Biometric Systems project
DL based iris feature extractor trained with few-shot learning

Eduardo Rinaldi 1797800

Contents

1 Introduction

For my bachelor's thesis I decided to work on an iris recognition system which relies on IS_{IS} $v.2$ method for segmentation (Iris segmentation for Identification Systems) and both LBP and $Spatiogram$ operators for feature extraction. For this project instead I decided to circle back and try other approaches regarding iris recognition (for both segmentation and feature extraction).

1.1 Pipeline
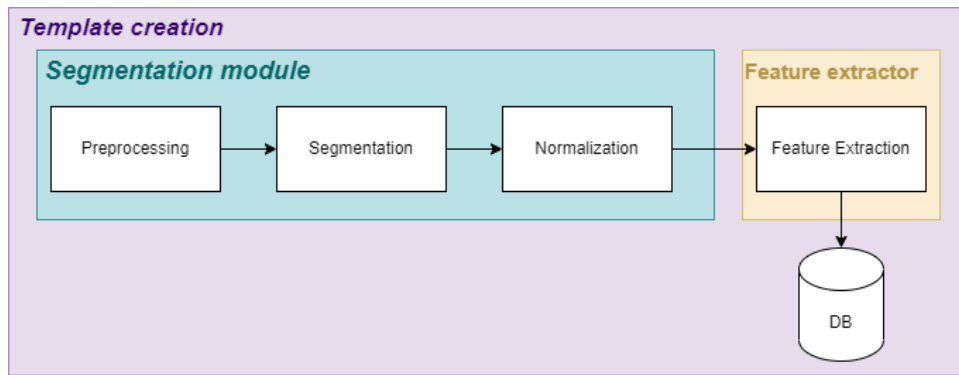
Figure 1: Main pipeline

The pipeline is the *"classical"* one, I just subdivided it in easy to change submodules:

- **Segmentation**: 2 different segmentation modules, one which is IS_{IS} $v.2$ and another based on a slightly different approach that will be discussed later, for simplicity I'll name it in both this report and code as **"Hough approach"**. I decided to use only this latter for time limitation (obviously in the code I put both modules).

- **Feature Extraction**: 4 different feature extractors (only one is used in the demo, so no ensemble) all based on **deep learning approaches**; the main difficulty here lies in the training part because as will be explained in next section the dataset used is relatively "small". Since I have decided to spend most of my time on this part and the approaches tested are quite "recent" and "unusual", the project is more focused on feature extraction.

2 Technologies and libraries

- **Languages**:

2

- `C++17` for segmentation, providing CMake file for compiling code as both executable and static library (in case someone want to use the segmentation module as an external library).

- `Python 3.9.4` for the whole pipeline and demo; I created a tiny API for calling the segmentation compiled executable from python.

- `IPython notebook` (hosted on *Google Colab*), I provide python notebook that has been used for creating the feature extractor models. In this notebook all the steps are explained in details.

- **Libraries**: OpenCV, PyTorch, Pandas, Numpy

In the section **"How to test the pipeline"** are explained all the steps needed to run each module and the demo.

3 Dataset

Due to limited time I decided to train and test the whole project on only one dataset which is **Utiris** [1]. The database is constructed with 1540 images from 79 individuals from both right and left eyes demonstrated in 158 classes in total. The individuals are numbered the same in NIR and VW sessions, but for the feature extraction I only used VW images.

3.1 Train - test split

I decided to split the dataset giving **75% of the subjects for training set** and the remaining **25% for testing set**. Notice here that I **split by subjects** not by probes, so the final models will be evaluated on subjects they never seen (overfit can be easily spotted); the reason behind this choice lies in the training approach, but it will be explained and discussed in the *"Feature extractors"* section.

4 Segmentation modules

As said before, segmentation modules provided here are two, but the code has been structured in such a way that I can implement a new one and then use it with rest of the pipeline. Indeed, these segmentation modules can be seen as a function that takes an image cropped on an eye and returns **two circles**: one for the **limbus** and one for the **pupil**, so in each module we're "describing" how to find these two circles. They share a common part which consists in:

1. **Preprocessing**: this part is needed for **speeding up computation time** and for **removing reflections**. First, I find ROI (Region of interest) of the eye using an Haar Cascade classifier obtaining a "squared" image, then I scale the image to a fixed size (in my case 250x250).

