Machine Learning Course Project Report

Indian Institute of Space Science and Technology

Parthasarathi Samanta SC17B106
Kiran, L. SC17B150

2 May 2020

Abstract

Human face is one of the easiest ways to distinguish and identity individuals. Face recognition is an identification system that identifies a person with his/her physical traits. Human face recognition procedure consists of two steps - face detection and recognition. Face detection and recognition is a much researched topic among community dealing with the branch of Pattern Recognition and Machine Learning. Auto-tagging feature of Facebook is a popular example of application of Machine learning algorithms for facial identification. There are two kinds of methods that are currently popular in developed face recognition namely, Eigenface method and Fisherface method. Facial image recognition through Eigenface method is based on the reduction of face-space dimension using Principal Component Analysis (PCA) for facial features. The main purpose of the use of PCA on face recognition using Eigen faces was formed (face space) by finding the eigenvectors corresponding to the largest eigenvalue of the face image. The software requirements for this project is Jupyter Notebook.

Keywords: Face detection,Face recognition, Eigenface, Image Procesing, Principal Component Analysis, Python, Open-CV, PyTorch, kNN, SVM, Neural Network, Ensemble Classifier.

Extension: There are vast number of applications from this face detection project. This project can be extended to build a system to replace finger print based biometric systems, to improve the search methods in matching a persons's frontal face photo with existing database of criminals, ATM user security, and many more. Further, the process of feature extraction can be included in the neural network by building deep convolutional neural networks to achieve the same.

Acknowledgements

We take this opportunity to express our profound gratitude and deep regards to our guide Dr. Deepak Mishtra, Associate Professor & Head, Department of Electronics and Communications, for his exemplary guidance, monitoring and constant encouragement throughout the course of this project. The blessing, help and guidance given by him, time to time shall carry us a long way in the journey of life on which we are about to embark.

We thank our beloved parents, siblings and dear friends for their continued support and encouragement for the successful completion of our project.

Contents

Chapter 1

Introduction

1.1 Principal Component Analysis[1]

Basic Idea

PCA is about projecting (Dot Product) data points onto a line, i.e. data points are projected into a lower dimension. PCA is a projection such that a variance in that projection is as high as possible. PCA is just finding the projection such that most of the variance of the data is accounted for.
The first thing in a PCA is a shift of the data onto a new coordinate system by calculating the mean for every dimension in the data, then subtracting every observation in that dimension by the mean.
$\widetilde{X}$ is the new X,while μ (mu) is the mean,
$\widetilde{X} = X - \mu$.
The origin of our coordinate system would now have been shifted to the center of our data, by subtracting the mean- $\sum = \frac{1}{n-1} \sum_{i=1}^{n} x_i$

Finding Principal Components

the objective would be to find the direction in which we get the most variance (imagine a line through origin and the data points). To find the most variance we want the directions where there are most data points covered (e.g. by drawing a line in that direction).
What we really want is the line, after having projected all data points, where we have maximized the sum of the distance from all points to origin. That is what most variance is. So for each data point, we would have a

distance d and we would have n data points, so we would sum all of the squared distances and divide them by $n-1$-
$\frac{d_1^2+d_2^2+...+d_n^2}{n-1} = variance$

Finding the Eigenvectors and Eigenvalues

How to get the line that accounts for most variance for our data. This is what we call an eigenvalue problem and here is the formula: $A\vec{v} = \lambda\vec{v}$
A is a transformation, a squared matrix,$\vec{v}$ is the eigenvector,and λ is the eigenvalue.Most of the time, the litterature will represent or derive matrix A as- $\tilde{x}^T\tilde{x}$.
The procedure for finding the line is-

- Finding the eigenvalues.
- Finding the eigenvectors from the eigenvalues.

For first part,To find the eigenvalues, we have- $det(A-\lambda I) = 0$
For the second part- To find the eigenvectors, we would want to change up the formula. So we move everything from the right side of the formula to the left side and set it equal to 0: $A\vec{v} = \lambda\vec{v} \iff (A-\lambda)\vec{v} = 0$
we would find a resulting eigenvector for each eigenvalue. To solve this,we can use a Gaussian process (e.g. Gaussian Elimination).
The eigenvector corresponding to the highest eigenvalue is the principal component.We can have more than one principal component if we use eigenvalues which are significant as compared to the highest eigenvalue.[2]

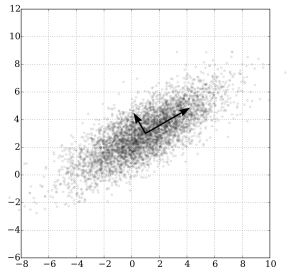


Figure 1.1: Principal Components

1.1.1 Singular Value Decomposition

In linear algebra, the singular value decomposition (SVD) is a factorization of a real or complex matrix that generalizes the eigendecomposition of a square normal matrix to any $m \times n$ matrix via an extension of the polar decomposition.

The singular value decomposition (SVD) provides another way to factorize a matrix, into singular vectors and singular values. The SVD allows us to discover some of the same kind of information as the eigendecomposition. However, the SVD is more generally applicable.

Specifically, the singular value decomposition of an $m \times n$ real or complex matrix $\mathbf{M}$ is a factorization of the form $\mathbf{U\Sigma V}^*$, where $\mathbf{U}$ is an $m \times m$ real or complex unitary matrix, $\mathbf{\Sigma}$ is an $m \times n$ rectangular diagonal matrix with non-negative real numbers on the diagonal, and $\mathbf{V}$ is an $n \times n$ real or complex unitary matrix. If $\mathbf{M}$ is real, $\mathbf{U}$ and $\mathbf{V^T} = \mathbf{V}^*$ are real orthonormal matrices.

The diagonal entries $\sigma_i = \Sigma_{ii}$ of $\mathbf{\Sigma}$ are known as the singular values of $\mathbf{M}$. The number of non-zero singular values is equal to the rank of $\mathbf{M}$. The columns of $\mathbf{U}$ and the columns of $\mathbf{V}$ are called the left-singular vectors and right-singular vectors of $\mathbf{M}$, respectively.

The singular value decomposition is very general in the sense that it can be applied to any $m \times n$ matrix, whereas eigenvalue decomposition can only be applied to diagonalizable matrices.

$$\overset{\hat{X}}{\underset{m \times n}{\begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & \\ \vdots & \vdots & \ddots & \\ x_{m1} & & & x_{mn} \end{pmatrix}}} \approx \overset{U}{\underset{m \times r}{\begin{pmatrix} u_{11} & \cdots & u_{1r} \\ \vdots & \ddots & \\ u_{m1} & & u_{mr} \end{pmatrix}}} \overset{S}{\underset{r \times r}{\begin{pmatrix} s_{11} & 0 & \cdots \\ 0 & \ddots & \\ \vdots & & s_{rr} \end{pmatrix}}} \overset{V^{\mathsf{T}}}{\underset{r \times n}{\begin{pmatrix} v_{11} & \cdots & v_{1n} \\ \vdots & \ddots & \\ v_{r1} & & v_{rn} \end{pmatrix}}}$$

Figure 1.2: Singular Value Decomposition

1.2 Problem Statement

A face recognition system (software) is to be designed. The program reduces each face image to a vector, then uses Principal Component Analysis (PCA) to find a linear subspace for faces. This space is spanned by just a few vectors, which means each face image can be defined by a small set of coefficients weighting these vectors, thus reducing and considering only the information from large data for further analysis and processing.

1.3 Objective

The primary objective of this project is to design a system which is able to take in an image, locate faces of people in the image and identify each one of them using Eigenface method. The objective can be broken down into two broad sub-objectives as:

1. Face Detection
2. Face Recognition

Machine Learning algorithms and classifiers are to be used for identifying (classifying) the detected faces in given test image. The input test image may contain faces of more than one person (i.e., group photo).

Chapter 2

Literature Survey

2.1 Survey on Real Life Requirements of Face recognition systems

The literature survey was performed to get an insight to the real life situations where face detection and recognition systems can reduce effort and time.

The most crucial requirement of face recognition system is in **criminology**. The database of photographs of the past accused and convicts is huge. The management of the database itself if a tiresome task. This data contains much of redundant information, for example the pixels corresponding to the background of the person, etc. Often when new cases arise, the photographs of the suspects is matched with the pre-developed database. This task could take long time (increase in time with database size) without use of optimized search algorithms. These two problems of storage size of database and speed of identification can be overcome by building machine learning systems and optimizing over various search optimization algorithms and GPU execution.

Next prominent use of the face recognition system is in social media such as Facebook, Twitter, etc., Facebook, has already implemented and optimized its face recognition system - evident from the accuracy of auto-tagging feature.

Another upcoming technology is the phone face unlock feature which requires use of face recognition systems.

Also, with the increase in the number of CCTV cameras throughout the

globe, the effort and time required for finding missing people (especially, children) can be reduced drastically by incorporating a face recognition system into CCTV cameras in public places with direct data relay to the appropriate police stations. Such applications require real time face identification to identify the people in the video frames.

The face recognition system find its need in ATMs to validate identity of people transacting in the ATMs.

One of the requirements which is much advocated by higher authorities but strongly opposed by students is face recognition based automatic attendance system.

Similarly, many such real life requirements do exist, where face recognition systems can play a cruical role.

2.2 Literature Review

Various students, research scholars and industrialists working in **Pattern Recognition and Machine Learning** have developed face recognition systems for various applications. The methods used by each vary and so does the accuracy and other performance metrics.

A team of 4 students from Jawaharlal Nehru Technological University Kakinada, designed a "Face Recognition System With Face Detection" under the guidance of Ms. SK. AYESHA, M.Tech, Assistant Professor of E.C.E dept during the academic year 2013-17. The implementation was done in MATLAB software. The face detection was achieved through face localization (using features such as skin color detection), followed by mouth, nose and eyes and verifying eyes-mouth triangle. Fully automated face detection of frontal view faces was implemented using a deformable template algorithm relying on image invariants of human faces. Face recognition was based on Principal Component Analysis. 30 test subjects were used for the analysis.

