite loop to capture video frames using 'cap.read()'.
    - Submit the current frame to the 'process_frame()' function using 'executor.submit()' and store the future object in 'future'.
    - Get the processed frame from the future object using 'future.result()' and store it in 'imgOrignal'.
    - Display the processed frame using 'cv2.imshow()'.
    - Wait for a key press for 1 millisecond using 'cv2.waitKey(1)' and store the key code in 'k'.
    - If the key pressed is 'q', break out of the loop.

11. Release the video capture object using 'cap.release()' and destroy all open windows using 'cv2.destroyAllWindows()'.

- **EXPLANATION:**
  The code performs face recognition using a pre-trained model. The Haar Cascade Classifier is used to detect faces in the video stream, and then each face is cropped and resized to fit into a 224x224 pixel image. The pre-trained model is then used to predict the class of the image, which corresponds to a specific person. The class label and the probability score are then displayed on the screen along with a bounding box around the face. The processing is done in parallel using a thread pool of 4 threads. The video stream is read from the default camera device (index 0), and the program continues until the user presses 'q' to quit the application.

5.2. SOURCE CODE

- **Haarcascade code:**

- **DATACOLLECT.PY**

```
import cv2
import os
import multiprocessing as mp

def detect_faces(frame, nameID, count):
  facedetect=cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
  faces=facedetect.detectMultiScale(frame,1.3, 5)
  for x,y,w,h in faces:
    count=count+1
    name='./images/'+nameID+'/'+ str(count) + '.jpg'
    print("Creating Images........." +name)
    cv2.imwrite(name, frame[y:y+h,x:x+w])
    cv2.rectangle(frame, (x,y), (x+w, y+h), (0,255,0), 3)
  return frame, count

if __name__ == '__main__':
  video=cv2.VideoCapture(0)

  count=0

  nameID=str(input("Enter Your Name: ")).lower()
```

```
    path='images/'+nameID

    isExist = os.path.exists(path)

    if isExist:
        print("Name Already Taken")
        nameID=str(input("Enter Your Name Again: "))
    else:
        os.makedirs(path)

    # create a pool of processes
    num_processes = 4  # set the number of processes you want to create
    pool = mp.Pool(num_processes)

    while True:
        ret,frame=video.read()

        # create a list of arguments to pass to the detect_faces function
        args_list = [(frame[i::num_processes], nameID, count) for i in range(num_processes)]

        # run detect_faces on each process in parallel
        results = pool.starmap(detect_faces, args_list)

        # combine the results
        count = sum([res[1] for res in results])
        frame = results[0][0]  # assume all frames are the same for simplicity

        cv2.imshow("WindowFrame", frame)
        cv2.waitKey(1)
        if count>500:
            break

    video.release()
    cv2.destroyAllWindows()
```

- **TEST.PY**

```
import cv2
from keras.models import load_model
import numpy as np
from concurrent.futures import ThreadPoolExecutor
```

```
facedetect = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')

cap=cv2.VideoCapture(0)
cap.set(3, 640)
cap.set(4, 480)
font=cv2.FONT_HERSHEY_COMPLEX


np.set_printoptions(suppress=True)
model = load_model("keras_Model.h5")


def get_className(classNo):
   if classNo==0:
      return "shivani"
   elif classNo==1:
      return "tom"


def process_frame(frame):
   faces = facedetect.detectMultiScale(frame, 1.3)
   for x,y,w,h in faces:
      crop_img = frame[y:y+h,x:x+h]
      img = cv2.resize(crop_img, (224,224))
      img = img.reshape(1, 224, 224, 3)
      prediction = model.predict(img)
      classIndex = np.argmax(model.predict(img), axis=1)
      print("CLASS PREDICTIONS: ",prediction)
      probabilityValue = np.amax(prediction)
      cv2.rectangle(frame, (x,y),(x+w,y+h),(0,255,0),2)
      cv2.rectangle(frame, (x,y-40),(x+w, y), (0,255,0),-2)
      cv2.putText(frame, str(get_className(classIndex)), (x,y-10), font, 0.75, (255,255,255),1,
cv2.LINE_AA)

      cv2.putText(frame, str(round(probabilityValue*100, 2))+"%", (30, 30), font, 0.75,
(255,0,0),2, cv2.LINE_AA)
   return frame
```

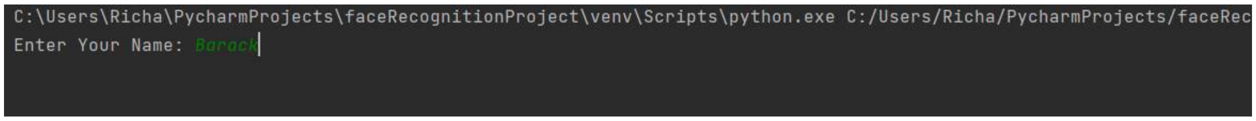
```
with ThreadPoolExecutor(max_workers=4) as executor:
    while True:
        success, frame = cap.read()
        if not success:
            break
        future = executor.submit(process_frame, frame)
        imgOrignal = future.result()
        cv2.imshow("Result", imgOrignal)
        k=cv2.waitKey(1)
        if k==ord('q'):
            break


cap.release()
cv2.destroyAllWindows()
```

