San Jose State University

SJSU ScholarWorks

Master's Theses Master's Theses and Graduate Research

Fall 2020

Adaptive Learning Technique For Facial Recognition

Rachana Dineshkumar Bumb

San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Recommended Citation

Bumb, Rachana Dineshkumar, "Adaptive Learning Technique For Facial Recognition" (2020). *Master's Theses*. 5138.

DOI: https://doi.org/10.31979/etd.x3ps-ftpm

https://scholarworks.sjsu.edu/etd_theses/5138

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

ADAPTIVE LEARNING TECHNIQUE FOR FACIAL RECOGNITION

A Thesis

Presented to

The Faculty of the Department of Computer Engineering

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Rachana Dineshkumar Bumb

December 2020

© 2020

Rachana Dineshkumar Bumb

ALL RIGHTS RESERVED

The Designated Thesis Committee Approves the Thesis Titled

ADAPTIVE LEARNING TECHNIQUE FOR FACIAL RECOGNITION

by

Rachana Dineshkumar Bumb

APPROVED FOR THE DEPARTMENT OF COMPUTER ENGINEERING

SAN JOSÉ STATE UNIVERSITY

December 2020

Dr. Harry Li, Ph.D.	Department of Computer Engineering
Sirkeci, Birsen, Ph.D.	Department of Electrical Engineering
Younghee Park, Ph.D.	Department of Computer Engineering

ABSTRACT

ADAPTIVE LEARNING TECHNIQUE FOR FACIAL RECOGNITION

by Rachana Dineshkumar Bumb

This research describes the adaptive learning technique for facial recognition. It is a common practice in convolutional neural network(CNN) based facial recognition to save its trained result on a large dataset and then load and apply it to ongoing facial recognition tasks. This generally used method lacks adaptation, and the ongoing evolution of new knowledge poses a key technical challenge. In this research, we propose a continued learning technique to incorporate new knowledge derived in each facial recognition process. A positive recognition with confidence score is assigned, and then the image associated with this confidence is added to the image dataset for ongoing training. Pre-trained CNN on a similar small dataset serves as the starting point for this ongoing training technique, which leads to a significant reduction in the training time and enhancement of the recognition rate. This research is inspired by the evolutionary adaptive learning talk given by Dr. Harry Li in the 2019 SiliconValley AI Event. This research is conducted to provide proof-of-concept groundwork to demonstrate the feasibility of continued learning and adaptation while executing the FaceNet/ResNet facial recognition algorithm. In this research, the proof-of-the-concept algorithm is demonstrated on a simple feed-forward neural network, then tested with an adaptive face technique to demonstrate the learning acceleration from the adaptive process. Experiments confirm the adaptive learning on FaceNet and ResNet as per the proof-of-the-concept by reducing the number of epochs required to reach a convergence point by approximately 50%. This states the use of adaptive learning techniques in software that require identifying images of aging people concerning time.

ACKNOWLEDGMENTS

First and foremost, praises and thanks to God, for his showers of blessings throughout my research journey. I would like to express my deep and sincere gratitude to my research advisor Professor Harry Li, for providing me invaluable guidance and pushing me to complete this research. His vision and motivation have deeply inspired me. I would also like to thank my thesis committee members, Professor Birsen Sirkeci and Professor Kaikai Liu for taking time out of their busy schedule to be on the committee and help provide any helpful suggestions for the paper. I would like to thank Minh Duc Ong and the CTI One Corporation team for helping me create the foundation of this research by having timely discussions and sharing GitHub repositories. I would also like to thank all my fellow students and friends, both undergraduate and graduate, for encouraging me to push on forward when challenges came up.

Last, but not least, I would like to thank my family and Niraj. Without their support, I would have never been able to complete this thesis to pursue my dreams.

TABLE OF CONTENTS

List of Tables viii

List of Figures x

1 Introduction 1
- 1.1 Problem Statement 1
- 1.2 Research Objectives 2

2 Literature Survey 5
- 2.1 Facial Recognition 5
  - 2.1.1 Feature Extraction 6
  - 2.1.2 Decision Making by Cluster Analysis 8
- 2.2 FaceNet 9
- 2.3 ResNet 11

3 Methodology 13
- 3.1 FaceNet Architecture Design 13
- 3.2 Triplet Loss Function 14
- 3.3 Training FaceNet with Triplet Loss and its Convergence 19

4 Implementation 24
- 4.1 Feasibility Study with Adaptive Feed Forward Network 24
- 4.2 David Sandberg - Baseline Implementation 34
- 4.3 Adaptive Learning for FaceNet 36

5 Comparative Study of Adaptive ResNet 44
- 5.1 Discussion of ResNet 44
- 5.2 Results of Adaptive ResNet Training 47

6 Future Work 50

7 Conclusions 52

Literature Cited 53

Appendix A: Max-pooling in FcaeNet 57
- A.1 Convolutional, max-pooling and L2 layer in FaceNet 57

Appendix B: Adaptive Architecture 58
- B.1 Code for Simple Feed Forward Neural Network 58

B.2 Code for Adaptive Simple Feed Forward Neural Network 64
B.3 Code for FaceNet 69
B.4 Code for Adaptive FaceNet 71

LIST OF TABLES

Table 1. Range of Landmark Points for Different Facial Parts 6

Table 2. Sample Embeddings of test-image 2, test-image 3, test-image 4 7

Table 3. Dataset for Feasibility Study of Feed Forward Network Training without Prior Knowledge 25

Table 4. Dataset for Feasibility Study of Feed Forward Network Training with Prior Knowledge 28

Table 5. Architecture of FaceNet Code 34

Table 6. Our Customer Designed Dataset for FaceNet Training without Prior Knowledge. 38

Table 7. Our Customer Designed Dataset for FaceNet Training with Prior Knowledge. 39

Table 8. Updated FaceNet Parameters for Adaptation 39

Table 9. Our Customer Designed Dataset for ResNet Training without Prior Knowledge 45

Table 10. Our Customer Designed Dataset for ResNet Training with Prior Knowledge 46

LIST OF FIGURES

Fig. 1. Detection, extraction, classification: facial recognition pipeline 5

Fig. 2. Two important steps of facial recognition: feature extraction and clustering.......... 5

Fig. 3. Test-image 1 to plot facial landmarks.......... 6

Fig. 4. Sixty-Eight facial landmarks plotted on a sample test-image 2......... 7

Fig. 5. Sixty-Eight facial landmarks plotted on a real-time sample test-images. 7

Fig. 6. Four types of cluster analysis (modified from by Bijl and Erik [11]). . 8

Fig. 7. FaceNet input: face image (test_image_5), FaceNet output: embedding vector.......... 9

Fig. 8. Two dimensional plot(dimensional reduction for illustration purpose) of faces based on clustering.......... 10

Fig. 9. Skip Connection in ResNet.......... 11

Fig. 10. Block diagram of FaceNet by Schroff et al. [1].......... 13

Fig. 11. Triplet loss block diagram of FaceNetby Schroff et al. [1]............ 14

Fig. 12. Embeddings for triplet loss.......... 14

Fig. 13. Learning from Triplets.......... 15

Fig. 14. Flowchart representation of FaceNet training.......... 22

Fig. 15. 2D plot of dataset for feed-forward network.(Red circles: points of class 1, Blue circles: points of class 0).......... 26

Fig. 16. Graph of pre-adaptive learning loss in feed-forward network. 27

Fig. 17. 2D plot of dataset for feed-forward network with additional data...... 28

Fig. 18. Graph of post-adaptive learning loss in feed-forward network. 29

Fig. 19. Comparison of pre-adaptive and post-adaptive learning losses in feed-forward network.......... 30

Fig. 20. Loss vs Epochs for pre-adaptive and post-adaptive simple feed-forward network. 31

Fig. 21. Loss vs Epochs for pre-adaptive and post-adaptive simple feed-forward network (contd.). 32

Fig. 22. Loss vs Epochs for pre-adaptive and post-adaptive simple feed-forward network(contd.). 33

Fig. 23. Architecture of pilot test platform for adaptive learning. 33

Fig. 24. Code architecture: David Sandberg's FaceNet github repo. 35

Fig. 25. Code architecture: David Sandberg's FaceNet github repo(contd.). 35

Fig. 26. Images of number of persons in the dataset to train FaceNet. 36

Fig. 27. Our customer designed small pre-adaptive FaceNet dataset. 37

Fig. 28. Our customer designed small adaptive FaceNet dataset. 38

Fig. 29. Graph of loss vs epochs of Adaptive FaceNet. 40

Fig. 30. Pre-adaptive and post-adaptive FaceNet learning loss. 41

Fig. 31. Pre-adaptive and post-adaptive FaceNet learning loss(contd.). 42

Fig. 32. Our customer designed small pre-adaptive ResNet dataset. 45

Fig. 33. Our customer designed small adaptive ResNet dataset. 46

Fig. 34. Graph of Adaptive ResNet learning loss. 47

Fig. 35. Pre-adaptive and Post-adaptive ResNet learning loss. 48

1 INTRODUCTION

1.1 Problem Statement

Face recognition and face detection are two of the most discussed problems/ use-cases in the fields of computer vision and deep learning [1], [2], [3]. These tasks are achieved by manually extracting features from faces and then computing the patterns such as eye, nose, lips, etc. Deep learning has been used to automate this task and has been successful largely; it faces scaling with great accuracy. For example, if a softmax layer is used to classify images into respective classes, the trained neural network has to be retrained each time that a new face is added to an existing database, and hence, this becomes a huge bottleneck. The FaceNet paper by Schroff et al. [1] significantly addresses this issue.

However, lack of adaptation is one of the problems faced by various models, including FaceNet by Schroff et al. [1] and ResNet by He et al. [2]. Researchers at Google developed Facenet, and researchers at Microsoft developed ResNet. David Sandberg's implementation of FaceNet architecture acts as the baseline implementation for this research. He provided an implementation of FaceNet with triplet loss for training. His implementation provides two pre-trained models by Sandberg [4] on FaceNet to be used as part of transfer learning. Transfer learning uses the knowledge learned for some tasks to solve another similar task by Pan et al. [5]. Well-trained and well-constructed neural networks trained over large datasets are loaded to boost the performance of training the network on a new dataset of comparatively small size. Using pre-trained models, the model's weights and architecture can be directly used to apply the learning to the new dataset for training on the same neural network architecture. One has to be extremely careful while using a pre-trained model. If the new problem statement is completely different from one of the pre-trained models, the learning is not useful and hence provide wrong predictions and be inaccurate. To avoid this, the adaptive learning technique is

proposed in this research by confirming the results obtained on the proof-of-the-concept simple feed-forward network and then testing it on FaceNet and ResNet.

If a neural network has to be trained from scratch on a large dataset, it may take a long time to complete a big dataset training. For FaceNet, it takes hours or days to train the model from scratch on datasets such as CASIA-WebFace by Yi et al. [6], or VGGFace2 by Cao et al. [7] because it has 7.5M parameters with 1.6B FLOPS, described by Schroff et al. [1]. This is not good enough to train a neural network from scratch each time for a new dataset. If there is a technique that can save the model and checkpoints on a small trained dataset and then adapt this learning to the same dataset with some additional data to be trained on the same network, the amount of time required for training would be reduced to a large extent and also the performance of the model increases.

1.2 Research Objectives

The first objective of this thesis is to study and investigate FaceNet implementation for adaptive development. Two different pre-trained models, trained on CASIA-WebFace by Yi et al. [6] and VGGFace2 by Cao et al. [7] datasets, are made available on the GitHub repository of David Sandberg for FaceNet implementation by Sandberg [4]. Although these pre-trained models can be used to retrain the FaceNet architecture on custom datasets, they lack the feature of adaptation for ongoing training with the addition of new faces to the dataset. This research studies and investigates the FaceNet architecture and its implementation in detail to understand the parameters used in training.

The second objective of this research is to design and develop an adaptive learning algorithm to demonstrate the feasibility of adaptive learning on a simple feed-forward network. A simple feed-forward neural network is considered the foundation for testing the proposed research's feasibility for ongoing learning with an increasing number of data points in the dataset.

The third objective of this research is to implement, test, and verify the proposed adaptive algorithm on FaceNet and also compare the results with ResNet. These facial recognition algorithms are considered two of the best algorithms for facial recognition. Hence a comparative study of adaptive FaceNet with ResNet becomes necessary for the scope of the proposed research.

While transfer learning is a wonderful technique because pre-trained models can be easily downloaded, some compelling reasons to train a neural network from scratch are:

1) If the images are preprocessed properly, the network trained on custom data should be able to classify those images.
2) Training a network from scratch gives better accuracy than a pre-trained model on unique training data.
3) Tuning of parameters as required.
4) Fragile checkpoints on pre-trained models by Corporation [8].

However, it is still impossible to train the entire network from scratch each time the new data are available. To overcome this again, a new adaptive learning technique is proposed in this thesis. A neural network is trained from scratch on a small dataset. The weights and checkpoints of this network/model are saved and restored when some new data points are available and added to the existing dataset. Consider this case for facial recognition algorithms. The dataset here consists of images of different people. Once the saved weights and checkpoints are restored, the model is again trained. This proposes an ongoing technique to train a neural network to incorporate new data. A positive confidence score is assigned to the new data and added to the ongoing training process dataset. Already trained networks serve as the starting point for this ongoing training phenomenon, which leads to a significant reduction in training time. Also, the network converges at a significantly earlier stage compared to training a neural network from scratch. This also enhances the recognition rate.

The proof-of-the-concept for adaptive learning is provided on a simple feed-forward network in the following sections. The results of this experiment confirm the proposed adaptive learning technique.

2 LITERATURE SURVEY

2.1 Facial Recognition

Recognizing or detecting human faces by computers has been under process and investigation for a long time. There is widespread use of facial recognition technology in various domains. Face recognition has a long pipeline of processes. Fig. 1 depicts the high-level pipeline of face recognition technology (modified from Dulčić [3]).

Fig. 1. Detection, extraction, classification: facial recognition pipeline

First, a face is localized and detected in an input image by a face detection algorithm. The output face with a bounding box from the face detection algorithm is then passed to a feature extraction algorithm. Based on these extracted features, faces are passed to the face classification algorithm to be assigned with a unique ID resulting in clusters with images of the same person in one cluster. The extracted features are important in recognizing faces. This phase of facial recognition is a two-step process, as shown in Fig. 2.

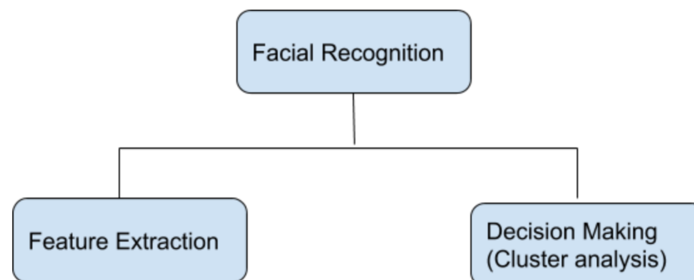

Fig. 2. Two important steps of facial recognition: feature extraction and clustering.