A Face recognition system was built by Yang Li and Sangwhan Cha, PhD. Assistant Professor of Computer Science,[3] in which ORI dataset of 400 frontal face images (of 40 subjects, with 10 faces per person) was used. A Siamese network was designed for measuring similarity between two subject's faces. The model training and complete execution was done with torch tensors on Nvidia GPU. The system so made produces appreciable results only when number of subjects is less and number of faces per person

is large. No block was built for face detection.

A famous project on face detection and recognition is : **Open Face**[4]. Implemented in Python and Torch, the workflow of the system includes following steps: Detect faces with a pre-trained models from dlib or OpenCV, transform the face for the neural network using OpenCV's affine transformation, crop out the face from the image, use a deep neural network for training and use the trained model classifier for clustering, similarity detection and classification.

The above list of projects is non-exhaustive, as many research papers of new architectures for face recognition systems with deep learning, convolutional neural networks, etc., are coming up every now and then with a large community or group of teams putting in effort to build real time applicable face recognition systems.

Chapter 3

Requirement Specification

The following problem scope for this project was arrived at after reviewing the literature on face detection and face recognition, and determining possible real-world situations where such systems would be of use. The following system(s) requirements were identified:

1. A system to detect frontal view faces in static images
2. A system to recognize a given frontal view face
3. All implemented systems must display a high degree of lighting invariance.
4. All systems must posses near real-time performance.
5. A fully automated face detection must be supported
6. Frontal view face recognition will be realised using only a set of known frontal view face images
7. The face recognition sub-system must display a slight degree of invariance to scaling and rotation errors in the segmented image extracted by the face detection sub-system.

Unfortunately although we may specify constricting conditions to our problem domain, it may not be possible to strictly adhere to these conditions when implementing a system in the real-world.

Chapter 4

Design

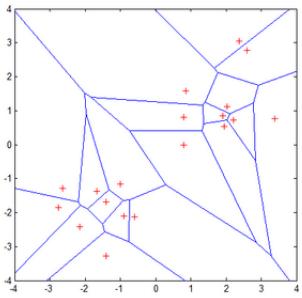
The present chapter is dedicated to explain the conceptual overview of the face detection and recognition system developed. The theory and algorithms are described with aid of diagrams and flow charts.

4.1 Classifiers

4.1.1 k Nearest Neighbours[5]

The Basic Idea of KNN and Voronoi set

The Basic idea of KNN is based on minimum euclidean distance between a test sample and specified training sample.Let $\mathbf{x}_i$ be input with p features $(x_{i1}, x_{i2}, \ldots, x_{ip})$, n be the number of samples $(i = 1, 2, \ldots, n)$, and$p$ be the total number of features(j=1,2,. . . ,p) .The euclidean distance between $\mathbf{x}_i$ and $\mathbf{x}_l$ $(l = 1, 2, \ldots, n)$is given by
$d(\mathbf{x}_i, \mathbf{x}_l) = \sqrt{(x_{i1} - x_{l1})^2 + (x_{i2} - x_{l2})^2 + \cdots + (x_{ip} - x_{lp})^2}$.
The concept of nearest neighbor is explained using a diagram here. The minimum distance points corresponding to one specific training samples are defined by the boundaries of the Voronoi cell corresponding to that specific training samples.



A Voronoi cell corresponding to one training point is defined by-
$R_i = \{\mathbf{x} \in \mathrm{R}^p : d(\mathbf{x}, \mathbf{x}_i) \leq d(\mathbf{x}, \mathbf{x}_m), \quad \forall \quad i \neq m\}$,the fundamental properties reflected from these sets are that-

- all possible points within a sample's Voronoi cell are the nearest neighboring points for that sample.
- for any sample, the nearest sample is determined by the closest Voronoi cell edge.

Using the 2nd property, the k-nearest-neighbor classification rule is to assign to a test sample the majority category label of its k nearest training samples. In practice, k is usually chosen to be odd, so as to avoid ties.

Classification decision rule

Classification typically involves partitioning samples into training and testing categories.Let $\mathbf{x}_i$ be a training sample and $\mathbf{x}$ be a test sample, and let ω be the true class of a training sample and $\hat{\omega}$ be the predicted class for a test sample $(\omega, \hat{\omega} = 1, 2, \ldots, \Omega)$. Here, Ω is the total number of classes. During the training process, we use only the true class ω of each training sample to train the classifier, while during testing we predict the class $\hat{\omega}$ of each test sample. It warrants noting that kNN is a "supervised" classification method in that it uses the class labels of the training data. With 1-nearest neighbor rule, the predicted class of test sample $\mathbf{x}$ is set equal to the true class ω of its nearest neighbor, where $\mathbf{m}_i$ is a nearest neighbor to $\mathbf{x}$ if the distance-
$d(\mathbf{m}_i, \mathbf{x}) = \min_j\{d(\mathbf{m}_j, \mathbf{x})\}$.
For k-nearest neighbors, the predicted class of test sample $\mathbf{x}$ is set equal to the most frequent true class among k nearest training samples. This forms the decision rule $D : \mathbf{x} \rightarrow \hat{\omega}$.

Confusion Matrix

The confusion matrix used for tabulating test sample class predictions during testing is denoted as $\mathbf{C}$ and has dimensions $\Omega \times \Omega$.

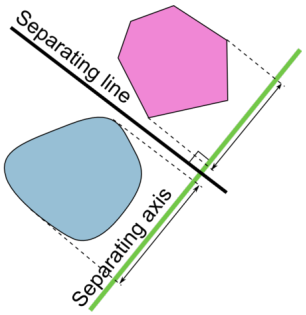
- During testing, if the predicted class of test sample $\mathbf{x}$ is correct (i.e.,$\hat{\omega}$ =ω), then the diagonal element $c_{\omega\omega}$ of the confusion matrix is incremented by 1.
- However, if the predicted class is incorrect (i.e., $\hat{\omega} \neq \omega$), then the off-diagonal element $c_{\omega\hat{\omega}}$ is incremented by 1.

Once all the test samples have been classified, the classification accuracy is based on the ratio of the number of correctly classified samples to the total number of samples classified, given in the form-
$Acc = \frac{\sum_{\omega}^{\Omega} c_{\omega\omega}}{n_{total}}$
where $c_{\omega\omega}$ is a diagonal element of $\mathbf{C}$ and n_{total} is the total number of samples classified.

4.1.2 Support Vector Machine[6]

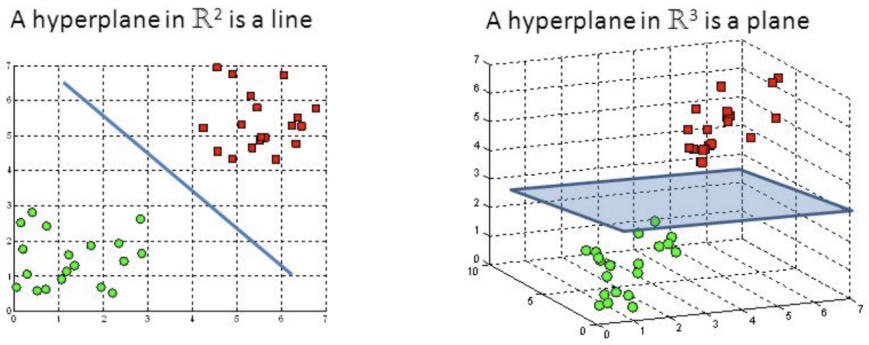
Objective

The objective of SVM is to find a hyperplane in an N-dimensional space (N-Number of features) that distinctly classifies the data points.In the context of support-vector machines, the optimally separating hyperplane or maximum-margin hyperplane is a hyperplane which separates two convex hulls of points and is equidistant from the two.

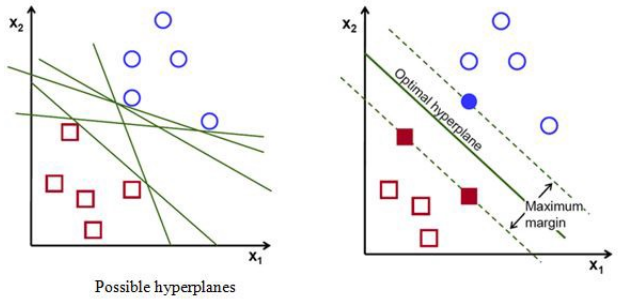


Hyperplane

In an N-dimensional space, a hyperplane is a flat affine subspace of dimension N-1. Visually, in a 2D space, a hyperplane will be a line and in 3D space, it will be a flat plane.In simple terms, hyperplane is a decision boundary that helps classifying data points.

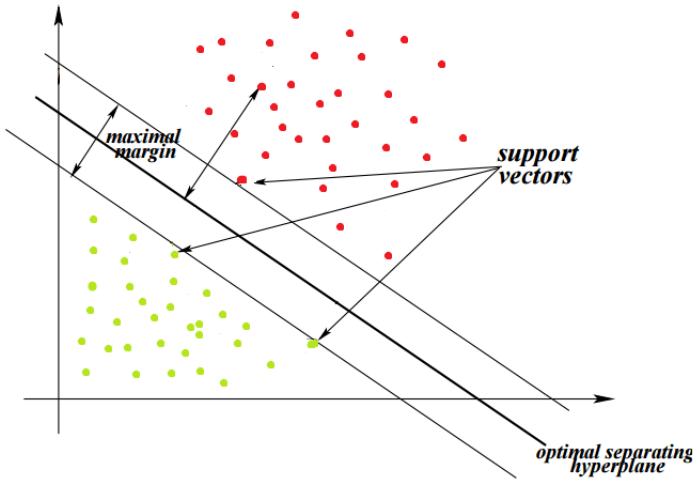


Now, to separate two classes of data points, there are many possible hyperplanes that could be chosen. Our objective is to find a plane that has the maximum margin i.e. the maximum distance between data points of both classes and below figure clearly explains this fact.