<u>Output:</u>

Datacollect.py

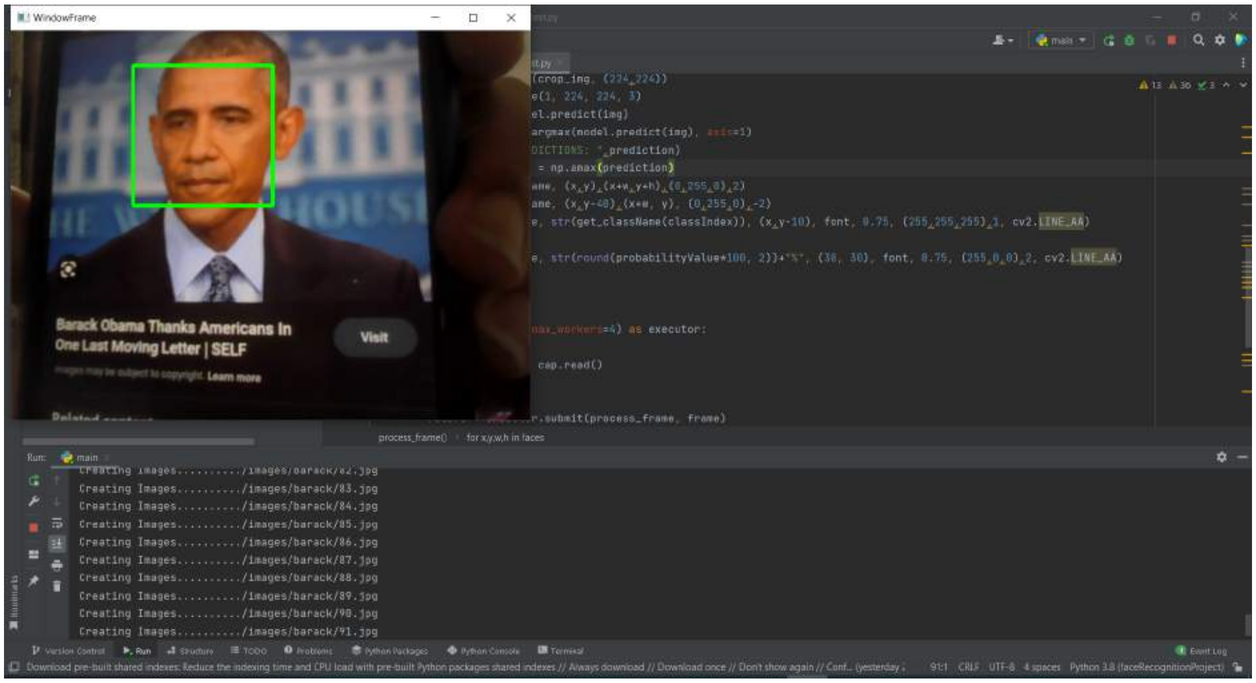
- **Dataset creation**

  Creating the directory to save captured images:



```
C:\Users\Richa\PycharmProjects\faceRecognitionProject\venv\Scripts\python.exe C:/Users/Richa/PycharmProjects/faceRec
Enter Your Name: Barack
```

  Capturing images:

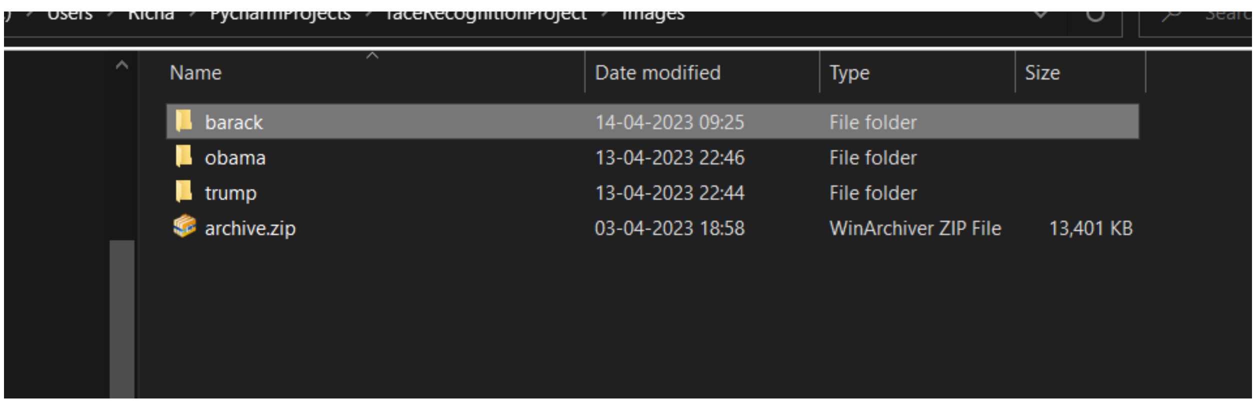


```
(crop_img, (224,224))
e(1, 224, 224, 3)
el.predict(img)
argmax(model.predict(img), axis=1)
DICTIONS: ",prediction)
 = np.amax(prediction)
ame, (x,y),(x+w,y+h),(0,255,0),2)
ame, (x,y-40),(x+w, y), (0,255,0),-2)
e, str(get_className(classIndex)), (x,y-10), font, 0.75, (255,255,255),1, cv2.LINE_AA)

e, str(round(probabilityValue*100, 2))+"%", (30, 30), font, 0.75, (255,0,0),2, cv2.LINE_AA)



max_workers=4) as executor:

 cap.read()


r.submit(process_frame, frame)
```

```
Creating Images........../images/barack/82.jpg
Creating Images........../images/barack/83.jpg
Creating Images........../images/barack/84.jpg
Creating Images........../images/barack/85.jpg
Creating Images........../images/barack/86.jpg
Creating Images........../images/barack/87.jpg
Creating Images........../images/barack/88.jpg
Creating Images........../images/barack/89.jpg
Creating Images........../images/barack/90.jpg
Creating Images........../images/barack/91.jpg
```

WindowFrame
Barack Obama Thanks Americans In One Last Moving Letter | SELF
Visit
Learn more
main
process_frame() for x,y,w,h in faces
Run: main
Version Control
Run
Structure
TODO
Problems
Python Packages
Python Console
Terminal
Event Log
Download pre-built shared indexes: Reduce the indexing time and CPU load with pre-built Python packages shared indexes // Always download // Download once // Don't show again // Conf... (yesterday
91:1 CRLF UTF-8 4 spaces Python 3.8 (faceRecognitionProject)

File stored in the project directory:



