Mathematics for AI Term Project

Face Recognition System

Group Members	ID
1. Dawit Getahun	UGR/9220/13
2. Metsakal Zeleke	UGR/1027/13
3. Tewodros Berhanu	UGR/9715/13
4. Tinsae Shemalise	UGR/6075/13
5. Yohannes Dessie	UGR/7612/13

Report on our Face Recognition System

Introduction

In this assignment, we tried to build our own version of a Face Recognition System for our class. The goal was to build a mobile app that relied on a model that could accurately recognize each of our classmates. Through the process of building this system we have been able to look at a few methods of face recognition:

- Siamese networks
- Eigenfaces
- Support Vector Machine(SVM) based algorithms

After training and testing, we selected the best model and created an API to serve as an interface for our model.

I) Siamese Network

A Siamese Network at its core is just a neural network that uses multiple neural networks of the same weight, i.e. redundant identical neural networks, to compare the features of different inputs. What it essentially does is during training, instead of accepting a single input at a time it accepts 2 or 3 inputs then it passes those inputs through the same network individually, after that it checks to see if the inputs are closely represented (if they are similar) or differently represented (if they aren't similar). It uses the difference of output between the multiple inputs and uses it to learn how to properly represent the inputs, i.e. encode them. This very nature of the Siamese Network helps us to encode images, voice, signatures, and other things so that identification could be as simple as computing a cosine similarity between different encodings done by the network.

There are generally 2 ways of training a Siamese Network:

- Using a Contrastive loss: uses pairs of inputs and a binary label indicating whether they belong to the same class or not. Then the loss function tries to minimize the distance between similar inputs and maximize the distance between dissimilar inputs. This is generally used to train a network to say whether two inputs are the same or not.

- Using a Triplet loss: uses three sets of inputs: an anchor, a positive, and a negative. While the anchor and the positive belong to the same class the

negative belongs to a different class. Again the loss function tried to minimize the distance between the anchor and the positive input while maximizing the distance between the anchor and negative input.

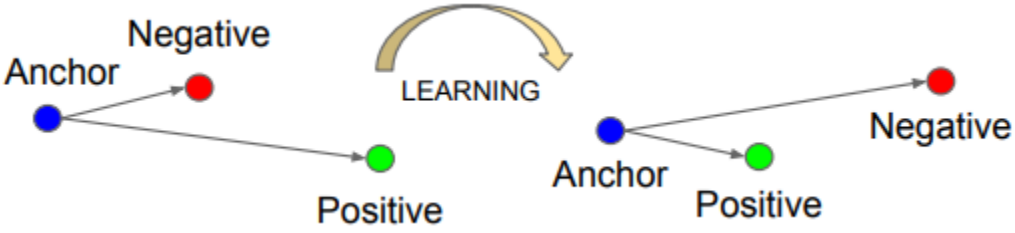

Figure 1. Triplet Loss minimizes the distance between the anchor and the positive and maximizes the distance between the anchor and the negative

So for our intended purpose, we think that using the second method is better. To accomplish this we read and followed a paper called **FaceNet: A Unified Embedding for Face Recognition and Clustering** [1] written in 2015. It shows how one can build an accurate face recognition model(system) using a Siamese Network with a triplet loss.

Reading through this paper we found that there are 2 different ways of choosing our Anchor, Positive, and Negative.

- **Hard Off-line Learning**: The idea is to choose triplets that are hard, meaning that the distance between the anchor and the positive is close to or larger than the distance between the anchor and the negative. This way, the model is forced to learn more discriminative features and reduce the intra-class variance. There are a couple of problems with this:
    - It is infeasible to compute the arg-min(most closest) and arg-max(most furthest) across the whole training set.
    - Additionally, it might lead to poor training, as mislabelled and poorly imaged faces would dominate the hard positives and negatives.

- **Semi-Hard Online Learning**: This method selects triplets where the anchor-positive distance is smaller than the anchor-negative distance, but the negative is still within the margin of the loss function. This ensures that the triplets are not too easy or too hard, and avoids the issue of slow convergence or bad local optima.
  As shown in the paper, the best results are from these kinds of triplets. And to find these efficiently we could utilize online learning and only train from the Semi-Hard examples in each batch.

Now we have a basic understanding of how to build this network and how to train it we went straight to writing the code. We created a simple pipeline for preparing our data,

which just resizes the images to 200 by 200 and then we use **cv2** to create a tight face crop, i.e. we create a tight box around the faces and then crop them. We did this because the images collected weren't all of the same shape and cropped properly. After that, we built a simple neural network described below:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 200, 200, 64)      832

 max_pooling2d (MaxPooling2  (None, 100, 100, 64)      0
 D)

 dropout (Dropout)           (None, 100, 100, 64)      0

 conv2d_1 (Conv2D)           (None, 100, 100, 32)      8224

 max_pooling2d_1 (MaxPoolin  (None, 50, 50, 32)        0
 g2D)

 dropout_1 (Dropout)         (None, 50, 50, 32)        0

 flatten (Flatten)           (None, 80000)             0

 dense (Dense)               (None, 256)               20480256

 lambda (Lambda)             (None, 256)               0

=================================================================
Total params: 20489312 (78.16 MB)
Trainable params: 20489312 (78.16 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 2. Structure of our Neural Network

For handling the triplet loss selection, we used TensorFlow addons which is a library of extensions for TensorFlow that provide additional functionality not available in the core TensorFlow package.