Support Vectors

Support Vectors are the data points that are on or closest to the hyperplane and influence the position and orientation of the hyperplane. Using these support Vectors we maximize the margin of the classifier and deleting these support vectors will change the position of the hyperplane. These are actually the points that help us build SVM.



Support Vectors are equidistant from the hyperplane. They are called support vectors because if their position shifts, the hyperplane shifts as well. This means that the hyperplane depends only on the support vectors and not on any other observations.

Non-linear Classification

Concept of Kernel in SVM to classify non-linearly separated data. A kernel is a function which maps a lower-dimensional data into higher dimensional data. There are two ways by which kernel SVM will classify non-linear data.

- Soft Margin
- Kernel Tricks

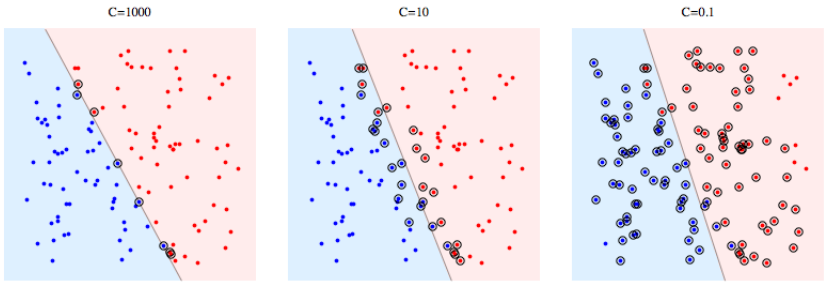
Soft Margin

In this method,SVM tolerates a few dots to get misclassified and tries to balance the tradeoff between finding the line that maximizes the margin and minimizes misclassification.
There are two types of misclassifications that can happen:

- The data point is on the wrong side of the decision boundary but on the correct side.
- The data point is on the wrong side of the decision boundary and on the wrong side of the margin.

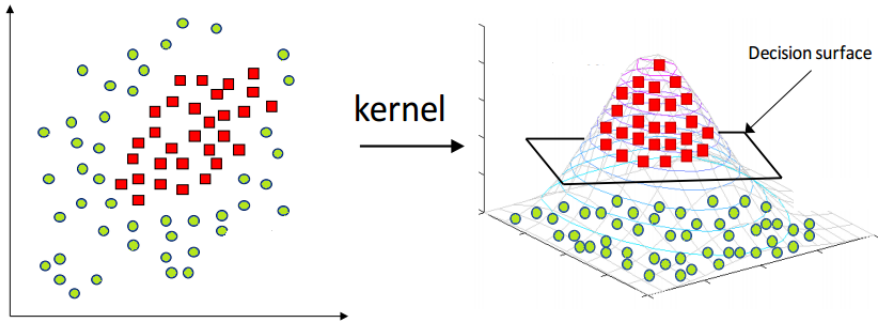
Degree of Tolerance

How much tolerance we want to set when finding the decision boundary is an important hyper-parameter for the SVM (both linear and nonlinear solutions). In Sklearn, it is represented as the penalty term — 'C'.
The bigger the C, the more penalty SVM gets when it makes misclassification. Therefore, the narrower the margin is and fewer support vectors the decision boundary will depend on.



Kernel Trick

The idea is mapping the non-linear separable data from a lower dimension into a higher dimensional space where we can find a hyperplane that can separate the data points.
Kernel Functions are generalized functions that take 2 vectors(of any dimension) as input and output a score(dot product) that denotes how similar the input vectors are. If the dot product is small, vectors are different and if the dot product is large, vectors are more similar.
Pictorial representation of the Kernel Trick-



Different types of kernel Functions are:

- Linear
- Polynomial
- Radial Basis Function
- Sigmoid

The most popular rbf kernel is Gaussian Radial Basis function.
Mathematically:

$$k(\mathbf{x_i}, \mathbf{x_j}) = \exp(-\gamma \|\mathbf{x_i} - \mathbf{x_j}\|^2)$$

where gamma(γ) controls the influence of new features on the decision boundary. Higher the value of γ more influence of features on the decision boundary.
Similar to Regularization parameter/penalty term(C) in the soft margin, Gamma(γ) is a hyperparameter that can be tuned when we use kernel trick.

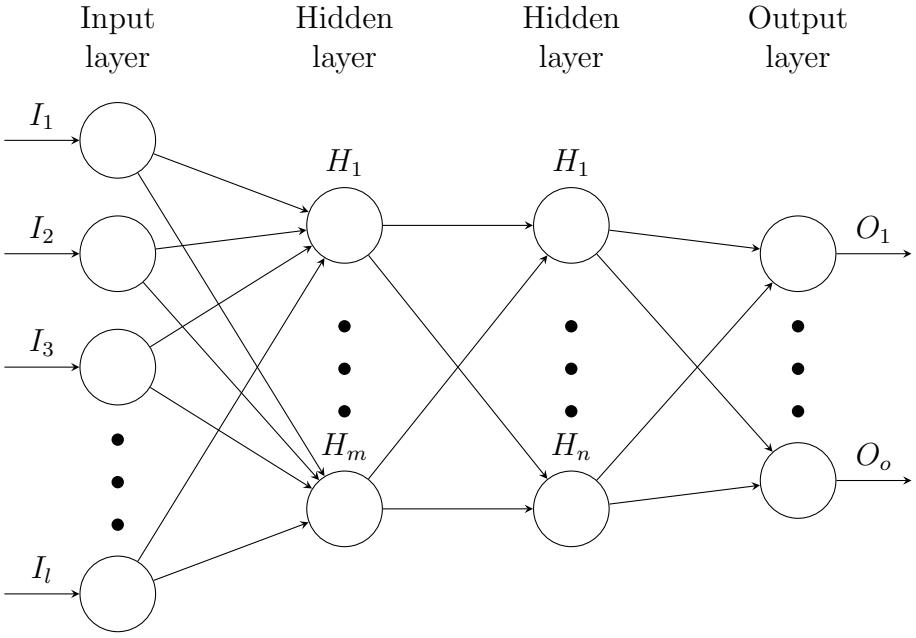
4.1.3 Feed Forward Neural Network[7]

Basic Idea

A neural network simply consists of neurons (also called nodes). These nodes are connected in some way. Then each neuron holds a number, and each connection holds a weight.
These neurons are split between the input, hidden and output layer. In practice, there are many layers and there are no general best number of layers.

Input Layer

The input data is just the dataset, where each observation is run through sequentially from $x = 1, ..., x = i$ Each neuron has some activation — a value between 0 and 1, where 1 is the maximum activation and 0 is the minimum activation a neuron can have



Input Layer to Hidden Layer

each neuron has an activation a and each neuron that is connected to a new neuron has a weight w. Activations are typically a number within the range of 0 to 1, and the weight is a double, one could multiply activations by weights and get a single neuron in the next layer, from the first weights and activations w_1a_1 all the way to w_na_n:
$w_1a_1 + w_2a_2 + ... + w_na_n = new\ neuron.$
The procedure is the same moving forward in the network of neurons, hence the name feed forward neural network, and finally we reach the output layer.

Hidden Layer to Output Layer

The neuron outputs from the final hidden layer is given to the Output layer, which contains neurons among which one activation will be higher than all others corresponding to a particular class, sometimes it can be related to a probability, where each neuron in the output layer contains some activation value(some small positive value for all neurons except the one corresponding to the right class which will have activation number close to 1.)

Activation Function

An activation function, most commonly a sigmoid function, just scales the output to be between 0 and 1 again — so it is a logistic function.
$sigmoid = \sigma = \frac{1}{1+e^{-x}}$ = number between 0 and 1.

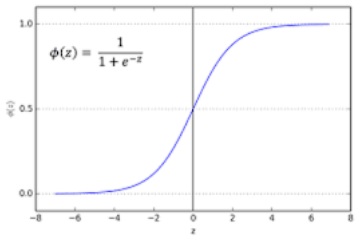


Figure 4.1: Sigmoid Function

We wrap the equation for new neurons with the activation, i.e. multiply summarization of the result of multiplying the weights and activations $\sigma(w_1a_1 + w_2a_2 + ... + w_na_n)$ = new neuron.
Other commonly used activation functions include: tanh, ReLU, leaky ReLU, etc., ReLU and leaky ReLU were introduced to overcome the issue of vanishing gradients when the activation function saturates, causing no effective learning.

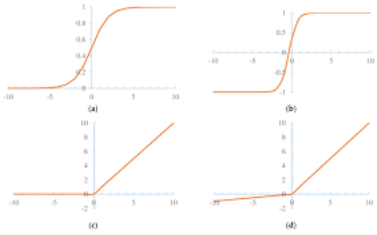


Figure 4.2: Different Activation Functions

Bias

Bias is trying to approximate where the value of the new neuron starts to be meaningful. We just need to add or subtract bias from the multiplication of weights and activations. $\sigma(w_1a_1 + w_2a_2 + ... + w_na_n + b)$ = new neuron.

Cross Entropy Loss

Of the various loss function used for training a neural network such as MSE, 1-0 loss function, etc., cross entropy loss is most commonly used in practice, along with L2 regularization. Below is the description of the cross entropy loss function:

Cross-entropy loss is often simply referred to as "cross-entropy," "logarithmic loss," "logistic loss," or "log loss" for short.