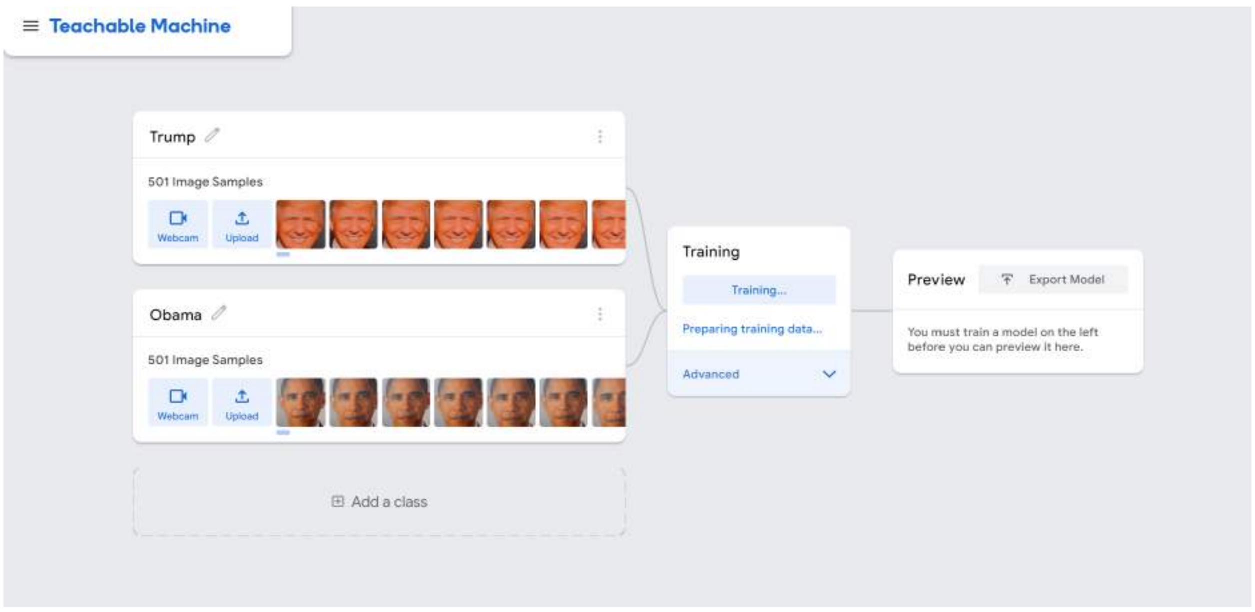
Users › Richa › PycharmProjects › faceRecognitionProject › images

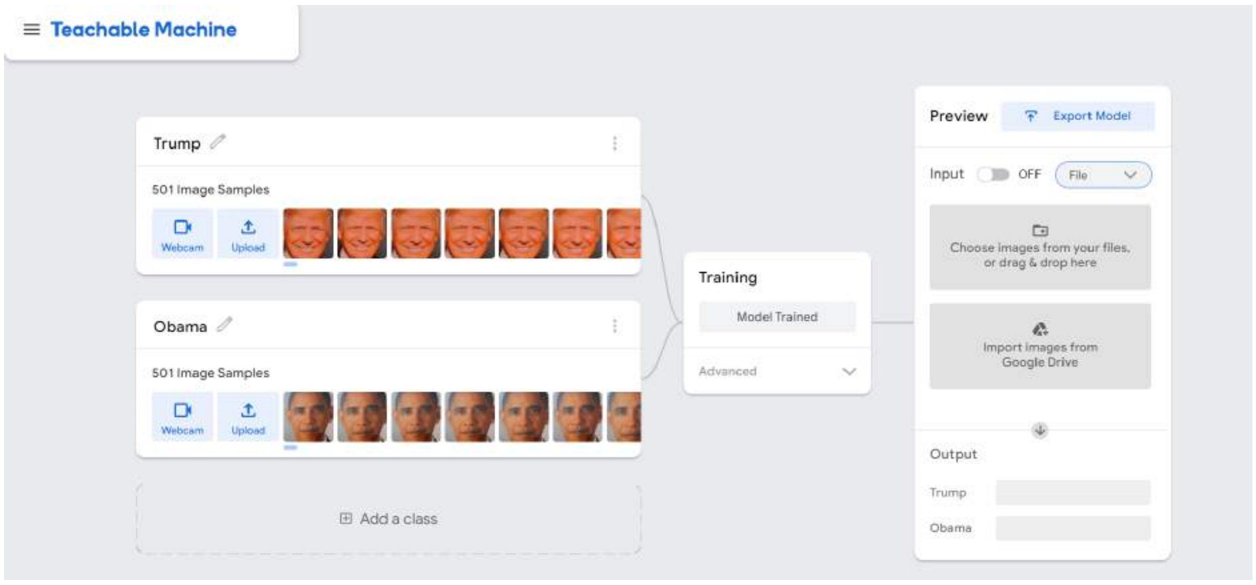
Name	Date modified	Type	Size
barack	14-04-2023 09:25	File folder	
obama	13-04-2023 22:46	File folder	
trump	13-04-2023 22:44	File folder	
archive.zip	03-04-2023 18:58	WinArchiver ZIP File	13,401 KB