2. **Normalization**: given as input limbus and pupil circle to normalization module we obtain an image normalized following the rubber sheet model. (Example of normalized iris in fig 2)
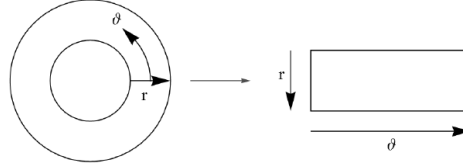
(a) Rubber sheet model

(b) Normalized iris

Figure 2: Iris normalization

4.1 $IS_{IS}v2$

The main idea behind this module is to find **first the limbus** and **then the pupil** using a **"bruteforce" approach**. For bruteforce here I mean that every time the algorithm try to find one of the two circles it first generates all the possible ones using different parameters and then select the best circle according to a specific metric. So we can subdivide the "circle finding" in two main big steps: *"circle candidates generation"* and *"circle selection"*.

4.1.1 Circle candidates generation

In this step, in order to increase the contrast in the image, a **posterization filter** is applied to the preprocessed image. The filter consists in:

1. Convert image in gray scale

2. A window W of size $(2k + 1) \times (2k + 1)$ is scrolled over the entire image, and at each scrolling a histogram H_W is calculated on the pixels inside W

3. The most frequent colour within H_W is assigned to the central pixel of W

At every iteration, k is increased (ranging in $[1, 17]$).

After this "preprocessing" step a list of edges are detected using **"Canny edge detector"** and these are then given as input to a circle fitting algorithm (for this module is the **Taubin** method) which will output a list of candidates circles.

4.1.2 Circle selection

This step receives as input a list containing all the candidates circles obtained after each iteration of the previous step and returns only one circle based on some criteria:

- For limbus circle we have the following constraints:

    - $size_{img} * 0.15 \leq radius_{limbus} \leq size_{img} * 0.5$
    - must have the higher score (based on **homogeneity** and **separability**)

- For pupil circle we have the following constraints:

    - $diameter_{limbus} * 0.1 \leq radius_{pupil} \leq diameter_{limbus} * 0.2$
    - must be inside the limbus circle
    - must have the higher score (based on **homogeneity** and **separability**)

4.2 Hough approach

This approach is a custom one created by mixing some of the approaches used in IS_{IS} and some of the ones used here (this latter is thought to work well on infrared images).

The main idea here is to find **first the pupil** and **then the limbus** still using a "bruteforce" approach based on *"circle candidates generation"* and *"circle selection"* steps.

4.2.1 Circle candidates generation

This step starts by transforming the input image in gray scale for then apply a **median blur filter** (at each iteration the size of the window used by the filter is increased), this helps reducing noise in the image.

Then as in IS_{IS}, I use **canny edge detector** for obtaining circle edges.

When the algorithm is looking for pupil circles I apply the following "trick": since pupil circle is composed by very dark pixels, I can inverse threshold the blurred image with small threshold values (at each iteration this threshold is increased) for then giving the resulted image to the canny edge detector; this latter in this case will have more chances to detect edges of pupil.

Figure 3: From left to right: blurred image, thresholded image, edge detected by canny

Instead when the algorithm is looking for limbus circle, the edge detector receives as input the blurred image.

Final step consists in giving the *"edge map"* (i.e. the output of canny) as input to **"Hough circle transform"** which will give as output a list of all possible circles (something "similar" to **Taubin** circle fitting). At each iteration we decrease the *"param2"* parameter of Hough transform, which is the accumulator threshold, the smaller it is the more false circles may be detected (it ranges in $[35, 120]$).

4.2.2 Circle selection

This step for **pupil** is very simple: just take the mean circle (i.e. mean position and mean radius).

For **limbus** a filtering process is applied:

1. every circle with $radius_{limbus} < 1.5 * radius_{pupil}$ is discarded from the "candidates" list.

2. every circle positioned outside pupil circle is discarded from the "candidates" list.

3. mean circle μ and std circle σ are calculated, then every circle with position or radius outside the range $[\mu - 1.5\sigma, \mu + 1.5\sigma]$ is discarded from the "candidates" list.

