

St. Cloud State University

The Repository at St. Cloud State

Culminating Projects in Electrical Engineering

Department of Electrical and Computer
Engineering

8-2022

Real-Time Deep Learning-Based Face Recognition System

Aarathi Rajagopalan

Follow this and additional works at: https://repository.stcloudstate.edu/ece_etds

Recommended Citation

Rajagopalan, Aarathi, "Real-Time Deep Learning-Based Face Recognition System" (2022). *Culminating Projects in Electrical Engineering*. 8.

https://repository.stcloudstate.edu/ece_etds/8

This Starred Paper is brought to you for free and open access by the Department of Electrical and Computer Engineering at The Repository at St. Cloud State. It has been accepted for inclusion in Culminating Projects in Electrical Engineering by an authorized administrator of The Repository at St. Cloud State. For more information, please contact tdsteman@stcloudstate.edu.

Real-Time Deep Learning-Based Face Recognition System

by

Aarthi Rajagopalan

A Starred Paper

Submitted to the Graduate Faculty

of

Saint Cloud State University

in Partial Fulfillment of the Requirements

for the Degree

Master of Science

in Electrical Engineering

August, 2022

Starred Paper Committee:

Dr. Yi Zheng, Chairperson

Dr. Ling Hou

Dr. Aiping Yao

Abstract

This research proposes Real-time Deep Learning-based Face recognition algorithms using MATLAB and Python. Generally, Face recognition is defined as the process through which people are identified using facial images. This technology is applied broadly in biometrics, security information, accessing controlled areas, etc. The facial recognition system can be built by following two steps. In the first step, the facial features are picked up or extracted, then the second step involves pattern classification. Deep learning, specifically the convolutional neural network (CNN), has recently made more progress in face recognition technology. Convolution Neural Network is one among the Deep Learning approaches and has shown excellent performance in many fields, such as image recognition of a large amount of training data (such as ImageNet). However, due to hardware limitations and insufficient training datasets, high performance is not achieved. Therefore, in this work, the Transfer Learning method is used to improve the performance of the face-recognition system even for a smaller number of images. For this, two pre-trained models, namely, GoogLeNet CNN (in MATLAB) and FaceNet (in Python) are used. Transfer learning is used to perform fine-tuning on the last layer of CNN model for new classification tasks. FaceNet presents a unified system for face verification (is this the same person?), recognition (who is this person?) and clustering (finds common people among these faces) using the method based on learning a Euclidean embedding per image using a deep convolutional network.

Keywords: CNN, Face Recognition, Transfer Learning, GoogLeNet, FaceNet.

Acknowledgement

I would like to express my sincere gratitude to my advisor, Dr. Yi Zheng for his constant support, help and guidance throughout my academic career. His immense knowledge and passion towards engineering has always inspired me. I would like to thank my committee members, Dr. Ling Hou and Dr. Aiping Yao for their encouragement, insightful comments, and questions. I am extremely grateful to my parents for their love and support throughout my endeavors.

Table of Contents

	Page
List of Tables.....	6
List of Figures.....	7
Chapter	
1: Introduction	
Background.....	9
Transfer Learning.....	11
Face Recognition.....	12
Outline.....	13
2: Review of Existing Face Detection and Recognition Algorithms	
Introduction.....	15
Principle Component Analysis (PCA).....	15
Linear Discriminant Analysis (LDA).....	16
Skin Color-Based Algorithm.....	17
Wavelet Based Algorithm.....	18
Artificial Neural Network-Based Algorithm.....	19
3: Transfer Learning From Pre-Trained Models	
Introduction.....	21
Convolutional Neural Networks.....	21
Transfer Learning Process.....	26
Pre-Trained Models for Computer Vision.....	27

Chapter	Page
4: Design Methodology	
Transfer Learning Using Pretrained GoogLeNet CNN.....	32
Face Recognition Using Google’s Deep Convolutional Network – FaceNet....	41
5: Experiments and Results	
Implementation Using Pretrained GoogLeNet CNN.....	48
Implementation Using FaceNet.....	69
6: Conclusion.....	82
References.....	83

List of Tables

Table	Page
1: Architectural details of GoogLeNet.....	37

List of Figures

Figure	Page
1: Convolutional neural network architecture.....	10
2: The Convolution operation	22
3: Examples of kernel filters for CNN	23
4: An example of CNN.....	24
5: Features of a convolutional layer.....	25
6: Features of a pooling layer	25
7: Features of a fully connected layer.....	26
8: Transfer Learning using Pretrained Network.....	26
9: An illustration of the VGG-19 Network.....	28
10: An illustration of the Inceptionv3 Network	29
11: An illustration of the ResNet50 Network.....	30
12: The network architecture of EfficientNet.....	31
13: Dataset splitting.....	33
14A,14B: An example of GoogLeNet convolution operation.....	34,35
15A,15B: Inception module	36
16: Architecture of GoogLeNet CNN	38
17: Flow Chart of Train and Evaluate the Model	39
18: Flow Chart of Testing on Trained Model	41
19: FaceNet model.....	42
20: Architecture of FaceNet.....	43
21: Triplet-loss and learning	44

Figure	Page
22: Triplet Selection.....	47
23: Dataset.....	48
24: Application Workflow	61
25: Email setup.....	62
26: GUI Design.....	64
27: Real-time Face Detection and Tracking	66
28A-B: Result 1 - when known people come in front of the webcam	67
29: Result 2 - Unknown face detected.....	68
30: Result 2 – Intruder Email Alert with image of unknown person attached.....	68
31A,31B: Implementation using FaceNet: Results - Example #1	77,78
32A,32B: Implementation using FaceNet: Results - Example #2.....	78,79
33A,33B: Implementation using FaceNet: Results - Example #3.....	79,80
34A,34B: Implementation using FaceNet: Results - Example #4.....	80,81

Chapter 1: Introduction

The background and objectives of this study will be briefly reviewed in this chapter. This chapter will also outline the contents of this starred paper.

1.1 Background

Deep learning is a subdivision of machine learning, which is essentially a neural network with three or more layers. These neural networks attempt to imitate the behavior of the human brain by “learning” from large amounts of data, although not completely matching its ability. They do so through a combination of data inputs, weights and bias. These elements work together in order to accurately recognize, classify, and describe objects within the data.

There are various types of neural networks to address specific problems or datasets. For example, Convolutional Neural Networks (CNNs) are used primarily in computer vision and image classification applications detect features and patterns within an image, enabling tasks, like object detection or recognition.

Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a type of Neural Network that concentrates in image processing and classification. It basically takes the pixel values of an image in vector/matrix form as its input, runs it through a sequence of layers, and outputs a classification for the image. The Convolutional Neural Networks are mainly made of up four types of layers:

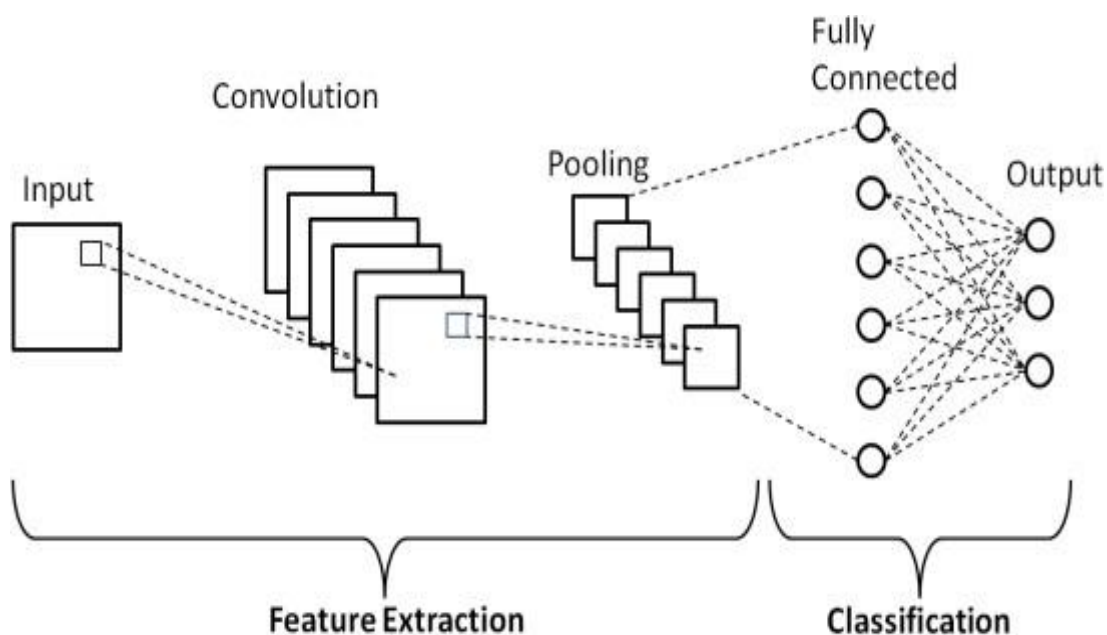
1. A *Convolutional Layer* takes patterns from the image by running its matrix through a set of learnable filters/kernels, which portray different visual features in the image. These filters slide over the image with respect to a specified number of

pixels, or strides. During these *convolutions*, each of the filters produce their own feature map; the final output of this layer is a transformation of the original image, consisting of all the feature maps placed on top of each other.

2. The *Rectified Linear Unit Layer* (ReLU) is a non-linear activation function $f(x) = \max(x, 0)$. Without changing its shape, ReLU converts the elements of the output of the convolutional layer in the range from 0 to infinity, by replacing any negative values with zero.

Figure 1

Convolutional neural network architecture



Note. This figure shows an example of a simple schematic representation of a basic CNN. (Ray et al., 2021).

3. A *Pooling Layer* regulates the width by height dimensions by reducing the input volume spatial dimensions for the next convolutional layer without changing the dimensional depth of the volume. The process performed by the pooling layer is otherwise known as down-sampling or sub-sampling because the reduction of size

results in simultaneous information loss that benefits the network. The reduction becomes less computational as the information moves to the next pooling layers, and it also works against over-fitting. The most common strategies employed in the pooling layer networks are max-pooling and average-pooling. In a comprehensive theoretical analysis of the max pooling and average pooling, it has been observed that max pooling can result in faster convergence of information, and that the network picks the high-ranking features in the image thus enhancing generalization. Also, pooling layer has other variations such as stochastic pooling spatial pyramid pooling, and def-pooling that serves marked purposes.

4. *Fully-Connected Layer:* The filters and neurons in this layer are connected to all the activation from the previous layers, resulting in full connections as their name implies. The calculations in this level are done through the multiplication of matrix followed by the bias offset. FC layer goes through a process which converts the 2D feature map to the 1D feature vector. In addition, the vector formed in this process is either classified as classes for classification or the feature vector undergoing further processing.

1.2 Transfer Learning

Transfer learning is a machine learning technique which uses knowledge from one domain to get better in another domain. Transfer learning reduces the need to recollect the training data for a specific domain. It allows the distributions, tasks, and domains to be different for the training and testing data. Transfer learning is driven by the fact that people can intelligently apply knowledge from previously learned solutions to solve new problems or find better solutions. For instance, it is easier to learn French if you already know

Latin. Yet another example is: if you already learned to ride a scooter then it will be easier to learn to ride a motorcycle.

In the field of *machine learning*, transfer learning is often the same but it is meant to be used for sharing the weights of neural networks. The common approach is to train all the layers of one network and then copying the first n layers of this network to a second network. This step is known as *pre-training* of the network. When the layers are copied to the second network, the remaining layers of the second network can be randomly initialized and trained for the target task with a different dataset. There are two ways to use the pre-trained weights: The first way is to train these weights together with the weights of the last layers, the second approach is to not change the weights of the pre-trained layers. The weights are left frozen while training on the target task. This second approach is used in this starred paper.

1.3 Face Recognition

Face recognition is a modern security feature which deals with recognition and authentication of a face, and is a field that has been studied extensively for a long time. Face detection and tracking has been used for the purposes of surveillance, security, human computer interaction, etc. (Shah et al., 2016). Here, the Transfer learning technique is being used, which is commonly used in deep learning applications. One can take a pre-trained network and use it as a starting point to learn a new task. Fine-tuning a network with transfer learning is usually much faster and easier when compared to training a network with randomly initialized weights from scratch. The learned features can quickly be transferred to a new task using a smaller number of training images.

In the late 1980s, the development of computer technology and optical imaging technology was improved and the real entry into the application phase of face recognition occurred in the late 1990s. In early research on face recognition, the researcher mainly focused on methods called geometry methods to match simple features with image processing techniques. Later, the holistic methods such as principal component analysis (PCA) and linear discriminant analysis (LDA) became popular. Then, the feature-based method was developed for matching all the local features across face image. As time passed, a holistic and feature-based method was developed and then combined into hybrid methods. Hybrid methods stayed the state-of-the-art until recently when deep learning emerged as a leading approach to most applications of computer vision, including face recognition.

Deep Learning is a machine learning technique that has got attention lately because it can achieve high accuracy when trained with large amounts of data. In addition, Convolutional Neural Networks (CNNs) is one of the most popular deep learning algorithms used for image classification problems. As a result, a CNN based deep transfer learning for face recognition using small datasets is proposed. Transfer learning is a popular deep learning approach where the knowledge gained from a related task is transferred to a new task. Compared with training deep neural networks from scratch, this proposed method can reduce training time.

1.4 Outline

Chapter 2 reviews the existing face detection and recognition algorithms. The related work on transfer learning and face recognition are briefly described in this chapter. Chapter 3 gives details on transfer learning using pre-trained models. This chapter also explains about Convolutional Neural Networks in

detail. Chapter 4 describes the models used to perform the experiments on face recognition. Chapter 5 gives the implementation details for face recognition and discusses its results. Chapter 6 provides conclusions and future scope.

Chapter 2: Review of Existing Face Detection and Recognition Algorithms

2.1 Introduction

Many robust algorithms have been developed to have accurate performance for tackling face detection and recognition problems. These algorithms or methods are the most successful and widely used ones in face detection and recognition applications. The algorithms are as follows:

- Principle Component Analysis (PCA)
 - Eigenface
- Linear Discriminant Analysis (LDA)
 - Fisherface
- Skin color- based algorithm
 - a. Red-Green-Blue (RGB)
 - b. YCbCr (Luminance-Chrominance)
 - c. Hue-Saturation Intensity (HSI)
- Wavelet based algorithm
 - Gabor Wavelet
- Artificial neural networks-based algorithm
 - a. Fast Forward
 - b. Back Propagation
 - c. Radial Basis Function (RBF)

2.2 Principle Component Analysis (PCA)

PCA is a method used to simplify the problem of selecting the representation of eigenvalues and the corresponding eigenvectors in order to obtain a consistent representation. This can be attained by reducing the dimension space of the

representation. The dimension space needs to be reduced in order to obtain fast and robust object recognition. PCA also reserves the original information of the data. The PCA basis is applied in the Eigenface based algorithm.

Eigenface Based Algorithm

Eigenface based algorithm is the most widely used method to detect faces. This approach makes use of the presence of eyes, nose and mouth and the relative distances between them. This feature is called as *Eigenfaces* in the facial domain. This facial feature can be obtained by using a mathematical tool known as Principle Component Analysis (PCA). Any original image from the training set can be reconstructed by combining the Eigenfaces by using PCA. Usually, a face is classified by estimating the relative distance of the Eigenfaces.

2.3 Linear Discriminant Analysis (LDA)

LDA is otherwise known as Fisher's Linear Discriminant (FLD). It decreases the dimension space by using the FLD technique. FLD technique uses within-class information, thus reducing variation within each class and maximizing the class separation.

Fisherface Based Algorithm

The Fisherface approach is also one of the most widely used feature extraction algorithms in facial images. This approach attempts to find the projection direction in which the images belonging to different classes are segregated maximally.

According to Shang-Hung Lin, Fisherface algorithm is an improvement of the eigenface algorithm in order to cater the variation in the illumination. It is reported that Fisherface algorithm performs better than eigenface when the lighting condition is changed. This method needs various training images for each face. Thus, this

algorithm cannot be applied to the face recognition applications where only one example image per person is at hand for training.

2.4 Skin Color-Based Algorithm

Skin color is the most important feature of a human face. Human skin colors are set apart from different ethnic with respect to the intensity of the skin color, and not the chromatic features. One of the facial feature methods involves skin color-based processing method. In this algorithm, each pixel is classified as either skin color or non-skin color. For an input image, this method uses color space for the skin region as the criteria of classification. Threshold is applied to the mask of the skin region. Lastly, a bounding box is drawn to obtain the face from the input image.