2.1.1 Feature Extraction

Simply detecting a face in an image is not helpful in facial recognition. Additional information about the face is required to take the recognition process further. Taking this into consideration, facial landmarks are used to extract more information about the face. Facial landmarks are used to extract salient features from different facial parts, as shown in Table 1.

Table 1
Range of Landmark Points for Different Facial Parts

Face part	**Landmark points**
Jaw	0–16
Right Brow	17-21
Left Brow	22–26
Nose	27–35
Right Eye	36–41
Left Eye	42–47
Mouth	48–60
Lips	61–67

With the help of extracted facial landmarks, faces can be aligned in an image. In general, there are 68 facial features/landmarks that can be extracted and plotted on a face to gain important information about a face. There are various facial landmark detectors, and all of them similarly mark the landmark points. Fig. 3, Fig. 4 and Fig. 5 contains 7 images of 7 different people to extract and plot the facial features composed of 68 landmark points.

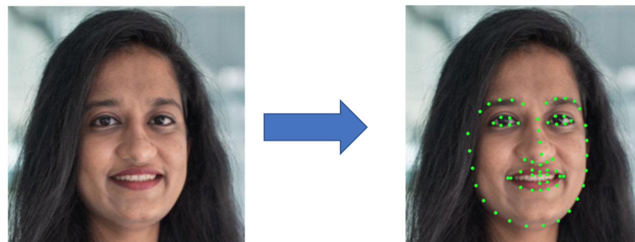

Fig. 3. Test-image 1 to plot facial landmarks.

Fig. 4. Sixty-Eight facial landmarks plotted on a sample test-image 2.

Fig. 5. Sixty-Eight facial landmarks plotted on a real-time sample test-images.

These facial features are extracted by using landmark detection DLib algorithms[32]. The facial landmark points on test_images are mapped according to Fig. 3. Table 2 describes the facial features extracted from test-images and, these feature values form embeddings.

Table 2
Sample Embeddings of test-image 2, test-image 3, test-image 4

Landmark	**Landmark points**	**ID**	**x_start**	**y_start**	**Arc Length**
Left_eye	37-42	test-image 4	128	307	10.81
Left_eye	37-42	test-image 3	226	425	13.85
Left_eye	37-42	test-image 2	483	240	9.64
Nose	28-31	test-image 2	556	266	43

Feature extraction plays an important role in FaceNet. We will look into this in the next section. These extracted feature points are then used as feature vectors in vector space to make decisions.

2.1.2 Decision Making by Cluster Analysis

This is the third stage in facial recognition. It is the decision making phase, wherein decisions are made to classify an image in its respective class/cluster. This decision is based on the vector distances between a new face and already known faces. Cluster analysis can be performed in various ways, as shown in Fig. 6.

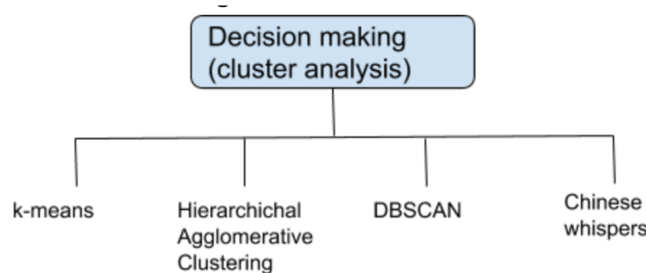

Fig. 6. Four types of cluster analysis (modified from by Bijl and Erik [11]).

One of them is by using the Softmax classifier by Bijl, and Erik [11]. Softmax classifier classifies a face based on the embeddings of the face. The k-nearest neighbor can also be used for clustering faces by Bijl and Erik [11]. Cluster analysis is generally unsupervised. When a neural network generates 128-dimensional feature vectors/embeddings [25], these embeddings are then passed to the face cluster algorithm. DBSCAN cluster analysis algorithm is usually used for programs that make use of a small dataset. In contrast, the Chinese whispers algorithm is used when there is a large dataset to be trained for facial recognition by Bijl and Erik [11]. In DBSCAN, points or embeddings which are closely related in an N-dimensional space are clustered together. In FaceNet, images are represented in 128-bytes embeddings. These embeddings themselves

lead to the formation of clustering, usually called agglomerative clustering of faces by Schroff et al. [1].

2.2 FaceNet

As discussed, one of the most exciting features of deep learning is facial recognition. There is an immense rise in the adoption of facial recognition in numerous domains. Various facial recognition architectures have been developed over the years. However, the one worth commenting and experimenting about is FaceNet. FaceNet is a neural network architecture developed by researchers at Google that performs face recognition, clustering, and verification tasks. Deep CNN plus triplet loss forms the basis of FaceNet by Schroff et al. [1]. The input to FaceNet is an image of a person, and the output of FaceNet is a 128-dimensional embedding that represents the most important features of that particular input face as shown in Fig. 7 (modified from Schroff et al. [1]). These embeddings are nothing but vectors. Almost all of the important information about the face in an image is embedded in this vector. In short, the image of a person or the face in an image is compressed into a 128-dimensional vector. Moreover, embeddings coming from the same person are similar. High-dimensional data like images are now compressed into low-dimensional data called embeddings.

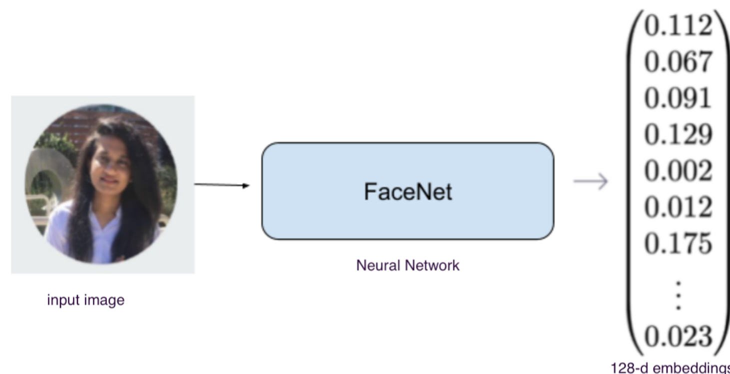

Fig. 7. FaceNet input: face image (test_image_5), FaceNet output: embedding vector.

In the last section, we have seen that there are 68 facial landmarks. Moreover, the dataset used to train FaceNet contains images that may have turned in different directions for the machine to consider them as different images. These images/faces can be aligned using 68 point facial landmarks to solve this problem of alignment. Rather than using any bottleneck layer, FaceNet learns to map the images and create embeddings. This marks the difference between FaceNet and other facial recognition architectures. So, FaceNet architecture creates embeddings of facial images based on 128 dimensions. How are these embeddings useful in FaceNet? These embeddings can be represented on a coordinate system, which means with known embeddings, the face of an image/person can be plotted on the x-y axis, with similar embeddings into the same cluster. Because 128-dimensional vector/embeddings practically cannot be plotted, converting this to a 2-dimensional vector and then plotting it on a graph is fairly easy as shown in Fig. 8.

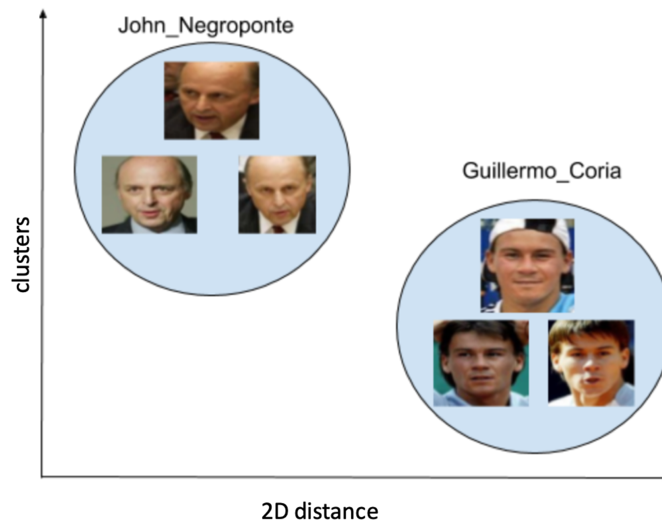

Fig. 8. Two dimensional plot(dimensional reduction for illustration purpose) of faces based on clustering.

FaceNet provides unified embedding for facial recognition and clustering. Each face in the dataset is mapped into a euclidean space matching its similarity for facial

recognition and clustering. Once the embeddings are created and stored, they are used to perform facial recognition and verification in a standard manner by defining a threshold value. “So, the most important thing to note here is that FaceNet does not define any new algorithm to carry out the aforementioned tasks, rather it just creates the embeddings, which can be directly used for face recognition, verification, and clustering.” by Schroff et al. [1]. Another important phenomenon of FaceNet is the triplet loss function by Schroff et al. [1]. This will be explained in the next section.

2.3 ResNet

ResNet refers to the Residual Neural Network developed by researchers at Microsoft by He et al. [2]. An important concept called “skip connections” by He et al. [2] identity mapping is introduced by this architecture as shown in Fig. 9.

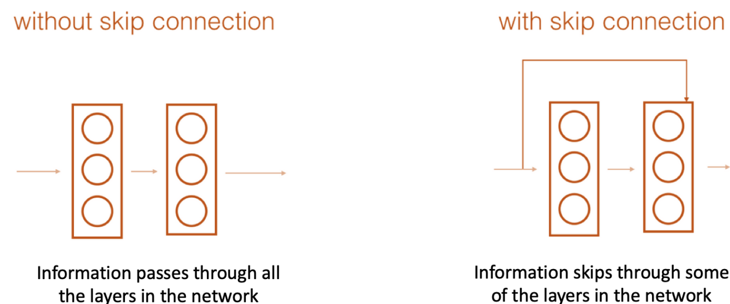

Fig. 9. Skip Connection in ResNet.

This identity mapping does not contain any parameters. Instead, it just adds the output from the previous layer to the layer ahead. The important point is, the output of one layer serves as the input to the next layer, and eventually, somewhere in the middle, the output of one layer as an input to layer over 2-3 hops. This means that a few layers in a deep network can be skipped using skip connections, which are usually referred to as a residual block. By looking at backpropagation and vanishing gradient problems, it is clear that

skip connections can solve this issue for deeper networks, and the initial layers of the network can learn/train as fast as the last layers of the network by Sahoo [12].

3 METHODOLOGY

3.1 FaceNet Architecture Design

FaceNet is a state-of-art neural network architecture for facial recognition, clustering, and verification. This neural network is 22 layers deep by Schroff et al. [1] and trains itself to form an output to be in the form of 128-dimensional embeddings. The triplet loss function is used in the last layer of the network. The architectural block diagram of FaceNet is shown in Fig. 10.

Fig. 10. Block diagram of FaceNet by Schroff et al. [1]

The deep architecture in figure above is from GoogleNet architecture by Szegedy et al. [13] which has many revisions. This deep neural network has some ground-breaking features:

- 22-layers of deep network
- Efficient
- Faster computational power and 2 times less computational cost compared to AlexNet
- Low power consumption and low memory usage
- Although network is large, parameters are 12 times less than AlexNet by Deore [14]

The reduction in parameters and faster computational power is the inspiration behind GoogleNet architecture. This eventually got transferred as the ”Inception module.” The inception-v1 module has different versions: Inception-v2 and Inception-v3 (Factorization, BatchNormalization, Label smoothing) by Szegedy [15], Inception-v4 and Inception-ResNet-v1 by Moindrot [16]. For the ResNet version, a residual connection is

added, replacing the inception module's pooling layer. Specifically, David Sandberg's implementation of FaceNet uses the 'Inception-ResNet-v1' version by Sandberg [9].

3.2 Triplet Loss Function

The Google team introduced triplet loss in their 2015 paper titled "FaceNet: A Unified Embedding for Face Recognition and Clustering." by Schroff et al. [1]. In Fig. 11 below, the box selected in the red outline acts as the triplet loss function for FaceNet architecture. Embeddings, or in general terms called feature vectors of images, act as input to the loss function. Fig. 12 describes the inception module's position in the architecture to generate embeddings from an input image. FaceNet is a convolutional neural network that forms embeddings of an input image into a 128-dimensional vector encoding.

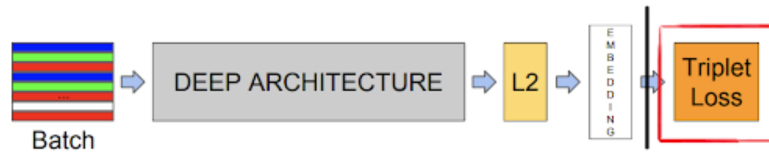

Fig. 11. Triplet loss block diagram of FaceNetby Schroff et al. [1]

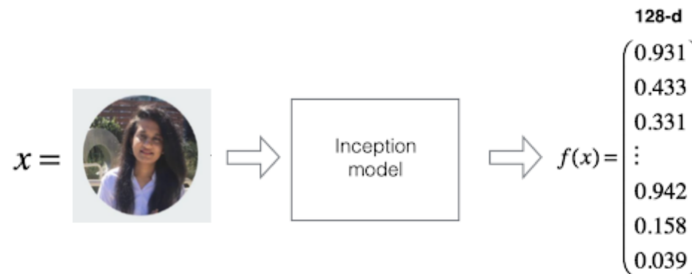

Fig. 12. Embeddings for triplet loss.

$$D = \sum_{i=1}^{M} ||f(x_i^a) - f(x_i^p)||_2^2 \tag{1}$$

Here in equation (1), f represents a function operated on image x_i. These functions act as embeddings of anchor and positive images. Cost function of triplet loss looks as [20]:

$$\text{Cost Function} = \sum_{i}^{N} \text{Triple Loss Function} + L_2 \ \text{Regularization} \tag{2}$$

The deciding factors to calculate cost function in equation (2) are the face triplets and L2 regularization. Then it tries to minimize the distance between an anchor and a positive sample of the same identity and maximizes the distance between the anchor and a negative sample of a different identity by Schroff et al. [1]. Fig. 13(modified from Works [18]) shows the learning and adaptation of learning loss to form proper triplets.

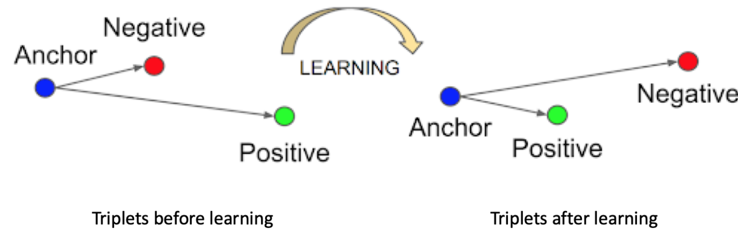

Fig. 13. Learning from Triplets.