At the end of this filtering process, as for pupil, the *new* mean circle among the candidates is taken.

5 Feature extractors

Since the project focuses more on feature extraction and since the methods proposed are fairly recent and therefore seem to be underused, this section will be the one with the most detail.

All code concerning this section and the evaluation phase can be found in the python notebook `"Iris Feature Extraction.ipynb"`, together with the code various explanations of each step are given.

5.1 Preface

Initially, the idea was to use pretrained networks such as *vgg*, *densenet* and *resnet* and then perform a fine-tuning operation for the proposed dataset. This idea was immediately catalogued as a failure for two main reasons:

- for each class there are very **few examples** (about 5), not enough even for a small fine-tuning process

- if a new subject is added to gallery we need to retrain the whole model (this occurs only if these models are used as "end" classifiers)

This was the starting point for my research into possible "training techniques" that could fit well with my needs: it was here that I came across two "techniques" called ***"few-shot learning"*** and ***"siamese networks"***.

5.1.1 Few-shot learning

Few-shot learning is the problem of making predictions based on a limited number of samples. The goal here is not to let the model recognize the images in the training set and then generalize to the test set, instead, the goal is to learn (*"learn to learn"*).

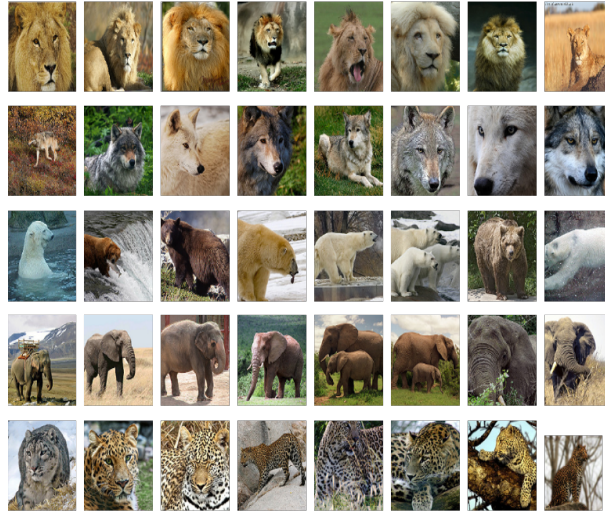Let's do an example: let's suppose that we have as training set the images in figure 4

Figure 4: Example training set

The goal of training is not to know what an elephant is and what a tiger is but instead, the goal is to **know the similarity and difference between objects**. So, later, I can give two images of animals that never appeared in the training set and the model can say if they are the same animal (it learns a similarity/distance measure).

Figure 5: Test pair

Let's now go back to our example; when the model sees the pair of images in figure 5 it does not know they are squirrels (we didn't train the model to recognize the class). However, the model can tell you with high confidence that they are the same kind of objects. This technique seems to be **well suited for biometric recognition models**.

5.1.2 Siamese Networks

A possible implementation of few-shot learning is through to the concept of Siamese networks [2], which consists of twin networks (generally two or three) which accept distinct inputs but are joined by an energy function and they share the same weights, this guarantees that two extremely similar images are not mapped by each network to very different locations in feature space because each network computes the same function. After the training phase we can then extract the network used at the base, i.e. the one that produces the feature vector, and use it as a "feature extractor" (I will use this term in the next sections to refer to the network at the base).

For this project I decided to try two different architectures of siamese networks: **pair siamese networks** and **triplet siamese network**, both based on conv. neural networks.

Pair siamese networks During training, as the name suggests, an image pair is fed into the model with their ground truth relationship y: $y = 1$ if the two images are similar and 0 otherwise. The loss function for a single pair is the *contrastive loss*:

$$yd^2 + (1 - y)max(m - d, 0)^2$$

where:

- d is the euclidean distance between the two feature vectors obtained by the network