According to Singh et al. (2003), the skin color processing method has a faster processing time when compared to other facial feature methods. However, Chae et al. (2008) has a different opinion that skin color method is time consuming. He felt so because the method scans the target image linearly, that involves a large space of scanning. Thus, a novel method using sub-windows scanning has been proposed instead of the standard linear scanning. This proposed method works by scanning the image sparsely on the basis of the facial color density by determining the horizontal and vertical intervals.

From the experiment, the results revealed that this proposed method was successful in detecting faces in a shorter period of time when compared to the standardized method. The sub-windows scanning method has a less computational time since it skips the sub-windows that do not comprise possible faces. The three most popular color spaces are RGB, YCbCr, and HSI.

RGB (Red-Green-Blue)

In RGB color space, a normalized color histogram is used in order to detect the pixels of skin color of an image and can be further normalized for intensity changes when dividing by luminance. This enables localization and detection of the face. However, this color space is not preferred for color-based detection methods when compared to YCbCr or HSI.

YCbCr (Luminance-Chrominance)

This color space gives a good coverage for different human races. The responsible values in this color space are luminance (Y) and chrominance (C). It basically separates luminance and chrominance. This algorithm can only be carried out if the chrominance component is used. It eliminates the luminance as much as possible by selecting the Cb-Cr plane from the YCbCr color space. A pixel represents the skin tone if the values [Cr, Cb] lie within the thresholds.

HSI (Hue-Saturation-Intensity)

On the basis of the studies performed by Zarit et al. (1999), HSI is said to yield the best performance for skin color approach. Skin color classification in the HSI color space is similar to the YCbCr color space, but the responsible values are hue (H) and saturation (S). Unfortunately, all of these algorithms become unsuccessful when there are regions other than the face such as arms, legs and other objects in background having the same color value.

2.5 Wavelet based algorithm

In wavelet-based algorithm, each face image is characterized by a subset of band filtered images that contain wavelet coefficients. Wavelet transform likely provides a robust multi-scale method of analysis for an image. Wavelets are very

flexible. This is because several bases exist, from which the most suitable basis can be selected for an application. Gabor wavelet method is the most widely used wavelet method in image texture analysis.

Gabor Wavelet

Gabor wavelet transform uses the spatial frequency structures and the orientation relation. This method is a kind of Gaussian modulated sinusoidal wave of the Fourier transform. Gabor wavelet approach detects the short lines, ending lines and sharp changes in curvature. These curves match well with the important features of human face such as mouth, nose, eyebrow, jaw line and cheekbone. Thus, Gabor wavelet is widely known for detecting features.

2.6 Artificial neural network-based algorithm

Artificial neural network (ANN) is the most widely used method for the recognition process. ANN is implemented once a face has been detected to identify, by calculating the weight of the facial information to recognize who the person is.

ANN imitates the human brain, i.e., the biological neuron system. A neuron basically receives a signal from the previous layer and then transmits the signal to all neurons of the next layer. Before transmitting the signal to the next layer, the signal gets multiplied with a separate multi weight value and then the weighted input is summed.

ANN can be divided into several categories namely: feed forward neural network, back propagation neural network and Radial Basis Function (RBF) network. These three networks are most commonly used in ANN.

Feed Forward

The simplest type of ANN is the feed forward network, which is otherwise known as a multilayer perceptron. A feed forward network has no cycles since the information moves forward in only one direction, i.e., from the input nodes to the output nodes via the hidden nodes.

Back Propagation

Back propagation is the technique of estimating the error made in the output neuron and propagating them back to the inner neuron. This method is otherwise referred to as “learn by examples”. Thus, a learning set comprising the input examples for each case must be included. The output value and each of the examples in the training set are compared, and an error value is calculated. This error value is then propagated backwards to the neuron and is used to adjust the weights. The error value is reduced by making small changes to the weight value. This process is repeated until a pre-determined value is reached.

The back propagation neural network can be used to recognize and classify the different aspects of the image of a human’s face like the expression of the person in the image, or the orientation of the face in the image, or the presence of any accessories such as sunglasses, beard, etc.

Radial Basis Function (RBF)

Radial Basis Function (RBF) is used for estimating the functions and identifying patterns. It applies the Gaussian potential functions whereby these functions are used in the networks. RBF provides advantages such as localization, cluster modelling, functional approximation, interpolation and quasi-orthogonality.

Chapter 3: Transfer Learning From Pre-Trained Models

3.1 Introduction

Deep learning is fast becoming popular in artificial intelligence applications. For instance, in areas such as computer vision, natural language processing, and speech recognition, deep learning has been bringing about remarkable results. Therefore, the interest in deep learning is growing. One of the areas where deep learning is exceptionally good is *image classification*. The goal in image classification is to categorize a specific picture according to a set of possible categories. A good example of image classification is to identify cats and dogs in a set of pictures. From a deep learning perspective, the image classification problem can be solved via *transfer learning*. Several state-of-the-art results in image classification are based upon transfer learning solutions.

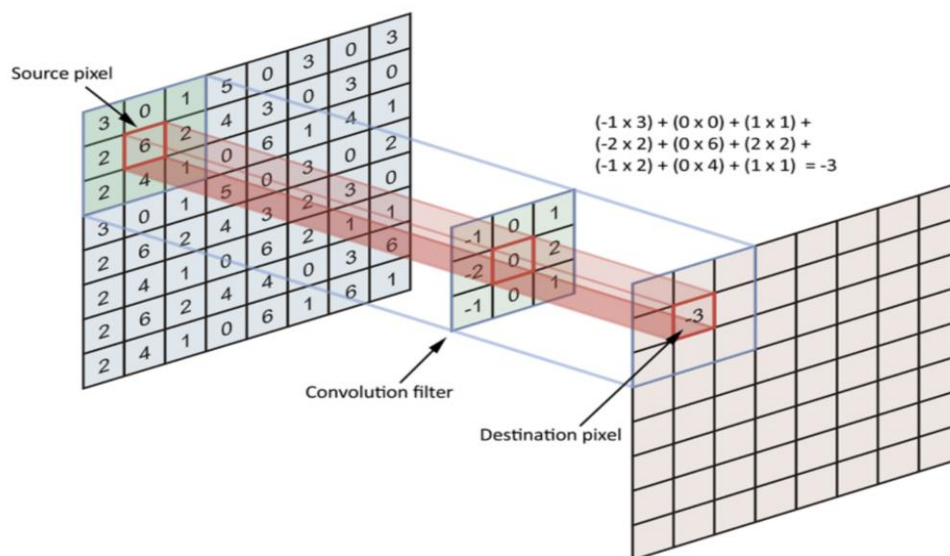
3.2 Convolutional Neural Networks

A convolutional neural network (CNN) is a type of artificial neural network used in image recognition and processing that is especially designed to process pixel data. CNNs have their “neurons” arranged more like those of the frontal lobe, the area responsible for processing visual stimuli in humans and other animals. A CNN has one or more convolutional layers and are mainly used for image processing, classification, segmentation and also for other auto correlated data. Convolution refers to sliding of a filter over the input. The influence of nearby pixels is studied using something called a filter. A filter of a size specified by the user (a rule of thumb is 3x3 or 5x5) is taken and moved across the image from top left to bottom right. For each point on the image, a value is estimated based on the filter using a convolution operation.

A filter could be related to anything, like for pictures of humans, one filter could be associated with seeing noses, and the nose filter would indicate how strongly a nose seems to appear in the image, and how many times and in what locations they occur. This reduces the number of weights that the neural network must learn compared to a MLP, and also means that when the location of these features differs it does not throw the neural network off.

Figure 2

The Convolution Operation



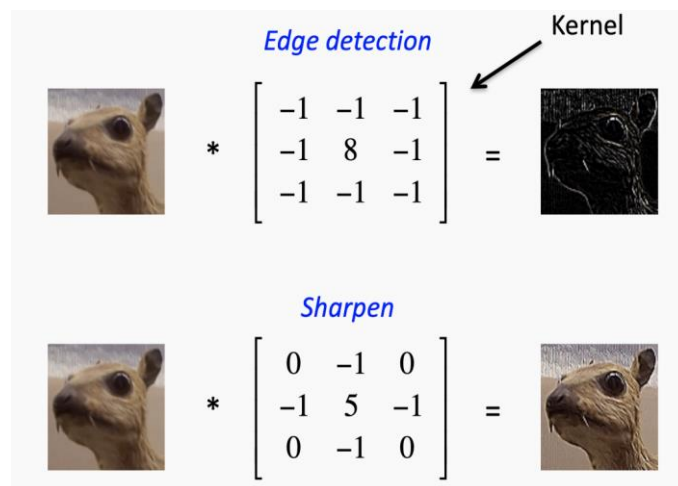
Note. A convolution is an integral that expresses the amount of overlap of one function (kernel or filter) as it is shifted over another function (input). (Kana, 2020).

It may be wondered how the different features are learned by the network, and whether it is possible that the network will learn the same features i.e., having 10 nose filters would be kind of redundant and is highly unlikely to happen. While building the network, random values are specified for the filters, which then continuously update themselves as the network is trained. It is very unlikely that two

filters that are the same will be produced unless the number of chosen filters is extremely large. Some examples of filters, or kernels, are given below:

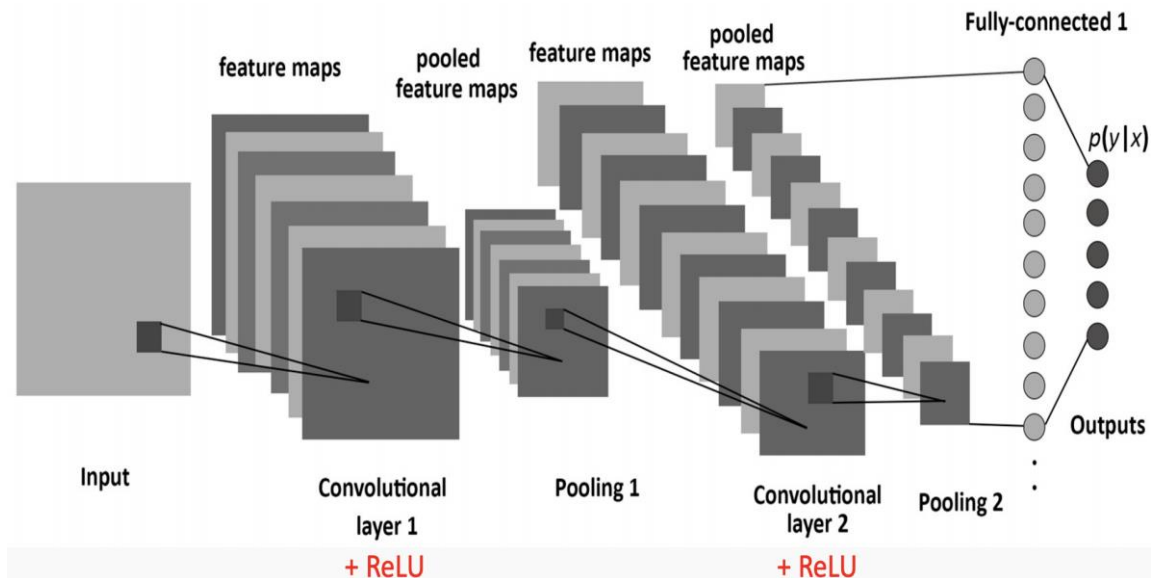
Figure 3

Examples of Kernel Filters for CNN



Note. Specific kernels can be used to extract edges from an image. We can have many of those filters to detect all the valuable features of the image. (Kana, 2020).

After the filters pass over the image, a feature map is generated for each filter. These then go through an activation function, which decides whether a certain feature is present at a given location in the image. Then a lot of things can be done, such as adding more filtering layers and creating more feature maps, which become more and more abstract as a deeper CNN is created. We can also use pooling layers to select the largest values on the feature maps and then use these as inputs to subsequent layers. In theory, any type of operation can be carried out in pooling layers, but in practice, only max pooling is used in order to find the outliers - these are when our network spots the feature.

Figure 4*An example of CNN*

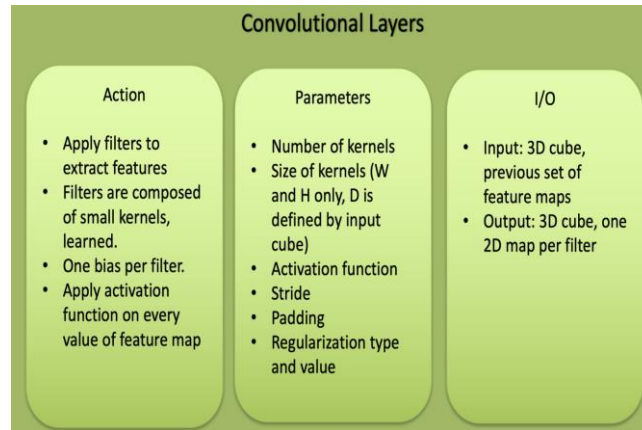
Note. This figure shows a CNN with two convolutional layers, two pooling layers, and a fully connected layer which decides the final classification of the image into a particular category (Nigam, 2018).

ReLU

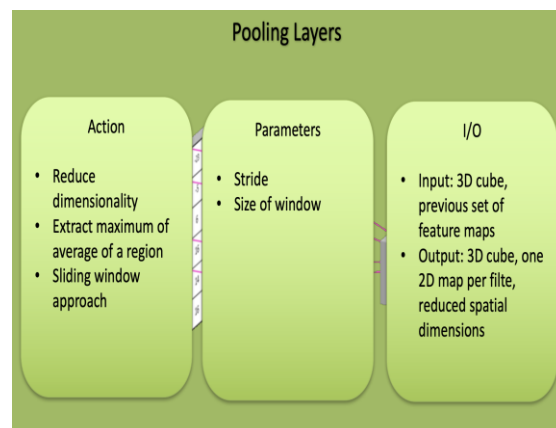
The most successful non-linearity for CNN's is the Rectified Linear unit (ReLU), which combats the vanishing gradient problem occurring in sigmoids. ReLU is easy to compute and generates sparsity, which is not always beneficial.

Comparison of Different Layers

There are three types of layers in a convolutional neural network namely; convolutional layer, pooling layer, and fully connected layer. Each of these layers have different parameters that can be optimized to perform a different task on the input data.

Figure 5*Features of a Convolutional Layer*

In the convolutional layers, the filters are applied to the original image, or to other feature maps in a deep CNN. Most of the user-specified parameters are there in these layers of the network. The most significant parameters are the number of kernels and the size of the kernels.

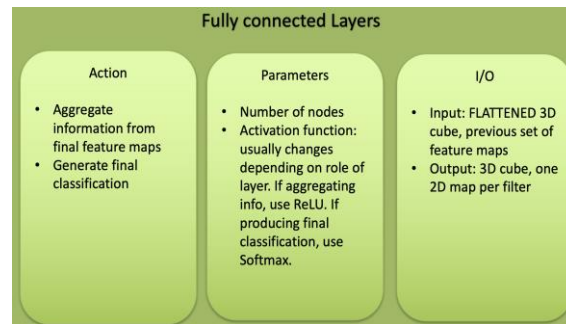
Figure 6*Features of a Pooling Layer*

Pooling layers, similar to convolutional layers perform a specific function such as max pooling, which takes the maximum value in a certain filter region, or average

pooling, which takes the average value in a filter region. These are essentially used to reduce the dimensionality of the network.

Figure 7

Features of a Fully Connected Layer



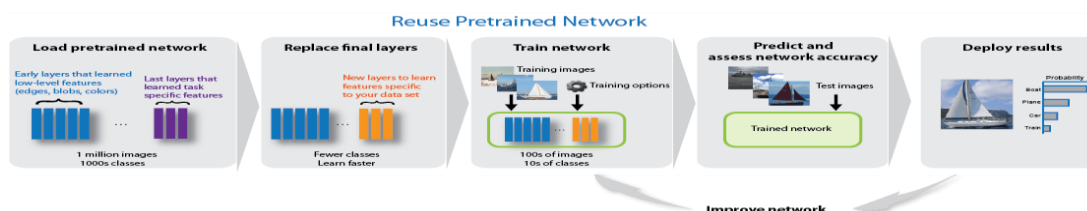
Fully connected layers are located before the classification output of a CNN and are used to flatten the results prior to classification.