Each predicted probability is compared to the actual class output value (0 or 1) and a score is calculated that penalizes the probability based on the distance from the expected value. The penalty is logarithmic, offering a small score for small differences (0.1 or 0.2) and enormous score for a large difference (0.9 or 1.0).

Cross-entropy loss is minimized, where smaller values represent a better model than larger values. A model that predicts perfect probabilities has a cross entropy or log loss of 0.0.

Cross-entropy for a binary or two class prediction problem is actually calculated as the average cross entropy across all examples.

Cross-entropy can be calculated for multiple-class classification. The classes have been one hot encoded, meaning that there is a binary feature for each class value and the predictions must have predicted probabilities for each of the classes. The cross-entropy is then summed across each binary feature and averaged across all examples in the dataset.

4.2 Data

The data used for this project was borrowed from *https://courses.cs.washington.edu/courses/cse455/09wi/projects/project4/web/project4.html*. The data comprises of totally 24 different people with two images per person - one smiling and one non-smiling. This cropped (but, not aligned) images form the training samples. The test images are the group photos, with three people in a group. All the images were colored images (having R,G,B components).

4.3 Training Pipeline

The smiling and non-smiling cropped, frontal face images of all 24 people from the training samples. As the training dataset size was small, the data was appended (data augmentation) to the training samples by adding Gaussian (white) noise to each image and rotating it by random angle within a finite interval comprising of rotation in both sense (clockwise and anti-clockwise) and generating new images, so as to increase number of samples per person. The training samples are first pre-processed using OpenCV's image processing tool-box. The image processing steps include:

1. Resizing: All the training images were resized to an image of 128×128 pixels. Each training images made available are not of identical dimension.

2. Histogram Equalization: It is a technique to improve contrast in images, which is accomplished by effectively spreading out the most frequent intensity values, i.e. stretching out the intensity range of the image. This method usually increases the global contrast of images when its usable data is represented by close contrast values. This allows for areas of lower local contrast to gain a higher contrast.

3. Face Alignment: The face alignment step is used to ensure that all the faces are aligned such that face is not tilted (i.e., is straight).

4. RGB to Gray scale conversion: All the input images were converted to gray scale images as individual components of the images were all identical to each other, expect for the scale of each image pixel range. As no extra advantage over gray scale image is provided by colored images, all the images (both test and train samples) were converted to gray scale images.

Next, is the step relating to Principal Component Analysis (PCA). Before proceeding, all the images were linearized (into linear 1D arrays). Through the method of PCA, the input training images were mapped from a image space of dimensions 16384(128×128) to face space of dimensions 54 ($\frac{1}{8}$ of the original number of dimensions). Thus, the number of principal components used is 54. In the process, the first 54 eigenfaces were considered for further analysis. Also, a transformation matrix was constructed to map input images from image (pixel) space to face space

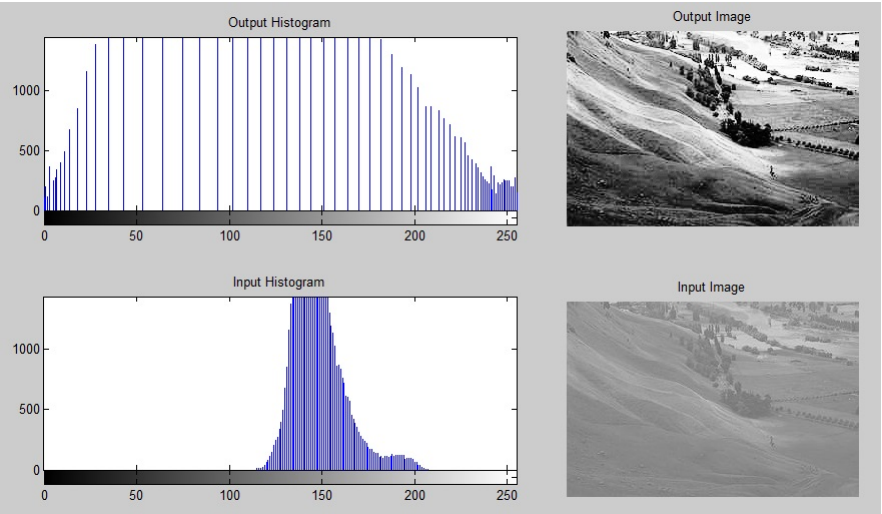


Figure 4.3: Histogram Equalization

(matrix dimension: 54×16384). Next, for training of the classifier, these face space vector representation of the training images were used.
For classification purpose three different classifiers were tested and finally, an ensemble of classifier was designed for improved performance of the system. The three classifiers used are:

1. k Nearest Neighbour
2. Support Vector Machine
3. (Fully Connected) Neural Network

The theory of each of the above classifiers is described in detail in the section of "Classifiers" of this chapter.

4.4 Face Detection and Recognition Pipeline

As mentioned previously, the test images are group photos of three random people (out of 24 people) in a group. The first step in this pipeline is detecting of faces in a group photo. This is achieved using OpenCV's Haar cascade classifiers (based on Viola Jones Algorithm).
Once the classifier locates the faces of the people in the test image, a rectangle is drawn around the face. Then, the coordinates of these

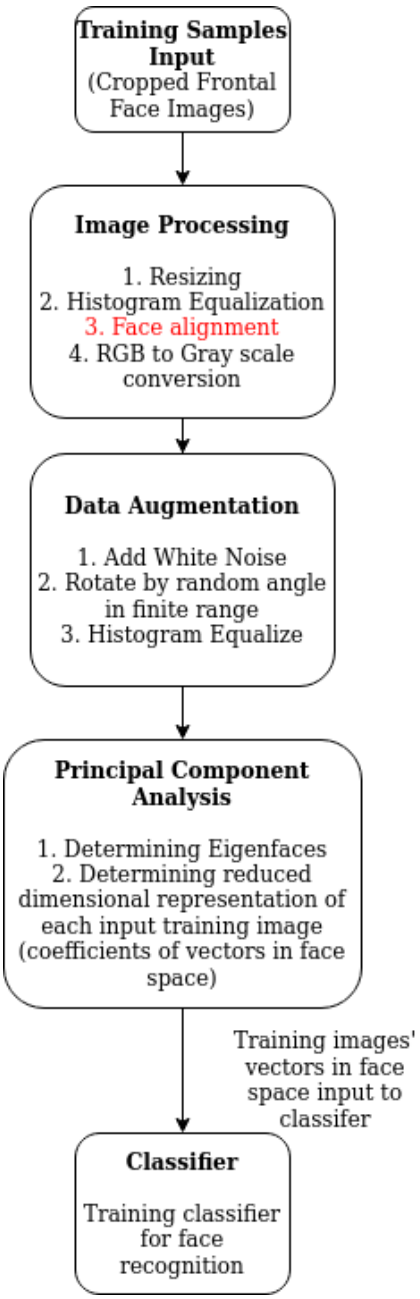


Figure 4.4: Training Pipeline

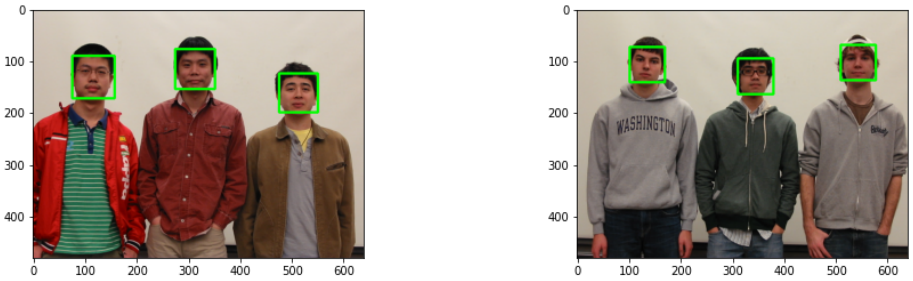


Figure 4.5: Face Detection using OpenCV's Haar Cascade Classifier (Viola Jones Algorithm)

rectangle's corner and width and height of each rectangle are used to crop out the faces from the image.

The step after cropping out the faces is basically image processing step. Not all the rectangles drawn by the algorithm are of same dimensions, hence, each is cropped face needs to be resized. The crucial step here is re-cropping the faces cropped out of group photos. The step is crucial because, the accuracy of the classifier critically depends on the degree of resemblance of the cropped test face images with the training set images (the training set cropped images are in fact cropped out of these group photos manually and uploaded, thanks to the Computer Vision group of University of Washington, Computer Science & Engineering).

After cropping the images to match the training samples as much as possible, all the images are resized to the same dimensions as pre-processed training samples, followed by histogram equalization and then RGB to Gray scale conversion.

Once, the pre-processing the test images is done, the images are mapped to face space vectors and plugged into trained classifier(s) for face recognition. The predicted outputs are compared with the ground truth and the top-1,5,10 accuracy scores and confusion matrix are determined.

