# Face Recognition Hackathon

Bhavya Kohli

October 2021

## Introduction

The aim of this study was to develop an approach to find out whether two given images were of the same person. I tried to attempt this using two methods – one, using the face_recognition library, and two, using Convolutional Neural Networks.

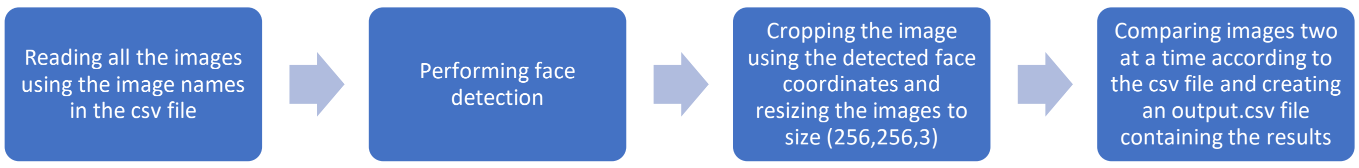
## Using the face_recognition library

The face_recognition library provides a convenient way of comparing two faces, whether they are within the same image or are in two different images. It utilizes dlib's state of the art face recognition model built using deep learning. The module can be used to load images and detect faces in those images, and can then be used to compare any two images using the "encoding" of the image.
The image encoding obtained using the module can be used to track facial features such as the eyes, eyebrows, nose, mouth and general face outline. Comparing the encodings of two images accounts for all these structures and only then computes the final result.

## Image preprocessing

Loading images was done using the matplotlib imread function for consistency with the subsequent imshow function calls, and face detection was performed using the opencv library, using the Haar cascade detector. Face detection was possible using the face_recognition module itself, but it had many failure cases and the opencv method was preferred. On detecting the face locations the faces were suitably cropped using array slicing. All loaded images were also resized to be of the same size for fulfilling the input shape consistency requirements in the Convolutional Neural Network approach, but they also improved performance using the face_recognition method. Functions were defined for performing each of these tasks and were linked to each other to give a clean look.

## Brief overview of the workflow


Reading all the images using the image names in the csv file
Performing face detection
Cropping the image using the detected face coordinates and resizing the images to size (256,256,3)
Comparing images two at a time according to the csv file and creating an output.csv file containing the results

## Challenges Faced

The opencv face detection hyperparameters *scaleFactor* and *minNeighbors* had to be tuned, which posed a challenge because of the average time it took to complete one run.
The face_recognition had problems in detecting faces after cropping in the previous step, but this issue was solved by specifying the *known_face_location* parameter.

## Results

Using the optimum parameters, the f1 score of the process was able to reach around 0.7 on using 10000 samples out of the 23917 given in the train.csv file, with an average accuracy of 0.70.

## Using a Convolutional Neural Network

This was my first attempt at the problem statement, i.e. using a Convolutional Neural Network architecture to predict whether the two input images were "correlated" or not. Using sufficient training samples this correlation would be recognized by the model and the output would then signify whether the faces in the two input images are of the same person or not.

For the task, I utilized a two input CNN which is similar to having two CNNs running parallely. The two input streams were combined together using a *concatenate* layer and then fed into a Dense layer before finally reaching the softmax activated output layer. The architecture used is attached
As seen in the figure, the output layer has 2 nodes because of the labels "0" and "1" were converted to a one-hot encoded scheme and hence were represented as a 2x1 vector.

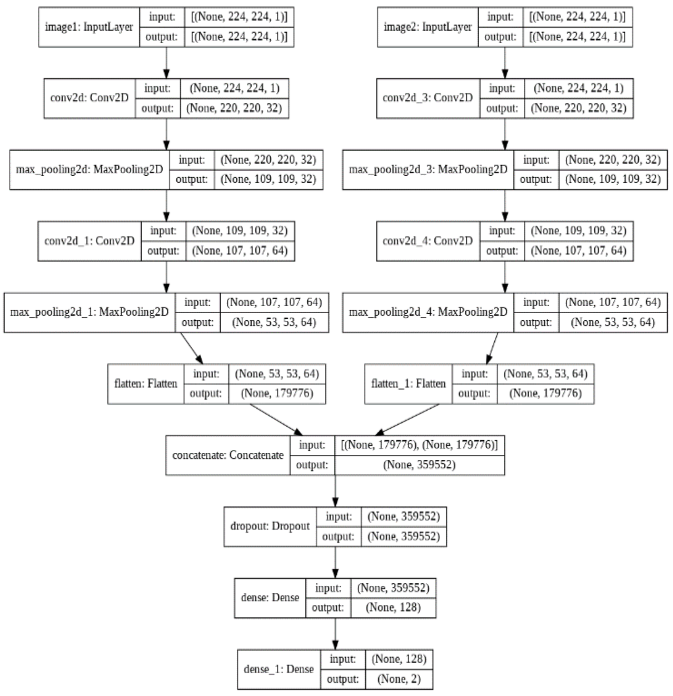


image1: InputLayer | input: [(None, 224, 224, 1)] | output: [(None, 224, 224, 1)]
image2: InputLayer | input: [(None, 224, 224, 1)] | output: [(None, 224, 224, 1)]
conv2d: Conv2D | input: (None, 224, 224, 1) | output: (None, 220, 220, 32)
conv2d_3: Conv2D | input: (None, 224, 224, 1) | output: (None, 220, 220, 32)
max_pooling2d: MaxPooling2D | input: (None, 220, 220, 32) | output: (None, 109, 109, 32)
max_pooling2d_3: MaxPooling2D | input: (None, 220, 220, 32) | output: (None, 109, 109, 32)
conv2d_1: Conv2D | input: (None, 109, 109, 32) | output: (None, 107, 107, 64)
conv2d_4: Conv2D | input: (None, 109, 109, 32) | output: (None, 107, 107, 64)
max_pooling2d_1: MaxPooling2D | input: (None, 107, 107, 64) | output: (None, 53, 53, 64)
max_pooling2d_4: MaxPooling2D | input: (None, 107, 107, 64) | output: (None, 53, 53, 64)
flatten: Flatten | input: (None, 53, 53, 64) | output: (None, 179776)
flatten_1: Flatten | input: (None, 53, 53, 64) | output: (None, 179776)
concatenate: Concatenate | input: [(None, 179776), (None, 179776)] | output: (None, 359552)
dropout: Dropout | input: (None, 359552) | output: (None, 359552)
dense: Dense | input: (None, 359552) | output: (None, 128)
dense_1: Dense | input: (None, 128) | output: (None, 2)

The challenges faced were much tougher using this method mostly because of hardware limitations and the nature of CNNs. Using lesser training samples or reducing complexity could reduce time, but that also affects the generalized performance of the model. Grid search could not be properly performed because of these limitations and the best result of the training with a validation subset was an accuracy of around 63%.
The ipynb notebook containing the code for the model is also present in the repository, but due to time constraints I could not create a script to use the model for testing on a new set of images as per the guidelines.