3.3 Transfer Learning Process

Transfer learning is a method in deep learning used to use an existing network as a starting point to learn a new task. This concept is used to perform face recognition. The various steps involved are shown in the following figure:

Figure 8

Transfer Learning Using Pretrained Network



Note. This example shows how to fine-tune a pretrained GoogLeNet convolutional neural network to perform classification on a new collection of images. (MathWorks, 2022).

Designing a new network, optimizing the architecture for maximum accuracy, specifying the effective initial weights of the hidden nodes is a time-consuming and lengthy process. If we go for transfer learning, we get an already optimized network ready to learn new features to perform new tasks. Transfer learning helps us get things done using a neural network with minimal effort. Hence, we can perform face recognition with minimal effort by modifying a convolutional neural network (CNN).

3.4 Pre-Trained Models for Computer Vision

Here are the four pre-trained networks one can use for computer vision tasks such as ranging from image generation, neural style transfer, image classification, image captioning, anomaly detection, and so on:

1. VGG19
2. Inceptionv3 (GoogLeNet)
3. ResNet50
4. EfficientNet

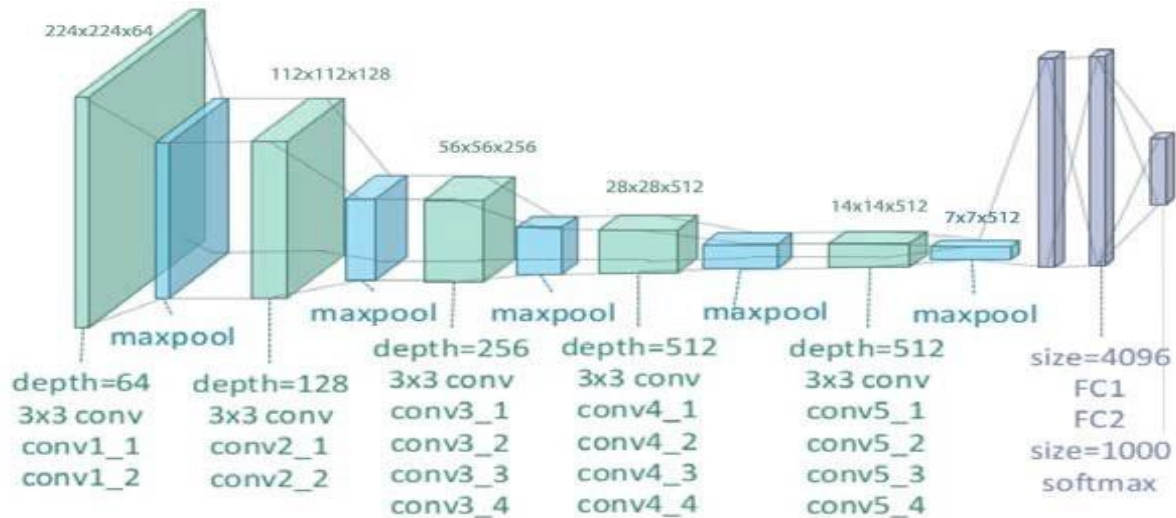
VGG-19

VGG is a convolutional neural network having a depth of 19 layers. It was built and trained by Karen Simonyan and Andrew Zisserman at the University of Oxford in 2014.

The VGG-19 network is trained on more than 1 million images from the ImageNet database. Naturally, one can import the model with the ImageNet trained weights. This pre-trained network can categorize up to 1000 objects. The network was trained on 224x224 pixels coloured images.

Figure 9

An Illustration of the VGG-19 Network



Note. A brief info about the size and performance of the VGG-19 Network: Size: 549 MB, Top-1: Accuracy: 71.3%, Top-5: Accuracy: 90.0%, Number of Parameters: 143,667,240, Depth: 26 (Yalcin, 2020).

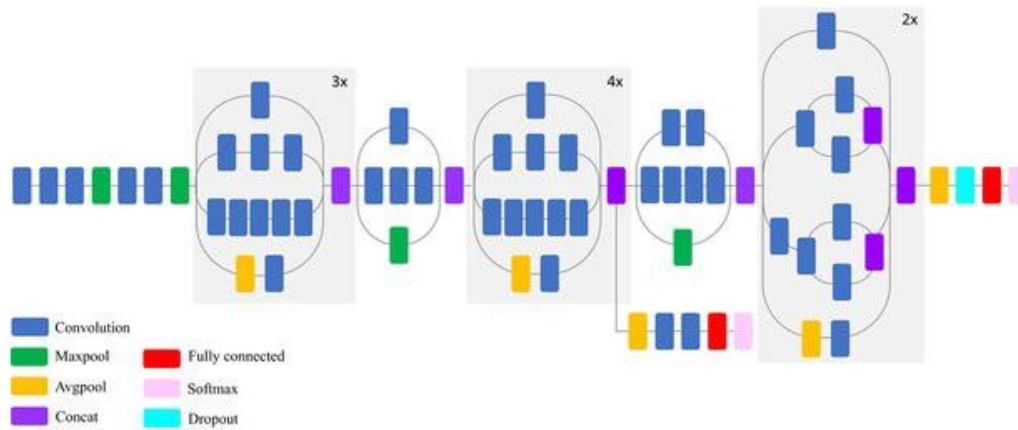
Inceptionv3 (GoogLeNet)

Inceptionv3 is a convolutional neural network having a depth of 50 layers. It was built and trained by Google.

The pre-trained version of Inceptionv3 with the weights of ImageNet can classify up to 1000 objects. The image input size of this network is 299x299 pixels, that is larger than the VGG19 network.

Figure 10

An Illustration of the Inceptionv3 Network



Note. The brief summary of Inceptionv3 features is as follows: Size: 92 MB, Top-1: Accuracy: 77.9%, Top-5: Accuracy: 93.7%, Number of Parameters: 23,851,784, Depth: 159 (Yalcin, 2020).

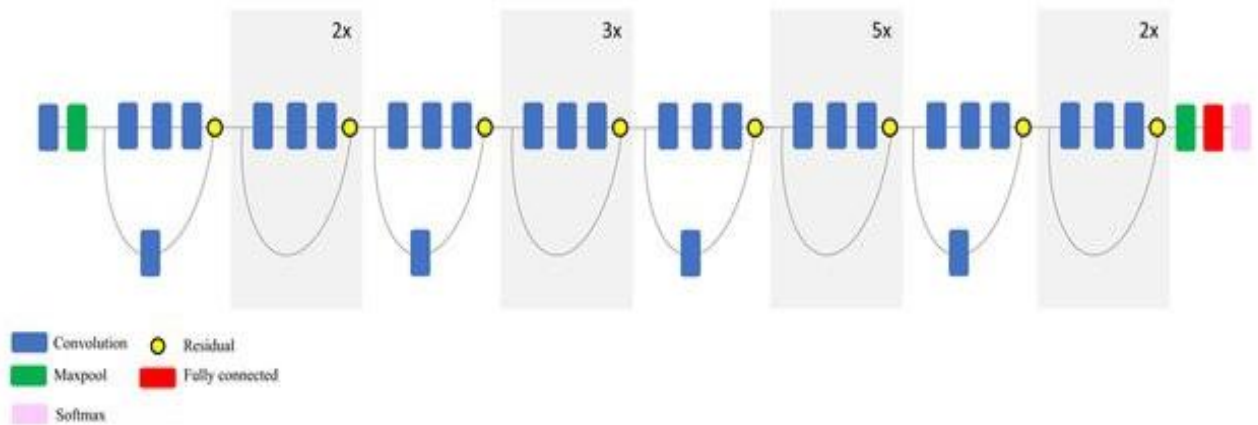
ResNet50 (Residual Network)

ResNet50 is a convolutional neural network having a depth of 50 layers. It was built and trained in 2015 by Microsoft. This model is trained on more than 1 million images from the ImageNet database.

Just like VGG-19, it can classify up to 1000 objects and the network was trained on 224x224 pixels coloured images. If ResNet50 is compared to VGG19, it can be seen that ResNet50 actually outperforms VGG19 even though it has lower complexity.

Figure 11

An Illustration of the ResNet50 Network



Note. A brief info about the size and performance of the ResNet50 Network: Size: 98 MB, Top-1: Accuracy: 74.9%, Top-5: Accuracy: 92.1%, Number of Parameters: 25,636,712 (Yalcin, 2020).

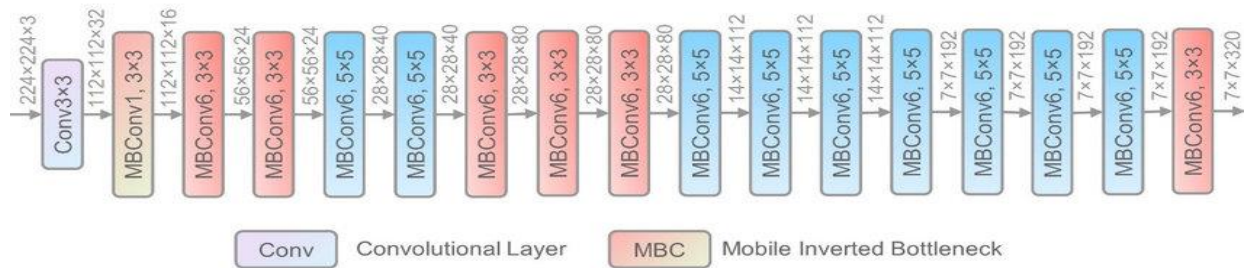
EfficientNet

EfficientNet is a state-of-the-art convolutional neural network which was trained and released to the public by Google in 2019 with the paper “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks”.

There are 8 alternative implementations of EfficientNet (B0 to B7) and even the simplest one, EfficientNetB0, is great. With 5.3 million parameters, it attains a 77.1% Top-1 accuracy with respect to performance.

Figure 12

The Network Architecture of EfficientNet



Note. The brief summary of EfficientNetB0 features is as follows: Size: 29 MB, Top-1: Accuracy: 77.1%, Top-5: Accuracy: 93.3%, Number of Parameters: ~5,300,000, Depth: 159 (Yalcin, 2020).

Chapter 4: Design Methodology

In this starred paper, face recognition is performed using two pretrained models and their performance is compared. These models are:

1. GoogLeNet Convolutional Neural Network
2. Google's deep convolutional network - FaceNet

4.1 Transfer Learning Using Pretrained GoogLeNet CNN

For the evaluation of face recognition system using transfer learning, techniques with pre-trained CNN model will be presented in this work.

The proposed system, of face recognition algorithm is implemented in MATLAB environment. In this, we are using the Deep Learning Toolbox GoogLeNet Network. We are creating a database to perform training, validation and testing on the proposed method.

The images are captured using the inbuilt laptop webcam, and the resolution of the images for the same is 1280x720 pixels. The resolution of the images captured is 640x480 pixels and the average size of the faces in these images is 224x224 pixels.

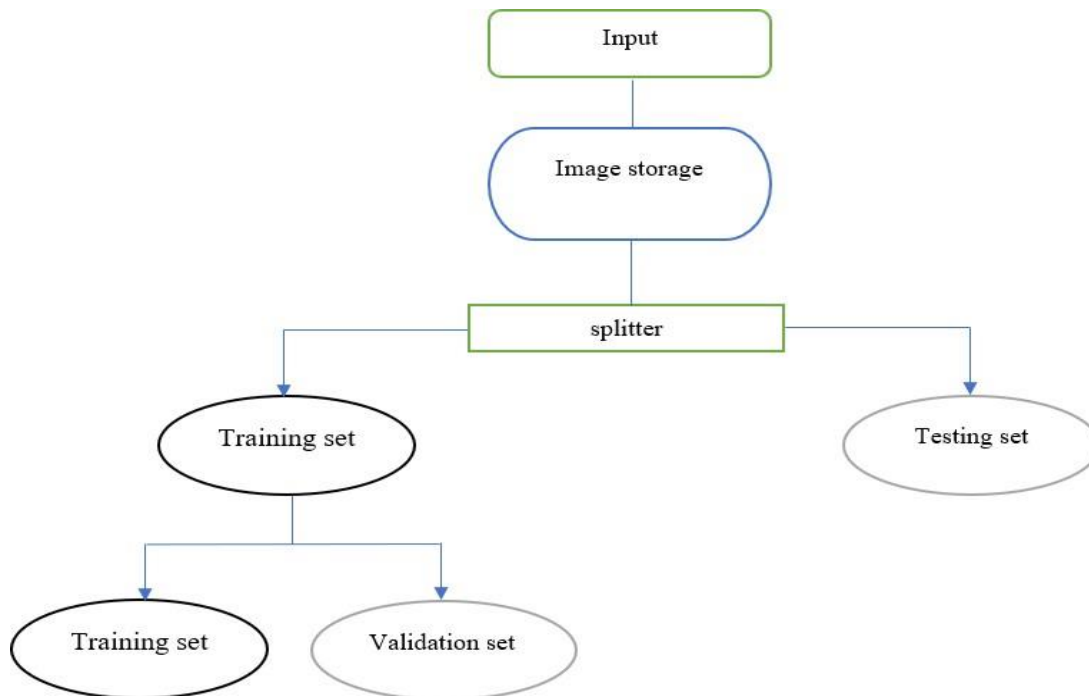
4.1.1 Data Splitting

The first step is to split the dataset into a few sets. In MATLAB, input images are stored inside data-store by using the "imageDatastore" function and the dataset is randomly split into two sets which are training and test set using the "split label" function. Images from the training set are then randomly split into training and validation set for train and evaluate the model. The training set is used to train the network while the validation set is used to predict the accuracy of the trained model. The flow chart of the dataset splitting is shown in Figure 13.

4.1.2 Compose the model

Figure 13

Dataset Splitting



In this proposed system, pre-trained CNN is used as a feature extractor and transfer learning by fine-tuning technique is used to transfer the extracted features from the pre-trained CNN model to a new task. In this section, the pre-trained CNN architecture and the fine-tuning pre-trained CNN architecture is described in detail.

4.1.3 Pre-trained CNN Model as Feature Extractor

GoogLeNet pre-trained CNN model is selected as the feature extractor in this proposed system. GoogLeNet has been trained on more than a million images from the ImageNet database and it can classify images into 1000 object categories with about 60 million parameters.

Features of GoogLeNet

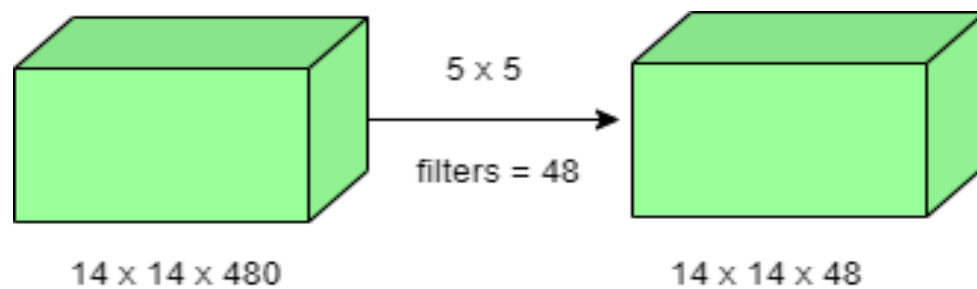
The GoogLeNet architecture is very different from previous state-of-the-art architectures like AlexNet and ZF-Net. It uses different kinds of methods such as 1×1 convolution and global average pooling that enables it to create a deeper architecture. Some of the methods in the architecture are:

1×1 Convolution. 1×1 convolution is used in the inception architecture. These convolutions are used to minimize the number of parameters (weights and biases) of the architecture. By reducing the parameters, the depth of the architecture is increased. An example of a 1×1 convolution is shown below:

For instance, if one wants to perform 5×5 convolution having 48 filters without using 1×1 convolution as intermediate:

Figure 14A

An example of GoogLeNet Convolution Operation Without 1×1 Convolution



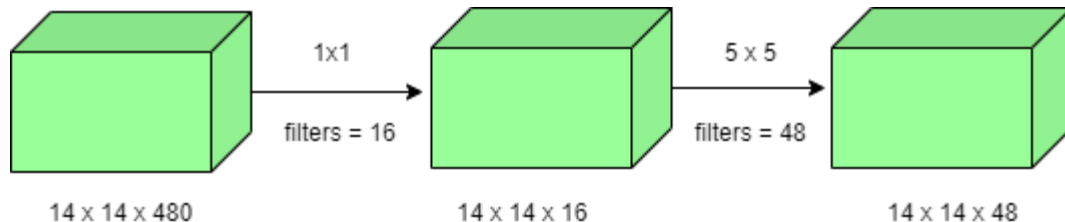
Note. Total Number of operations: $(14 \times 14 \times 48) \times (5 \times 5 \times 480) = 112.9 \text{ M}$

(GeeksforGeeks, 2021).