The triplet loss function takes face embeddings of three images: anchor, positive and negative images. An anchor image and a positive image are the same person's images, whereas the negative image is a different person's image. To recognize a person in an unseen image, firstly, we need to calculate its embeddings. We calculate the distance between known embeddings and the embedding of an unseen image based on these embeddings. If this unseen face embedding is close to one of the known embeddings, we can conclude that the image belongs to this particular person. The FaceNet model aims to generate encodings/embeddings. There is less distance between the encoding of the images belonging to the same person and comparatively large distance between the encoding of the images belonging to different people. To accomplish this goal, triplet loss

tries to minimize the distance between similar images, and at the same time, tries to maximize the distance between different images.

To achieve this, while training a model for classification, weights of parameters are modified to minimize the loss function. This is a feature vector, highly referred to as ‘face embeddings,’ which are related by Euclidian distance, such that faces belonging to the same class produce embeddings with small distances. This triplet loss approach is used in FaceNet systems that achieve state-of-the-art results on benchmark face recognition datasets. The triplets that are used to train the model are carefully chosen to be hard triplets. Because triplets that are considered easy result in a small loss, which is good but not effective at training the model efficiently. At the same time, hard triplets support changes to the model and the face embeddings to be predicted.

Objective:

$$Loss = \sum_{i=1}^{N} \left[\|f_i^a - f_i^p\|_2^2 - \|f_i^a - f_i^n\|_2^2 + \alpha \right]_+ \tag{3}$$

The objective of the FaceNet neural network is to minimize equation (3).

$$\|f(x_i^a) - f(x_i^p)\|_2^2 \alpha < \|f(x_i^a) - f(x_i^n)\|_2^2 \tag{4}$$

More formally, for an embedding function f(x) $\in Rd^2$ that embeds input data x into a d-dimensional vector, we want equation (4) for all N possible triplets of x_i^a, x_i^p, x_i^n:

- f(x) takes x as an input and returns a 128-dimensional vector w.
- i denotes i’th input.
- Subscript a denotes Anchor image, p denotes Positive image, n denotes Negative image.

The objective is to minimize the above equation, which implicitly means:-

Minimizing first term $\rightarrow$ distance between Anchor and Positive image.

Maximizing second term $\rightarrow$ distance between Anchor and Negative image.

The third term is a bias which acts as the threshold.

$$\forall (f(x_i^a), f(x_i^p), f(x_i^n)) \in T \tag{5}$$

For any triplet as in equation (5), embedding feature vectors from anchor images, embedding feature vectors from positive images, and embedding feature vectors from negative images, equation (2) holds good. As long as this combination of triplets satisfies equation (2), it belongs to a meaningful dataset for training. The α symbol stands for margin to ensure that the model does not make the embeddings f(x_i^a), f(x_i^p), and f(x_i^n) equal to each other in order to satisfy the above inequality.

$$\sum_i^N \left[\|f(x_i^a) - f(x_i^p)\|_2^2 - \|f(x_i^a) - f(x_i^n)\|_2^2 + \alpha \right]_+ \tag{6}$$

This leads to the following loss function over the N possible triplets as in equation (6) The $[x]^+$ operator stands for max(0,x).

If,

$$\|f(x_i^a) - f(x_i^p)\|_2^2 - \|f(x_i^a) - f(x_i^n)\|_2^2 < 0 \tag{7}$$

From equation (7), the distance of a negative image from an anchor image is very large as compared to the distance of a positive image from an anchor image, which is not a suitable combination of triplet for training.

And if,

$$\|f(x_i^a) - f(x_i^p)\|_2^2 - \|f(x_i^a) - f(x_i^n)\|_2^2 > 0 \tag{8}$$

From equation (8), the distance of a positive image from an anchor image is very large as compared to the distance of a negative image from an anchor image, which is

also not a suitable combination of triplet for training. Therefore, this states that the triplets are chosen in such a way that it forces the loss function towards zero(0).

Definition 1: P classes objective function by Li [19]:

$$D = \alpha_1 \sum_{j=1}^{M_1} \| f_t rue(x_j) - f(x_j) \|_2^2 + \\ \alpha_2 \sum_{j=M_1+1}^{M_2} \| f_t rue(x_j) - f(x_j) \|_2^2 + ... \\ ... + \alpha_p \sum_{j=M_p-1+1}^{M_p} \| f_t rue(x_j) - f(x_j) \|_2^2 \tag{9}$$

where

$$\sum_{i=1}^{p} \alpha_i = \alpha_2 + ... + \alpha_p = 1 \tag{10}$$

Lemma 1: P classes recognition problem can be decomposed as two classes detection at a time, by grouping P-1 classes into one combined negative class by Li [19].

Property 1: Triplet loss functions for training based on the equally likely assumptions. For P classes, if the detection task is to recognize an image, based on above lemma, then its loss function is defined as equation (3) by. Li [19].

Let P=2, so definition 1 becomes:

$$D = \alpha_1 \sum_{j=1}^{M_1} \| f_t rue(x_j) - f(x_j) \|_2^2 + \\ \alpha_2 \sum_{j=M_1+1}^{M_2} \| f_t rue(x_j) - f(x_j) \|_2^2 \tag{11}$$

where

$$\alpha_1 = \alpha_2 = \frac{1}{2} \tag{12}$$

Equation (12) proves property 1. Alpha is added in the equation to further minimize the distance of the positive class from the anchor class.

3.3 Training FaceNet with Triplet Loss and its Convergence

During FaceNet training, the deep architecture extracts facial features. These features are then converted into 128-dimensional embeddings. To keep positives far apart from negatives, a margin represented by alpha is added to the positive value, and this is how the positives are moved further apart from negatives. Triplet selection: This forms the basis for the training of FaceNet architecture. The main intention behind training the FaceNet is selecting (anchor, positive) and (anchor, negative) pairs of images. If these pairs are selected randomly, the loss function would be satisfied to zero very easily, but the network will not learn much from it. Additionally, the gradient descent may converge to the wrong weights by Deore [17]. Triplets in FaceNet are selected by online mining by Schroff et al. [1] There are three categories of triplets by Moindrot [16]:

- easy triplets: triplets which have a loss of 0, because
  Distance of (anchor,positive)+alpha(margin)¡distance of(aanchor,negative)
- hard triplets: triplets where the negative is closer to the anchor than the positive, i.e.
  Distance of (anchor,negative) ¡ distance(anchor,positive)
- semi-hard triplets: triplets where the negative is not closer to the anchor than the positive, but which still have positive loss:
  Distance of (anchor,positive)¡distance of (anchor,negative)¡distance of (anchor,positive)+alpha(margin)

These definitions are highly dependent on the position/distance of the negative from anchor and positive. Thus, these three categories can be extended to easy negative triplets, hard negative triplets, and semi-hard negatives as well. In the original paper of FaceNet, authors pick random semi-hard negative triplets for each pair of anchor and positive images in a triplet. Then the network is trained on these triplets by Schroff et al. [1].

The FaceNet paper suggests that using extremely hard triplets to train the model may result in early convergence but may cause a broken model. To avoid this, using semi-hard triplets becomes the preferred option. This is achieved by using a reasonably small mini-batch of images. In the original paper, the author suggests using 40 faces in a mini-batch. The condition required to satisfy the ‘semi-hard’ triplet selection is:

$$\text{Distance of (Anchor,Positive)} \approx \text{Distance of (Anchor,Negative)}$$

Algorithm 1 describes the selction of anchor image and formation of minibatches in original FaceNet paper. FaceNet requires many images to train itself by Deore [17]. To

Algorithm 1 Anchor selection algorithm for Facenet

input: anchor image x_i^a , positive image x_i^p, negative image x_i^n
1: **repeat**
2: generate mini-batch of 40 images per identity/person.
3: randomly sample x_i^n for each mini-batch.
4: **for** each mini-batch **do**
5: select all (x_i^a, x_i^p) pairs within the batch.
6: **for** each (x_i^a, x_i^p) **do**
7: select x_i^n from the mini-batch
8: **until** all triplets are formed
return (x_i^a, x_i^p, x_i^n)

keep the experiment simple and understandable, assume that there are just a couple of images of 3 persons. Similar logic can be applied if there are more than 3 numbers of people. At first, FaceNet generates vectors/embeddings for each image of all the persons. Moreover, hence, these images are randomly scattered on the coordinate system. Algorithm 2 describes the modified adaptive FaceNet training pseudocode based on this research.

Fig. 14 represents the flowchart of FaceNet training process.

Learning/ training process of FaceNet starts this way:

1) Randomly initialize the FaceNet parameteres.

Algorithm 2 Adaptive FaceNet algorithm

input: 10 images per person (3 persons)

1: **repeat**
2: randomly initialize the FaceNet parameteres or load the saved checkpoint.
3: **for** each image **do**
4: preprocess images to 160x160
5: generate random vectors/embeddings 128-d
6: form triplets:
7: randomly select anchor image
8: select negative and positive image
9: adjust facenet parameteres:

$$f(x_i) = [128d]$$

10: **for** each triplet **do**
11: triplet loss function:

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

12: **for** fast convergence **do**
13: select triplets:
14:

$$HardPositive : Argmax \left\| f(x_i^a) - f\left(x_i^p\right) \right\|_2^2$$

15:

$$HardNegative : Argmin \| f(x_i^a) - f(x_i^n) \|_2^2$$

16: train on Inception Network Architecture
17: save the checkpoints
18: **until** loss $\geq$ stop_threshold [convergence point]

return saved checkpoint

2) Preprocess images to 160x160.

3) Generate embeddings of all the images.

4) Form triplets.

5) Selection of an anchor image at random.

6) Selection of a positive image(belonging to the same person as that of anchor image) at random.

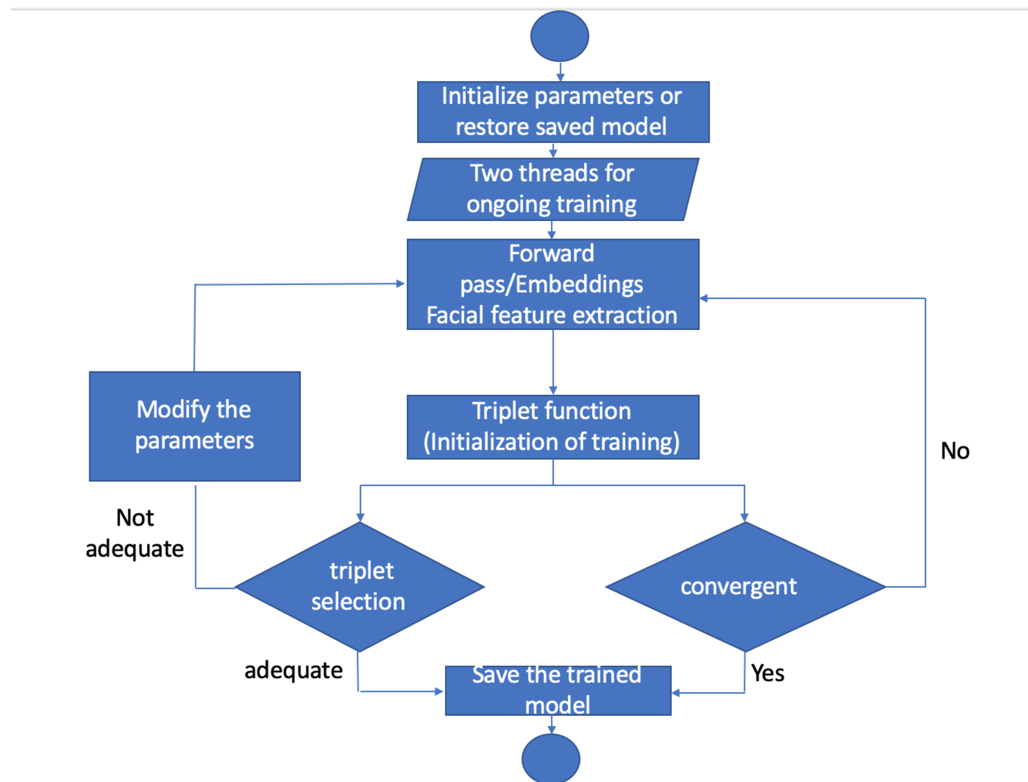

Fig. 14. Flowchart representation of FaceNet training.

7) Selection of a negative image(belonging to a different person from that of anchor and positive image) at random.
8) Adjusting the parameters of Facenet architecture in such a way that anchor and positive images are closer to each other than negative image.

All the above steps are repeated until there are no possibilities left.

The above transformation results in forming clusters eventually with similar images in one cluster. However, considering all these possibilities of triplets for training a Facenet model will be huge. Thus, triplets are selected with proper parameters. FaceNet paper by Schroff et al. [1] suggests two different methods for triplet selection:

- Offline selection of triplets: at the beginning of each epoch embeddings on the training set are computed, and then, only hard or semi-hard triplets are selected.

Compute the loss of these B triplets and then backpropagate in the network by Moindrot [16]. Updating the offline mined triplets is necessary and so this method is inefficient.

- Online selection of triplets: for each batch of inputs, compute useful triplets on the fly. For sure, most of these triplets are not valid, because they do not satisfy the condition of having 1 anchor, 1 positive and 1 negative image. But as this method gives more number of triplet pairs than offline mining of triplets, hence it is more efficient

4 IMPLEMENTATION

4.1 Feasibility Study with Adaptive Feed Forward Network

As stated, this research's research objective is to minimize the time required for training and keep an ongoing process of training when additional data points are added to the dataset. This adaptive learning process is proved by experimenting with it on a small and simple feed-forward network, which stands as the proof-of-the-concept. This proof is used to implement adaptive learning on the basic neural network and should not be considered a final piece of work. Algorithm 3 describes the overview of the adaptive feed-forward network.

Algorithm 3 Adaptive Feed Forward Network Pseudo Code

input: (x_1, x_2) coordinates of point
1: **repeat**
2: initialize the weights w_i_j and biases b_i_j or load the saved weights and biases.
3: Forward pass through sigmoid activation function:

$$y = f(x_1 w_1 + x_2 w_2 + b)$$

4: **for** each element in dataset **do**
5: train/minimize the loss:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_{true} - y_{pred})^2$$

6: backpropagation: partial derivative:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

7: update weight equation(SGD):

$$w_1 \leftarrow w_1 - \eta \frac{\partial L}{\partial w_1}$$

8: Save the updated weights and biases (model)
9: **until** loss $\geq$ stop_threshold [convergence point]
return 0 or 1

Feed-forward neural network dataset only serves to prove the concept of adaptive learning. It is not used to demonstrate the capability of network decision purpose, nor for linear/non-linear decision-making functions. The amount of data points in the dataset is

not the point of concern. Instead, it is the evolution of data points concerning time and the prior knowledge being used for adaptive learning. Training a neural network from scratch requires a double expense of time, utilizes resources, and increases computation. All of this can be reduced by adopting adaptive learning. It uses prior knowledge plus new data to train the network, which reduces computation, required resources, and time for training. It trains the network on an already saved network model and updates the last weights based on newly added data to the already trained dataset. First, consider a simple feed-forward neural network with two hidden layers and 2 inputs (x_1, x_2) as coordinates of a point. This acts as the input to the network. A fine line can be drawn between red circles and blue circles in the plot. This fine line represents 2 different clusters. Red circles belong to class 1, and blue circles belong to class 0. Graphical representation of dataset for simple feed-forward neural network to train without prior knowledge: Table 3 provides the exact coordinates represented by the red and blue circles.