Figure 4.6: Testing Samples: Cropped Faces from the group photos

4.5 Evaluation Metrices

Confusion Matrix

A confusion matrix is a summary of prediction results on a classification problem. The number of correct and incorrect predictions are summarized with count values and broken down by each class. This is the key to the confusion matrix. The confusion matrix shows the ways in which your classification model is confused when it makes predictions. It gives us insight not only into the errors being made by a classifier but more importantly the types of errors that are being made. Here

- Class1:positive
- Class2:negative

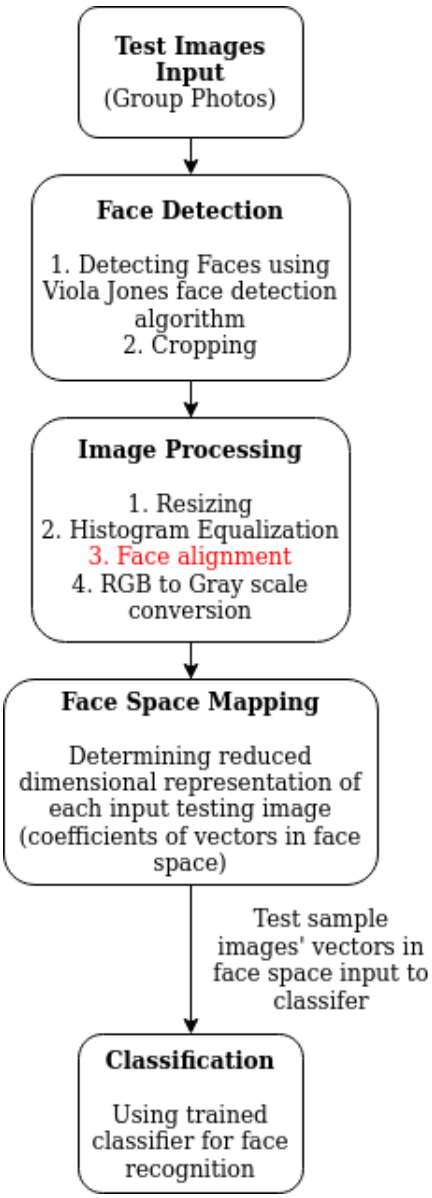


Figure 4.7: Face Detection and Recognition Pipeline

Definition of the Terms:

- Positive (P) : Observation is positive (for example: is an apple).
- Negative (N) : Observation is not positive (for example: is not an

	Class 1 Predicted	Class 2 Predicted
Class 1 Actual	TP	FN
Class 2 Actual	FP	TN

Figure 4.8: Confusion Matrix

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Figure 4.9: Accuracy

apple).

- True Positive (TP) : Observation is positive, and is predicted to be positive.
- False Negative (FN) : Observation is positive, but is predicted negative.
- True Negative (TN) : Observation is negative, and is predicted to be negative.
- False Positive (FP) : Observation is negative, but is predicted positive.

Classification Rate/Accuracy:

Classification Rate or Accuracy is given by the relation: However, there are problems with accuracy. It assumes equal costs for both kinds of errors. A 99% accuracy can be excellent, good, mediocre, poor or terrible depending upon the problem.

$$Recall = \frac{TP}{TP + FN}$$

Figure 4.10: Recall

$$Precision = \frac{TP}{TP + FP}$$

Figure 4.11: Precision

Recall:

Recall can be defined as the ratio of the total number of correctly classified positive examples divide to the total number of positive examples. High Recall indicates the class is correctly recognized (a small number of FN).

Precision:

To get the value of precision we divide the total number of correctly classified positive examples by the total number of predicted positive examples. High Precision indicates an example labelled as positive is indeed positive (a small number of FP).

- **High recall, low precision** This means that most of the positive examples are correctly recognized (low FN) but there are a lot of false positives
- **Low recall, high precision:** This shows that we miss a lot of positive examples (high FN) but those we predict as positive are indeed positive (low FP).

F-measure:

Since we have two measures (Precision and Recall) it helps to have a measurement that represents both of them. We calculate an F-measure which uses Harmonic Mean in place of Arithmetic Mean as it punishes the extreme values more. The F-Measure will always be nearer to the smaller value of Precision or Recall.

$$F\text{ - measure} = \frac{2*\text{Recall}*\text{Precision}}{\text{Recall} + \text{Precision}}$$

Figure 4.12: F-measure

4.6 Block Diagram

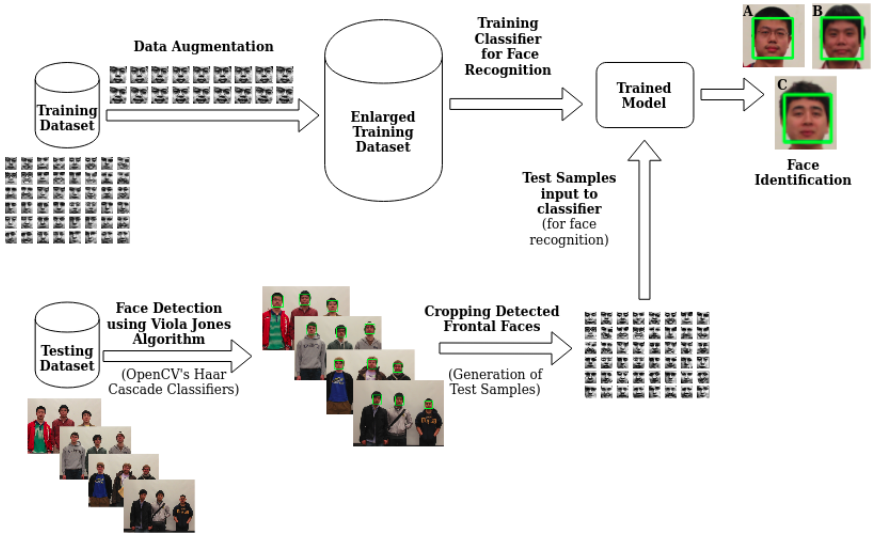


Figure 4.13: Block Diagram of Face Recognition System Designed

Chapter 5

Implementation

The present chapter is dedicated to explain the implementation of the algorithms described in previous chapter and also describe the code structure and the work flow.

The system design described in above chapter was implemented in Python programmin language (in Jupyter notebook on Google Colab platform - with GPU hardware accelerator). Effort was made to hand-code most of the important functions in the pipeline and minimize the usage of inbuild functions for the same. The input images were all ".tga" images. Hence **PIL** module's **Image** function was used to read the images. Further, image processing was done using **OpenCV**'s module functions. As the number of images per person for training was very low (2 per person), data augmentation was done by appending noisy images to the data. Random Gaussian (white) noise was added to each original training samples and the number of images per person was extended to 22 (11 per facial expression per person). In the process, the training images were even rotated by an angle and the range $(-5°,5°)$ so as to train the model to perform well even on data in which the input images are not aligned straight. At the end, each new augmented image was histogram equalized. Histogram equalization of augmented data is necessary, since all the test images are also histogram equalized. It was, in fact observed that the performance of each classifier improved significantly (increase in test accuracy by $\approx 7\%$ for kNN, $\approx 2\%$ for SVM and $\approx 4\%$ for FFNN) after histogram equalizing the augmented noisy data. It is observed that the process of data augmentation is the most time consuming part of the code.

Next, the Principal Component Analysis was performed using user defined

functions (without using inbuilt library function of Python). The PCA was performed as shown in the flow chart below. The dimensions of the matrices, in the implementation are:

Matrix	Dimensions	Dimension Used
M	(432, 16384)	(432, 16384)
U	(432,432)	(432,54)
S	(432,432)	(54,54)
V	(16384, 16384)	(54,16384)

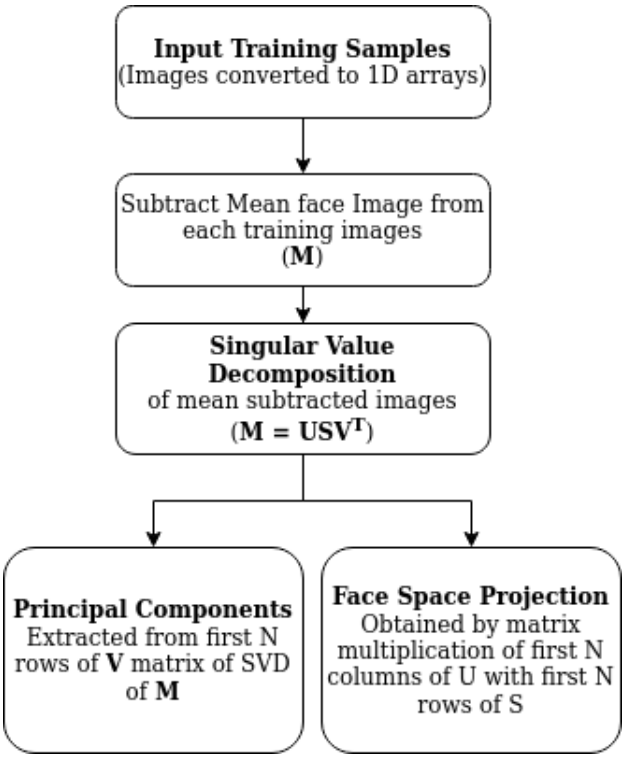


Figure 5.1: PCA Pipeline

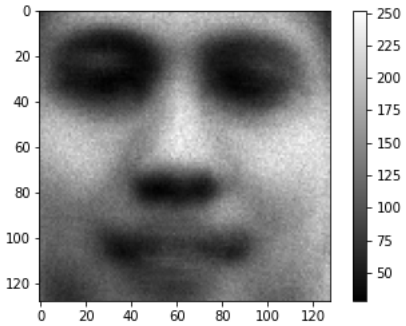


Figure 5.2: Mean Face (computed from training set)

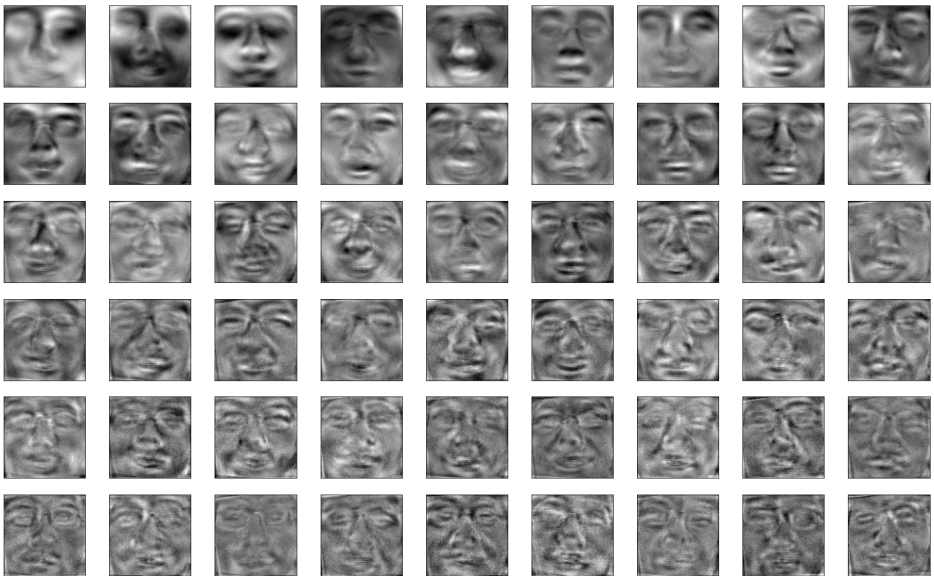


Figure 5.3: Eigen Faces used for Face Recognition (54 Prinicpal Components)

The face detection was achieved using Haar cascade classifier for face detection - $haarcascade\_frontalface\_alt.xml$ - with the parameters of scaleFactor and minNeighbors to 1.2 and 5 respectively. Finer values of the same were required for detecting all the faces in other images, given for practicing face detection. For example, in order to detect all the faces in the following image, values of 1.1 and 1 for scaleFactor and minNeighbors

were required to be used.