With 1×1 convolution:

Figure 14B

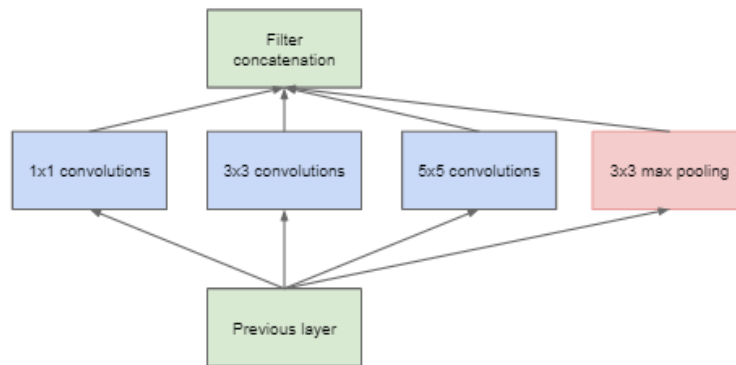
An Example of GoogLeNet Convolution Operation with 1×1 Convolution



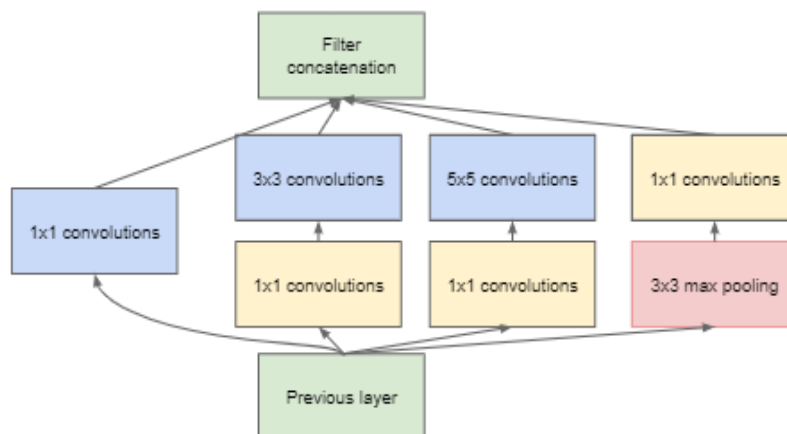
Note. $(14 \times 14 \times 16) \times (1 \times 1 \times 480) + (14 \times 14 \times 48) \times (5 \times 5 \times 16)$ is $1.5M + 3.8M = 5.3M$ that is much smaller than $112.9M$. (GeeksforGeeks, 2021).

Global Average Pooling. In the previous architecture such as AlexNet, the fully connected layers are utilized at the end of the network. These fully connected layers comprise the majority of parameters of many architectures that causes an increase in computation cost. In GoogLeNet architecture, there is a method called global average pooling that is used at the end of the network. This layer considers a feature map of 7×7 and averages it to 1×1 . This also reduces the number of trainable parameters to 0 and enhances the top-1 accuracy by 0.6%

Inception Module. The inception module differs from previous architectures such as AlexNet, ZF-Net. In this architecture, the convolution size is fixed for each layer. In the Inception module 1×1 , 3×3 , 5×5 convolution and 3×3 max pooling performed parallelly at the input and the output of these are stacked together to generate the final output. The idea behind this is that convolution filters of different sizes will handle objects at multiple scale better.

Figure 15A*Inception Module*

Note. The inception module is different from previous architectures such as AlexNet, ZF-Net (GeeksforGeeks, 2021).

Figure 15B*Inception Module with Dimension Reductions*

Note. In the Inception module 1×1 , 3×3 , 5×5 convolution and 3×3 max pooling performed in a parallel way at the input and the output of these are stacked together to generated final output. (GeeksforGeeks, 2021).

Auxiliary Classifier for Training. Inception architecture use some intermediate classifier branches in the middle of the architecture, these branches are used while training only. These branches comprise a 5×5 average pooling layer with a stride of 3, a 1×1 convolutions with 128 filters, two fully connected layers of 1024 outputs and 1000 outputs and a softmax classification layer. The generated loss from these layers is added to total loss with a weight of 0.3. These layers help in withstanding gradient vanishing problem and also provide regularization.

Model Architecture

The table below shows the layer-by-layer architectural details of GoogLeNet.

Table 1

Architectural Details of GoogLeNet

type	patch size/ stride	output size	depth	# 1×1	# 3×3 reduce	# 3×3	# 5×5 reduce	# 5×5	pool proj	params	ops
convolution	$7 \times 7 / 2$	$112 \times 112 \times 64$	1							2.7K	34M
max pool	$3 \times 3 / 2$	$56 \times 56 \times 64$	0								
convolution	$3 \times 3 / 1$	$56 \times 56 \times 192$	2		64	192				112K	360M
max pool	$3 \times 3 / 2$	$28 \times 28 \times 192$	0								
inception (3a)		$28 \times 28 \times 256$	2	64	96	128	16	32	32	159K	128M
inception (3b)		$28 \times 28 \times 480$	2	128	128	192	32	96	64	380K	304M
max pool	$3 \times 3 / 2$	$14 \times 14 \times 480$	0								
inception (4a)		$14 \times 14 \times 512$	2	192	96	208	16	48	64	364K	73M
inception (4b)		$14 \times 14 \times 512$	2	160	112	224	24	64	64	437K	88M
inception (4c)		$14 \times 14 \times 512$	2	128	128	256	24	64	64	463K	100M
inception (4d)		$14 \times 14 \times 528$	2	112	144	288	32	64	64	580K	119M
inception (4e)		$14 \times 14 \times 832$	2	256	160	320	32	128	128	840K	170M
max pool	$3 \times 3 / 2$	$7 \times 7 \times 832$	0								
inception (5a)		$7 \times 7 \times 832$	2	256	160	320	32	128	128	1072K	54M
inception (5b)		$7 \times 7 \times 1024$	2	384	192	384	48	128	128	1388K	71M
avg pool	$7 \times 7 / 1$	$1 \times 1 \times 1024$	0								
dropout (40%)		$1 \times 1 \times 1024$	0								
linear		$1 \times 1 \times 1000$	1							1000K	1M
softmax		$1 \times 1 \times 1000$	0								

Note. The overall architecture is 22 layers deep. (GeeksforGeeks, 2021).

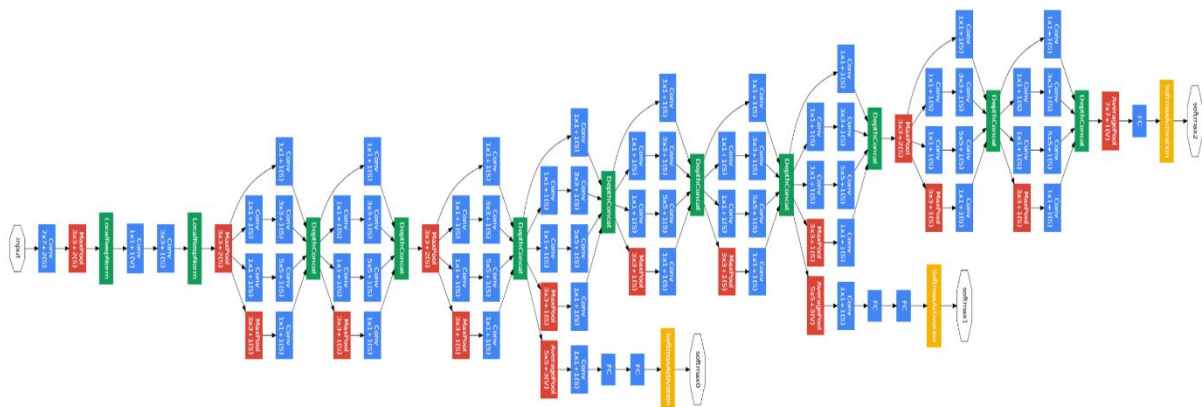
The architecture was designed keeping computational efficiency in mind. The idea behind this was that the architecture can be run on individual devices even with low computational resources. The architecture also comprises two auxiliary classifier layers connected to the output of Inception (4a) and Inception (4d) layers.

The architectural details of auxiliary classifiers are as follows:

1. An average pooling layer of filter of size 5x5 and stride 3.
2. A 1x1 convolution having 128 filters for dimension reduction and ReLU activation.
3. A fully connected layer containing 1025 outputs and ReLU activation
4. Dropout Regularization having dropout ratio = 0.7
5. A softmax classifier with output of 1000 classes similar to the main softmax classifier.

Figure 16

Architecture of GoogLeNet CNN



Note. This architecture considers image of size 224 x 224 with RGB color channels. (GeeksforGeeks, 2021).

All the convolutions inside this architecture have Rectified Linear Units (ReLU) as their activation functions.

4.1.4 Fine-tuning pre-trained CNN model

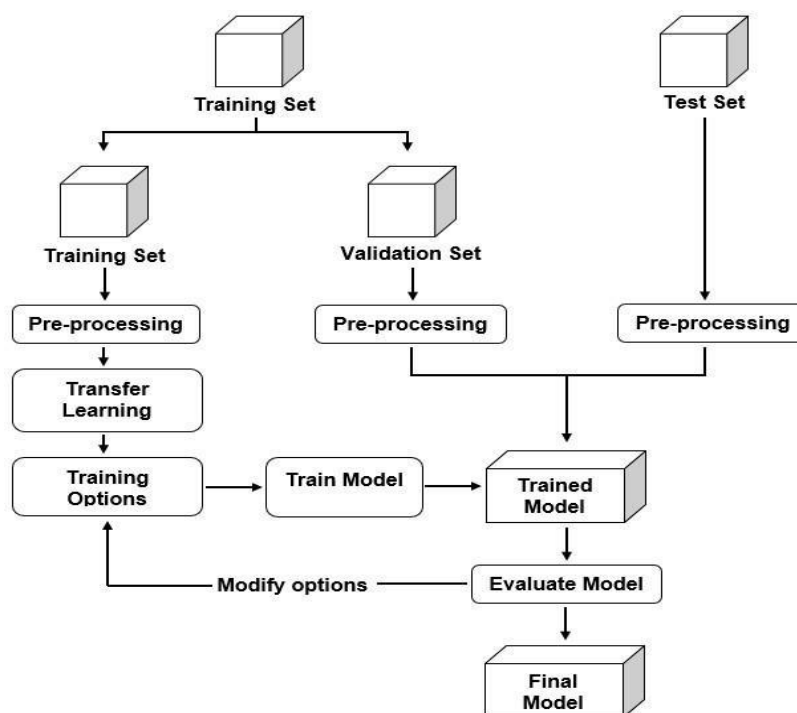
In this, transfer learning by fine-tuning of a pre-trained CNN model is used to classify the images by using a smaller number of training images. Pre-trained CNN model has learned rich feature representations for a large scale of images.

Therefore, transfer learning from the pre-trained CNN model is much faster and

easier than training a network from scratch. This proposed approach is using GoogLeNet pre-trained CNN architecture and transfer the layers to the new classification task by replacing the final layers: fully connected layer, and the classification output layer for a new task.

Figure 17

Flow Chart of Train and Evaluate the Model



Training Process

In transfer learning, training data with labels, fine-tune network, and training algorithm options were the three important things that need before train a network. Training data is stored in an augmented data store and pre-processed while the fine-tuned network is the network that transfers learning from GoogLeNet CNN pre-trained network to perform a newtask. For training algorithm options, it was the most important component in the training process as it can control the behavior of the training algorithm. In MATLAB, the “trainingoptions” function is used

to set up the training options such as optimizer uses, initial learning rate, mini-batch size, maximum epochs, optimizer, validation frequency, and many more should be specified before starting to train the model. In this proposed system, root mean square prop (RMSprop) optimizer is used as the solver for a training network, which can also accelerate gradient descent.

Optimizer is updated the weight parameters to minimize the loss function. For the learning rate, it was the most important hyper-parameter when configuring the network as it controls the change of the model in response to the estimated error each time the model weights are updated. Choosing a learning rate is challenging as the value too small may result in a long training process that could get stuck, whereas the value too large may result in learning a sub-optimal set of weights too fast or an unstable training process. At first, I am fine-tuning the learning rate and choose a suitable learning rate for the network. A larger value such as 0.1 is started to try on the network then reduce the initial learning rate exponentially until 0.0001. Finally, the learning rate was set to a small value as 0.0001 to slow learning down, since the pre-trained CNN is used. Besides, the size of the mini-batched is set to 30. In this study, different number of epochs are used to test the model until it reaches a good accuracy. The loss and gradient calculated for each mini-batched approximates the loss and gradient for the full training set.

Evaluating on Trained Model

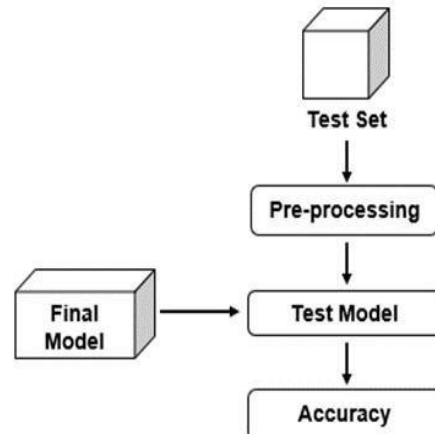
In this, training accuracy, validation accuracy, testing accuracy, mini-batch loss, and validation loss are the evaluation parameters used for the experiments. For the training process, the validation set is used to evaluate and validate the performance of during training. In MATLAB, training progress can be plot using

“Plots” and “training-progress” function in training options. From the training progress, training accuracy, validation accuracy, training loss value, and validation loss value can be observed to prevent over-fitting. It happens if the training accuracy is greater than the validation accuracy and the validation loss value training loss value is greater than the training loss value. In the other hand, if the validation accuracy is higher than the training accuracy and the validation loss value is lower than the training loss value, it was under-fitting. Apart from that, the confusion matrix table also used to observe the correct and error between the ground truth images and the predicted images.

4.1.5 Test on final model

Figure 18

Flow Chart of Testing on Trained Model



In this, training accuracy, validation accuracy, testing accuracy, mini-batch loss, and validation loss are the evaluation parameters used for the experiments. For the training process, the validation set is used to evaluate and validate the performance of during training.

4.2 Face Recognition Using Google’s Deep Convolutional Network - FaceNet

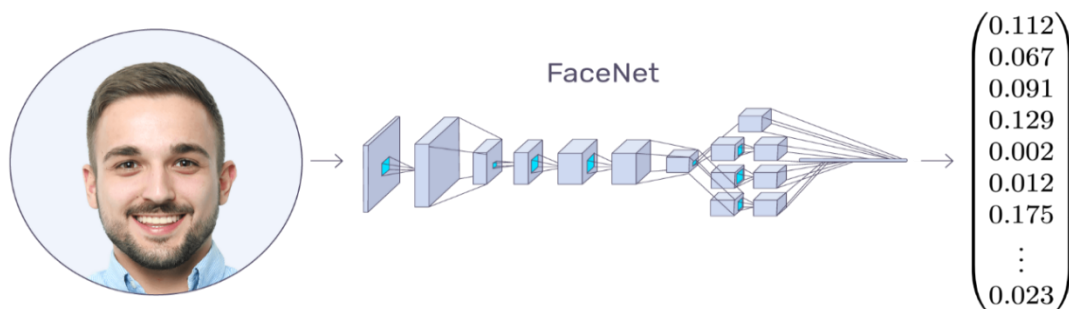
FaceNet is a deep neural network used for obtaining features from an image of a person's face. It was published by Google researchers Schroff et al in 2015.

FaceNet employs two different core architectures:

1. Zeigler & Fergus Style Network
2. Inception Network

Figure 19

Facenet Model



Note. FaceNet model takes an image of a face as input and outputs the embedding vector. (Dulčić, 2020).

FaceNet takes an image of the person's face as input and outputs a vector of 128 numbers which characterize the most important features of a face. In machine learning, this vector is known as *embedding*. Embedding is required because all the important information from an image is *embedded* into this vector. Basically, FaceNet takes a person's face and flattens it into a vector of 128 numbers. Ideally, embeddings of similar faces are similar as well.