Table 3
Dataset for Feasibility Study of Feed Forward Network Training without Prior Knowledge

x_1	x_2	**Classes**
1	2.5	1
1	3	1
2.1	3.4	1
2.1	1	0
3.3	1	0
3	2.3	0

Fig. 15 represents the 2D plot of data points used to train feed-forward neural network. Based on the above dataset, a simple feed-forward network is trained by setting the stop_threshold value to be 0.004. Below is the stepwise methodology to implement and save the trained feed-forward network. Step 1: Load data (CSV file):

Input Variables (X): Array of x and y co-ordinates.

Output Variables (y): Class variable (0 or 1)

Once the CSV file is loaded into memory, split the columns of data into input and output

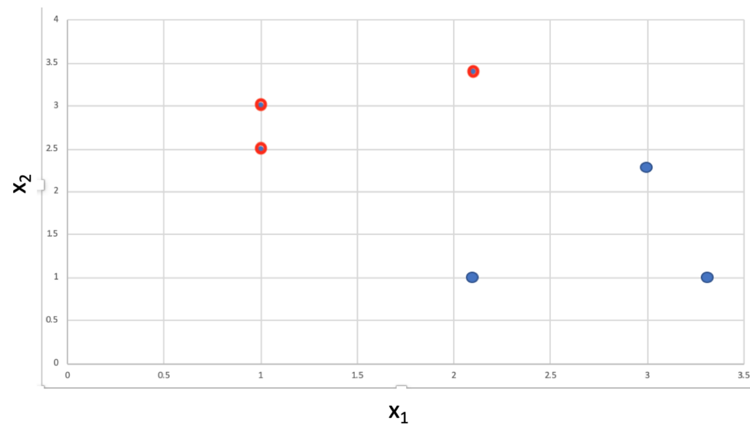

Fig. 15. 2D plot of dataset for feed-forward network.(Red circles: points of class 1, Blue circles: points of class 0).

variables.

Step 2: Define the Keras Model:

A sequential model is created by adding 1 input and 1 output layer. The preferred activation function used is the sigmoid activation function.

Step 3: Compile Keras Model:

Compiling the model uses efficient numerical libraries such as Theano or TensorFlow. When compiling, additional properties are specified which are required to train the network. The loss function: “binary_crossentropy“ is used to evaluate a set of weights, the Adam optimizer is used to search through different weights for the network and any optional metrics to be collected and reported during training.

Step 4: Fit Keras Model:

Fitting of model means to execute or to train the model on some data. Training occurs over epochs and each epoch is split into batches.

- Epoch: One pass through all of the rows in the training dataset.
- Batch: One or more samples considered by the model within an epoch before weights are updated.

While training the network for the very first time , epoch value is set to 500.

Step 5: Evaluate the model:

The simple feed-forward neural network is now trained on the entire dataset and evaluated for the performance of the network on the same dataset. The model.evaluate() function returns a list with two values. The first is the loss of the model on the dataset and the second is the accuracy of the model on the dataset.

Step 6: Save the model

To save the model, the Keras checkpoint feature is used. To decide which version should be stored, Keras observes the loss function and chooses the model version that has a minimal loss.

checkpoint = ModelCheckpoint(filepath, monitor = 'loss', verbose = 1, save_best_only = True, mode = 'min')

Save the model in the x.h5 file. Fig. 16 shows the graph of decreasing loss concerning the number of epochs for training a simple feed-forward neural network without adaptive learning. Once the model weights and biases are saved in x.h5 file, now, one more data

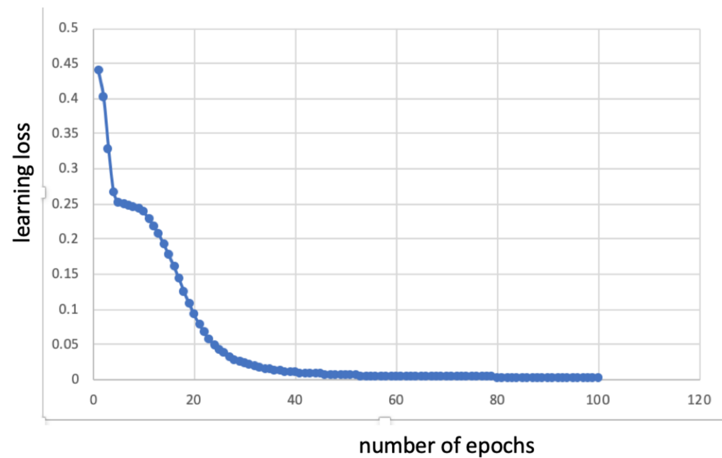

Fig. 16. Graph of pre-adaptive learning loss in feed-forward network.

point is added in each class in the previous dataset as shown in Fig. 17 with its exact coordinates shown in Table 4.

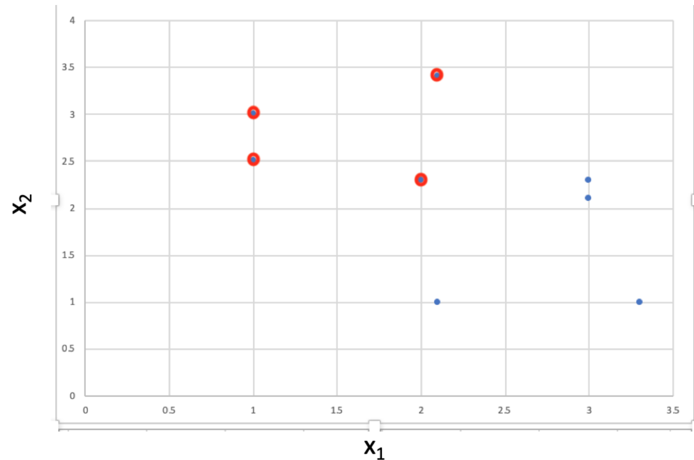

Fig. 17. 2D plot of dataset for feed-forward network with additional data.

Table 4
Dataset for Feasibility Study of Feed Forward Network Training with Prior Knowledge

x_1	x_2	**Classes**
1	2.5	1
1	3	1
2.1	3.4	1
2	2.3	1
2.1	1	0
3.3	1	0
3	2.3	0
3	2.1	0

For adaptive learning on a simple feed-forward neural network, all the steps implemented for non-adaptive simple feed-forward neural networks are implemented except for one change. Here, the initialization of weights and biases changes from random initialization to restoring the previously saved weights and biases and then continuing the entire process from step one to step 6. And so step 7 becomes:

Step 7: Restore the saved model: Restore the file's model by using the load_model() function.

new_model = load_model(”x.h5”)

The model is now restored and ready to start the training. Fit the new model with stop_threshold or number of epochs (whichever reaches first acts as the stopping criteria for the training to stop). Observe the log output of loss and number of epochs required to converge the model before saving the model and after loading the model from the saved file and calling the fit function again.

An important point to notice is that the training did not start from scratch. Instead, Keras continued fitting the model from it left off with reduced loss and training time to get converged. Fig. 18 shows the graph of decreasing loss concerning the number of epochs for training a simple feed-forward neural network with adaptive learning. In this experiment, pre-adaptive training and post-adaptive training are performed for 1000 epochs and the learning losses are recorded. For the first time, the stop_threshold loss = 0.004 in pre-adaptive learning was recorded at epoch 640, and in post-adaptive learning, it was recorded at epoch 18. The training time required to converge the model is reduced compared to the previous model and can be seen in Fig. 19.

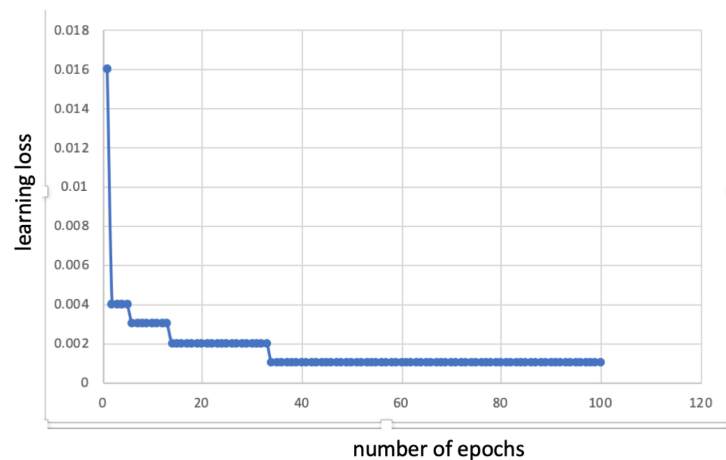

Fig. 18. Graph of post-adaptive learning loss in feed-forward network.

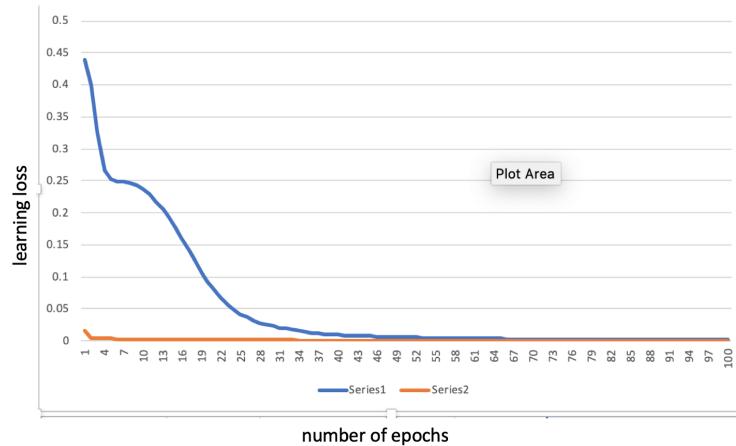

Fig. 19. Comparison of pre-adaptive and post-adaptive learning losses in feed-forward network.

Observations: Training starts from where the last training was saved. Because the previously trained model was saved in a file and loaded again to resume training on the new data and adapted into the model. This has saved the computation of training from scratch and also reduced the time required for training. Comparison of training epochs and losses is based on N and N_new.

Here,

N= number of epochs for the network to converge without prior training knowledge i.e. without adaptive learning.

N_new = number of epochs for the network to converge at the same point with prior training knowledge i.e. with adaptive learning. Learning losses recorded are as shown in Fig. 20, Fig. 21 and Fig. 22. Consider the number of epochs required to reach a convergence point in training a neural network without prior knowledge is N. And the number of epochs required to reach the same convergence point in training a neural network with prior knowledge is N_new. If N_new ¡ N, only then this proof-of-the-concept can be proved. Experimental results: A simple feed-forward neural network with one input layer, one hidden layer, and one output layer.

Epochs	Pre-adaptive learning loss	Post-adaptive learning loss	
0	0.439	0.016	
10	0.401	0.004	
30	0.327	0.004	
40	0.267	0.004	
50	0.252	0.004	
60	0.249	0.003	
70	0.248	0.003	
80	0.246	0.003	
90	0.243	0.003	
100	0.238	0.003	
110	0.229	0.003	
120	0.218	0.003	
130	0.206	0.003	
140	0.192	0.002	
150	0.177	0.002	
160	0.16	0.002	
170	0.143	0.002	
180	0.125	0.002	
190	0.108	0.002	
200	0.092	0.002	
210	0.079	0.002	
220	0.067	0.002	
230	0.057	0.002	
240	0.049	0.002	
250	0.042	0.002	
260	0.037	0.002	
270	0.032	0.002	
280	0.028	0.002	
290	0.025	0.002	
300	0.023	0.002	
310	0.02	0.002	
320	0.019	0.002	
330	0.017	0.002	
340	0.015	0.001	
350	0.014	0.001	
360	0.013	0.001	
370	0.012	0.001	
380	0.011	0.001	

Fig. 20. Loss vs Epochs for pre-adaptive and post-adaptive simple feed-forward network.

The number of epochs to reach the convergence point are:

N = 640, N_new = 180 with loss function being = 0.004

N_new/N = 180/640 = 28.125%

This leads to an accuracy of A= (640-180)/640= 71.875%

Epochs	Pre-adaptive learning loss	Post-adaptive learning loss
390	0.011	0.001
400	0.01	0.001
410	0.009	0.001
420	0.009	0.001
430	0.008	0.001
440	0.008	0.001
450	0.008	0.001
460	0.007	0.001
470	0.007	0.001
480	0.007	0.001
490	0.006	0.001
500	0.006	0.001
510	0.006	0.001
520	0.006	0.001
530	0.005	0.001
540	0.005	0.001
550	0.005	0.001
560	0.005	0.001
570	0.005	0.001
580	0.004	0.001
590	0.004	0.001
600	0.004	0.001
610	0.004	0.001
620	0.004	0.001
630	0.004	0.001
640	0.004	0.001
650	0.004	0.001
660	0.003	0.001
670	0.003	0.001
680	0.003	0.001
690	0.003	0.001
700	0.003	0.001
710	0.003	0.001
720	0.003	0.001
730	0.003	0.001
740	0.003	0.001
750	0.003	0.001

Fig. 21. Loss vs Epochs for pre-adaptive and post-adaptive simple feed-forward network (contd.).

This is calculated based on the losses and epochs recorded for pre-adaptive learning and post-adaptive learning of a simple feed-forward network. The architecture of the pilot test platform for an adaptive feed-forward network is as shown in Fig. 23. When the same feed-forward network is modified by adding one more hidden layer, which means the new neural network has one input layer, two hidden layers, and one output layer.

Epochs	Pre-adaptive learning loss	Post-adaptive learning loss
760	0.003	0.001
770	0.003	0.001
780	0.003	0.001
790	0.003	0.001
800	0.002	0.001
810	0.002	0.001
820	0.002	0.001
830	0.002	0.001
840	0.002	0.001
850	0.002	0.001
860	0.002	0.001
870	0.002	0.001
880	0.002	0.001
890	0.002	0.001
900	0.002	0.001
910	0.002	0.001
920	0.002	0.001
930	0.002	0.001
940	0.002	0.001
950	0.002	0.001
960	0.002	0.001
970	0.002	0.001
980	0.002	0.001
990	0.002	0.001
1000	0.002	0.001

Fig. 22. Loss vs Epochs for pre-adaptive and post-adaptive simple feed-forward network(contd.).