- m is called margin, is used to "tighten" the constraint: if two images in a pair are dissimilar, then their distance should be at least m, or a loss will be incurred
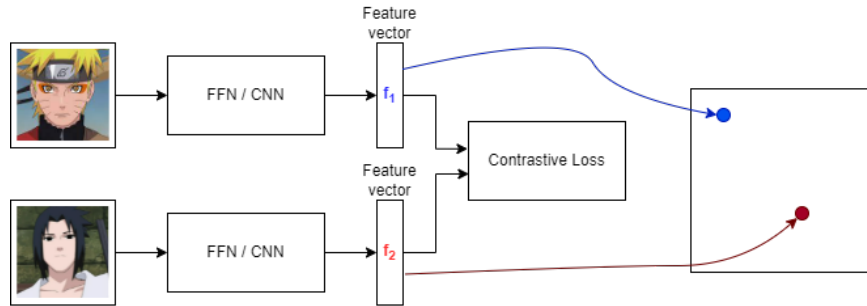

Figure 6: Example of pair siamese network

Triplet siamese networks   During training process, an image triplet $< I_a, I_p, I_n >$ is fed into the model as a single sample, where I_a, I_p and I_n represent the anchor, postive and negative images respectively. The idea behind is that distance between anchor and positive images should be smaller than that between anchor and negative images. The loss function used is called *triplet loss*:

$$max(\|f_a - f_p\|^2 - \|f_a - f_n\|^2 + m, 0)$$
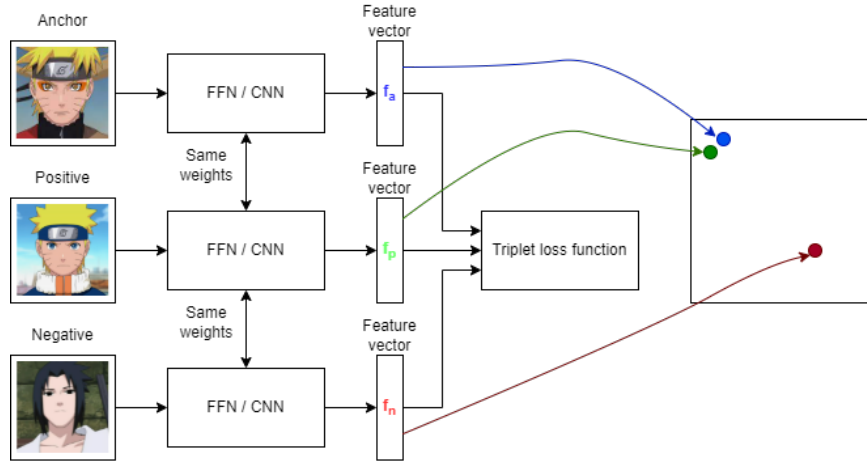

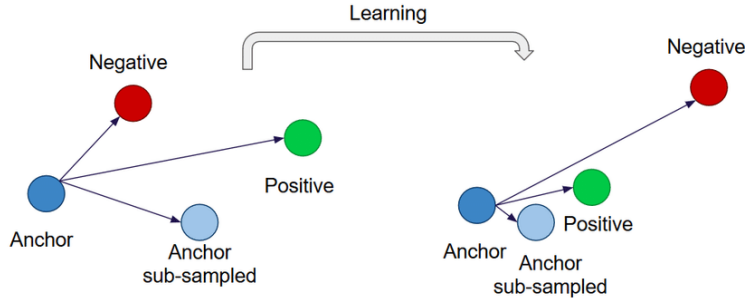Figure 7: Example of triplet siamese network

Figure 8: Triplet loss

5.2   Trained and tested models

Now that I have explained, although in a very simple way, the underlying mechanism, I can list the feature extractors trained and tested:

1. `vggFEPair`: pretrained model of vgg fine-tuned using pair few-shot learning (pair siamese network)

2. `vggFETriplet`: pretrained model of vgg fine-tuned using triplet few-shot learning (triplet siamese network)

3. `vggFEPretrained`: pretrained model of vgg as it is, no fine-tuning applied, used as feature extractor

4. `featNetTriplet`: inspired by *featNet* architecture in [3], but most important **trained from scratch** (weights are initialized randomly)

Training was done for 100 epochs.

6 Evaluation

For the evaluation phase I measured the performance of the different models on:

- **All-vs-all verification** multiple template, obtaining 3 main informations: *FAR*, *FRR* and *EER*.

- **All-vs-all identification** open-set multiple template, obtaining the following main informations: *FAR*, *FRR*, *EER*, *DIR* (Detection and Identication Rate) and *CMS* (Cumulative Match Score)

To simplify the test phase, I calculated the distance matrix between all samples in the test set, using the Euclidean distance (distance values are not normalized). In the following sections the results obtained for each model will be presented.