FaceNet presents a unified system for face verification (is this the same person?), recognition (who is this person?) and clustering (finds common people

among these faces) using the method based on learning a Euclidean embedding per image using a deep convolutional network. The similarity of faces relates to the squared L2 distances of the embeddings of 128 dimensions learned using triplet loss function.

Once these embedding are produced, then the next tasks become straight-forward:

1. face verification simply refers to thresholding the distance between the two embeddings;
2. recognition represents a k-NN classification problem;
3. and clustering can be achieved with the help of off-the-shelf techniques like k-means or agglomerative clustering.

Embeddings are vectors and the vectors can be interpreted as points in the Cartesian coordinate system. This means one can plot an image of a face in the coordinate system by using its embeddings. FaceNet embedding vectors have 128 numbers, i.e., they are 128-dimensional. Since we live in a 3-dimensional world, we cannot plot a 128-dimensional vector. It can be pretended that faces can be plotted in 2D for simplicity, the same logic applies for 2D and 128D, but unlike 128D, one can visualize 2D.

One possible method of recognizing a person on an unseen image would be to estimate its embedding, calculate distances to images of known people and if the face embedding is close enough to embeddings of person A, one can say that this image contains the face of person A.

Figure 20

Architecture of FaceNet



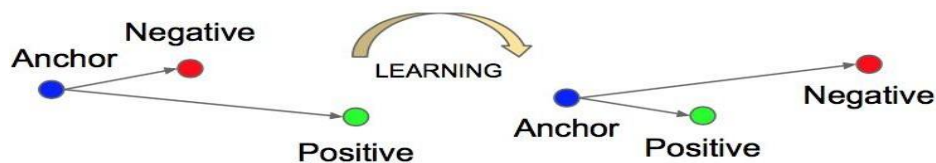
Note. When the CNN architecture is treated as a blackbox, the most important aspect of FaceNet lies in the end-to-end learning of the system. (GeeksforGeeks, 2022).

FaceNet searches for an embedding $f(x)$ from an image into feature space \mathbb{R}^d , such that the squared L_2 distance between all face images (independent of imaging conditions) of the same identity is small, while on the contrary, the distance between a pair of face images from different identities is large.

While the previously used losses encourage all faces of the same identity onto a single point in \mathbb{R}^d , the triplet loss additionally tries to impose a margin between each pair of faces from one person (anchor and positive) to all others' faces. This margin applies discriminability to other identities.

Figure 21

Triplet-loss and learning



Note. Generating triplets on every step on the basis of previous checkpoints and compute minimum and maximum on a subset of data. (GeeksforGeeks, 2022).

Triplet Loss

Triplet-loss training intends at learning score vector identity verification by comparing facial descriptors in Euclidean space. This is similar to “metric learning”, and, like many other metric-learning approaches, is used to learn a projection that is at the same time distinguishing and compact, achieving dimensionality reduction at the same time. The function is defined as,

$$\sum_i^N \left[\|f(x_i^a) - f(x_i^p)\|_2^2 - \|f(x_i^a) - f(x_i^n)\|_2^2 + \alpha \right]_+$$

Triplet Function

Triplet loss is a loss function in artificial neural networks where a baseline (anchor) input is compared to a positive (truthy) input and a negative (falsy) input. The distance from the baseline (anchor) input to the positive (truthy) input is decreased, and the distance from the baseline (anchor) input to the negative (falsy) input is increased. Once the FaceNet model is trained, one can create the embedding for the face by feeding it into the model. In order to compare two images, the embedding is created for both images by feeding through the model separately. Then, the above formula can be used to find the distance, which will be lower for similar faces and higher for different faces. The formula for calculating the Euclidean distance between two points is as below,

$$\begin{aligned} d(\mathbf{p}, \mathbf{q}) = d(\mathbf{q}, \mathbf{p}) &= \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \dots + (q_n - p_n)^2} \\ &= \sqrt{\sum_{i=1}^n (q_i - p_i)^2}. \end{aligned}$$

Triplet Selection

Producing all possible triplets would result in many triplets that satisfy the distance condition mentioned above, thus not contributing towards learning and would also slow down the convergence. These triplets are referred to as Easy Triplets and will always give loss 0 as $d(x^a, x^{p_i}) + \alpha < d(x^a, x^{n_i})$.

Thus, to have faster convergence and better learning, those triplets which violate the triplet constraint (referred as Hard Triplets) i.e. $d(x^a, x^{n_i}) < d(x^a, x^{p_i})$ are required.

Choosing the triplets to be used plays an important role in achieving good performance. Inspired by curriculum learning, a novel online negative exemplar mining strategy is presented, which ensures consistently increasing complexity of triplets as the network trains. The two methods discussed were:

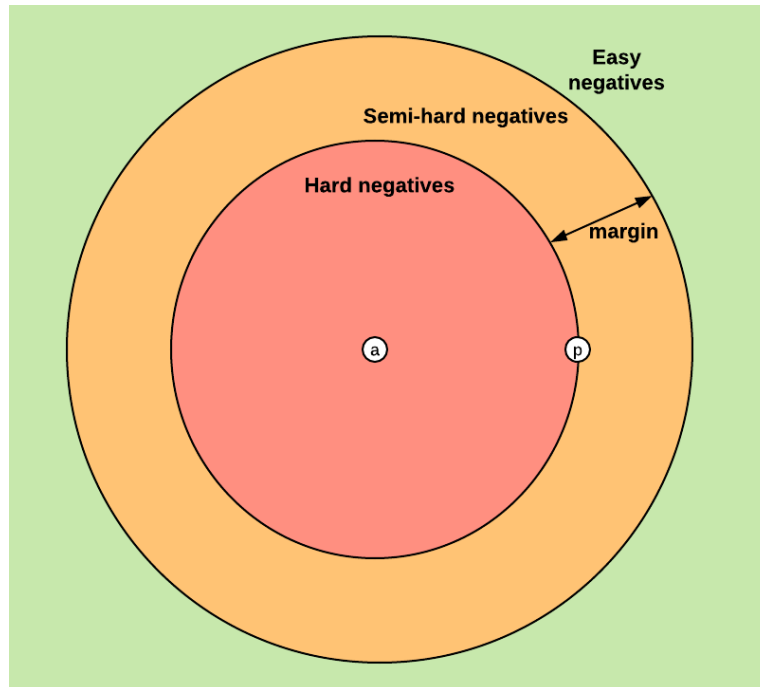
1. *Generate triplets offline*

For every n epochs, the embeddings on the subset of data are computed and only the hard triplets i.e $\operatorname{argmax}_{x^a, x^p_i} (d(x^a, x^p_i))$ and $\operatorname{argmin}_{x^a, x^n_i} (d(x^a, x^n_i))$ are selected. Evaluating offline triplet mining resulted in inconclusive results and is hence not very efficient.

2. *Generate triplets online*

This can be done by choosing the hard positive/negative exemplars from within a mini-batch. Instead of just the hardest positive ($\operatorname{argmax}_{x^a, x^p_i} (d(x^a, x^p_i))$), all anchor-positive pairs in a mini batch are chosen, while still selecting the hard negatives. There is no side-by-side comparison of hard anchor-positive pairs versus all anchor-positive pairs within a mini-batch, but it can be found in practice that all anchor-positive method was more stable and converged slightly faster during the beginning of training.

Also, instead of choosing just hardest negative ($\operatorname{argmin}_{x^a, x^n_i} (d(x^a, x^n_i))$), the negative exemplars are selected in such a way that: $d(x^a, x^p_i) < d(x^a, x^n_i)$. These are known as Semi-Hard Triplets and lie within a margin α . Selecting the hardest negatives can in practice cause bad local minima early on in training, specifically it can lead to a collapsed model (i.e., $f(x)=0$).

Figure 22*Triplet Selection*

Note. Choosing which triplets to use turns out to be very important for achieving good performance and, inspired by curriculum learning, we present a novel online negative exemplar mining strategy which ensures consistently increasing difficulty of triplets as the network trains. (Abdullah, 2021).

Chapter 5: Experiments and Results

5.1 Implementation using Pretrained GoogLeNet CNN

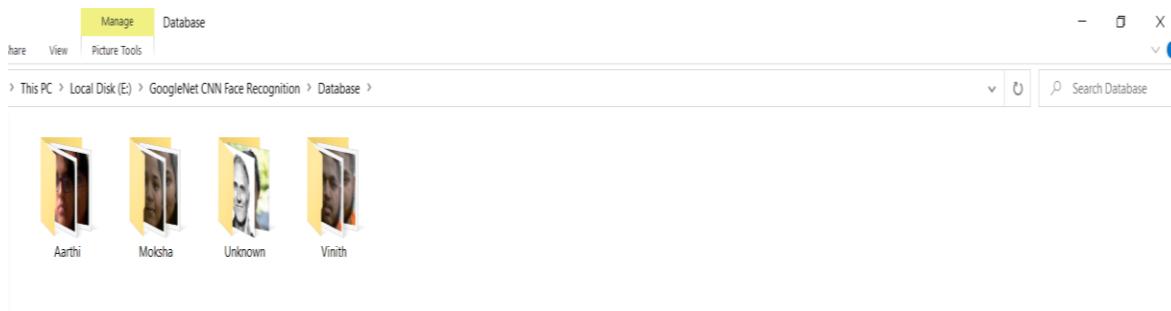
Initial steps

Preparing the Dataset. The dataset has been prepared manually. Following is the procedure to generate the dataset:

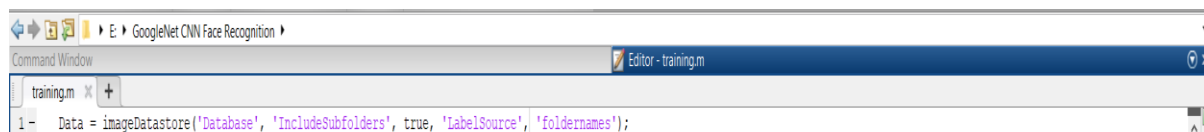
Here the name of the dataset is 'Database.' Inside of this 'Database' folder, there are folders named by the person. Each of these folders have images of corresponding person. There is also a folder with faces of random people taken from the internet. It is saved as 'Unknown'.

Figure 23

Dataset



Loading the Dataset. After preparing the dataset, the first step is to load the dataset. The dataset should then be split into training and validation set. In MATLAB, the function used to handle a large collection of images is 'imageDatastore()'. This function needs the location of the image folder. The subfolders are included and the labels of the images are specified using this function. The code below shows the use of 'imageDatastore()'.



The dataset is inside the 'Database' folder. It is located in the working directory. That is 'Database' has been used as the first argument of the 'imageDatastore()' function. The data inside the dataset are arranged into different folders.

To force the function to include the subfolder, 'IncludeSubfolders' should be set to 'true'. The folder names refer to the labels of the categories of dataset.

This is why 'LabelSource', 'foldernames' has been used as the argument to the function. Finally, the dataset is stored in a variable named 'Dataset'. After organizing the dataset, it is split into training and validation set. In MATLAB, the 'splitEachLabel()' function is used to do it. This function needs two arguments. The first argument is the dataset one needs to split, and the second argument is the ratio of splitting. In the following code, the ratio is set to 7/10. It means that the data will be split into two groups wherein one group will have 70% of the data and other group will have 30% of the data.

```
2- [Training_Data, Validation_Data] = splitEachLabel(Data, 7.0);
3
```

Here, the 70% data are stored in 'Training_Dataset' variable and rest of the 30% of the data are stored in 'Validation_Dataset' variable. The dataset loading part is now completed. Next, the pre-trained network must be loaded.

5.1.1 Loading the Pre-Trained Network

Loading the pre-trained network is the simplest step. Nothing is needed except calling the network. However, it should be made sure that it is installed in MATLAB.

Otherwise, one will not be able to call it. Using the following line, the 'GoogleNet' is called and stored in a variable named 'net':

```
net = googlenet;
```

This network needs to be modified. In order to do it, some idea about the structure of the network is required. The following code can be used to visualize the network:

```
analyzeNetwork(net)
```

It will depict the graphical structure of the network. It is a lengthy network; however, it is not required to analyze the entire network. The main focus is on the fully connected classification layer named as 'loss3-classifier'. Apart from the classification layer, it is important to have idea about the input layer as well. The information about the input layer can be accessed using the following code:

```
net.Layers(1)
```

The name of the input layer of GoogleNet is 'data'. The size of the input is 224×224 pixels, and it considers images with three channels for example – RGB image. The size of the input layer is defined using the following code:

```
Input_Size = net.Layers(1).InputSize;
```

The line above will get the input size of the input layer and store it the 'Input_Size' variable. This variable layer will be used. So, the pre-trained network is loaded. The next step is to replace the final layer.

5.1.2 Replacing the Layers

It is needed to replace the layers of the GoogleNet which are trained to classify 1000 objects with our own layers. However, before that one needs to find out which layers are the task specific layers. The following code can be used to find information about the network:

```
Layer__Graph = layerGraph(net);
```

The 'layerGraph()' function considers the network as an argument, inspects the architecture and returns general information like number of layers, input layer, output layer and number of connections. However, a more effective way to find out the layer one needs to replace is to use network analyzer.

The network can be examined using network analyzer to find out which network layer is responsible for task specific feature learning and which layer is meant for classifying the objects.

From the network analyzer, it can be seen that the layer no. 142 is the layer trained with task specific features. The name of this layer is known as 'loss3-classifier'. Layer no. 144 is the classification layer. This layer is meant for classifying objects. These two layers must be replaced with new layers so that they can be trained to classify objects. One can get access to the layers 142 and 144 using the following two lines:

```
Feature__Learner = net.Layers(142);
```

```
Output__Classifier = net.Layers(144);
```

Now the layers to be replaced are stored in 'Feature__Learner' and 'Output__Classifier' variable. Our new training dataset has 4 categories of data. This means that we are trying to classify images into 5 categories. Thus, the classifier layer will have 4 classes and the feature learner layer will have 5 fully connected layers.

The 'numel()' function can be used to find out the number of classes present in our dataset. The 'numel' is the short form for 'number of elements' and it counts the number of elements. The number of classes in the training dataset can be found using the following code:

```
Number__of__Classes = numel(categories(Training_Data.Labels));
```

The 'numel()' function has an argument called 'categories(Training_Data.Labels)'. The 'Training_Data.Labels' refers to the labels contained in the training dataset. The 'categories()' function returns the number of data categories. The 'numel()' function returns the number of categories present in the training data.

Now, our own layers can be defined. First of all, the fully connected layer will be defined, which is stored in 'Feature__Learner' variable. The following block of code is used to improve a fully connected layer:

```
New__Feature__Learner = fullyConnectedLayer(Number__of__Classes,...  
  
'Name','Facial Feature Learner', ...  
  
'WeightLearnRateFactor',10, ...
```

```
'BiasLearnRateFactor',10);
```

The 'fullyConnectedLayer()' function is used to define a fully connected layer. The number of classes is the first argument of this function. The number of classes is stored in 'Number__of__Classes' variable. Then using three dots (...), one can jump to the next line. Here, the name of the layer is defined as 'Facial_Feature_Learner'. Then, in the next line, the weighted learning rate is assigned to 10. After that, in the next line, the bias is set to 10. Both the learning rate and the bias values are weighted values, not the actual values. Now, the classification layer needs to be defined. To define the classification layer, a function named 'classificationLayer()' is used. Our new classification layer can be defined using the following code:

```
New__Classifier__Layer = classificationLayer('Name', 'Face Classifier');
```

'Face Classifier' is the name of our new classification layer.

So, the layers are defined. Now, the existing layers can be replaced by our redefined layers. To replace the layers, one uses 'replaceLayer()' function. The 'replaceLayer()' function takes the layer-graph, new layer, and name of the existing layer as arguments. The following code shows the use of this function:

```
Layer__Graph = replaceLayer(Layer__Graph, Feature__Learner.Name,  
New__Feature__Learner);
```

'Layer_Graph' is the first argument of the function above. The entire graph of the network is stored in 'Layer_Graph' variable. This is why the 'Layer_Graph' is used as the first argument of the function. 'Feature__Learner.Name' is the second argument.