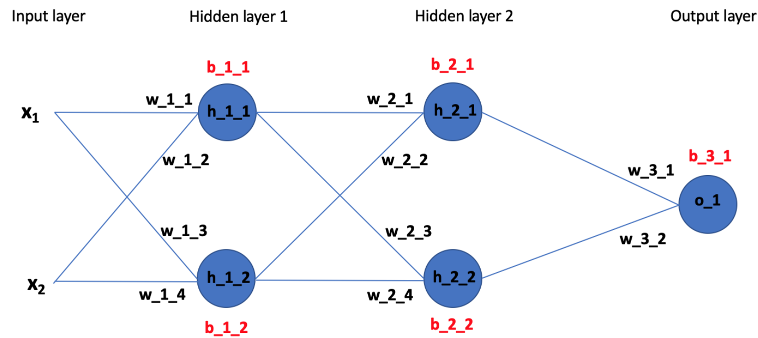

Fig. 23. Architecture of pilot test platform for adaptive learning.

For this new neural network,

N=570, N_new=10 with loss function being = 0.004

N_new/N = 10/570 = 1.75%

This leads to an accuracy of A_new = (570-10)/570 = 98.24%

Hence, A_new is greater than A

This proves the concept of adaptive learning based on a feed-forward network. Once the model is trained on some data points, it can then be extended by saving and reloading the model again with additional data points with reduced loss and time. The same theory and implementation can now be tested on FaceNet and ResNet for adaptive learning.

4.2 David Sandberg - Baseline Implementation

David Sandberg provides a baseline implementation by Sandberg [4] for the original FaceNet paper by Schroff et al. [1] with triplet loss and softmax loss. Although 2 pre-trained models are made available by David Sandberg on his GitHub repository, which is trained on CASIA-Webface and VGGFace2 datasets, we have not used any of them. In this research, FaceNet neural network is trained from scratch to validate proof-of-the-concept and receive significant results. Testing of FaceNet for adaptive learning is implemented the same way as an adaptive simple feed-forward neural network following all the 7 steps. The tree structure of David Sandberg's GitHub repository by Sandberg [4] is as shown in Table 5.

Table 5
Architecture of FaceNet Code

Folders	**Files**	**uses**
Data (images)	train_raw train_aligned test_raw test_aligned	Collection of dataset required for training FaceNet
src (align)	detect.py align_dataset_mtcnn.py	face detection / alignment algorithm align and crop the dataset in proper size
src	facenet.py train_tripletloss_copy1 train_tripletloss_copy2	triplet loss, center loss, train, crop functions Training a face recognizer without prior knowledge Training a face recognizer with prior knowledge

Fig. 24 and Fig. 25 shows the tree structure of the code architecture in local system.

The particular feature of FaceNet is its loss function. Triplet loss is the name of the function that is used for face validation. However, David's FaceNet implementation has

```
├── src
│   ├── __init__.py
│   ├── __pycache__
│   │   ├── facenet.cpython-36.pyc
│   │   └── lfw.cpython-36.pyc
│   ├── align
│   │   ├── __init__.py
│   │   ├── __pycache__
│   │   │   └── detect_face.cpython-36.pyc
│   │   ├── align_dataset_mtcnn.py
│   │   ├── det1.npy
│   │   ├── det2.npy
│   │   ├── det3.npy
│   │   └── detect_face.py
│   ├── calculate_filtering_metrics.py
│   ├── classifier.py
```

Fig. 24. Code architecture: David Sandberg's FaceNet github repo.

```
│   ├── classifier.py
│   ├── compare.py
│   ├── decode_msceleb_dataset.py
│   ├── detect_corrupt_images_script.py
│   ├── download_and_extract.py
│   ├── facenet.py
│   ├── freeze_graph.py
│   ├── generative
│   │   ├── __init__.py
│   │   ├── calculate_attribute_vectors.py
│   │   ├── models
│   │   │   ├── __init__.py
│   │   │   ├── dfc_vae.py
│   │   │   ├── dfc_vae_large.py
│   │   │   ├── dfc_vae_resnet.py
│   │   │   └── vae_base.py
│   │   ├── modify_attribute.py
│   │   └── train_vae.py
│   ├── lfw.py
│   ├── models
│   │   ├── __init__.py
│   │   ├── __pycache__
│   │   │   ├── __init__.cpython-36.pyc
│   │   │   └── inception_resnet_v1.cpython-36.pyc
│   │   ├── dummy.py
│   │   ├── inception_resnet_v1.py
│   │   ├── inception_resnet_v2.py
│   │   ├── model-1.ckpt-3.data-00000-of-00001
│   │   ├── model-1.ckpt-3.index
│   │   ├── model-1.meta
│   │   └── squeezenet.py
│   ├── train_softmax.py
│   ├── train_tripletloss.py
│   ├── train_tripletloss_copy.py
│   ├── train_tripletloss_copy1.py
│   ├── train_tripletloss_copy2.py
│   └── validate_on_lfw.py
```

Fig. 25. Code architecture: David Sandberg's FaceNet github repo(contd.).

two loss functions, 'Triplet loss' and 'Softmax activation with cross-entropy loss.' For training, David Sandberg's FaceNet implementation suggests making use of Softmax loss function over triplet loss for better results. Usually, in supervised learning, a network is trained over softmax cross-entropy loss because there are a fixed number of classes to be trained. However, in some cases, there can be a possibility of having a variable number of classes. For example, in facial recognition, there is a need to compare two unknown faces to know whether they belong to the same person/class or not. Here, triplet loss marks

significant importance. It is a way to learn good embeddings for all the faces individually. Separate clusters are formed in a euclidean space from the faces that belong to the same person. Considering this as a baseline, adaptive learning is implemented in the next section.

4.3 Adaptive Learning for FaceNet

Training Dataset:

A folder named “images” is created in the project directory of David Sandberg’s Github repository by Sandberg [4]. Place the training data/images in this folder. Create three separate folders inside this directory for each person. Now place the images of these three people in their respective folder as shown in Fig. 26.

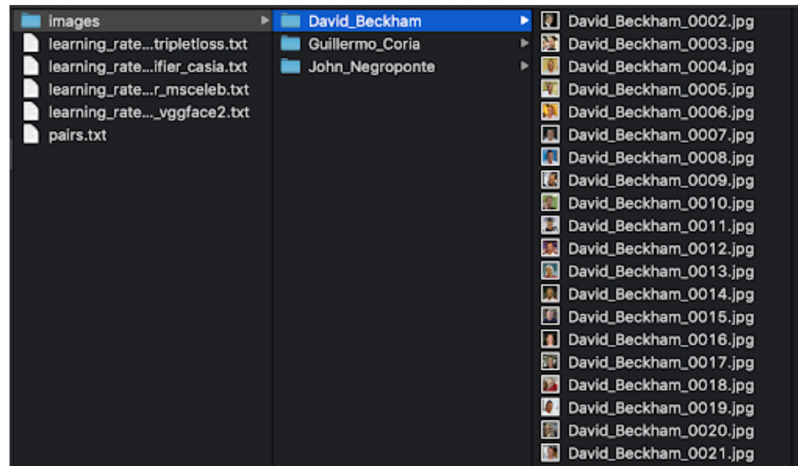

Fig. 26. Images of number of persons in the dataset to train FaceNet.

These original images have a different shape, size, lighting, orientation, background, etc. To train the FaceNet model, these images should have the same size required by the network, and they must contain faces only. This is obtained by generating tight bounding boxes around the faces and extracting these bounding boxes. To get the exact training data: the cropped faces, a face detection algorithm called Multi-task Cascaded Convolutional Neural Networks (MTCNN) is applied to the original set of images.

FaceNet takes an input of images with resolution 160x160. Use the script named align_dataset_mtcnn.py to align faces in 160x160 size. This code for cropping the images is taken from David Sandberg's repository by Sandberg [4]. The command given below is used to crop and align images in the required FaceNet format for training.

python align_dataset_mtcnn.py SOURCE_Path Target_Path

To have fewer computations on images and extract useful features, it is mandatory to align and crop the images around bounding boxes from the dataset that can be feed into FaceNet. It becomes fairly easy to compute embeddings over the cropped image because most of the information which is not important is being discarded in this process. To make this training simpler and test the results on a small dataset while training FaceNet from scratch, I have added 10 images of 3 persons each in the dataset as shown in Fig. 27. The resolution and count of images for each person are described in Table 6.

Fig. 27. Our customer designed small pre-adaptive FaceNet dataset.

FaceNet model is now trained on the above dataset with a triplet loss function. As this training process is from scratch, all the parameters, specific weights, and biases for the network are randomly initialized. This network is trained until a stop_threshold of loss = 1.4 (can be chosen based on your training and dataset) is reached. Once the training

reaches the convergence point either of stop_threshold or the number of epochs mentioned in the code, whichever occurs first, the model is saved in the form of checkpoints. The number of epochs and loss associated with it is noted to compare the result with adaptive FaceNet training. Once FaceNet is trained from scratch, and the model is saved, adaptive FaceNet is being implemented.

Table 6
Our Customer Designed Dataset for FaceNet Training without Prior Knowledge.

Person	**Number of images**	**Resolution**
David_Beckham	10	160 x 160
Guillermo_Coria	10	160 x 160
John_Negroponte	10	160 x 160

In order to do adaptive learning of FaceNet, I have added 10 more images to each person as shown in Fig. 28 and its description is mentioned in Table 7.

Fig. 28. Our customer designed small adaptive FaceNet dataset.

In David Sandberg's repository, some parameters have been set to different default values according to the dataset. But as we have our own custom dataset, it becomes necessary to change the default values of these parameters. To get proper results of

Table 7
Our Customer Designed Dataset for FaceNet Training with Prior Knowledge.

Person	**Number of images(Previous:10 + New:10)**	**Resolution**
David_Beckham	20	160 x 160
Guillermo_Coria	20	160 x 160
John_Negroponte	20	160 x 160

training and based on the dataset used, these parameters have been changed as stated in Table 8.

Table 8
Updated FaceNet Parameters for Adaptation

Parameter	**Value**
stop_threshold	1.4
max_nrof_epochs	1
batch_size	30
people_per_batch	3
images_per_person	10
alpha	0.2
learning_rate	0.1

Along with these parameters, the most important change to be implemented for adaptive learning is loading the previously saved checkpoints above instead of randomly initializing the parameters. Because the learning/prior knowledge of the subset of a new dataset is already available, which, when loaded and used in the network, helps in adaptive learning by reducing training time from scratch. The model converges faster than before.

Result:

The pre-adaptive and post-adaptive triplet learning losses and corresponding epochs are stored in a file to compare the results. This can be visualized in graphical format for training with prior knowledge compared to training without prior knowledge, as shown in Fig. 29.

The figure shows that the pre-adaptive learning loss is less than 1.4 at epoch number 24. Comparatively, the post-adaptive learning loss is less than 1.4 at epoch number 12.

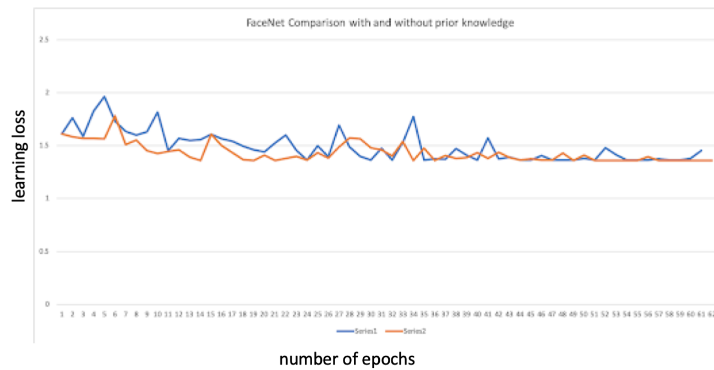

Fig. 29. Graph of loss vs epochs of Adaptive FaceNet.

which reduces the training time to reach the loss below 1.4 by 50%. The exact values of losses are shown in Fig. 30 and Fig. 31.

Observation 1:

- Less loss errors which means, in initial stages, loss recorded is less as compared to pre-adaptive learning losses.
- Converges faster, almost in half the time needed to train without prior knowledge.

Observation 2: In the triplet-loss function FaceNet algorithm, the network converges faster when the prior knowledge is introduced and gives better accuracy. The training with prior knowledge shown in the orange line in the graph almost converges at 43 epochs. Also, with prior knowledge and additional data, the training starts with less loss and much more accuracy than training without prior knowledge. With prior knowledge, hard triplets are usually found at the very start of the training with comparatively lesser loss.

As the research objectives are being accomplished, this can be further extended to multiprocessing. The training process keeps ongoing as the new data is being added to the dataset. Multiprocess algorithm for adaptive FaceNet training is as follows in Algorithm 4.

	FaceNet training loss		
EPOCH	Loss without prior knowledge	Loss without prior knowledge	
1	1.612	1.612	
2	1.763	1.585	
3	1.59	1.571	
4	1.828	1.568	
5	1.963	1.567	
6	1.733	1.783	
7	1.636	1.511	
8	1.6	1.555	
9	1.631	1.453	
10	1.819	1.427	
11	1.455	1.444	
12	1.571	1.461	
13	1.549	1.39	
14	1.559	1.362	
15	1.609	1.607	
16	1.566	1.501	
17	1.544	1.433	
18	1.496	1.367	
19	1.46	1.362	
20	1.44	1.411	
21	1.526	1.362	
22	1.599	1.38	
23	1.458	1.4	
24	1.363	1.363	
25	1.499	1.433	
26	1.396	1.384	
27	1.695	1.488	
28	1.487	1.574	
29	1.398	1.567	
30	1.363	1.482	
31	1.478	1.462	
32	1.363	1.402	
33	1.531	1.54	
34	1.773	1.362	
35	1.363	1.478	

Fig. 30. Pre-adaptive and post-adaptive FaceNet learning loss.

Number of experiments conducted: First, the experiment was conducted on training the FaceNet model on 3 persons with 5 images each. Added 5 more images to each person for adaptive learning. This training was executed for only 2 epochs because it becomes nearly impossible to update the FaceNet parameters concerning just 5 images per person for training. Hence, the training does not converge to give expected results on adaptive learning. Therefore, the experimental data was then modified to contain 10

36	1.375	1.362
37	1.37	1.406
38	1.473	1.381
39	1.409	1.389
40	1.363	1.432
41	1.574	1.381
42	1.375	1.436
43	1.393	1.387
44	1.363	1.363
45	1.363	1.376
46	1.408	1.365
47	1.363	1.363
48	1.363	1.43
49	1.363	1.362
50	1.379	1.412
51	1.363	1.362
52	1.482	1.362
53	1.414	1.362
54	1.363	1.362
55	1.363	1.362
56	1.363	1.394
57	1.374	1.362
58	1.363	1.362
59	1.363	1.362
60	1.378	1.362
61	1.454	1.362
		1.362

Fig. 31. Pre-adaptive and post-adaptive FaceNet learning loss(contd.).