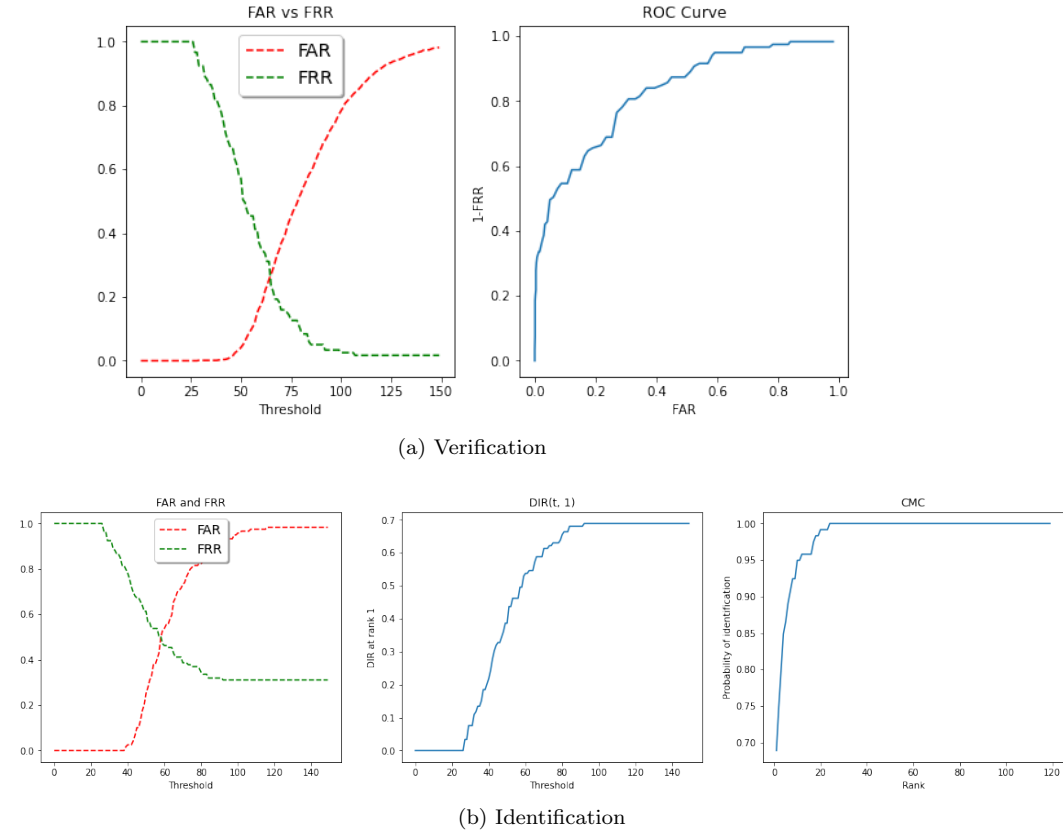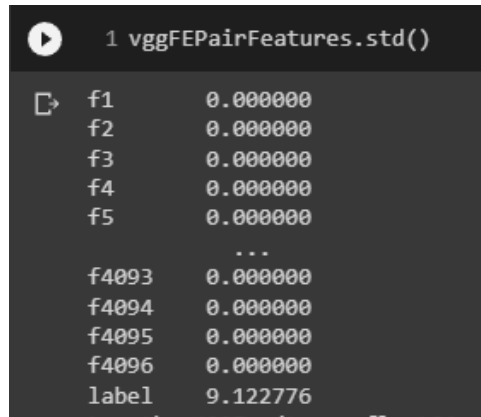## 6.1 vggFEPretrained

(a) Verification

(b) Identification

Figure 9: pretrained model of vgg as it is, no fine-tuning applied, used as feature extractor