The existing feature learner layer is stored in 'Feature_Learner' variable. 'Name' is one of the objects of the 'Feature_Learner'. When the 'Feature_Learner.Name' is passed as an argument to the 'replaceLayer()' function, the name of the layer to be replaced is specified. The last argument is the newly defined layer that is stored in 'New__Feature__Learner' variable.

The 'replaceLayer()' function is used to replace the layer and return the layer graph information about the new layer. This new information is stored in the existing 'Layer_Graph' variable. Thus, the feature learner layer has been replaced. The next step is to replace the output classification layer. This is similar to the procedure of replacing the feature learner layer. The following code is used to replace the output layer:

```
Layer__Graph = replaceLayer(Layer__Graph, Output__Classifier.Name,  
New__Classification__Layer);
```

The output layers have thus been replaced with our self-defined layers. Now, one can start training these layers. However, before starting the training process, it is better to freeze few of the initial layers.

5.1.3 Image Augmentation to Prevent Overfitting

The input layer size of GoogleNet is 224x224x3. This means that the network accepts images with three channels (such as RGB or HSV) with 224x224 pixels. However, the dataset might contain images of different sizes. Thus, it is necessary to resize the image to an appropriate size for the network to accept the training images as input image.

In addition, it is better to randomly flip the images or stretch or shrink the images during the training process in order to prevent the network from overfitting. In MATLAB, the 'imageDataAugmenter()' is used to modify the images before training process. The following code shows the use of 'imageDataAugmenter()' function:

1. Pixel__Range = [-30 30];
2. Scale__Range = [0.9 1.1];
3. Image__Augmenter = imageDataAugmenter(...
 - . 'RandXReflection',true, ...
 - A. 'RandXTranslation',Pixel_Range, ...
 - B. 'RandYTranslation',Pixel_Range, ...
 - C. 'RandXScale',Scale_Range, ...
 - D. 'RandYScale',Scale_Range);

Here, on the first line, the pixel range is defined. The images are randomly translated up to 30 pixels. Then on the second line, the scale range is defined. The images are scaled up to 10%. On the third line, the 'imageDataAugmenter()' is used to modify the images. The arguments of this function come in the form of {'Name', Value} pair. This means that the first part of the argument is the name and the second part which is separated by comma is the value. For easier understanding, the arguments are specified in new lines.

'RandXReflection' is the first argument and the value is 'true'. When the 'RandXReflection' becomes true, each image is reflected horizontally with probability of 50%. 'RandXTranslation' is the next argument and 'Scale_Range' is the value of this

argument, which is set to [-30 30]. The pixels are randomly moved horizontally within the given range using this argument. 'RandYTranslation' is the next argument.

'Scale_Range' is the value of this argument, which is [-30 30]. The pixels of an image are randomly shifted vertically within the defined range using this argument. On the 3(A), 'RandXScale' is used and the value is set to 'Scale_Range'. [0.9 1.1] is the 'Scale_Range'.

On the next line, a similar thing is done. However, this time 'RandYScale' is used to stretch or shrink the image along vertical axis. The reason of setting the range from 0.9 to 1.1 is to ensure that it allows both stretching and shrinking. If the value is in between 0.9 to 1.0, the image will shrink; while on the other hand, when the value is in between 1.0 to 1.1, the image is stretched. One can apply the augmentation using the following code:

```
Augmented__Training__Image =
```

```
augmentedImageDatastore(Input_Size(1:2),Training_Dataset, ...
```

```
'DataAugmentation',Image_Augmenter);
```

Any type of augmentation can be applied using the 'augmentedImageDatastore()' function. In the code above, the first argument is 'Input_Size(1:2)', which is the required resolution of the GoogleNet input layer, i.e., 224 x 224 pixels. The

'augmentedImageDatastore()' function will resize every image to the size 224x224 pixels. The next argument is the image where it is needed to apply the augmentation. 'Training_Dataset' is the name of our dataset. Apart from resizing images, it is required to modify images as well. The behavior of the image modifier is defined and stored in 'Image_Augmenter' variable. 'DataAugmentation' is the last argument of 'augmentedImageDatastore()' and the value of this argument is 'Image_Augmenter' that holds the definition of our image modification. The same augmentation process can be applied to the validation dataset using the following code:

```
Augmented_Validation_Image =  
augmentedImageDatastore(Input_Size(1:2),Validation_Dataset);
```

However, the images have already been resized to the appropriate size. Thus, the image modification is not mandatory for validation process. Since the training and validation data are now ready, the training process can be initiated.

5.1.4 Training the Face Recognition CNN

To train a network in MATLAB, the 'trainNetwork()' function is used. The 'trainNetwork()' function uses three arguments. These arguments are:

1. Training Data
2. The Architecture of the Network
3. Training Options

Training Data

The training data is already prepared and stored in a variable named 'Augmented__Training__Image'.

The Architecture of the Network

The entire network is not dealt with. Only the final layer and task specific feature learner layer is trained. In order to train these two layers, it is required to know the architecture of these two layers. The architecture is stored in the 'Layer__Graph' variable.

Training Options

There are many parameters related to the training process of a multilayer neural network. These parameters are known as training options. 'trainingOptions()' is the function used in MATLAB to specify the training options. The following block of code shows the use of 'trainingOptions()' function in the implementation:

1. Size__of__Minibatch = 10;
2. Validation__Frequency=
floor(numel(Augmented_Training_Image.Files)/Size_of_Minibatch);
3. Training__Options = trainingOptions('sgdm', ...
 - . 'MiniBatchSize',Size__of__Minibatch, ...
 - A. 'MaxEpochs',6, ...
 - B. 'InitialLearnRate',3e-4, ...
 - C. 'Shuffle','every-epoch', ...
 - D. 'ValidationData',Augmented_Validation_Image, ...
 - E. 'ValidationFrequency',Validation_Frequency, ...
 - F. 'Verbose',false, ...
 - G. 'Plots','training-progress');

The mini-batch method is used for training. This is why, the number of elements in per batch is defined at the first line. On the second line, the validation frequency is defined. Here, the total number of training image file is divided by the size of the mini-batch. Then, the 'numel()' function is used to find the number of elements. Finally, the 'floor()' function is used to convert fractions to integer. For instance, if we get floor(0.6), it will return 0, if we get floor(2.3), it will return 2 and so on. Finally, the value is stored in a variable named 'Validation__Frequency'.

On the third line, the training options is specified. To make it easier to understand, each of the arguments of the function and their corresponding values are presented in new line. The following list explains the role of the arguments used in the 'trainingOptions()' function.

sgdm: The term 'sgdm' refers to 'Stochastic Gradient Descent with Momentum'. This is the first argument of the 'trainingOptions()' function and it defines the algorithm used to update the weights

MiniBatchSize: With this argument, the size of the mini-batch is specified. The size of the mini-batch is stored in 'Size__of__Minibatch' variable. This is why, the 'Size__of__Minibatch' variable is used as the value of the 'MiniBatchSize' argument.

MaxEpochs: The maximum number of epochs is set as 6.

InitialLearningRate: This specifies the initial learning rate required for the training process. Here, 0.0003 is the initial learning rate. In MATLAB, 0.0003 is written as 3e-4.

Shuffle: This argument controls the shuffling of the training data and validation data.

The value of this argument is assigned to 'every-epoch'. It means that in every epoch, the training and validation data will be shuffled. This helps in reducing overfitting.

ValidationData: The validation data is specified using this argument. Our validation data is stored in 'Augmented_Validation_Image' variable. Thus, the value of this argument is 'Augmented_Validation_Image'

ValidationFrequency: The frequency of validation is specified using this argument. This is meant to indicate that after how many iterations, the trained network will be validated using the validation dataset. The frequency is already calculated and stored in a variable named 'Validation__Frequency'.

Verbose: The verbose shows the training progress information in the command window. It is not required to show the training related information during the training. So, the value of this argument is set to 'false'.

Plots: This argument is used to plot the training progress. The value of this property is set to 'training-progress' in order to see the training progress.

Once the training options are defined, three arguments are passed to the 'trainNetwork()' function as follows:

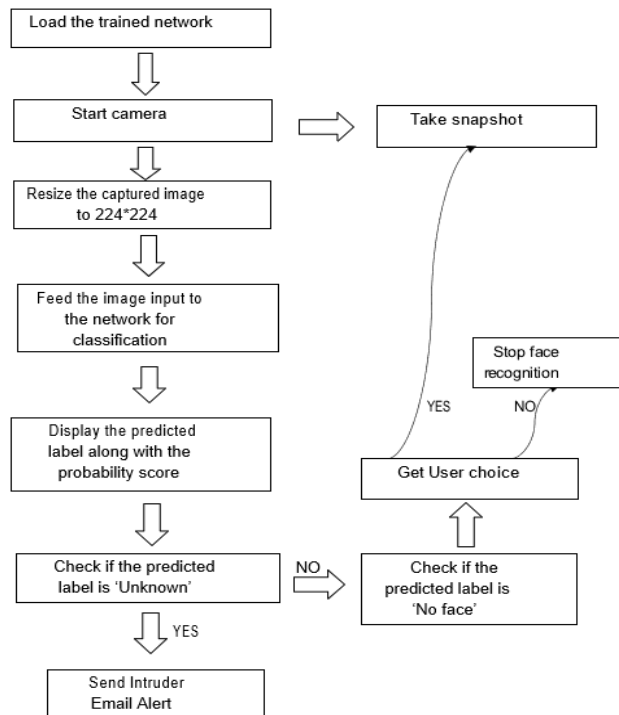
```
net = trainNetwork(Augmented_Training_Image, Layer__Graph, Training__Options);
```

Then, the 'trainNetwork()' function will train the newly added layers with the provided dataset. The 'net' variable will store the newly trained network. After the network is trained, the final step is to classify the validation images in order to check the performance of the network.

5.1.5 Testing the Face Recognition Network

Figure 24

Application Workflow



First, the trained network is loaded. Connection to the camera is established. The captured image is resized to 224*224. This is because; the first layer, which is the image input layer, requires input images of size 224-by-224-by-3, where 3 represents the number of color channels.

It is then fed as input to the network for face recognition. The network classifies amongst known faces and predicts the name of the person as label, along with the probability score. This score determines the probability of the prediction. It assigns the label as 'Unknown' if the face is not recognized.

The network has been trained to identify an unknown face. This is done by placing 10 face images in a folder called 'Unknown' in the prepared dataset. If the predicted label is unknown, an intruder email alert is sent from MATLAB.

Logic for unrecognizable face

```

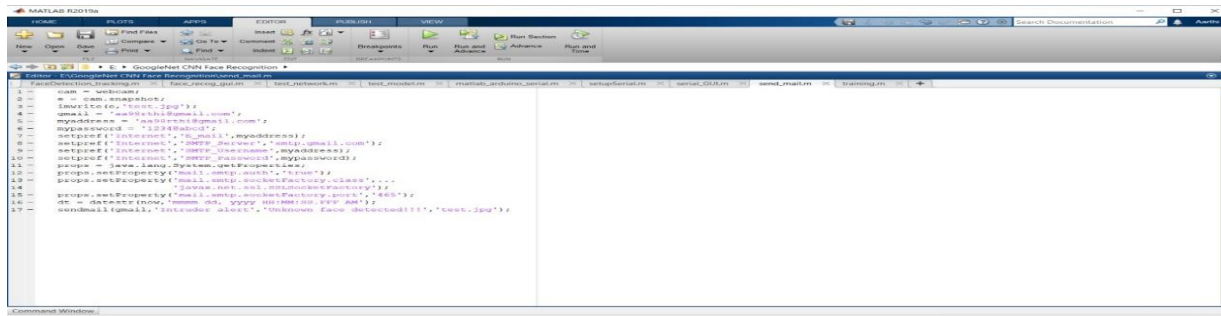
137 -         if isequal(char(label), 'Unknown')
138 -             opts.Interpreter = 'tex';
139 -             opts.Default = 'Yes';
140 -             quest = 'Face not recognized! Do you know this person?';
141 -             selection = questdlg(quest, 'Alert', 'Yes', 'No', opts);
142 -             if(isequal(selection, 'No'))
143 -                 handles.text1.String = 'Intruder!';
144 -                 drawnow;
145 -                 fprintf(s,0);
146 -                 pause on;
147 -                 break;
148 -             end
149 -         end

```

Intruder Email Alert

Figure 25

Email setup



```

1  email = webmail;
2  m = mail('example@');
3  emailSet('smtp','smtp.gmail.com');
4  emailSet('smtp','smtp.gmail.com');
5  myaddress = 'example@gmail.com';
6  myaddress = 'example@gmail.com';
7  myaddress = 'example@gmail.com';
8  myaddress = 'example@gmail.com';
9  myaddress = 'example@gmail.com';
10 myaddress = 'example@gmail.com';
11 myaddress = 'example@gmail.com';
12 myaddress = 'example@gmail.com';
13 myaddress = 'example@gmail.com';
14 myaddress = 'example@gmail.com';
15 myaddress = 'example@gmail.com';
16 myaddress = 'example@gmail.com';
17 myaddress = 'example@gmail.com';

```

The `setpref` function has two mail-related preferences:

Email address: This preference sets the email address appearing on the message.

SMTP server: This preference sets the outgoing SMTP server address, which can almost be any email server that either supports the Post Office Protocol (POP) or the Internet Message Access Protocol (IMAP).

Once email is configured properly in MATLAB, the `sendmail` function can be used. At least two arguments are required by the `sendmail` function: the recipient's email address and the email subject.

Files can also be attached to the email. Here, the image of the unknown person is sent as an attachment in the email.

Design of GUI using GUIDE

GUIDE stands for Graphical User Interface Development Environment. It provides with the tools to design user interfaces and create custom apps. The GUI can be accessed by typing `guide` in the workspace.

The components can be selected from the left pane and added to the workspace. Their size can be changed by dragging the edges. Changing the position can be done by double-clicking and dragging it. Three components have been used to design the GUI:

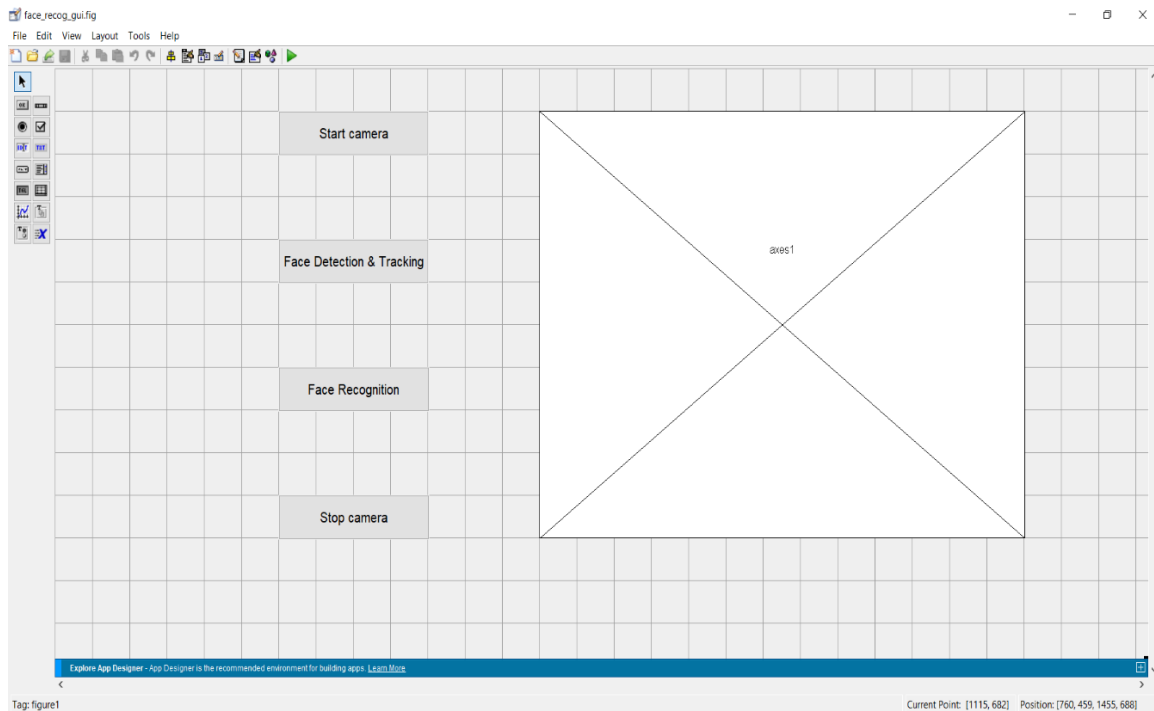
Push Button: Its function is to call the callback function for the execution of different programs

Axes: They are used to add images, charts, and plots to the GUI. They have no callback function

Static Text: It is used to add labels that remain unchanged in the GUI and has no callback

Figure 26

GUI Design



Logic for Real-time Face Detection and Tracking

The following steps describe the procedure for real-time face detection and tracking.