Algorithm 4 Multiprocess Adaptive FaceNet Pseudo Code

input: two processes
1: **repeat**
2: **process one:**
3: randomly initialize parameteres for data with 10 images per person.
4: form triplets based on MSE of triplet loss
5: train the dataset on triplet loss function.
6: save checkpoint
7: pass saved checkpoint to process two
8: **process two:**
9: reload the saved checkpoint [parameteres]
10: add 10 images to each person in previous dataset
11: form triplets based on MSE of triplet loss
12: train the new dataset on triplet loss function
13: save checkpoint
14: **until** loss $\geq$ stop_threshold [convergence point]
return recorded loss and accuracy

images of each person for pre-adaptive training with different FaceNet parameters to receive the expected result of proper convergence with the number of epochs.

In this experiment, firstly, the number of epochs was set to 15-20, and the results were recorded, and then adaptive learning on FaceNet was executed for the same number of epochs. FaceNet gives unexpected results for such a small number of epochs because hard triplets are generally at the initial stages, and hence the losses recorded are irregular. Moreover, therefore, there was a need to increase the training time by increasing the number of epochs. Finally, the number of epochs was set to 60 to get the converged result, which was then be saved and restored in adaptive learning.

5 COMPARATIVE STUDY OF ADAPTIVE RESNET

5.1 Discussion of ResNet

ResNet was developed in 2015 by Microsoft researchers by He et al. [2]. It is still prevalent in the field of computer vision for facial recognition. Its most important characteristic is that it can train hundreds or thousands of layers in the network without a vanishing gradient because of the introduction of skip connections. It has provided many pre-trained architectures/models, but ResNet is also available to be trained from scratch. Because of these skip connections, some of the network layers are skipped, and the learning becomes speedy because the output of previous layers reaches as an input to the next layers reusing the same activation functions by Link [20].

Algorithm 5 describes the pseudocode of adaptive ResNet.

Algorithm 5 Adaptive ResNet Algorithm

input: 10 images per person (3 persons)

1: **repeat**
2: randomly initialize the ResNet parameteres or load the saved model.
3: **for** each image x_i **do**
4: preprocess images to 250x250 resolution
5: generate name.npy and label.npy for each x_i
6: forward pass the parameters
7: pass these parameters through residual block
8:

$$y = f(x, \{w_i\}) + x$$

9: here x is input to residual block

$$f(x, \{w_i\})$$

10: where

$$1 \leq i \leq number_of_layers_in_residual_block$$

11: train the ResNet Network on input images.
12: save the model
13: **until** loss $\geq$ stop_threshold [convergence point]
return saved model

For this research, Kaihua Tang's GitHub repository by Tang [21] is used as baseline implementation. A folder named ”images” is created in the project directory of Kaihua

Tang's GitHub repository by Tang [21]. Place the training data/images in this folder. Create three separate folders inside this directory for each person. Now place the images of these three people in their respective folders. These original images have a different shape, size, lighting, orientation, background, etc. To train the ResNet model, these images should have the same size required by the network, and they must contain faces only. ResNet takes an input of images with resolution 224x224. It becomes fairly easy to extract features over the cropped image because most of the information which is not important is being discarded in this process. To make this training simpler and test the results on a small dataset while training ResNet from scratch, I have added 10 images of 3 persons each in the dataset as shown in Fig. 32.

The resolution and count of images for each person are described in Table 9.

Fig. 32. Our customer designed small pre-adaptive ResNet dataset.

Table 9
Our Customer Designed Dataset for ResNet Training without Prior Knowledge

Person	Number of images	Resolution
David_Beckham	10	224 x 224
Guillermo_Coria	10	224 x 224
John_Negroponte	10	224 x 224

The ResNet model is now trained on the above dataset. As this training process is from scratch, all the parameters, weights, and biases for the network are randomly initialized. This network is trained until a stop_threshold is reached. Once the training reaches the convergence point either of stop_threshold or the number of epochs mentioned in the code, whichever occurs first, the model is saved in the form of checkpoints. The number of epochs and losses associated with it is noted to compare the result with adaptive ResNet training. Once ResNet is trained from scratch, and the model is saved, adaptive ResNet is being implemented. To do adaptive learning of ResNet, I have added 10 more images to each person as shown in Fig. 33, and its description is mentioned in Table 10.

Fig. 33. Our customer designed small adaptive ResNet dataset.

Table 10
Our Customer Designed Dataset for ResNet Training with Prior Knowledge

Person	Number of images(Previous:10 + New:10)	Resolution
David_Beckham	20	250 x 250
Guillermo_Coria	20	250 x 250
John_Negroponte	20	250 x 250

According to the dataset in Kaihua Tang's repository, some parameters have been set to different default values. However, as we have our own custom dataset, it becomes necessary to change these parameters' default values. To get proper results of training and based on the dataset used, these parameters have been changed. Along with these parameters, the most important change to be implemented for adaptive learning is loading the previously saved checkpoints above instead of randomly initializing the parameters. Because the learning/prior knowledge of the subset of a new dataset is already available, which, when loaded and used in the network, helps in adaptive learning by reducing training time from scratch. The model converges faster than before.

5.2 Results of Adaptive ResNet Training

The pre-adaptive and post-adaptive learning losses and their corresponding epochs are stored in a file to compare the results. This can be visualized in graphical format for training with prior knowledge compared to training without prior knowledge, as shown in Fig. 34.

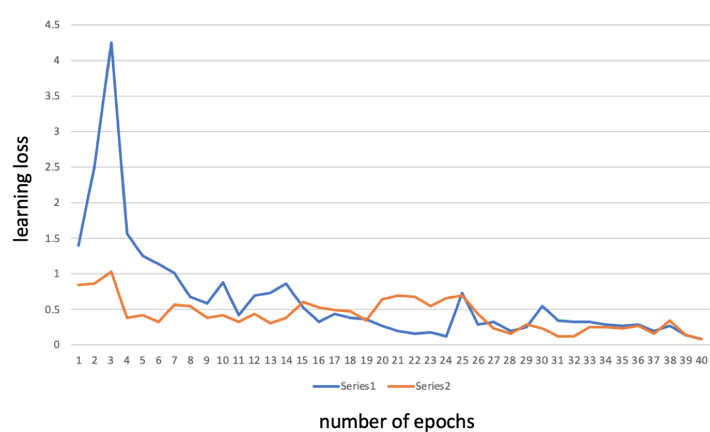

Fig. 34. Graph of Adaptive ResNet learning loss.

From the figure it clear that, the post-adaptive learning loss is less than that of pre-adaptive learning loss at initial stages. This can be visualized in tabular format for

training with prior knowledge as compared to training without prior knowledge as shown in Fig. 35.

Epoch		pre-adaptive loss	post-adaptive loss	
0		1.39	0.84	
1		2.48	0.85	
2		4.24	1.02	
3		1.56	0.37	
4		1.25	0.41	
5		1.14	0.33	
6		1	0.56	
7		0.67	0.54	
8		0.58	0.37	
9		0.87	0.41	
10		0.42	0.32	
11		0.7	0.44	
12		0.73	0.31	
13		0.86	0.38	
14		0.53	0.6	
15		0.33	0.52	
16		0.44	0.49	
17		0.37	0.47	
18		0.36	0.34	
19		0.27	0.63	
20		0.2	0.69	
21		0.16	0.67	
22		0.17	0.54	
23		0.12	0.65	
24		0.72	0.7	
25		0.29	0.43	
26		0.33	0.23	
27		0.2	0.16	
28		0.25	0.29	
29		0.54	0.23	
30		0.35	0.11	
31		0.33	0.12	
32		0.32	0.24	
33		0.28	0.25	
34		0.26	0.23	
35		0.29	0.26	
36		0.2	0.16	
37		0.26	0.34	
38		0.13	0.14	
39		0.09	0.08	

Fig. 35. Pre-adaptive and Post-adaptive ResNet learning loss.

Observation 1:

- Less loss errors which means, in initial stages, loss recorded is less as compared to pre-adaptive learning losses.
- Converges faster, less than the time needed to train ResNet without prior knowledge.

Observation 2: In the ResNet algorithm, when the prior knowledge is introduced, the network converges faster and gives better accuracy. The training with prior knowledge shown in the orange line in the graph almost converges at epoch 28. Also, with prior knowledge and additional data, the training starts with less loss and much more accuracy than training without prior knowledge. Hence, the research objectives are accomplished based on proof-of-the-concept.

The number of experiments conducted: Firstly, the experiment was conducted on training the ResNet model on 10 images for 3 persons each for 20 epochs. The losses were recorded for pre-adaptive and post-adaptive learning. The losses did not give significant results because in pre-adaptive learning with 20 epochs, the amount of learning saved was not significant enough to train the adaptive ResNet on prior knowledge. Moreover hence, the results did not converge as expected. Therefore, the experimental data was then modified to contain 10 images of each person for pre-adaptive training with 40 epochs to receive the expected results, as stated above.

6 FUTURE WORK

The proof-of-the-concept is now proved on 2 of the most commonly used facial recognition algorithms: FaceNet and ResNet. Although all of the research objectives of reduced time for training the neural network, early convergence on the ongoing adaptive training process, and training the network from scratch by using prior knowledge on the same dataset, there is still a scope to improve this adaptive learning algorithm. First of all, the selection of triplets plays an important role in FaceNet architecture. it works fine until now but can be improved on the following basis: The disadvantage of triplet loss function: As long as the negative value in equation (2) is larger than the positive value + alpha, there is no gain to condense the positive embeddings, and the anchor embeddings to the minimum distance than before by Arsenault [22]. Consider:

- Alpha is 0.2.
- Negative Distance is 2.0.
- Positive Distance is 1.1.

The result of the loss function will be 1.1 – 2.0 + 0.2 = -1. As stated above, once the loss function result is nearly equal or less than zero, the gain remains constant, which means a loss of information. With this result and condition, it is tough for the FaceNet algorithm to reduce the existing distance between the anchor and the positive images or embeddings by Arsenault [22].

The above results of adaptive learning on FaceNet or ResNet can be extended to test or implement on datasets with large size (more than 20) to adapt adaptive learning for ongoing additions in data points. Datasets in this research for the FaceNet and ResNet training are customized. The small size of data is for the feasibility of study purpose. Adaptive learning can be used to develop algorithms other than facial recognition algorithms.

FaceNet has a different backend neural network composition as compared to ResNet. However, as ResNet uses skip connections, it is possible in the future to integrate ResNet as a backend of FaceNet. Then a triplet loss training can be conducted on the dataset. This would prominently give significant results.

7 CONCLUSIONS

In this thesis/research, developing a new adaptive technique on FaceNet and ResNet for ongoing training on a dataset to reduce the training time and help faster convergence of the model is introduced. The main emphasis is on using prior knowledge to train the network for adaptive learning based on the changing target data. David Sandberg's Gihut repository is used as FaceNet baseline implementation, and Kaihua Tang's Github repository as ResNet baseline implementation. Firstly, to train the neural network from scratch and save the trained model for restoring and using it for adaptive training, our custom dataset of 10 images for 3 persons is generated and then adapted with 20 images for 3 persons each. A proof-of-the-concept simple feed-forward network is developed to demonstrate the research for ongoing adaptive learning techniques as a stepping-stone. After achieving the desired results on a simple feed-forward neural network, the same approach is tested and verified on FaceNet and ResNet. The network is then trained to save the checkpoints in a file that can be restored in the future for adaptive learning. This prior knowledge, saved in checkpoints, is now restored with additional data points in the dataset to start the adaptive training. Both the FaceNet and ResNet give significant results based on the research objectives. However, with the current state of implementation, we recommend extending this approach with a large dataset on FaceNet. It reduces the time required to converge the model at a threshold point of 50% reduction in epochs and training time. Finally, the results represent the comparison of the research objectives on ResNet and FaceNet in graphical format. The purpose of the investigation of FaceNet and ResNet for facial detection is based on these two architectures as they act as state-of-the-art technology. However, the lack of adaptation and the utilization of these networks is based on pre-trained models. Therefore this research addressed these voids.

Literature Cited

[1] Schroff, Florian, Kalenichenko, Dmitry, and Philbin, James. ”FaceNet: A Unified Embedding for Face Recognition and Clustering.” (2015): 815-23. Web.

[2] K. He, X. Zhang, S. Ren, and J. Sun. ”Deep Residual Learning for Image Recognition.” (2016): 770-78. Web.

[3] Y. Taigman, M. Yang, M. Ranzato and L. Wolf, ”DeepFace: Closing the Gap to Human-Level Performance in Face Verification,” *2014 IEEE Conference on Computer Vision and Pattern Recognition, Columbus, OH, 2014, pp. 1701-1708*, doi: 10.1109/CVPR.2014.220.

[4] D. Sandberg, ”FaceNet”, *https://github.com/davidsandberg/facenet*, 2019.

[5] Pan, S. Jialin, and Yang, Qiang. ”A Survey on Transfer Learning.” *IEEE Transactions on Knowledge and Data Engineering 22.10* (2010): 1345-359. Web.

[6] Dong Yi, Zhen Lei, Shengcai Liao and Stan Z. Li, “Learning Face Representation from Scratch”. *arXiv preprint arXiv*:1411.7923. 2014.

[7] Cao, Qiong, Shen, Li, Xie, Weidi, Parkhi, Omkar M, and Zisserman, Andrew. ”VGGFace2: A Dataset for Recognising Faces across Pose and Age.” (2017). Web.

[8] E. Corporation, ”Hands-on TensorFlow Tutorial: Train ResNet-50 From Scratch Using the ImageNet Dataset”, *https://blog.exxactcorp.com/deep-learning-with-tensorflow-training-resnet-50-from-scratch-using-the-imagenet-dataset/*, 2019.

[9] L. Dulčić, ”Face Recognition with FaceNet and MTCNN”, *https://arsfutura.com/magazine/face-recognition-with-facenet-and-mtcnn/*, 2019.

[10] J. CruzMartinez, ”Detecting Face Features with Python”, *https://towardsdatascience.com/detecting-face-features-with-python-30385aee4a8e*, 2018.

[11] Bijl, Erik, ”A comparison of clustering algorithms for face clustering”. *Bachelor’s Thesis, Computing Science*, 2018.

[12] S. Sahoo, ”Residual blocks — Building blocks of ResNet”, *https://towardsdatascience.com/residual-blocks-building-blocks-of-resnet-fd90ca15d6ec*, 2018.