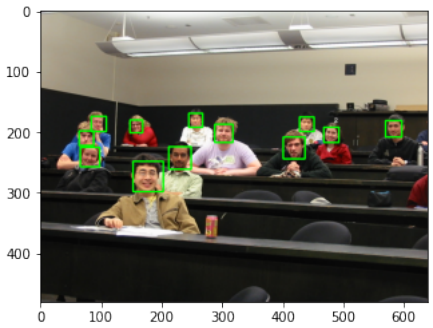


Figure 5.4: Face Detection - on Practice Image

The *rectangle* function is used for visualizing face detection process. The cropping of test face images to best match the training images of frontal face, a common crop dimensions and crop location was used (as mentioned above). This is a trade off between degree of resemblance and manual effort. It is possible to crop out each image manually to best resemble the training samples, yet not a good practice in general as real life situations contain images of numerous (1000s of) people. No metric was used to test the best crop dimensions, instead was completed by simple visual aid.

By trial and error, it was found that cropping the section [24:120, 16:112] of each image, produced an accuracy of the order of 70% in contrast to an accuracy of the order of 25% when the Viola Jones algorithm based cropped images were straight forwardly used for face recognition. Thus, resemblance of test images with training samples is pivotal in performance of any trained classifier in face recognition.

Post face detection, the test images' pre-processing was done using following OpenCV's module functions:

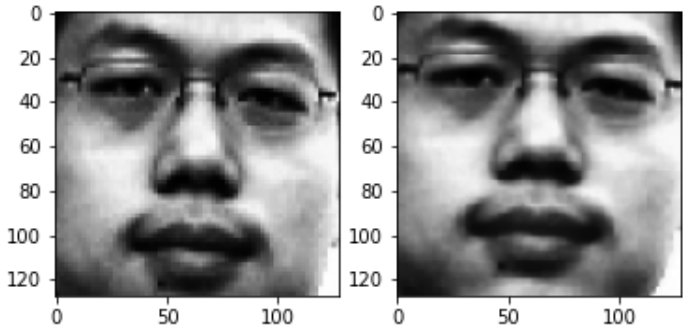


Figure 5.5: Training image vs Cropped and resized test image (an example)

Function	Usage description
resize	Used to resize all the images (both train and cropped test images) to a common dimensions of $128X128$ along with interpolation
hist_equalizer	Histogram Equalization
crop	Used to crop detected face images to match training samples ($[24:120, 16:112]$)
rectangle	Used to draw a rectangle around the detected faces in the group photo

Next is the step of training the classifier(s). The first classification algorithm used is **k Nearest Neighbour**. An accuracy of 72.916% was obtained using this algorithm with k = 1. A significant decline in the accuracy was obtained with increase in the value of k.

Next classifier used was Support Vector Machine. For this classifier, python libraries and inbuilt functions were used. Attempt was made to hand-code the function for SVM. Due to lack of knowledge in optimization algorithms and their implementation, inbuilt functions and kernels were used. This classifier produced an accuracy of 79.16%. The specifications of the SVM classifier used are as follows:

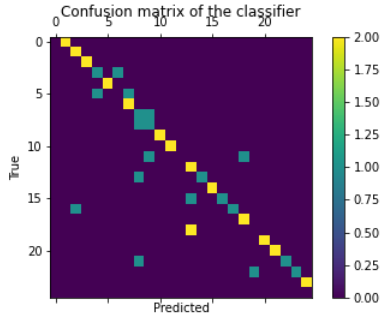


Figure 5.6: Confusion Matrix: kNN

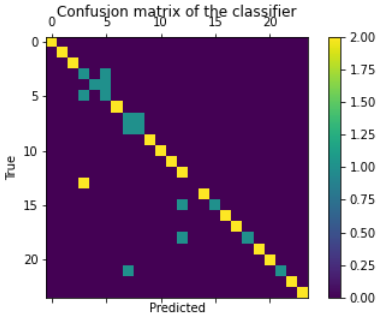


Figure 5.7: Confusion Matrix: SVM

Parameter	Value
C	10
Kernel	rbf
γ	0.01

Next classifier used is Fully Connected Neural Network. A shallow neural network with 2 hidden layers was implemented. Variety of neural network configurations were tried and top-1,5,10 accuracy values were used to choose among the models tried. Further, hyper-parameters of the neural network, which include:

1. activation function (sigmoid, tanh, ReLU, Leaky ReLU, ..)
2. learning rate (0.01,0.001,..)
3. number of epochs (100s,1000s,..)
4. regularization method (L1, L2, ...)
5. type of initialization method (random, Xavier, He, ...)
6. loss function (cross entropy loss, MSE, ..)

Below is the description of the experiments conducted to check over fitting of the model and also visualize the efficiency of regularization.
For experimentation purpose and partially as a learning exercise, the complete neural network (not the optimal or final configuration presented) class was hand coded, including the back propagation. For this, no

Figure 5.8: Class FFNetwork

optimization algorithm was implemented and simple (Vanilla) Gradient Descent method was used. The loss function used was cross entropy loss.

First, the simple model (configuration: [54,250,24]) was trained without regularization. To prevent over-fitting of the model to the training data, the input images, during data augmentation was divided into training and validation sets. The validation set accuracy of the model was monitored to prevent over fitting. It was observed that the accuracy of model on validation set was correlated (and in par) with training set, with both reaching 100% accuracy within 100 epochs.

Further, as not much variability was present in the training set, with only some noise added and two different expressions, we expect the validation set accuracy to not truly reflect lack of over-fitting. Thus, **L2 regularization** was included in the model. The weightage given to the regularization term in the loss function was varied and loss plots were observed. The results are displayed below.

Once, the code for neural network was optimized in terms of hyper-parameters, focus was then shifted towards satisfying the requirement of real time performance by the system. The time taken to fit the model by learning for 6000 epochs using serial execution was found to be close to 2 minutes (107.90 s). To optimize over time, all the data was

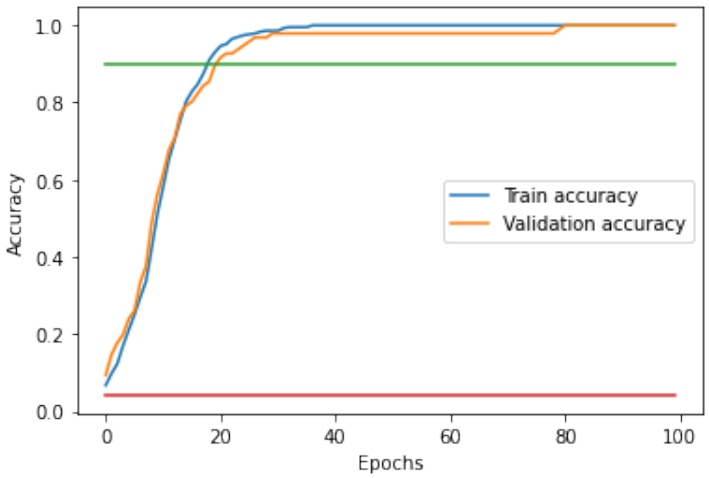


Figure 5.9: Without Regularization

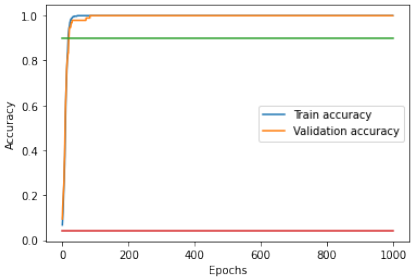
ported from Numpy to PyTorch.

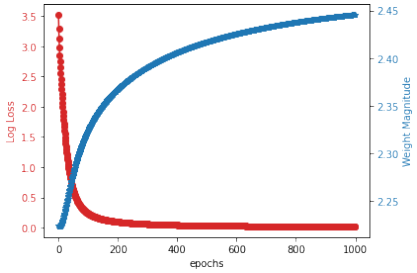
The class was re-defined using PyTorch's **nn.Sequential** with regularization included. The set of hyper-parameters along with their values chosen are displayed in the table below:

Parameter	Value
Activation Function	tanh
Regularization Method	L2 Regularization
Number of Epochs	6000
Type of Initialization	Random
Optimization Algorithm	Stochastic Gradient Descent
Neural Network	[n_components, 250, 150, n_persons]

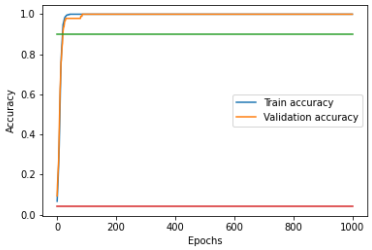
The above are the parameters of the final neural network used for classification, which was trained using PyTorch on GPU. Here, n_components is the number of principal components (54) and n_persons is the number of output classes (24).