6.2 vggFEPair

Figure 10: Standard deviation of feature extracted
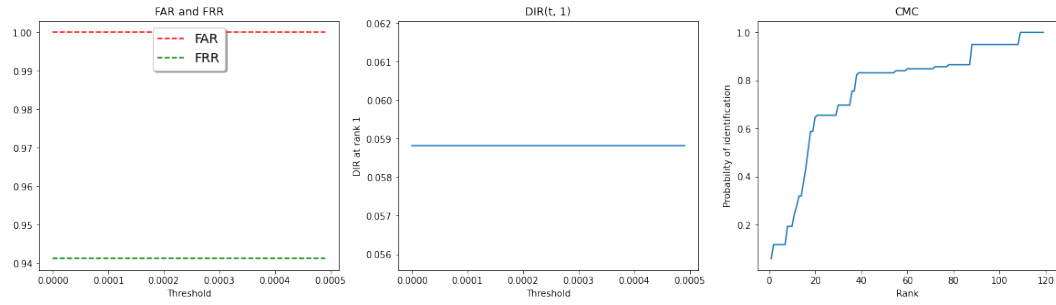
As can be seen in figure 10, the standard deviation on each feature is rounded to 0; this suggests that the features extracted from this model may not be so distant from each other. We can see this as a first red flag.
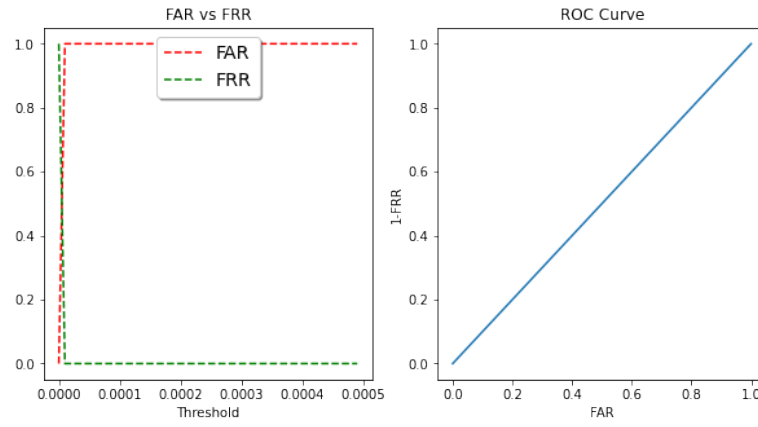
(a) Verification

(b) Identification

Figure 11: pretrained model of vgg fine-tuned using pair few-shot learning (pair siamese network)
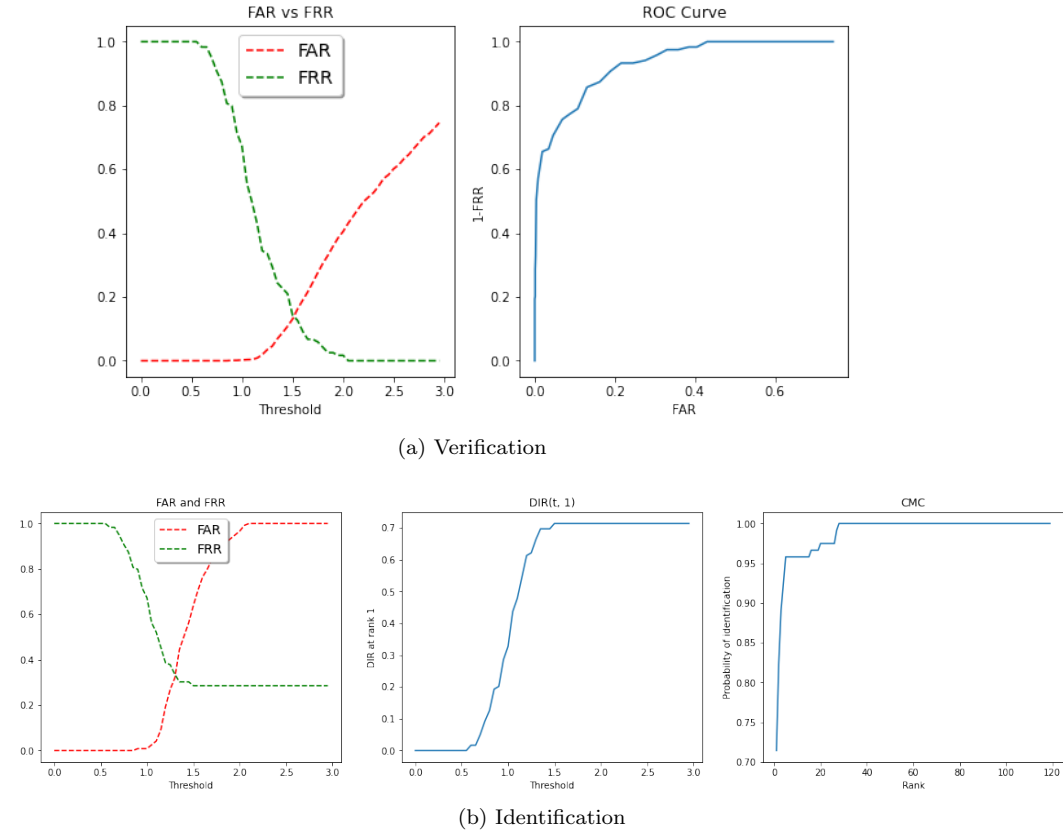
6.3 `vggFETriplet`