1. Create a 'cam' variable and initiate a 'webcam()' object
2. After that, take a variable named 'video_Frame,' and we will use the 'snapshot()' function to read the frames one by one from the 'cam' object
3. Now it is time to initiate the video player object. For that, create a variable named

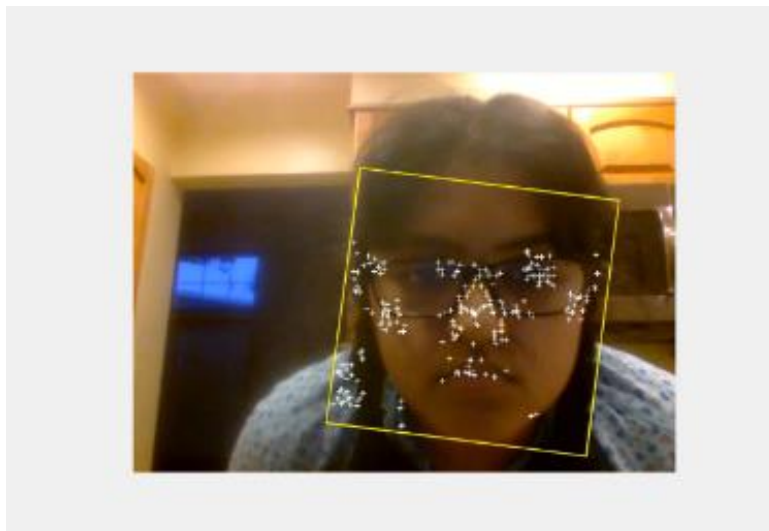
'video_Player,' and assign the 'video player' object to it. The first argument of this object is the 'Position.' In the second argument, we define the position in a set of square brackets. The first two values are for the left and bottom corners. The second two values are the width and height of the video, respectively.

4. Create another variable named 'face-Detector' and assign the 'cascade object detection()' object. We use this object to detect the face.
5. To track the face, a point tracker object is needed. Create a variable named 'point_Tracker' and put the 'point tracker object' in it.
6. Now, I am going to initiate three variables for the while loop. The first one is 'run_loop' equals true. The second one is 'number of points' equals 0, and the third is 'frame_Count' equals 0.
7. Now, declare a while loop. It will keep looping as long as run_loop is true and frame_Count is less than 400. If you want to run the webcam for longer, you can increase the number of frames. Then end this loop. Inside this loop, we are going to do the detection and tracking.
8. First, inside the 'video_Frame' variable, we will store the frames from the 'cam' object using the 'snapshot()' function. Then using the 'rgb2gray()' function, we convert the frames into grayscale and store them in a variable named 'gray_Frame.'
9. Then write, 'frame_Counter = frame_Counter+1' to increase the 'frame_Counter' by 1.
10. Initiate an if condition when the 'number of points is less than 10.
11. At the beginning of this if condition, we locate the rectangle that encloses the face on 'gray_Frame' using step function and faces 'face_Detector' object.

12. Now, if the 'face_Rectangle' is not empty, then using the 'detect Min Eigen Features' function, we will find the rectangle points. The first argument of this function is the frame where the image is located. In our case, it is the 'gray_Frame.' Then, we need to tell the function whether we are interested in getting the feature of the entire image or a particular region. We are interested in a particular region; we specify the region of interest or 'ROI' here. The ROI is the location where the 'face_Rectangle' is located. We are using this 'one comma colon' to get the Value of the first row of the 'face rectangle' matrix, which is the starting location of this rectangle.
13. Next, create a variable named 'xy_Points' and write 'points. Location' to convert the points to 'x y values.' Now the x values of the 'xy_Points' variable have the number of points we need to initialize the point tracker.

Figure 27

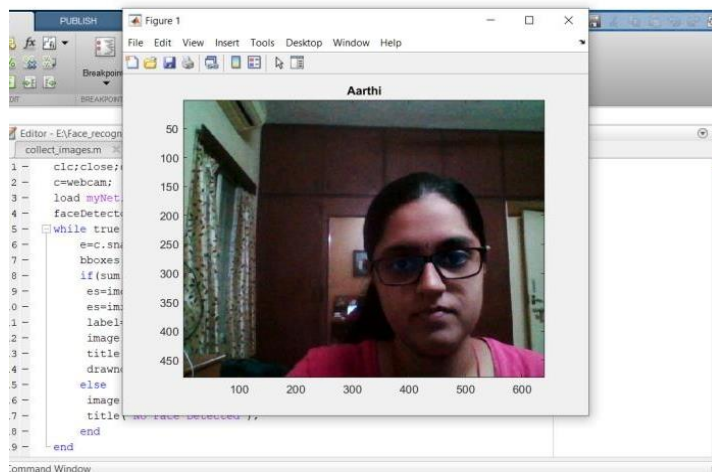
Real-time Face Detection and Tracking



5.1.6 Results

Figure 28A

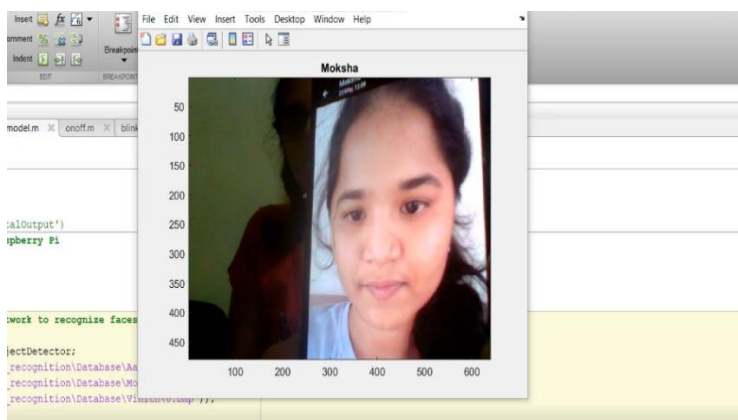
When a Known Person Comes in Front of the Webcam



Note. Person 1: Predicted label -> name of the person.

Figure 28B

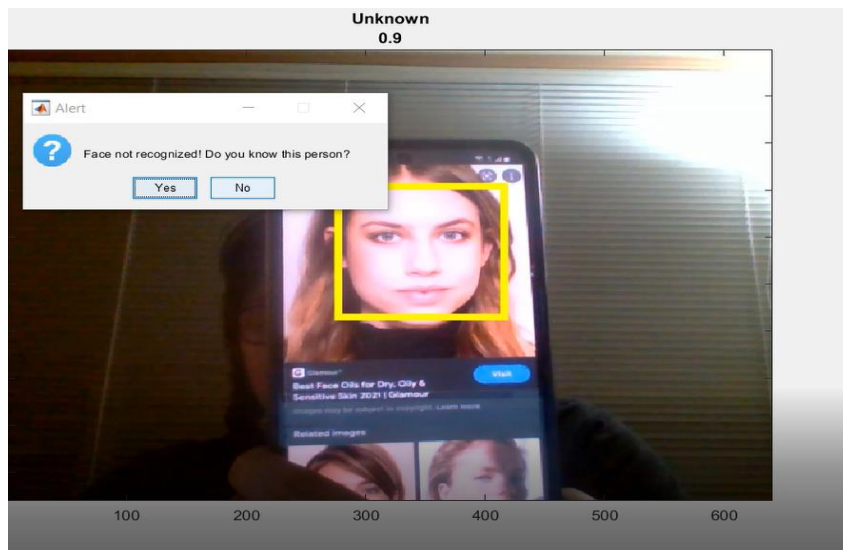
When another known person comes in front of the webcam



Note. Person 2: Predicted label -> name of the person.

Figure 29

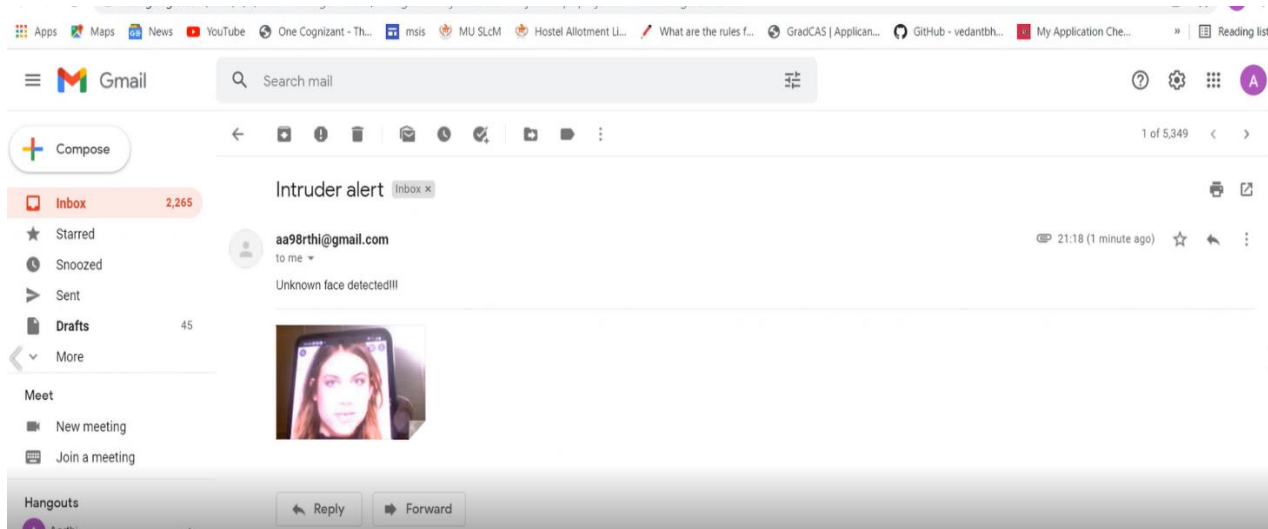
Unknown Face Detected



Note. When an unknown person comes in front of the webcam

Figure 30

Intruder Email Alert with Image of Unknown Person Attached



5.2 Implementation Using FaceNet

This face recognition system is implemented on a pre-trained FaceNet model, thus achieving a state-of-the-art accuracy. The system comes with both Live recognition & Image recognition. It is trained on faces of some celebrities.

Installing dependencies:

1. For Anaconda users: `conda install --file requirements.txt`
2. For python users: `pip install -r requirements.txt`

(even Anaconda users can use this if they use anaconda prompt instead of terminal)

The requirements.txt file is as follows:

```
numpy==1.16.4
Pillow==9.0.1
matplotlib==3.1.0
opencv-contrib-python==4.2.0.32
scipy==1.2.1
scikit-learn==0.21.2
tensorflow==1.14.0
Keras==2.2.4
```

5.2.1 Loading a FaceNet Model in Keras

There are a number of projects which provide tools to train FaceNet-based models and use pre-trained models.

The most prominent one is called OpenFace that provides FaceNet models, which are built and trained using the PyTorch deep learning framework.

Another prominent project called FaceNet by David Sandberg provides FaceNet models built and trained by using TensorFlow.

Keras FaceNet by Hiroki Tanai is a notable example. His project provides a script to convert the Inception ResNet v1 model from TensorFlow to Keras. A pre-trained Keras model ready for use is also provided.

Here, the pre-trained Keras FaceNet model provided by Hiroki Tani ai is used. It was trained on MS-Celeb-1M dataset, which expects input images to be color and to have their pixel values whitened, i.e., uniform across all three channels, and to have a square shape of 160×160 pixels.

The Keras FaceNet Pre-Trained model file is downloaded and placed in our current working directory with the filename `'facenet_keras.h5'`. One can load the model directly in Keras by using the `load_model()` function; for instance:

```
# An example to load the keras facenet model
```

```
from keras.models import load_model
```

```
# loading the model
```

```
model1 = load_model('facenet_keras.h5')
```

```
# sum up the input and output shape
```

```
print(model1.inputs)
```

```
print(model1.outputs)
```

Executing the above example loads the model and prints the shape of the input and output tensors. It can be seen that the model expects square color images as input with the shape 160×160, and outputs a face embedding as a 128-element vector.

```
# [<tf.Tensor 'input_1:0' shape=(?, 160, 160, 3) dtype=float32>]
```

```
# [<tf.Tensor 'Bottleneck_BatchNorm/cond/Merge:0' shape=(?, 128) dtype=float32>]
```

Now that we have a FaceNet model, we can explore the use of it.

5.2.2 Detecting Faces for Face Recognition

Before performing face recognition, we need to detect faces. The process of automatically locating faces in a photograph and localizing them by drawing a bounding box around their extent is known as Face detection.

The `CascadeClassifier` class is provided by OpenCV. It can be used to create a cascade classifier for face detection. The constructor can take a filename as an argument, which specifies the XML file for a pre-trained model. A pre-trained model for frontal face detection is downloaded from the OpenCV GitHub project and placed in our current working directory with the filename `'haarcascade_frontalface_default.xml'`. Once downloaded, one can load the model as follows:

```
# load the pre-trained model
```

```
classifier1 = CascadeClassifier('haarcascade_frontalface_default.xml')
```

Once loaded, face detection can be performed on a photograph using the model by calling the `detectMultiScale()` function. This function returns a list of bounding boxes for all the faces detected in the photograph.

```
# perform face detection
```

```
bboxes = classifier1.detectMultiScale(pixels)
```

```
# print bounding box for each detected face
```

```
for box in bboxes:
```

```
    print(box)
```

The `detectMultiScale()` function provides some arguments that help tune the usage of the classifier. Two parameters to be noted are `scaleFactor` and `minNeighbors`; for instance:


```
# perform face detection
```

```
bboxes = classifier.detectMultiScale(pixels, 1.1, 3)
```

The *scaleFactor* controls the scaling of input image prior to detection, e.g., is it scaled up or down, that can help to better find the faces in the image. The default value is 1.1 (10% increase), although this can be decreased to values such as 1.05 (5% increase) or increased to values such as 1.4 (40% increase).

The *minNeighbors* determines how robust each detection must be, so that it can be reported, e.g., the number of candidate rectangles that located the face. The default is 3, but this can be reduced to 1 to detect a lot more faces and will probably increase the false positives, or increase to 6 or more to need a lot more confidence before a face is detected.

The *scaleFactor* and *minNeighbors* often need tuning for a given image or dataset to best detect the faces. It may be useful to perform a sensitivity analysis across a grid of values and see what works well or best in general on one or many photographs. A fast strategy may be to reduce (or increase for small photos) the *scaleFactor* until all faces are detected, then increase the *minNeighbors* until all false positives vanish, or close to it.

5.2.3 Create Face Embeddings

The next task is to create a face embedding. A face embedding refers to a vector that represents the features obtained from the face. This can then be compared with the vectors produced for other faces. For example, a vector that is near, by some measure, may be the same person, whereas another vector that is far (by some measure) may be a different person.

The classifier model that we want to build takes a face embedding as input and predicts the identity of the face. The FaceNet model generates this embedding for a given image of a face.

The FaceNet model can be used as part of the classifier itself, or can be used to pre-process a face to create a face embedding which can be stored and made use of as input to our classifier model. This latter approach is preferred since the FaceNet model is both large and slow for creating a face embedding.

We can, thus, pre-compute the face embeddings for all faces of the train and test (formally '*validation*') sets in our 5 Celebrity Faces Dataset.

First, our detected faces dataset can be loaded using the `load()` NumPy function

```
# load the face dataset
```

```
data1 = load('5-celebrity-faces-dataset.npz')
trainX1, trainy1, testX1, testy1 = data1['arr_0'], data1['arr_1'], data1['arr_2'],
data1['arr_3']

print('Loaded the dataset: ', trainX1.shape, trainy1.shape, testX1.shape,
testy1.shape)
```

Next, the FaceNet model ready for converting faces into face embeddings can be loaded.

```
# loading the facenet model
```

```
model1 = load_model('facenet_keras.h5')
print('Loaded Model')
```

Then, each face can be enumerated in the train and test datasets to predict an embedding.