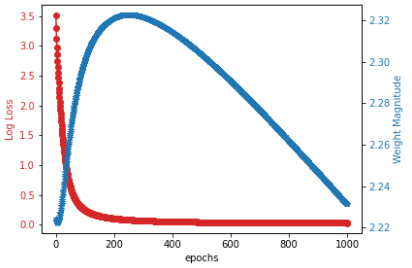
As mentioned in the requirements sections, one of the requirements is real time execution (fast execution). Thus, the neural network was trained on GPU using PyTorch, making use of the massive parallelization offered by Nvidia CUDA. As a good practice during training, the training samples were shuffled using shuffle functionality imported from sklearn.utils. The



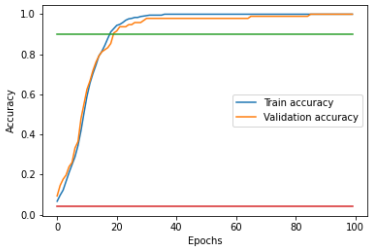


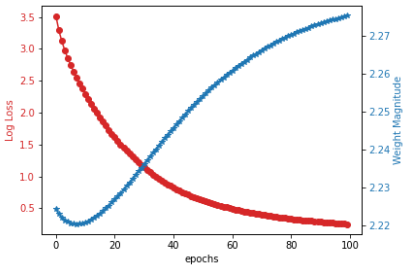
(a) $\lambda = 0.1$



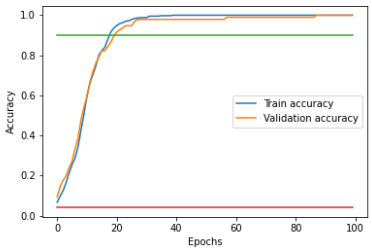


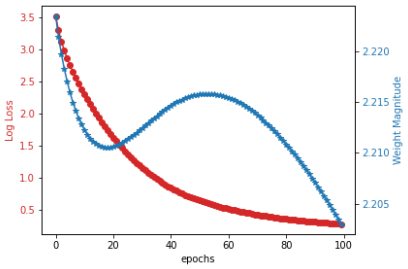
(b) $\lambda = 1$





(c) $\lambda = 2$





(d) $\lambda = 5$

(e) Model Performance vs Regularization fraction

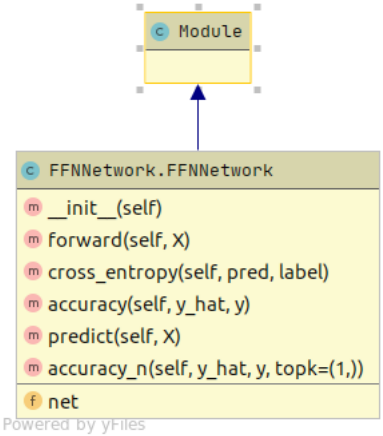


Figure 5.11: Class FFNNetowrk (without regularization)

testing accuracy did drop when training along with shuffling (from 81.25% to 79.16%), but not by a very significant amount.

The back propagation step, now was performed using PyTorch's Automatic Differenciation package. **torch.autograd** provides classes and functions implementing automatic differentiation of arbitrary scalar valued functions. It requires minimal changes to the existing code - one only need to declare torch Tensors for which gradients should be computed with the **requires_grad=True** keyword.[8]

Further, unlike in the previous execution, where simple Vanilla Gradient Descent was used for minimizing loss, here, Stochastic Gradient Descent was used. The oscillation are only observed initially, which get smoothed after a few 100s of epochs. Also, the updation of the weights and bias values is completed with just two command: **opt.step()** followed by **opt.zero_grad()**, where "opt" refers to the optimizer used, here, **optim.SGD** (**torch.optim** is a package implementing various optimization algorithm).

The activation functions tried are: sigmoid, tanh, ReLU and leaky ReLU. The performance of the FFNN with sigmoid and tanh were appreciable, with that with tanh better than with sigmoid. The use of ReLU (with Xavier initialization) or leaky ReLU (with He initialization) for the same purpose produces very bad accuracy values, and also some times "nan" value for loss (problem of exploding gradients). Thus, the two activation

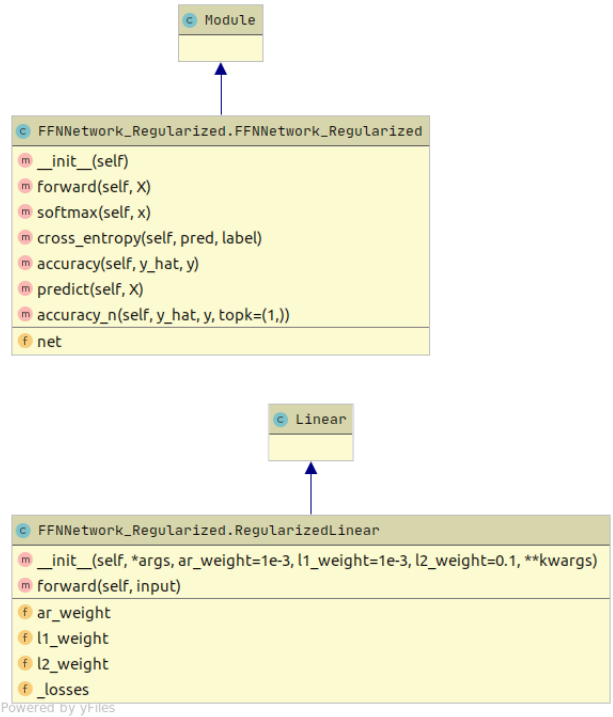


Figure 5.12: Class FFNNetowrk_Regularized

functions were not considered for further search in hyper parameter space.

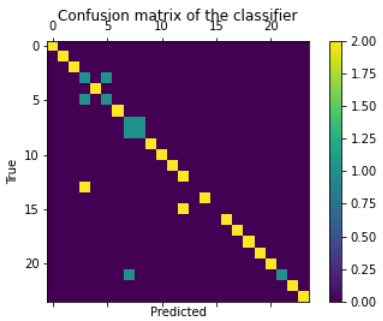


Figure 5.13: Confusion Matrix: FFNN

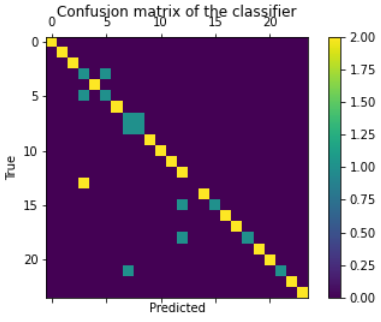


Figure 5.14: Confusion Matrix: Ensemble Classifier

In terms of performance, the accuracy values are same as that obtained

previously. Amazingly, the time for fitting the model was drastically reduces from of the order of a few minutes to a few seconds - **107.908s** to **15.089s!** (nearly 7x faster!)

5.1 Ensemble Classifier

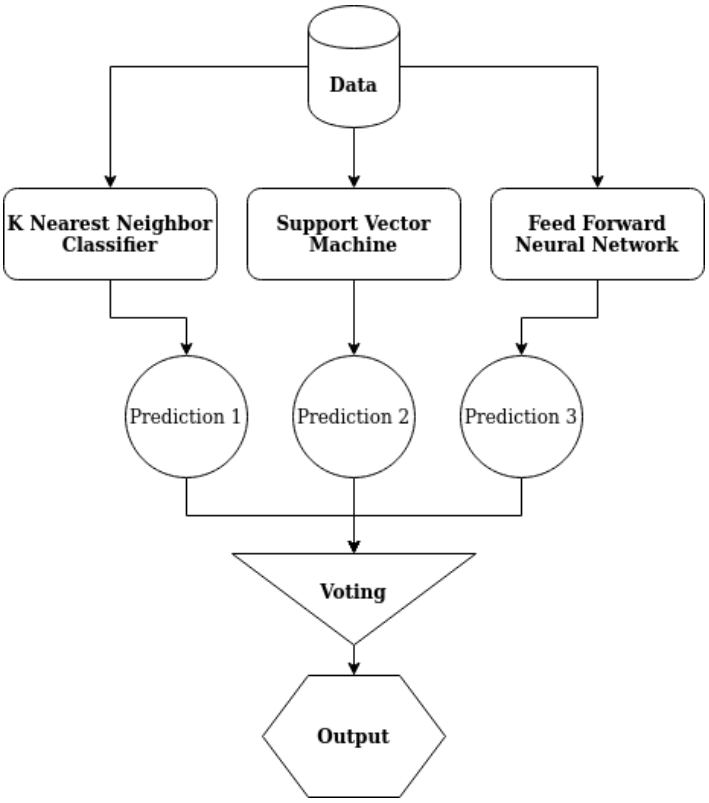


Figure 5.15: Ensemble Classifier

The above three classifier considered were further used to create an ensemble of classifier. The weightage for each classifier's prediction was given according to each individual classifier's accuracy on the test data.