We trained our neural net for 15 epochs and then we were now able to encode our images into 256 dimension vectors. So we utilized this to see how well our model can encode and cluster faces from the same people. We encoded every picture and stored it in a tsv(tab-separated value) file so that we could visualize it using a tool called **Embedding Projector** [2] and used **UMAP** (which is a dimension reduction technique that uses geometry and topology to map high-dimensional data onto a lower-dimensional manifold) and got the following visualization of the encoded images.

PS:- After implementing the application, we have noticed the model tends to do well on pictures taken with the back camera rather than a selfie picture. In hindsight, it would have been interesting to horizontally flip the images in the process and try it.

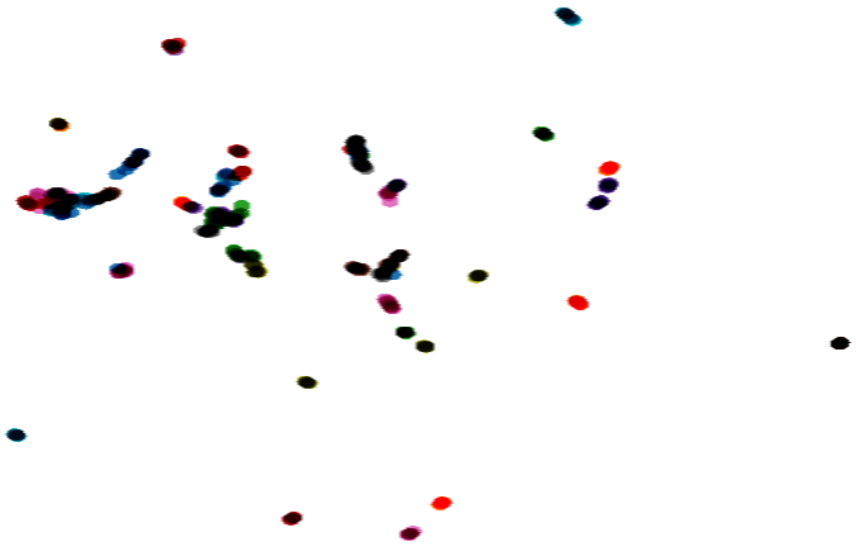

Figure 3. Visualization of the encoding images after UMAP dimensionality reduction

Here we can see that there is a good grouping(clustering) nature of the encoded images. Note that the images are colored based on the cluster they belong to. So stemming from this visualization we concluded that even though the encodings aren't terrible they aren't the best to perform cosine similarity on to find the best possible match for a new image submitted for face detection. So we used these embeddings to then train an SVM so that instead of simply using a cosine similarity this would be of help.

We trained an SVM classifier using **sklearn** with a linear, polynomial, and RBF kernel. Then we used grid search to find the best kernel, also a sklearn for model selection to choose the best-performing kernel, which was the polynomial kernel with a test accuracy of 97.11%.

Now the thing left to do was to test our model on new images taken by our phones. So we took images and then passed it to the preprocessing pipeline to reshape and to create a tight crop around the faces. Finally, when we ran the test we got an accuracy of 36.7%. Which is a very bad score considering what we got on our test split. We assumed it was from the way we took the photos like facial expressions and lighting but, no, the results were unchanged. So the only thing we have left to do is to try different architectures before moving on to the next technique. Hence we tried adding and using pre-trained models like ResNet50, InceptionV3, and VGG16 on top of our model but the results were almost the same: they had high accuracy scores for the test split but when

we tried them with new pictures taken by our phones we got around 30-37% accuracy which felt unacceptable.

We wanted to understand why this was happening but the reasons for this seem to be beyond us because we have made sure that our model didn't overfit by having a good accuracy result on our testing split; we also made sure that we didn't skip and preprocessing steps when we identified new images. But still, the problem persisted. So we decided to move on to the second method we had in mind.

II) Eigenfaces approach with Principal Component Analysis

Eigenfaces is a more traditional approach compared to the Siamese network we discussed above. It is based on a dimensionality reduction technique(PCA) that focuses on selecting the most important eigenvectors of the dataset and creating a projection of the data points onto the eigenvectors to create an eigen-space.

A bit about the data: We utilized the OpenCV python library's face cascade classifier to get the bounding boxes of the faces in the data to construct a new dataset with the faces cropped. And then performed an 80-20 train-test split on the data.

We performed the necessary steps to perform the Eigen-Decomposition, and then we selected the K most important eigenvectors to construct our eigenspace making sure we preserved 99% of the variance within selected vectors. In the process, we were able to see that 99% of the variance was captured by only 17 out of 423 vectors (4.02%).