To predict an embedding, the pixel values of the image first need to be suitably prepared to meet the requirements of the FaceNet model. This specific implementation of the FaceNet model expects the pixel values to be standardized.

```
# scaling of pixel values
```

```
face__pixels = face__pixels.astype('float32')
```

```
# standardizing pixel values across channels (global)
```

```
mean, std = face__pixels.mean(), face__pixels.std()
```

```
face__pixels = (face__pixels - mean) / std
```

To make a prediction for one example in Keras, one must expand the dimensions such that the face array is one sample.

```
# transforming face into one sample
```

```
samples1 = expand_dims(face__pixels, axis=0)
```

One can then use the model to make a prediction and obtain the embedding vector.

```
# making prediction to get embedding
```

```
yhat1 = model.predict(samples1)
```

```
# get embedding
```

```
embedding1 = yhat1[0]
```

The `get_embedding()` function defined below applies these behaviors and returns a face embedding, given a single image of a face and the loaded FaceNet model.

```
# obtain the face embedding for one face
```

```
def get_embedding(model, face__pixels):
```

```
    # scaling of pixel values
```

```
    face__pixels = face__pixels.astype('float32')
```

```
# standardizing pixel values across channels (global)
```

```
mean, std = face__pixels.mean(), face__pixels.std()
```

```
face__pixels = (face__pixels - mean) / std
```

```
# transforming face into one sample
```

```
samples1 = expand_dims(face__pixels, axis=0)
```

```
# making prediction to get embedding
```

```
yhat1 = model.predict(samples1)
```

```
return yhat1[0]
```

5.2.4 Perform Face Classification

Next, a model is developed to classify face embeddings as one of the known celebrities in the 5 Celebrity Faces Dataset. First, the face embeddings dataset must be loaded.

```
# loading dataset
```

```
data1 = load('5-celebrity-faces-embeddings.npz')
```

```
trainX1, trainy1, testX1, testy1 = data1['arr_0'], data1['arr_1'], data1['arr_2'],
```

```
data1['arr_3']
```

```
print('The Dataset: train=%d, test=%d' % (trainX1.shape[0], testX1.shape[0]))
```

Next, the data needs some minor preparation before modeling. Normalizing the face embedding vectors is a good practice because the vectors are often compared to each other by using a distance metric.

Here, vector normalization refers to scaling of the values until the length or magnitude of the vectors is 1 or unit length. This can be attained by using the Normalizer class in scikit-learn. It might be even more convenient to perform this step while the face embeddings are created in the previous step.

```
# normalizing input vectors
```

```
in_encoder1 = Normalizer(norm='l2')
```

```
trainX1 = in_encoder1.transform(trainX1)
```

```
testX1 = in_encoder1.transform(testX1)
```

Next, the string target variables need to be converted to integers for each celebrity name. This can be attained through the `LabelEncoder` class in `scikit-learn`.

```
# labelling encode targets
```

```
out_encoder1 = LabelEncoder()
```

```
out_encoder1.fit(trainy1)
```

```
trainy1 = out_encoder1.transform(trainy1)
```

```
testy1 = out_encoder1.transform(testy1)
```

Next, one can fit a model.

Usually, a Linear Support Vector Machine (SVM) is used while working with normalized face embedding inputs. This is because the method is very efficient at separating the face embedding vectors. One can fit a linear SVM to the training data by using the `SVC` class in `scikit-learn` and setting the `'kernel'` attribute to `'linear'`. We may also want probabilities later while making predictions, that can be configured by setting `'probability'` to `'True'`.

```
# fitting model
```

```
model1 = SVC(kernel='linear')
```

```
model1.fit(trainX1, trainy1)
```

Next, the model can be evaluated. This can be achieved using the fit model to make a prediction for each example of the train and test datasets and then calculate the classification accuracy.

```
# prediction
```

```
yhat_train1 = model1.predict(trainX1)
```

```
yhat_test1 = model1.predict(testX1)
```

```
# score
```

```
score_train1 = accuracy_score(trainy1, yhat_train1)
```

```
score_test1 = accuracy_score(testy1, yhat_test1)
```

```
# sum up
```

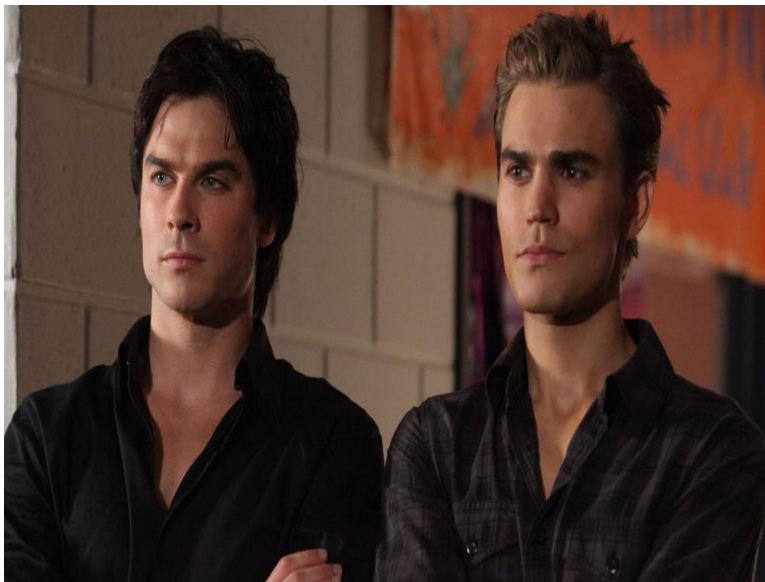
```
print('Accuracy is: train=%.3f, test=%.3f' % (score_train*100, score_test*100))
```

Note: Faces with Unidentified labels are faces on which the model is not trained.

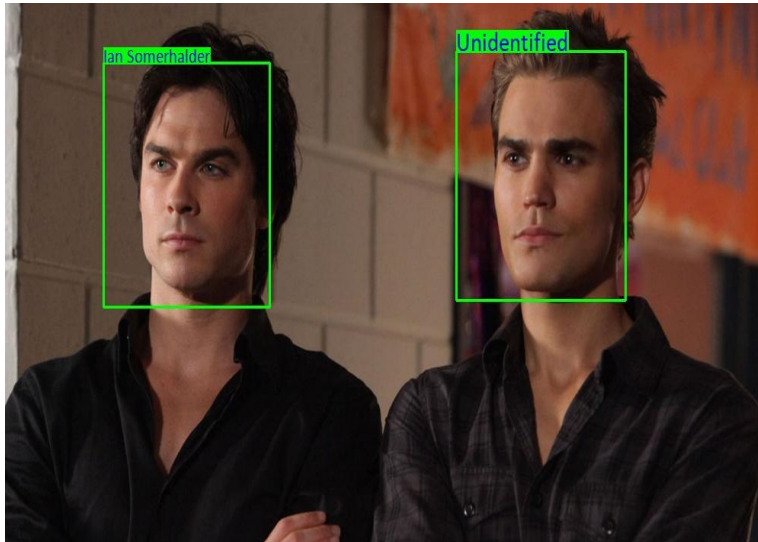
5.2.5 Results

Figure 31A

Input Image 1



Note. This figure shows an example image before facial recognition. (Goswami, 2022).

Figure 31B*Output Image 1*

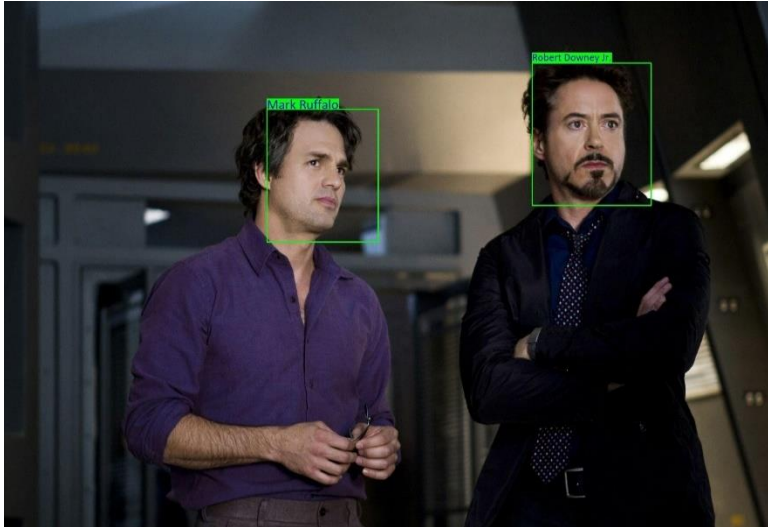
Note. This figure shows the image after facial recognition for the example.

(Goswami, 2022).

Figure 32A*Input Image 2*

Note. This figure shows a second example image before facial recognition.

(Goswami, 2022).

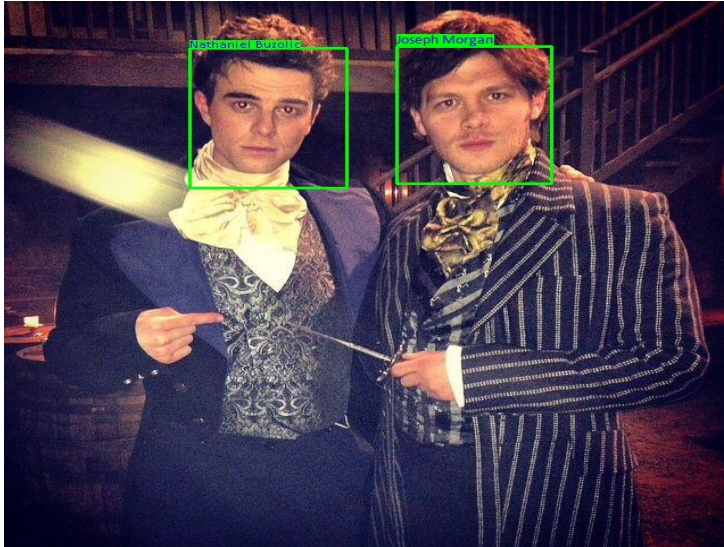
Figure 32B*Output Image 2*

Note. This figure shows the image after facial recognition for the second example.

(Goswami, 2022).

Figure 33A*Input Image 3*

Note. This figure shows a third example image before facial recognition. (Goswami, 2022).

Figure 33B*Output Image 3*

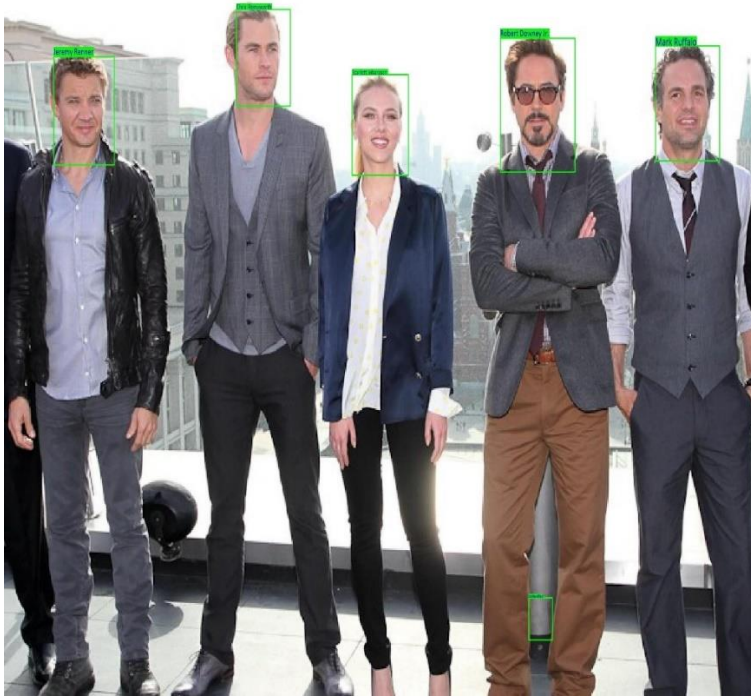
Note. This figure shows the image after facial recognition for the third example.

(Goswami, 2022).

Figure 34A*Input Image 4*

Note. This figure shows a fourth example image before facial recognition. (Goswami,

2022).

Figure 34B*Output Image 4*

Note. This figure shows the image after facial recognition for the fourth example.

(Goswami, 2022).

Chapter 6: Conclusion

Facial Recognition was performed using two pretrained models, GoogLeNet Convolutional Neural Network and Google's deep convolutional network – FaceNet. Transfer Learning was used to transfer the knowledge from existing network and is then modified according to the task (face recognition) to be accomplished.

The system was tested with faces inside and outside the dataset. When implemented upon pre-trained GoogLeNet Convolutional Neural Network, the accuracy of prediction was good, with probability score of 0.9 – 1 when image of database was shown and 0.75 – 0.86 when a different image was shown. At times, the network got confused, and gave out a wrong label. This could possibly be due to lack of clarity in the dataset images or while capturing. This was verifiable with the corresponding low probability score.

When implemented upon a pre-trained FaceNet, multiple faces could be recognized in a photograph. It was tested on different images of the same person, and the accuracy of prediction came out to be much good when compared to GoogLeNet Convolutional Neural Network. Yet another notable advantage was that, training the network required only one sample image for each person.

Through this research project, I learnt how to design a face recognition security system. It helped me get a good understanding of the working of a Convolutional Neural Network. It was interesting to learn the concept of Transfer Learning. I can improve the performance by analyzing errors (bad predictions) in the validation dataset or by inculcating Feature Engineering – a step to extract more information (features) from existing data.

References

- Abdullah, M. (2021). *Notes on "FaceNet: A Unified Embedding for Face Recognition and Clustering"*. [https://hackmd.io/@ABD/SJa0J7_Od#Notes-by-Muhammed Abdullah](https://hackmd.io/@ABD/SJa0J7_Od#Notes-by-Muhammed-Abdullah)
- Dulčić, L. (2020). *Face Recognition with FaceNet and MTCNN*. Arsfutura. <https://arsfutura.com/magazine/face-recognition-with-facenet-and-mtcnn/>
- Chae, Y.N., Chung, J.N., & Yang, H.S. (2008). Color filtering-based efficient face detection. *2008 19th International Conference on Pattern Recognition*, 1-4.
- GeeksforGeeks. (2021). *Understanding GoogLeNet Model – CNN Architecture*. <https://www.geeksforgeeks.org/understanding-googlenet-model-cnn-architecture/>
- GeeksforGeeks. (2022). *FaceNet – Using Facial Recognition System*. <https://www.geeksforgeeks.org/facenet-using-facial-recognition-system/>
- Goswami, A. (2022). *Face-Recognition-using-FaceNet*. Github. <https://github.com/TheAnkurGoswami/Face-Recognition-using-FaceNet>
- Kana, M. (2020, May 18). *How AI Can See Better Than Your Eyes Do*. Medium. <https://medium.com/dataseries/how-ai-can-see-better-than-your-eyes-do-93e5a5da1e8a>
- MathWorks. (2022). *Transfer Learning Using Pretrained Network*. <https://www.mathworks.com/help/deeplearning/ug/transfer-learning-using-pretrained-network.html>
- Nigam, V. (2018, September 10). *Understanding Neural Networks. From neuron to RNN, CNN, and Deep Learning*. Medium. <https://medium.com/analytics->

vidhya/understanding-neural-networks-from-neuron-to-rnn-cnn-and-deep-learning-cd88e90e0a90

Ray, S., Alshouli, K., & Agrawal, D.P. (2021). Dimensionality Reduction for Human Activity Recognition Using Google Colab. *Information* 2021, 12, 6.

<https://doi.org/10.3390/info12010006>

Shah, A.A., Zaidi, Z.A., Chowdhry, B.S., & Daudpoto, J. (2016). Real time face Detection/Monitor using raspberry pi and MATLAB. *2016 IEEE 10th International Conference on Application of Information and Communication Technologies (AICT)*, 1-4.

Singh, S.K., Chauhan, D.S., Vatsa, M. & Singh, R. (2003). A robust skin color based face detection algorithm. *Tamkang Journal of Science and Engineering*, 6(4), 227-234.

Yalcin, O.G. (2020, September 23). *4 Pre-Trained CNN Models to Use for Computer Vision with Transfer Learning*. Towards Data Science.

<https://towardsdatascience.com/4-pre-trained-cnn-models-to-use-for-computer-vision-with-transfer-learning-885cb1b2dfc>

Zarit, B.D., Super, B.J., & Quek, F.K.H. (1999). Comparison of five color models in skin pixel classification. *Proceedings International Workshop on Recognition, Analysis, and Tracking of Faces and Gestures in Real-Time Systems*, 58-63.