Training part

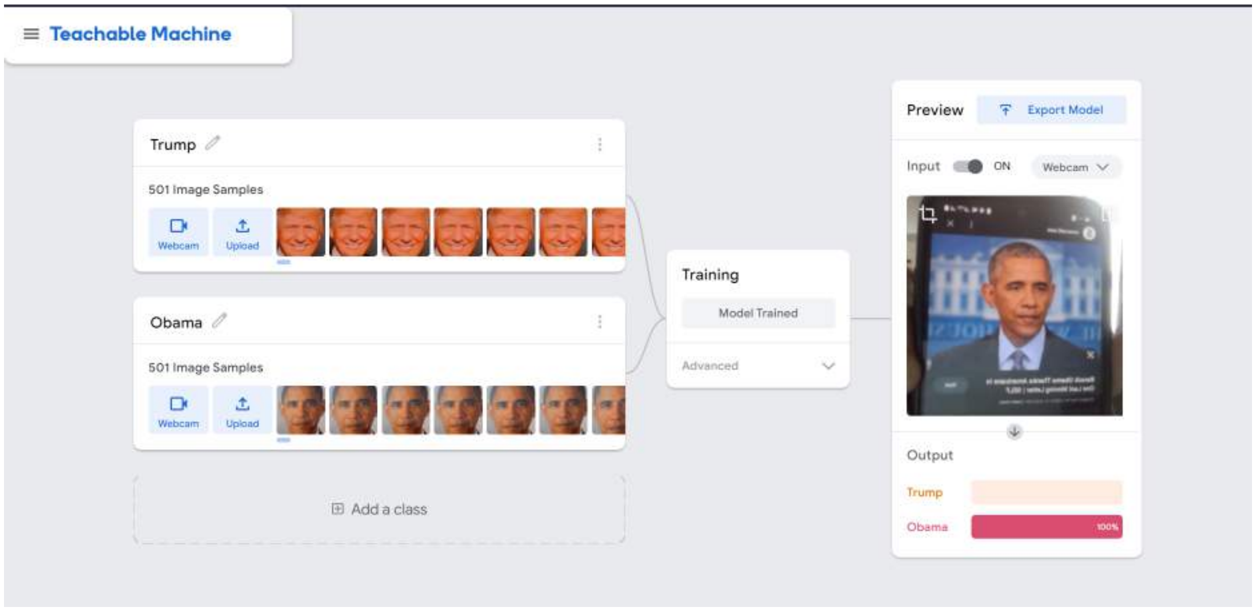
- **Training model on teachable machine and exporting the keras model**

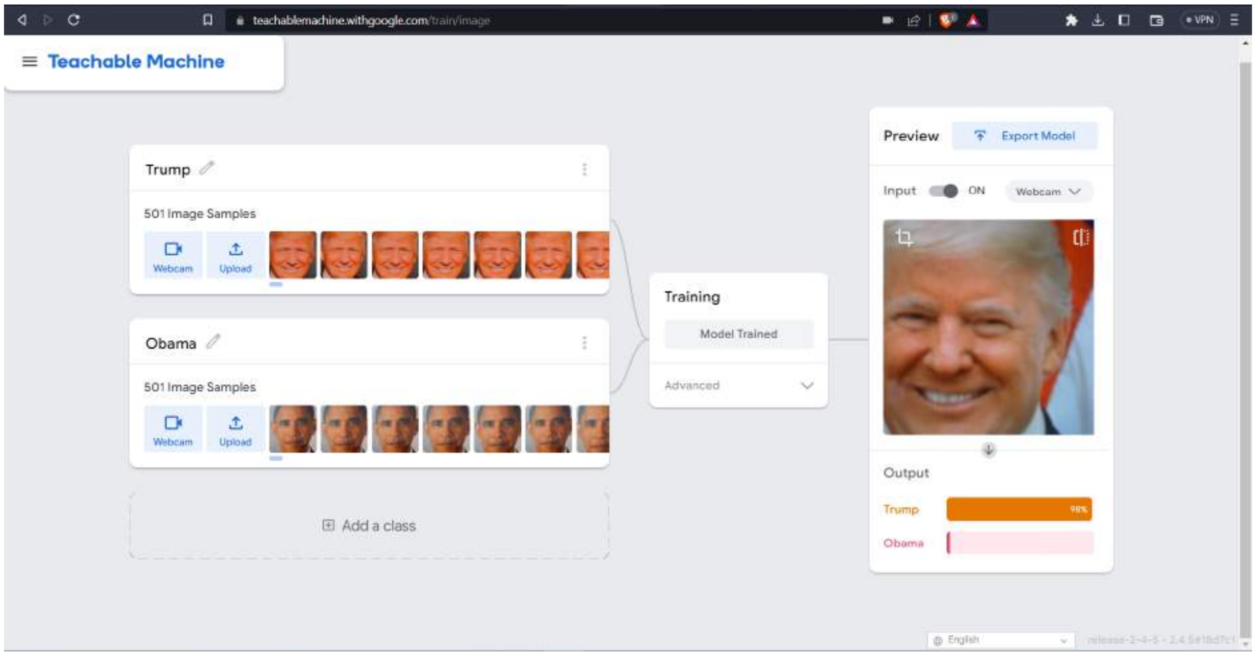
  Navigate to https://teachablemachine.withgoogle.com/train/image and create two or more classes for face detection:



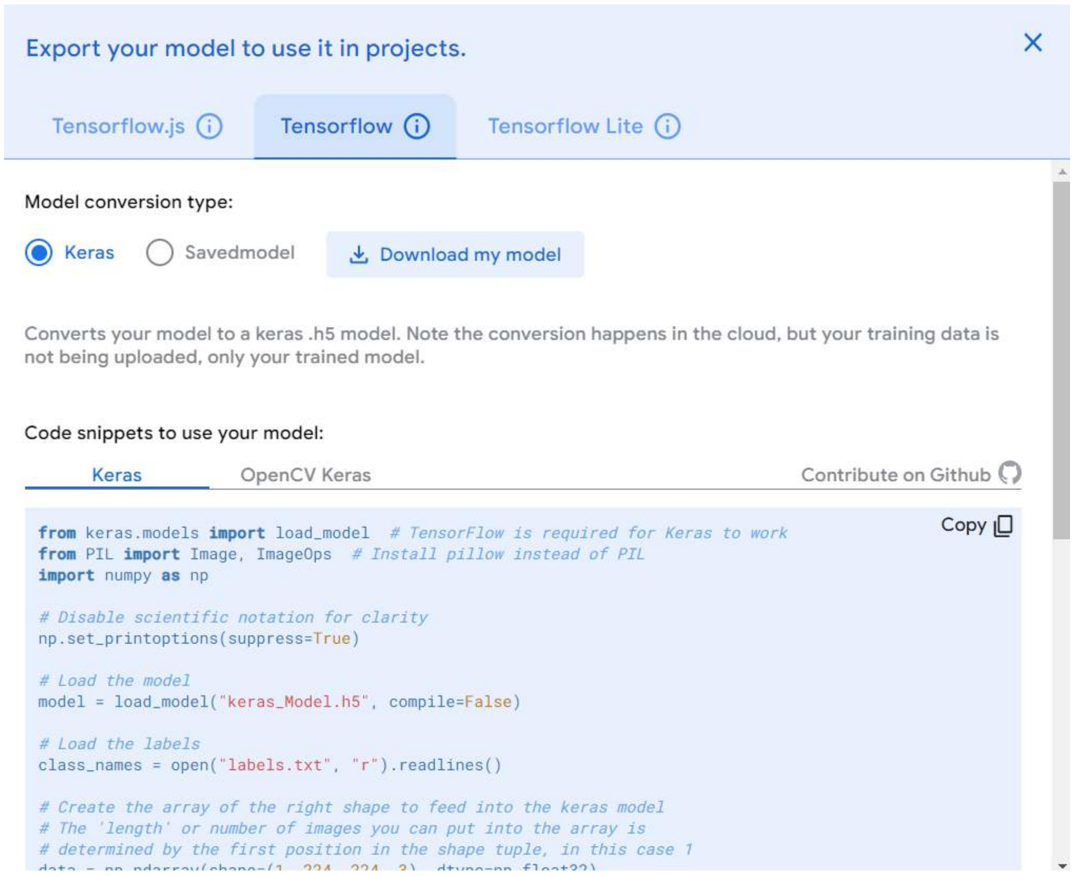


After click the train model button, the model will be ready and we can test it:





Next we will import the created model to our project:


Export your model to use it in projects.

Model conversion type:

[✓ Keras] Savedmodel

Converts your model to a keras .h5 model. Note the conversion happens in the cloud, but your training data is not being uploaded, only your trained model.

Code snippets to use your model:

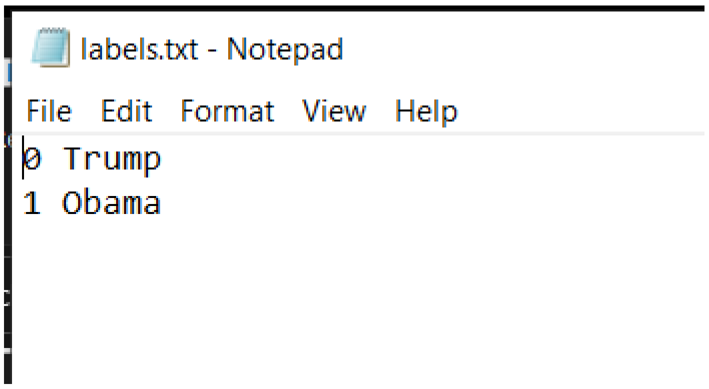
```
from keras.models import load_model  # TensorFlow is required for Keras to work
from PIL import Image, ImageOps  # Install pillow instead of PIL
import numpy as np

# Disable scientific notation for clarity
np.set_printoptions(suppress=True)

# Load the model
model = load_model("keras_Model.h5", compile=False)

# Load the labels
class_names = open("labels.txt", "r").readlines()

# Create the array of the right shape to feed into the keras model
# The 'length' or number of images you can put into the array is
# determined by the first position in the shape tuple, in this case 1
```

Test.py

- **Detecting faces in our parallelized code**

  Once detected, we change our code based on the label provided with the trained model:

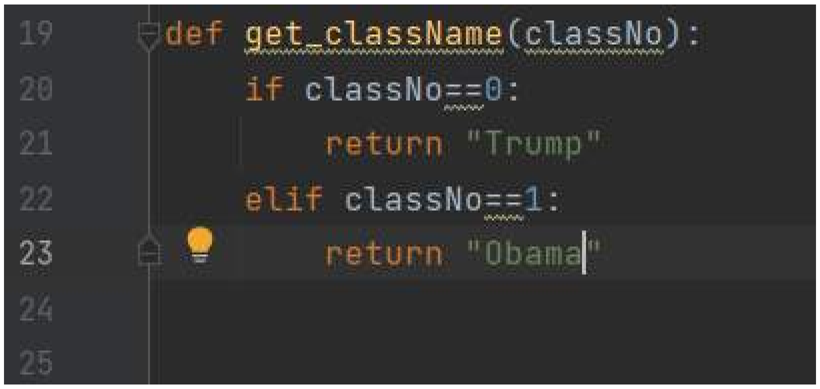
  Label.txt:


labels.txt - Notepad

```
0 Trump
1 Obama
```

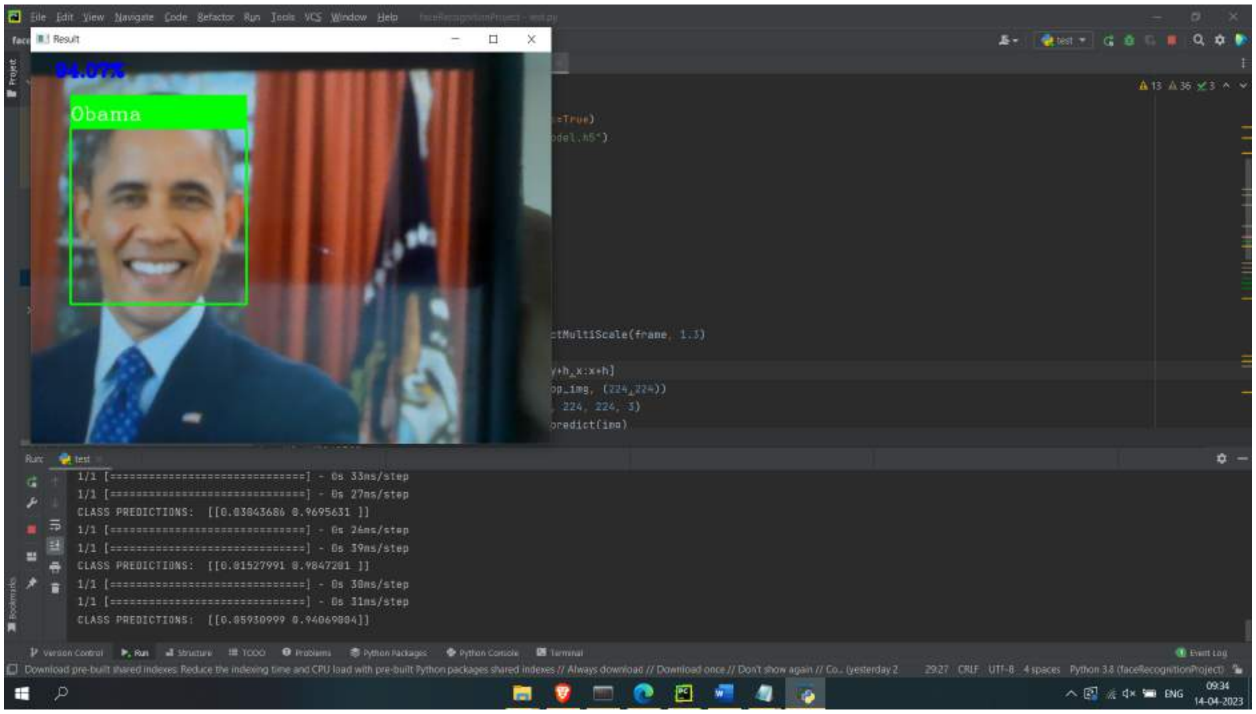
  We change our code likewise:

  Test.py:



```python
def get_className(classNo):
    if classNo==0:
        return "Trump"
    elif classNo==1:
        return "Obama"
```

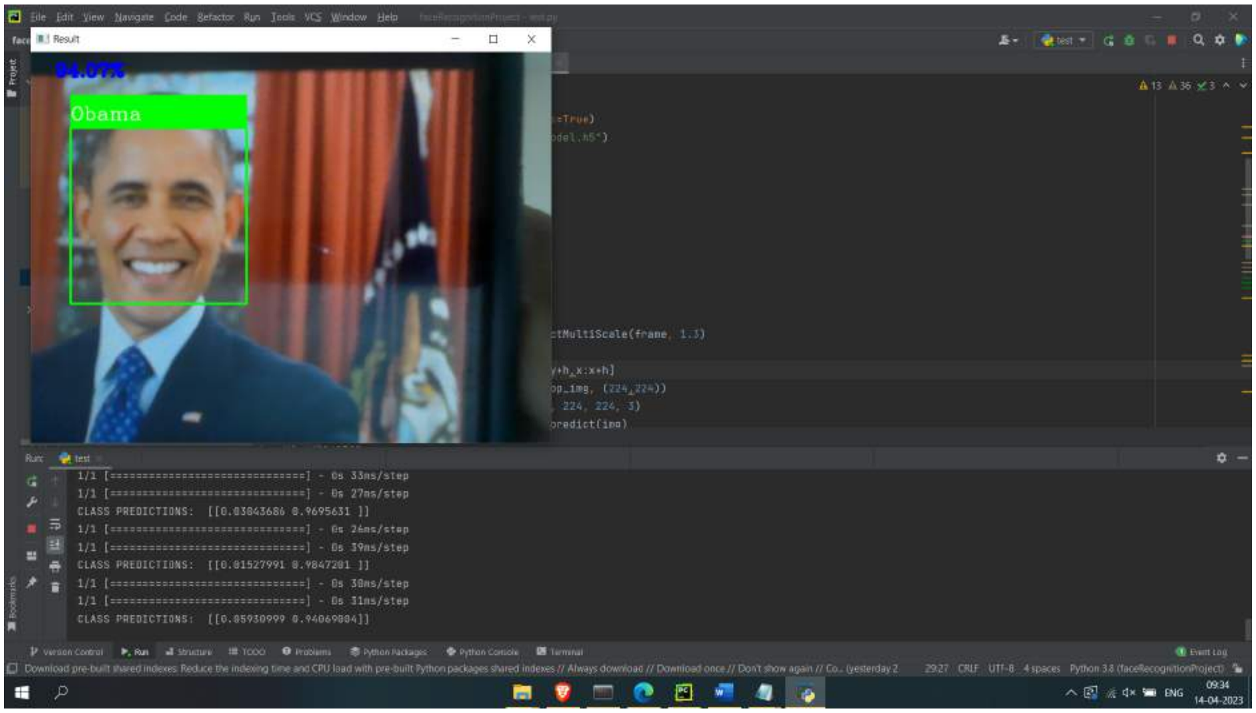
Next we run our code to see the results:



```
1/1 [==============================] - 0s 33ms/step
1/1 [==============================] - 0s 27ms/step
CLASS PREDICTIONS:  [[0.03043686 0.9695631 ]]
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 39ms/step
CLASS PREDICTIONS:  [[0.01527991 0.9847201 ]]
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 31ms/step
CLASS PREDICTIONS:  [[0.05930999 0.94069804]]
```