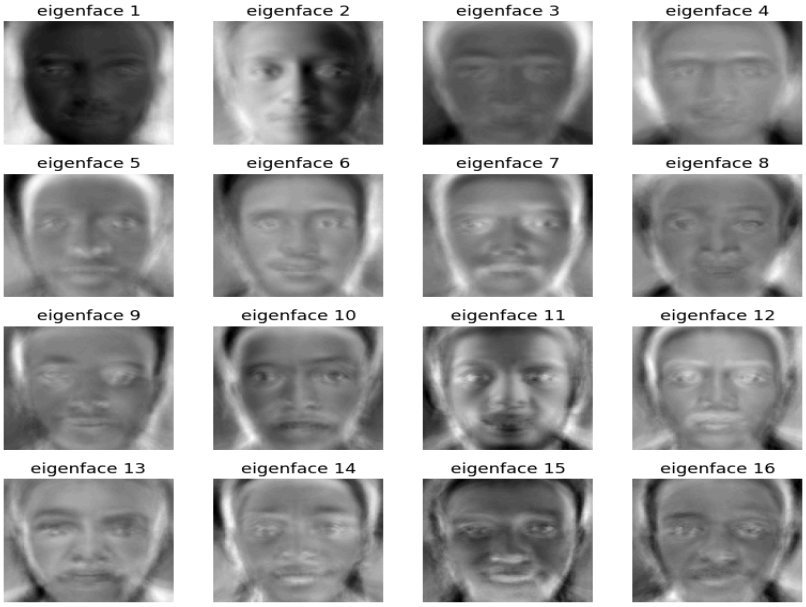

Fig: 16 out of 17 eigenfaces projected from the selected vectors as visualized here, in order from most important to list important

Note: One thing to note here, the described eigenfaces in the figure seem to share more similarities with male faces rather than female faces. And that is well in line with the male-female proportion in our classroom.

The performance of the eigenfaces algorithm on the testing set (i.e. from the 80-20 split) was a decent 88.78%. But what is more important is getting the algorithm to be efficient at generalizing outside of the dataset. So we picked a few pictures of ourselves and tested them. And mostly it showed that there wasn't enough separation between classes.

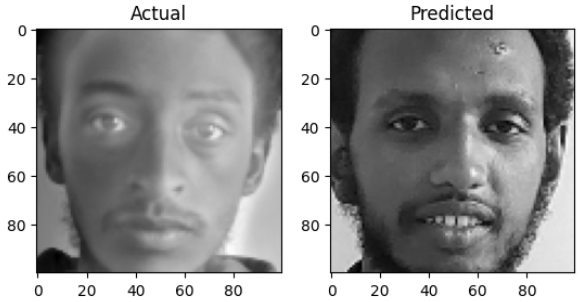

Fig: An example of a picture taken of one of our members(left) with the picture from the dataset that was predicted as most similar to(right)

So, to increase our chances of generalizing well outside of the dataset, we had to look elsewhere. And that is where we get to our next trial.

III) Support Vector Machine (SVM) with Principal Component Analysis

Support Vector Machines construct a hyperplane or set of hyperplanes in a high or infinite-dimensional space, which can be used for classification, regression, or other tasks like outliers detection [wikipedia]. SVMs are for the most part developed with the inception of maximizing the margin between classes.

The paper **Face Recognition with Support Vector Machines: Global versus Component-based Approach,** by B Heisele, P Ho, T Poggio, 2001 [link], gives a rundown of the steps to use this approach for face recognition. The paper does a comparison of the two approaches: global versus component-based. However, due to time constraints, we were only able to implement the global approach it outlined.

From the global approach the paper discussed, to get the face detection right, we have used an implementation from the OpenCV library as mentioned above. For the classification, it is calculated in a difference-space for two approaches: one-versus-all

and one-versus-one (pairwise). One-vs-all requires *q* support vector classifiers for *q* classes, while the pairwise approach needs *q(q-1)/2* classifiers to be trained. The paper tells us the classification performance of both approaches is very similar and suggests the use of the one-vs-all approach for it requires a lesser number of classifiers.

One-vs-All Decision:

- The final decision is essentially a voting mechanism, where each binary classifier "votes" for its assigned class.
- The class with the highest number of votes or the highest confidence score is chosen as the predicted class for the multiclass problem.

So, essentially we figured we could get a better result by using the Principal Component Analysis we showed earlier as an input for the classification with the one-vs-all approach for SVM. And we did. The model performed better at 89.62% accuracy on our initial 80-20 train-test split and also showed better performance on images picked from outside the dataset.

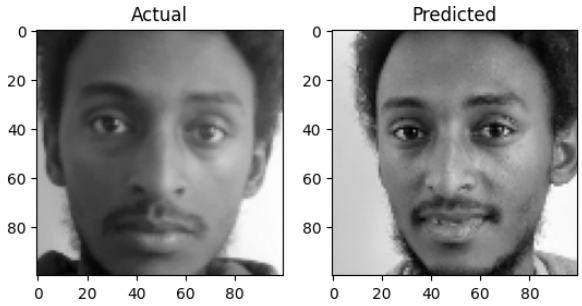

Fig: better performance than eigenfaces on image picked from outside the dataset.

In total, the SVM multi-class classifier was tested on 29 pictures picked from outside of the dataset and it was able to identify 21 of them correctly.