5.2 Face Alignment[9]

As a general procedure in any face recognition project, effort was made for face alignment using the algorithm described in the flowchart below.[10]

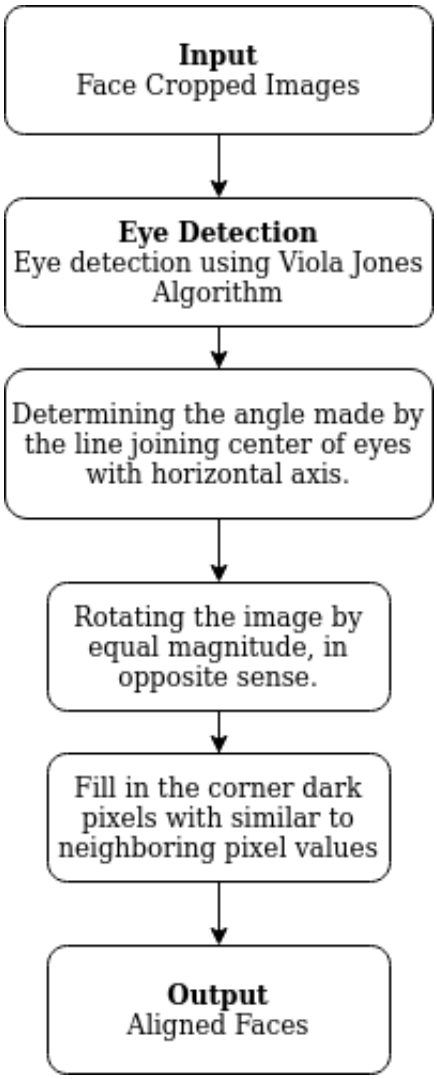


Figure 5.16: Face Alignment Flow chart

The effort went in vain as the eye detection classifier used - *haarcascade_eye.xml* - could only detect eyes properly from images which were not face cropped. In face cropped images, the classifier failed

miserable as it detected mouth (teeth), dark hair path (at corner of face cropped image), nose, and other irrelevant background portions as eyes and in some cases, did not detect any thing. A few examples of the performance of the algorithm (in spite of exhaustive hyper-parameter tuning) on the training samples. In very few images did the face alignment algorithm function efficiently.

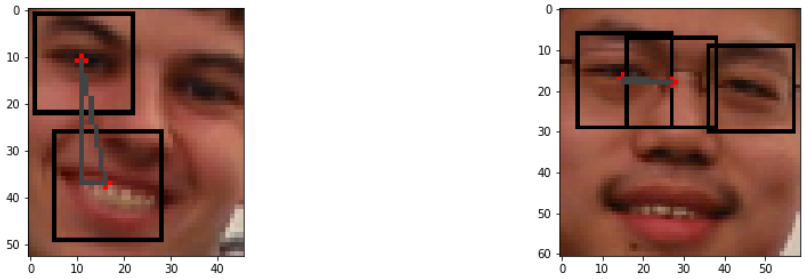


Figure 5.17: Face Alignment Results

5.3 Saving and Loading Models

The model so trained during the making of this project has been saved using PyTorch's save function as **state_dict** (using *torch.save(model.state_dict(), PATH)*). Two models have been trained and saved - one trained by looping orderly over the training data and other by looping over the training data randomly. These saved models can be loaded and either be used directly for classification purpose (on testing samples) or used for further training to improve the model further. This was achieved using PyTorch's **load_state_dict** function (i.e., using model.load_state_dict(torch.load(PATH))) [11]. For SVM, the values corresponding to optimum classifier has been listed in the sections above. These values can be plugged into the classifier direcly and used for classification purpose. Below is the comparison of all the classifiers considered above (including the ensemble classifier).

5.4 Comparison of Classifiers Performance

Along with the accuracy of the test samples based on prediction of each classifier the precision, recall and F-scores were used to compare the classifier performance. The numerical values are observed to depend on the initial random seeding of the augmented data. The best obtained model parameters of SVM and FFNN have been displayed in above relevant sections. The model file corresponding to best accuracy scores obtained using FFNN has been saved, which can be loaded and used.

Classifiers	KNN	SVM	Neural Network	Ensemble Classifier
Accuracy	i. For k=1:72.91666 ii. For k=2:50.0 iii. For k=3:50 iv. For k=4:35.41666 v. For k=5:29.1666	Training Set Score:1.000 Test Set Score:0.791667	• **Top-1** Training Accuracy- 100 Test Accuracy-79.16 • **Top-5** Training Accuracy-100 Test Accuracy-95.83 • **Top-10** Training Accuracy-100 Test Accuracy-97.916	• **Top-1** Testing Accuracy-81.25 • **Top-5** Testing Accuracy-89.5833 • **Top-10** Testing Accuracy-89.5833
Comments	Maximum Accuracy was obtained for K=1, with accuracy-72.9166. From this, we infer that test samples have high degree of similarity with the training data in the face space (as face space vectors are considered for classification)	**Best Performance Parameters:** • Best C (Degree of Tolerance)—10 • Best Kernel—RBF(Radial Basis Function) • Best Gamma (parameter of the Gaussian RBF)—0.01	• The loss was observed to decline rapidly within a few 100 epochs. After which loss nearly saturates. The model obtained by looping over training samples in perfect order produces higher accuracy than when looped over in random fashion in each epoch. (81.25% over 79.16%)	The top-1 accuracy value is observed to be determined primarily by the outputs of SVM and FFNN classifiers. While, the output of kNN influences top-5 and top-10 accuracy values.

Figure 5.18: Comparison of Classifiers

Chapter 6

Conclusion

The presented face detection and recognition system successfully detects faces from input group photos and identifies (classifies) the people with appropriate training set used. Higher the variability of the face images used for training and also higher the number of faces per person, better is the performance of the classifiers designed. Provision has been made for data augmentation in case of less number of training samples. The performance of the classifiers SVM and FFNN is appreciable. The performance of the FFNN may vary with the dataset. Thus, to account for this, a set of models with different neural network configuration, activation functions (some models having different activation functions at different layers), etc., which can be used in case the best model suggested does not provide satisfactory results. Necessary implementation of face alignment have been done, which can be tried on dataset under consideration, as the used algorithm based on haar cascade classifier for eye detection could not successfully detect eyes in all the training images in the dataset. The overall performance of the system crucially depends on the dataset used, in case the dataset contains images which can be aligned using the haar cascade eye-detection based algorithm, better performance can be expected.

Chapter 7

Future Enhancement

The work so reported can be extended by enhancing the robustness of the classifier by using datasets with more variability per face image. The existing work can be extended to be used in biometric systems such as the automatic attendence system. Further, the other methods for face detection can be used. One such example is the use of Convolutional Neural Networks, where the task of feature extraction (here, features extracted are face space vectors) can be done by the set of convolution layers and max pool or average pool layers. This can be followed by a set of fully connected layer for classification task. Some of the CNN architectures which are currently used for similar applications are: LeNet, AlexNet, GoogLe Net, Resnet, VGG-16, VGG-19, etc., by altering the output layer to output number of outputs as the number of people in the dataset. Further, with batch normalization and dropout, better architectures can be constructed for the same purpose.

Appendix A

Reconstructed Face images

Figure A.1: Reconstructed Training samples

Figure A.2: Reconstructed Testing samples

Figure A.3: Reconstructed Validation samples

Appendix A

Pseudo Code

Algorithm 1 Data Augmentation

1: **for** *iteration* $= 1, 2, \ldots, N$ **do**
2: Read training images
3: Resize and histogram equalize the image
4: Linearize the image into 1-D array form and save to training set
5: Add white noise to and rotate and fill the copied image ($-5°$,$5°$)
6: Histogram equalize and append image to training set
7: **end for**

Algorithm 2 Principal Component Analysis

1: Determine U,S,V matrices by SVD of mean subtracted images matrix
2: The first n rows of V corresponds to the Principal Components (PCs)
3: Map each image vector to face space vector using PC matrix

Algorithm 3 Face Detection

1: **for** *iteration* $= 1, 2, \ldots, \hat{N}$ **do**
2: Read in the testing (group) photo
3: Detect faces using OpenCV's haar cascade classifiers for face detection
4: Crop and resize detected frontal faces similar to training images
5: Histogram equalize and map linearized images to face space.
6: **end for**

Algorithm 4 Classification using kNN

1: Use Euclidean distance for kNN algorithm for required k values.
2: Compute accuracy scores and confusion matrix.

Algorithm 5 Classification using SVM

1: Search through the space of parameters of SVM (kernel, C, γ) for most optimal SVM based classifier and classify test samples.
2: Compute accuracy scores and confusion matrix.

Algorithm 6 Classification using FFNN

1: Map all data to torch tensor and design a neural network.
2: Port all data and network class to Nvidia GPU (if available)
3: **for** $epoch = 1, 2, \ldots, Max\_Epochs$ **do**
4: Train FFNN and compute *Cross Entropy Loss* with L2 regularization
5: Using Stochastic GD, back propagate and update weights and biases.
6: **end for**
7: Use trained model for face identification on the test samples.
8: Compute top-1,5,10 accuracy values and confusion matrix

Algorithm 7 Classification using Ensemble Classifier

1: Port predicted classes from FFNN to *numpy nd-array* from *torch.tensor*
2: One hot encode all the outputs (predictions) from each classifiers
3: Assign weightage to each classifier's prediction proportional to it's individual (top-1) accuracy on the test set and determine voted prediction
4: Determine top-1,5,10 accuracy values and confusion matrix

Bibliography

[1] *https://github.com/parulnith/Face-Detection-in-Python-using-OpenCV/blob/master/Face*

[2] *https://www.datacamp.com/community/tutorials/face-detection-python-opencv*

[3] *https://arxiv.org/pdf/1901.02452.pdf*

[4] *http://cmusatyalab.github.io/openface/*

[5] *http://www.scholarpedia.org/article/K-nearest_neighbor*

[6] *https://towardsdatascience.com/a-friendly-introduction-to-support-vector-machines-svm-925b68c5a079*

[7] *https://mlfromscratch.com/neural-networks-explained/*

[8] *https://pytorch.org/docs/stable/autograd.html*

[9] *https://github.com/contail/Face-Alignment-with-OpenCV-and-Python/blob/master/align_faces.py*

[10] *https://sefiks.com/2020/02/23/face-alignment-for-face-recognition-in-python-within-opencv/*

[11] *https://pytorch.org/tutorials/beginner/saving_loading_models.html*