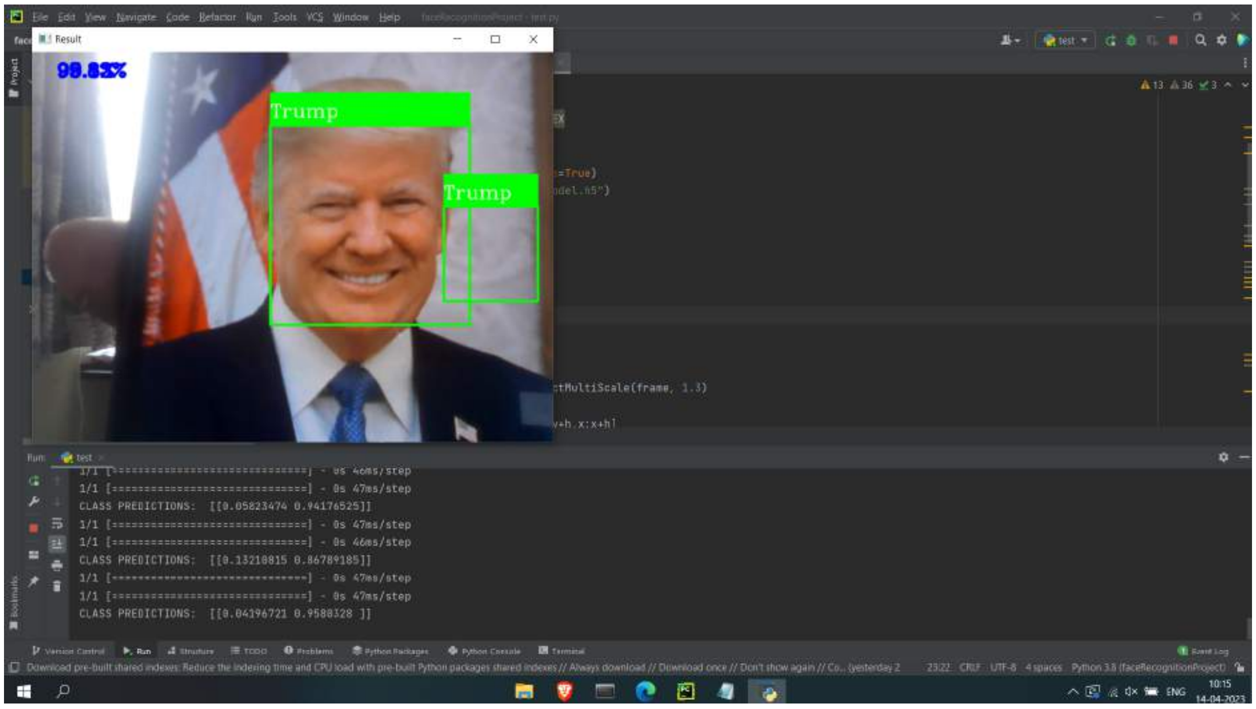
5.2. <u>Test Cases</u>

- Test using Obama's Picture:



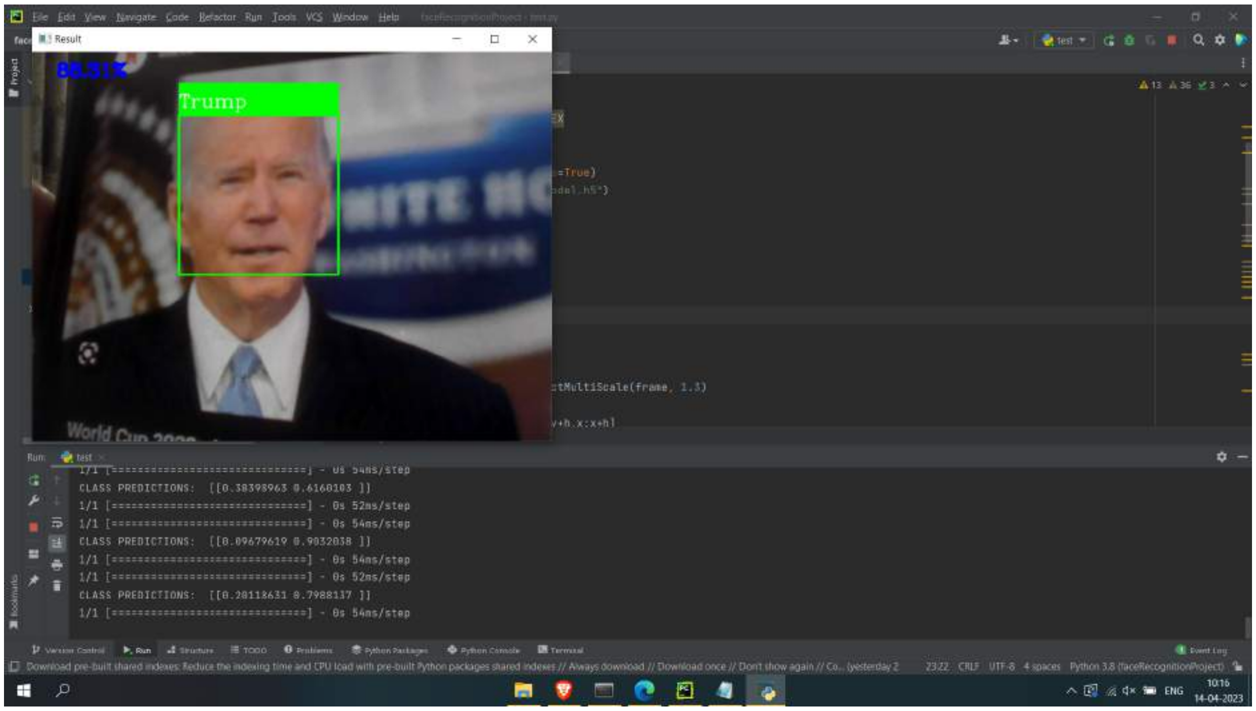
```
1/1 [==============================] - 0s 33ms/step
1/1 [==============================] - 0s 27ms/step
CLASS PREDICTIONS:  [[0.03043686 0.9695631 ]]
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 39ms/step
CLASS PREDICTIONS:  [[0.01527991 0.9847201 ]]
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 31ms/step
CLASS PREDICTIONS:  [[0.05930999 0.94069004]]
```

- Test using Trump’s picture:



```
1/1 [==============================] - 0s 46ms/step
1/1 [==============================] - 0s 47ms/step
CLASS PREDICTIONS:  [[0.05823474 0.94176525]]
1/1 [==============================] - 0s 47ms/step
1/1 [==============================] - 0s 46ms/step
CLASS PREDICTIONS:  [[0.13210815 0.86789185]]
1/1 [==============================] - 0s 47ms/step
1/1 [==============================] - 0s 47ms/step
CLASS PREDICTIONS:  [[0.04196721 0.9580328 ]]
```

- Test using any other person’s Picture:



```
1/1 [==============================] - 0s 54ms/step
CLASS PREDICTIONS:  [[0.38398963 0.6160103 ]]
1/1 [==============================] - 0s 52ms/step
1/1 [==============================] - 0s 54ms/step
CLASS PREDICTIONS:  [[0.09679619 0.9032038 ]]
1/1 [==============================] - 0s 54ms/step
1/1 [==============================] - 0s 52ms/step
CLASS PREDICTIONS:  [[0.20118631 0.7988137 ]]
1/1 [==============================] - 0s 54ms/step
```