[13] Szegedy, Christian, W. Liu, Y. Jia, Sermanet, Pierre, Reed, Scott, Anguelov, Dragomir, Erhan, Dumitru, Vanhoucke, Vincent, and Rabinovich, Andrew. ”Going Deeper with Convolutions.” 2015: 1-9. Web.

[14] M. Deore, ”FaceNet Architecture”, *https://medium.com/analytics-vidhya/facenet-architecture-part-1-a062d5d918a1*, 2019.

[15] Szegedy, Christian, Vanhoucke, Vincent, Ioffe, Sergey, Shlens, Jon, and Wojna, Zbigniew. ”Rethinking the Inception Architecture for Computer Vision.” (2016): 2818-826. Web.

[16] O. Moindrot,”Triplet Loss and Online Triplet Mining in TensorFlow”, *https://omoindrot.github.io/triplet-loss*, 2018.

[17] M. Deore, ”FaceNet Architecture”, *https://medium.com/analytics-vidhya/facenet-architecture-part-1-a062d5d918a1*, 2019.

[18] M. Works, ”Convolutional Neural Network”, *https://www.mathworks.com/discovery/convolutional-neural-network.html*

[19] H. Li, ”Train FaceNet with triplet loss for real time face recognition on keras”. *https://github.com/hualili/opencv/blob/master/deep-learning-2020S/10-2020F-107-part3-triple-loss-2020-10-6.pdf*, 2020.

[20] M. Link blog, ”PyTorch ResNet: Building, Training and Scaling Residual Networks on PyTorch”, *https://missinglink.ai/guides/pytorch/pytorch-resnet-building-training-scaling-residual-networks-pytorch/*, 2018.

[21] Kaihua Tang, ”ResNet50-Pytorch-Face-Recognition”, *https://github.com/KaihuaTang/ResNet50-Pytorch-Face-Recognition*, 2018.

[22] M. Arsenault, ”Lossless Triplet loss”, *https://towardsdatascience.com/lossless-triplet-loss-7e932f990b24*, 2018.

[23] M. Saini, ”Train FaceNet with triplet loss for real time face recognition on keras”. *https://medium.com/@mohitsaini_54300/train-facenet-with-triplet-loss-for-real-time-face-recognition-a39e2f4472c3*, 2019.

[24] D. Sandberg, ”Facenet”. *https://github.com/davidsandberg/facenet/blob/master/src/align/align_dataset_mtcnn.py*,2019.

[25] F.Li, ”CS231n Convolutional Neural Networks for Facial Recognition”, *https://cs231n.github.io/convolutional-networks/*, 2019.

[26] A. Pande, ”A Beginner’s Guide To Understanding Convolutional Neural Networks”, *https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner’s-Guide-To-Understanding-Convolutional-Neural-Networks/*, 2019.

[27] Ujjwalkarn, ”An Intuitive Explanation of Convolutional Neural Networks”, *https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/*, 2016.

[28] Datawow, ”Interns Explain CNN”, *https://blog.datawow.io/interns-explain-cnn-8a669d053f8b*, 2020.

[29] R. Prabhu, ”CNN Architectures — LeNet, AlexNet, VGG, GoogLeNet and ResNet”, *https://medium.com/@RaghavPrabhu/cnn-architectures-lenet-alexnet-vgg-googlenet-and-resnet-7c81c017b848*, 2018.

[30] B. Georgievski, ”Face Recognition using One-Shot Learning”, *https://mc.ai/face-recognition-using-one-shot-learning/*,2019.

[31] A. Ng, ”Deep Learning Resnet”, *http://deeplearning.ailectureC4W2L04*, 2017.

[32] C. Shorten, ”Introduction to ResNets”, *https://towardsdatascience.com/introduction-to-resnets-c0a830a288a4*, 2019.

[33] A. Krizhevsky, I. Sutskever, G. Hinton, ”ImageNet Classification with Deep Convolutional Neural Networks”, 2012.

[34] K. He, X. Zhang, S. Ren, Jian Sun, ”Deep Residual Learning for Image Recognition”, 2015.

[35] K. He, X. Zhang, S. Ren, J. Sun, ”Identity Mappings in Deep Residual Networks”, 2016.

[36] K. Simonyan, A. Zisserman, ”Very Deep Convolutional Networks for Large-Scale Image Recognition”, 2014.

[37] P. Dwivedi, ”Understanding and Coding a ResNet in Keras”, *https://towardsdatascience.com/understanding-and-coding-a-resnet-in-keras-446d7ff84d33*, 2019.

[38] Szegedy, Christian, W. Liu, Y. Jia, Sermanet, Pierre, Reed, Scott, Anguelov, Dragomir, Erhan, Dumitru, Vanhoucke, Vincent, and Rabinovich, Andrew. ”Going Deeper with Convolutions.” (2015): 1-9. Web

Appendix A

MAX-POOLING IN FCAENET

A.1 Convolutional, max-pooling and L2 layer in FaceNet

FaceNet architecture is based on two different convolutional neural networks(CNNs). These 2 different CNNs result into different number of parameteres. Facenet based on GoogleNet architecture uses 1x1 convolution to shrink the channels from 3 to 1 but keep the input size same. For e.g. 256x256x3 image to 256x256x1 image. These convolutions are used for dimensiona reduction. They are also called as filters. Other similar small filters are being used. Downsampling of the images is successfully done by max pooling layer in FaceNet architecture. Different filters are applied in parallel, and hence, eventually, all the outputs from one intermediate state are concatenated for next stage. This leads to the creation of deeper inception module.

Appendix B

ADAPTIVE ARCHITECTURE

B.1 Code for Simple Feed Forward Neural Network

Train and save Simple Feed Forward Neural Network on two classes with 3 data points each.

```
'''
  Program  : introNN.py;
  Version  : 1.0;
  Date     : Sept. 9, 2020
  Coded by : (see the reference below)
  Modified by: Dr. Harry Li  and Rachana Bumb for adapative learning // on Feed Forward Network
Ref: https://github.com/vzhou842/neural-network-from-scratch/blob/master/network.py
'''

import numpy as np

def sigmoid(x):
  # Sigmoid activation function: f(x) = 1 / (1 + e^(-x))
  return 1 / (1 + np.exp(-x))

def deriv_sigmoid(x):
  # Derivative of sigmoid: f'(x) = f(x) * (1 - f(x))
  fx = sigmoid(x)
  return fx * (1 - fx)
def mse_loss(y_true, y_pred):
  # y_true and y_pred are numpy arrays of the same length.
  return ((y_true - y_pred) ** 2).mean()

class OurNeuralNetwork:
  '''
  A neural network with:
    - 2 inputs
    - 2 hidden layer with 2 neurons each (h_1_1, h_1_2, h_2_1, h_2_2)
    - an output layer with 1 neuron (o_1)
```

```
*** DISCLAIMER ***:
The code below is intended to be simple and educational, NOT optimal.
Real neural net code looks nothing like this. DO NOT use this code.
Instead, read/run it to understand how this specific network works.
'''

def __init__(self):
  # Weights
  self.w_1_1 = np.random.normal()
  self.w_1_2 = np.random.normal()
  self.w_1_3 = np.random.normal()
  self.w_1_4 = np.random.normal()

  self.w_2_1 = np.random.normal()
  self.w_2_2 = np.random.normal()
  self.w_2_3 = np.random.normal()
  self.w_2_4 = np.random.normal()

  self.w_3_1 = np.random.normal()
  self.w_3_2 = np.random.normal()

  # Biases
  self.b_1_1 = np.random.normal()
  self.b_1_2 = np.random.normal()

  self.b_2_1 = np.random.normal()
  self.b_2_2 = np.random.normal()

  self.b_3_1 = np.random.normal()

def feedforward(self, x):
  # x is a numpy array with 2 elements.
  h_1_1 = sigmoid(self.w_1_1 * x[0] + self.w_1_2 * x[1] + self.b_1_1)
  h_1_2 = sigmoid(self.w_1_3 * x[0] + self.w_1_4 * x[1] + self.b_1_2)
```

```
    h_2_1 = sigmoid(self.w_2_1 * h_1_1 + self.w_2_2 * h_1_2 + self.b_2_1)
    h_2_2 = sigmoid(self.w_2_3 * h_1_1 + self.w_2_4 * h_1_2 + self.b_2_2)

    o_1 = sigmoid(self.w_3_1 * h_2_1 + self.w_3_2 * h_2_2 + self.b_3_1)
    return o_1

  def train(self, data, all_y_trues):
    '''
    - data is a (n x 2) numpy array, n = # of samples in the dataset.
    - all_y_trues is a numpy array with n elements.
      Elements in all_y_trues correspond to those in data.
    '''
    learn_rate = 0.1
    epochs = 1000 # number of times to loop through the entire dataset

    for epoch in range(epochs):
      for x, y_true in zip(data, all_y_trues):
        # --- Do a feedforward (we'll need these values later)
        sum_h_1_1 = self.w_1_1 * x[0] + self.w_1_2 * x[1] + self.b_1_1
        h_1_1 = sigmoid(sum_h_1_1)

        sum_h_1_2 = self.w_1_3 * x[0] + self.w_1_4 * x[1] + self.b_1_2
        h_1_2 = sigmoid(sum_h_1_2)

        sum_h_2_1 = self.w_2_1 * h_1_1 + self.w_2_2 * h_1_2 + self.b_2_1
        h_2_1 = sigmoid(sum_h_2_1)

        sum_h_2_2 = self.w_2_3 * h_1_1 + self.w_2_4 * h_1_2 + self.b_2_2
        h_2_2 = sigmoid(sum_h_2_2)

        sum_o_1 = self.w_3_1 * h_2_1 + self.w_3_2 * h_2_2 + self.b_3_1
        o_1 = sigmoid(sum_o_1)
        y_pred = o_1

        # --- Calculate partial derivatives.
        # --- Naming: d_L_d_w1 represents "partial L / partial w1"
```

```
d_L_d_ypred = −2 ∗ (y_true − y_pred)

d_ypred_d_h_1_1 = self.w_2_1 ∗ deriv_sigmoid(sum_h_2_1)
d_ypred_d_h_1_2 = self.w_2_2 ∗ deriv_sigmoid(sum_h_2_2)

# Neuron o1
d_ypred_d_w_3_1 = h_2_1 ∗ deriv_sigmoid(sum_o_1)
d_ypred_d_w_3_2 = h_2_2 ∗ deriv_sigmoid(sum_o_1)
d_ypred_d_b_3_1 = deriv_sigmoid(sum_o_1)

d_ypred_d_h_2_1 = self.w_3_1 ∗ deriv_sigmoid(sum_o_1)
d_ypred_d_h_2_2 = self.w_3_2 ∗ deriv_sigmoid(sum_o_1)


# Neuron h_1_1
d_h_1_1_d_w_1_1 = x[0] ∗ deriv_sigmoid(sum_h_1_1)
d_h_1_1_d_w_1_2 = x[1] ∗ deriv_sigmoid(sum_h_1_1)
d_h_1_1_d_b_1_1 = deriv_sigmoid(sum_h_1_1)

# Neuron h_1_2
d_h_1_2_d_w_1_3 = x[0] ∗ deriv_sigmoid(sum_h_1_2)
d_h_1_2_d_w_1_4 = x[1] ∗ deriv_sigmoid(sum_h_1_2)
d_h_1_2_d_b_1_2 = deriv_sigmoid(sum_h_1_2)

# Neuron h_2_1
d_h_2_1_d_w_2_1 = h_1_1 ∗ deriv_sigmoid(sum_h_2_1)
d_h_2_1_d_w_2_2 = h_1_2 ∗ deriv_sigmoid(sum_h_2_1)
d_h_2_1_d_b_2_1 = deriv_sigmoid(sum_h_2_1)

# Neuron h_2_2
d_h_2_2_d_w_2_3 = h_2_1 ∗ deriv_sigmoid(sum_h_2_2)
d_h_2_2_d_w_2_4 = h_2_2 ∗ deriv_sigmoid(sum_h_2_2)
d_h_2_2_d_b_2_2 = deriv_sigmoid(sum_h_2_2)

# −−− Update weights and biases
# Neuron h_1_1
```

```
      self.w_1_1 -= learn_rate * d_L_d_ypred * d_ypred_d_h_1_1 * d_h_1_1_d_w_1_1
      self.w_1_2 -= learn_rate * d_L_d_ypred * d_ypred_d_h_1_1 * d_h_1_1_d_w_1_2
      self.b_1_1 -= learn_rate * d_L_d_ypred * d_ypred_d_h_1_1 * d_h_1_1_d_b_1_1

      # Neuron h_1_2
      self.w_1_3 -= learn_rate * d_L_d_ypred * d_ypred_d_h_1_2 * d_h_1_2_d_w_1_3
      self.w_1_4 -= learn_rate * d_L_d_ypred * d_ypred_d_h_1_2 * d_h_1_2_d_w_1_4
      self.b_1_2 -= learn_rate * d_L_d_ypred * d_ypred_d_h_1_2 * d_h_1_2_d_b_1_2

      # Neuron h_2_1
      self.w_2_1 -= learn_rate * d_L_d_ypred * d_ypred_d_h_2_1 * d_h_2_1_d_w_2_1
      self.w_2_2 -= learn_rate * d_L_d_ypred * d_ypred_d_h_2_1 * d_h_2_1_d_w_2_2
      self.b_2_1 -= learn_rate * d_L_d_ypred * d_ypred_d_h_2_1 * d_h_2_1_d_b_2_1

      # Neuron h_2_2
      self.w_2_3 -= learn_rate * d_L_d_ypred * d_ypred_d_h_2_2 * d_h_2_2_d_w_2_3
      self.w_2_4 -= learn_rate * d_L_d_ypred * d_ypred_d_h_2_2 * d_h_2_2_d_w_2_4
      self.b_2_2 -= learn_rate * d_L_d_ypred * d_ypred_d_h_2_2 * d_h_2_2_d_b_2_2

      # Neuron o_1
      self.w_3_1 -= learn_rate * d_L_d_ypred * d_ypred_d_w_3_1
      self.w_3_2 -= learn_rate * d_L_d_ypred * d_ypred_d_w_3_2
      self.b_3_1 -= learn_rate * d_L_d_ypred * d_ypred_d_b_3_1

    # --- Calculate total loss at the end of each epoch
    if epoch % 10 == 0:
      y_preds = np.apply_along_axis(self.feedforward, 1, data)
      loss = mse_loss(all_y_trues, y_preds)
      print("Epoch %d loss: %.3f" % (epoch, loss))
  #write the last epoch weights and biases in text file
  with open('trained_weights_bias.txt','w') as f:
      f.write("%f, " % self.w_1_1)
      f.write("%f, " % self.w_1_2)
      f.write("%f\n" % self.b_1_1)

      f.write("%f, " % self.w_1_3)
```

```
        f.write("%f,␣" % self.w_1_4)
        f.write("%f\n" % self.b_1_2)

        f.write("%f,␣" % self.w_2_1)
        f.write("%f,␣" % self.w_2_2)
        f.write("%f\n" % self.b_2_1)

        f.write("%f,␣" % self.w_2_3)
        f.write("%f,␣" % self.w_2_4)
        f.write("%f\n" % self.b_2_2)

        f.write("%f,␣" % self.w_3_1)
        f.write("%f,␣" % self.w_3_2)
        f.write("%f\n" % self.b_3_1)

#------------------------------------
# Define dataset
#------------------------------------
data = np.array([
  [1, 2.5],    # Class 1
  [1, 3],      # Class 1
  [2.1, 3.4],  # Class 1
  [2.1, 1], # Class 0
  [3.3, 1], # Class 0
  [3, 2.3], # Class 0
])
all_y_trues = np.array([
  1, # Class 1
  1, # Class 1
  1, # Class 1
  0, # Class 0
  0, # Class 0
  0, # Class 0
])

# Train our neural network!
```

```
network = OurNeuralNetwork()
network.train(data, all_y_trues)
```

B.2 Code for Adaptive Simple Feed Forward Neural Network

Load and retrain Simple Feed Forward Neural Network on two classes with 4 data points each.

```
'''
  Program  : Adaptive Feed Forward Network;
  Version  : 1.0;
  Date     : Sept. 9, 2020
  Coded by : (see the reference below)
  Modified by: HL  and RB for adapative learning
Ref: https://github.com/vzhou842/neural-network-from-scratch/blob/master/network.py
'''

import numpy as np

def sigmoid(x):
  # Sigmoid activation function: f(x) = 1 / (1 + e^(-x))
  return 1 / (1 + np.exp(-x))

def deriv_sigmoid(x):
  # Derivative of sigmoid: f'(x) = f(x) * (1 - f(x))
  fx = sigmoid(x)
  return fx * (1 - fx)

def mse_loss(y_true, y_pred):
  # y_true and y_pred are numpy arrays of the same length.
  return ((y_true - y_pred) ** 2).mean()

class OurNeuralNetwork:
  '''
  A neural network with:
    - 2 inputs
```

```
  - 2 hidden layer with 2 neurons each(h_1_1, h_1_2, h_2_1, h_2_2)
  - an output layer with 1 neuron (o_1)

*** DISCLAIMER ***:
The code below is intended to be simple and educational, NOT optimal.
Real neural net code looks nothing like this. DO NOT use this code.
Instead, read/run it to understand how this specific network works.
'''

def __init__(self):
  # load prior Weights and biases from a text file
  s=[]
  with open('trained_weights_bias.txt','r') as f:
      for line in f:
          for num in line.split(','):
              s.append(float(num))
  self.w_1_1=s[0]
  self.w_1_2=s[1]
  self.b_1_1=s[2]

  self.w_1_3=s[3]
  self.w_1_4=s[4]
  self.b_1_2=s[5]

  self.w_2_1=s[6]
  self.w_2_2=s[7]
  self.b_2_1=s[8]

  self.w_2_3=s[9]
  self.w_2_4=s[10]
  self.b_2_2=s[11]

  self.w_3_1=s[12]
  self.w_3_2=s[13]
  self.b_3_1=s[14]
```

```
def feedforward(self, x):
  # x is a numpy array with 2 elements.
  h_1_1 = sigmoid(self.w_1_1 * x[0] + self.w_1_2 * x[1] + self.b_1_1)
  h_1_2 = sigmoid(self.w_1_3 * x[0] + self.w_1_4 * x[1] + self.b_1_2)

  h_2_1 = sigmoid(self.w_2_1 * h_1_1 + self.w_2_2 * h_1_2 + self.b_2_1)
  h_2_2 = sigmoid(self.w_2_3 * h_1_1 + self.w_2_4 * h_1_2 + self.b_2_2)

  o_1 = sigmoid(self.w_3_1 * h_2_1 + self.w_3_2 * h_2_2 + self.b_3_1)
  return o_1


def train(self, data, all_y_trues):
  '''
  - data is a (n x 2) numpy array, n = # of samples in the dataset.
  - all_y_trues is a numpy array with n elements.
    Elements in all_y_trues correspond to those in data.
  '''
  learn_rate = 0.1
  epochs = 1000 # number of times to loop through the entire dataset

  for epoch in range(epochs):
    for x, y_true in zip(data, all_y_trues):
      # --- Do a feedforward (we'll need these values later)
      sum_h_1_1 = self.w_1_1 * x[0] + self.w_1_2 * x[1] + self.b_1_1
      h_1_1 = sigmoid(sum_h_1_1)

      sum_h_1_2 = self.w_1_3 * x[0] + self.w_1_4 * x[1] + self.b_1_2
      h_1_2 = sigmoid(sum_h_1_2)

      sum_h_2_1 = self.w_2_1 * h_1_1 + self.w_2_2 * h_1_2 + self.b_2_1
      h_2_1 = sigmoid(sum_h_2_1)

      sum_h_2_2 = self.w_2_3 * h_1_1 + self.w_2_4 * h_1_2 + self.b_2_2
      h_2_2 = sigmoid(sum_h_2_2)
```

```
sum_o_1 = self.w_3_1 * h_2_1 + self.w_3_2 * h_2_2 + self.b_3_1
o_1 = sigmoid(sum_o_1)
y_pred = o_1

# --- Calculate partial derivatives.
# --- Naming: d_L_d_w1 represents "partial L / partial w1"
d_L_d_ypred = -2 * (y_true - y_pred)

d_ypred_d_h_1_1 = self.w_2_1 * deriv_sigmoid(sum_h_2_1)
d_ypred_d_h_1_2 = self.w_2_2 * deriv_sigmoid(sum_h_2_2)

# Neuron o_1
d_ypred_d_w_3_1 = h_2_1 * deriv_sigmoid(sum_o_1)
d_ypred_d_w_3_2 = h_2_2 * deriv_sigmoid(sum_o_1)
d_ypred_d_b_3_1 = deriv_sigmoid(sum_o_1)

d_ypred_d_h_2_1 = self.w_3_1 * deriv_sigmoid(sum_o_1)
d_ypred_d_h_2_2 = self.w_3_2 * deriv_sigmoid(sum_o_1)



# Neuron h_1_1
d_h_1_1_d_w_1_1 = x[0] * deriv_sigmoid(sum_h_1_1)
d_h_1_1_d_w_1_2 = x[1] * deriv_sigmoid(sum_h_1_1)
d_h_1_1_d_b_1_1 = deriv_sigmoid(sum_h_1_1)

# Neuron h_1_2
d_h_1_2_d_w_1_3 = x[0] * deriv_sigmoid(sum_h_1_2)
d_h_1_2_d_w_1_4 = x[1] * deriv_sigmoid(sum_h_1_2)
d_h_1_2_d_b_1_2 = deriv_sigmoid(sum_h_1_2)

# Neuron h_2_1
d_h_2_1_d_w_2_1 = h_1_1 * deriv_sigmoid(sum_h_2_1)
d_h_2_1_d_w_2_2 = h_1_2 * deriv_sigmoid(sum_h_2_1)
d_h_2_1_d_b_2_1 = deriv_sigmoid(sum_h_2_1)
```

```
# Neuron h_2_2
d_h_2_2_d_w_2_3 = h_1_1 * deriv_sigmoid(sum_h_2_2)
d_h_2_2_d_w_2_4 = h_1_2 * deriv_sigmoid(sum_h_2_2)
d_h_2_2_d_b_2_2 = deriv_sigmoid(sum_h_2_2)

# --- Update weights and biases

# Neuron h1
self.w_1_1 -= learn_rate * d_L_d_ypred * d_ypred_d_h_1_1 * d_h_1_1_d_w_1_1
self.w_1_2 -= learn_rate * d_L_d_ypred * d_ypred_d_h_1_1 * d_h_1_1_d_w_1_2
self.b_1_1 -= learn_rate * d_L_d_ypred * d_ypred_d_h_1_1 * d_h_1_1_d_b_1_1

# Neuron h2
self.w_1_3 -= learn_rate * d_L_d_ypred * d_ypred_d_h_1_2 * d_h_1_2_d_w_1_3
self.w_1_4 -= learn_rate * d_L_d_ypred * d_ypred_d_h_1_2 * d_h_1_2_d_w_1_4
self.b_1_2 -= learn_rate * d_L_d_ypred * d_ypred_d_h_1_2 * d_h_1_2_d_b_1_2

# Neuron h3
self.w_2_1 -= learn_rate * d_L_d_ypred * d_ypred_d_h_2_1 * d_h_2_1_d_w_2_1
self.w_2_2 -= learn_rate * d_L_d_ypred * d_ypred_d_h_2_1 * d_h_2_1_d_w_2_2
self.b_2_1 -= learn_rate * d_L_d_ypred * d_ypred_d_h_2_1 * d_h_2_1_d_b_2_1

# Neuron h4
self.w_2_3 -= learn_rate * d_L_d_ypred * d_ypred_d_h_2_2 * d_h_2_2_d_w_2_3
self.w_2_4 -= learn_rate * d_L_d_ypred * d_ypred_d_h_2_2 * d_h_2_2_d_w_2_4
self.b_2_2 -= learn_rate * d_L_d_ypred * d_ypred_d_h_2_2 * d_h_2_2_d_b_2_2

# Neuron o1
self.w_3_1 -= learn_rate * d_L_d_ypred * d_ypred_d_w_3_1
self.w_3_2 -= learn_rate * d_L_d_ypred * d_ypred_d_w_3_2
self.b_3_1 -= learn_rate * d_L_d_ypred * d_ypred_d_b_3_1



# --- Calculate total loss at the end of each epoch
if epoch % 10 == 0:
```

```
        y_preds = np.apply_along_axis(self.feedforward, 1, data)
        loss = mse_loss(all_y_trues, y_preds)
        print("Epoch %d loss: %.3f" % (epoch, loss))


#-----------------------------------
# Define dataset
#-----------------------------------
data = np.array([
  [1, 2.5],     # Class 1
  [2, 2.3],     # Class 1
  [1, 3],       # Class 1
  [2.1, 3.4],   # Class 1
  [2.1, 1],     # Class 2
  [3, 2.1],     # Class 2
  [3.3, 1],     # Class 2
  [3, 2.3]      # Class 2
])
all_y_trues = np.array([
  1, # Class 1
  1, # Class 1
  1, # Class 1
  1, # Class 1
  0, # Class 0
  0, # Class 0
  0, # Class 0
  0, # Class 0
])

# Train our neural network!
network = OurNeuralNetwork()
network.train(data, all_y_trues)
```

B.3 Code for FaceNet

Train and save FaceNet on three classes with 10 data points each.

```
with sess.as_default():
        if args.pretrained_model:
            print('Restoring pretrained model: %s' % args.pretrained_model)
        # Training and validation loop
        epoch = 0
        while epoch < args.max_nrof_epochs:
            step = sess.run(global_step, feed_dict=None)
            epoch = step // args.epoch_size
            # Train for one epoch
            train(args, sess, train_set, epoch, image_paths_placeholder, labels_placeholder, labels_batch,
                batch_size_placeholder, learning_rate_placeholder, phase_train_placeholder, enqueue_op, input_queue,
      global_step,
                embeddings, total_loss, train_op, summary_op, summary_writer, args.learning_rate_schedule_file,
                args.embedding_size, anchor, positive, negative, triplet_loss)

            # Save variables and the metagraph if it doesn't exist already
            save_variables_and_metagraph(sess, saver, summary_writer, model_dir, randir, step)
def train(..)
..
    stop_threshold = 1.4
        while i < nrof_batches:
            start_time = time.time()
            batch_size = min(nrof_examples-i*args.batch_size, args.batch_size)
            feed_dict = {batch_size_placeholder: batch_size, learning_rate_placeholder: lr, phase_train_placeholder:
        True}
            err, _, step, emb, lab = sess.run([loss, train_op, global_step, embeddings, labels_batch], feed_dict=feed_dict)
            #RB 09-25-2020
            if err < stop_threshold:
                break
            emb_array[lab,:] = emb
            loss_array[i] = err
            duration = time.time() - start_time

            print('Epoch: [%d][%d/%d]\tTime %.3f\tLoss %2.3f' %
                (epoch, batch_number+1, args.epoch_size, duration, err))
```

```
        batch_number += 1
        i += 1
        train_time += duration
        summary.value.add(tag='loss', simple_value=err)
..
```

B.4 Code for Adaptive FaceNet

Load and retrain FaceNet on three classes with 20 data points each.

```
with sess.as_default():
        if args.pretrained_model:
            print('Restoring pretrained model: %s' % args.pretrained_model)
            facenet.load_model(args.pretrained_model)
            # Get input and output tensors
            image_paths_placeholder = tf.get_default_graph().get_tensor_by_name("image_paths:0")
            embeddings = tf.get_default_graph().get_tensor_by_name("embeddings:0")
            phase_train_placeholder = tf.get_default_graph().get_tensor_by_name("phase_train:0")
            embedding_size = embeddings.get_shape()[1]
            #saver.restore(sess, os.path.expanduser(args.pretrained_model))
        # Training and validation loop
        epoch = 0
        while epoch < args.max_nrof_epochs:
            step = sess.run(global_step, feed_dict=None)
            epoch = step // args.epoch_size
            # Train for one epoch
            train(args, sess, train_set, epoch, image_paths_placeholder, labels_placeholder, labels_batch,
                batch_size_placeholder, learning_rate_placeholder, phase_train_placeholder, enqueue_op, input_queue,
      global_step,
                embeddings, total_loss, train_op, summary_op, summary_writer, args.learning_rate_schedule_file,
                args.embedding_size, anchor, positive, negative, triplet_loss)

            # Save variables and the metagraph if it doesn't exist already
            save_variables_and_metagraph(sess, saver, summary_writer, model_dir, randir, step)
def train(..)
...
    stop_threshold = 1.4
```

```
while i < nrof_batches:
    start_time = time.time()
    batch_size = min(nrof_examples−i∗args.batch_size, args.batch_size)
    feed_dict = {batch_size_placeholder: batch_size, learning_rate_placeholder: lr, phase_train_placeholder:
True}
    err, _, step, emb, lab = sess.run([loss, train_op, global_step, embeddings, labels_batch], feed_dict=feed_dict)
    #RB 09−25−2020
    if err < stop_threshold:
        break
    emb_array[lab,:] = emb
    loss_array[i] = err
    duration = time.time() − start_time

    print('Epoch:␣[%d][%d/%d]\tTime␣%.3f\tLoss␣%2.3f' %
            (epoch, batch_number+1, args.epoch_size, duration, err))
    batch_number += 1
    i += 1
    train_time += duration
    summary.value.add(tag='loss', simple_value=err)
...
```