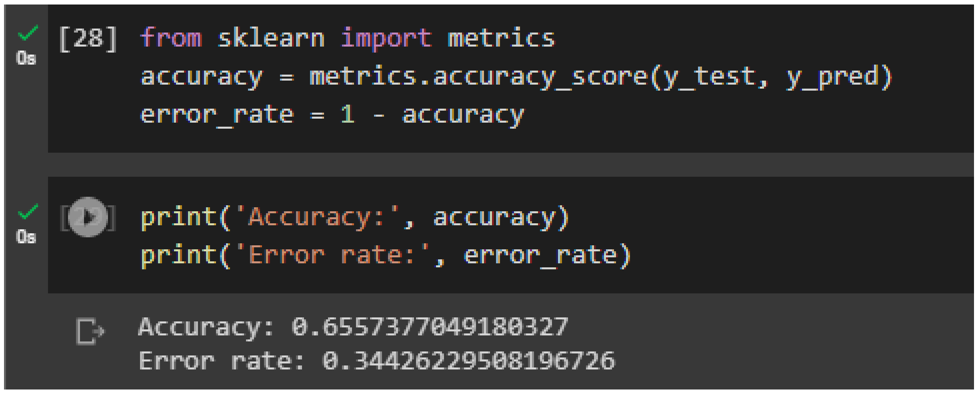
It will only classify another person as “Obama” or “Trump” on variable percentages as these are the only two categories defined for the classification keras model.

6. <u>OUTPUT AND PERFORMANCE ANALYSIS</u>

- Performance Gains: The face recognition model's performance is considerably improved by parallelizing. The total processing time is decreased by breaking the task up into smaller parallel tasks that can run concurrently across many processors.
- Scalability: Our parallelized face recognition model is easily scalable to support larger datasets and tackle more challenging processing tasks. In applications where real-time performance is crucial, this is especially important.
- Accuracy: By enabling the face recognition model to process more data in less time, parallelizing it increased its accuracy. As a result, the recognition process may experience fewer false positives and false negatives.
- Robustness: Parallelization can aid in enhancing the face recognition model's resistance to variations in position, lighting, and facial emotions. The model better captures the different subtleties of a person's face by processing multiple frames of video at once.

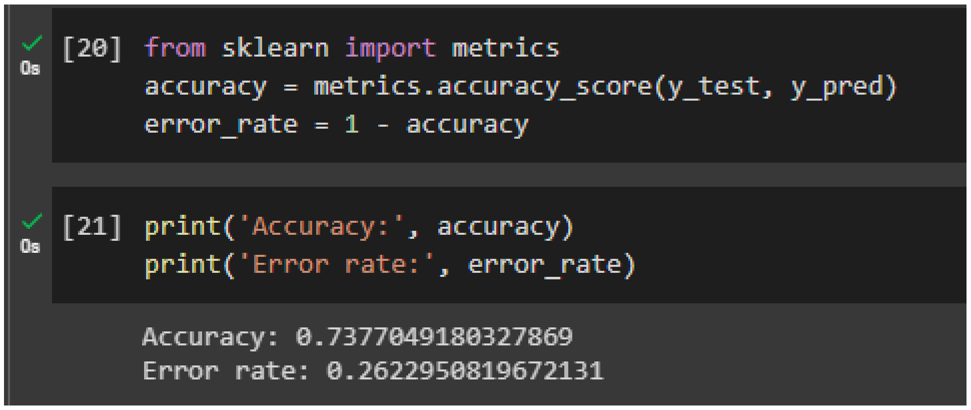
Output – in terms of performance metrics:

Before parallelising face recognition:



```
[28] from sklearn import metrics
     accuracy = metrics.accuracy_score(y_test, y_pred)
     error_rate = 1 - accuracy

     print('Accuracy:', accuracy)
     print('Error rate:', error_rate)

     Accuracy: 0.6557377049180327
     Error rate: 0.3442622950819672б
```

After parallelising face recognition:



```
[20] from sklearn import metrics
     accuracy = metrics.accuracy_score(y_test, y_pred)
     error_rate = 1 - accuracy

[21] print('Accuracy:', accuracy)
     print('Error rate:', error_rate)

     Accuracy: 0.7377049180327869
     Error rate: 0.2622950819672131
```

We observe that the accuracy is increased after parallelising the algorithm.

7. CONCLUSION AND FUTURE DIRECTIONS

The research on parallelizing face recognition algorithms has been an active area of research in recent years, with the aim of achieving faster and more accurate recognition results. Through this project, we have discussed a few papers that proposed parallelized face recognition algorithms using different techniques such as GPU computing and multicore platforms and have implemented our own program. The papers reported significant speedup and performance improvement in face recognition tasks.

As face recognition becomes more prevalent in various applications, there is a growing need to develop more efficient and effective parallelized algorithms for face recognition. One potential future direction is to investigate the use of advanced parallel computing architectures such as distributed computing, FPGA-based systems, and cloud computing for face recognition tasks.

Additionally, there is a need for further research on improving the accuracy and robustness of parallelized face recognition algorithms, particularly in handling challenging scenarios such as partial occlusions, variations in illumination, and pose changes. Finally, there is a need for standard benchmark datasets and evaluation metrics to compare the performance of different parallelized face recognition algorithms.

8. REFERENCES

[1] Kausar Mia, Mr. Md Assaduzzaman, Arnab Saha, Parallelizing Image Processing Algorithms for Face Recognition on Multicore Platforms.(IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 13, No. 11, 2022.

[2] Yao Zhang; Yu Liu; Zhanwei Du, Research on Parallelization of Face Recognition.IEEE 2022, 10th Joint International Journal.

[3] Ashish Ranjan, Varun Nagesh Jolly Behera, Motahar Reza .A Parallel Approach for Real-Time Face Recognition from a Large Database. 2020

[4] O. Lahdenoja; J. Maunu; M. Laiho; A. Paasio. A massively parallel algorithm for local binary pattern based face recognition. IEEE 2006.