(a) Verification

(b) Identification

Figure 12: pretrained model of vgg fine-tuned using triplet few-shot learning (triplet siamese network)

6.4 `featNetTriplet`
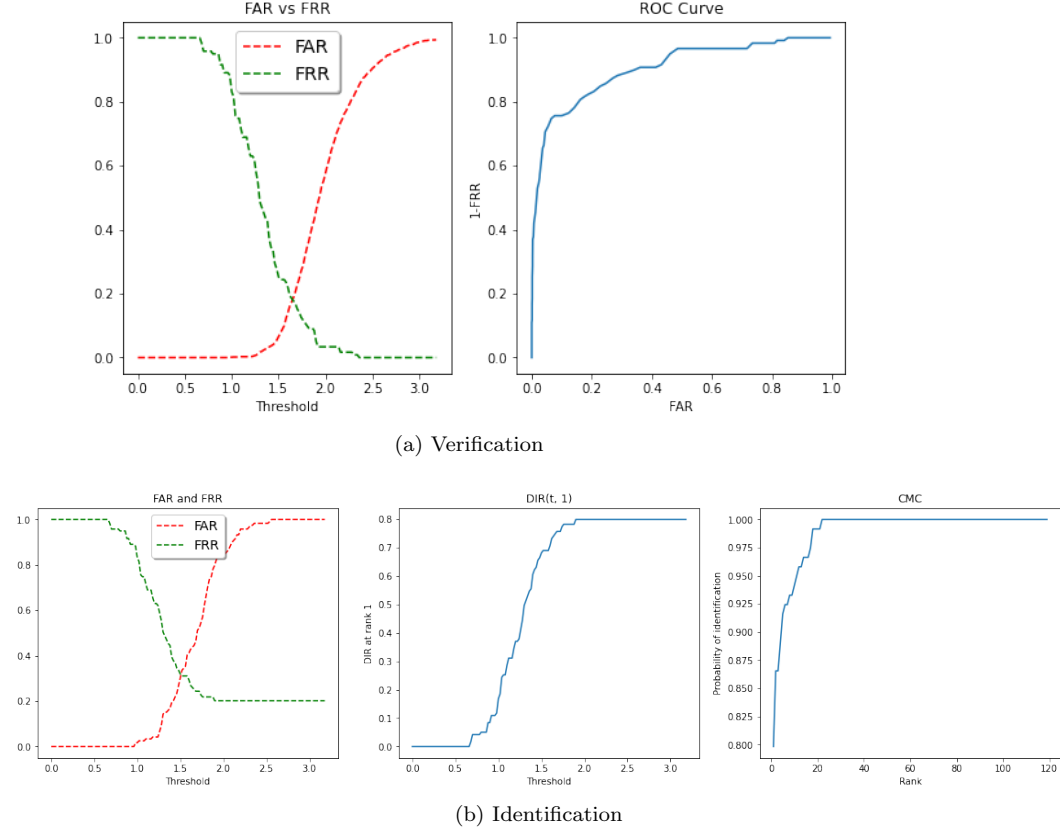

(a) Verification

(b) Identification

Figure 13: inspired by *featNet* architecture

6.5 Considerations after evaluation

After this evaluation phase we can take several considerations:

- `vggFEPretrained` (i.e. vgg without any kind of training/fine-tuning), compared to others, does not perform so bad on verification ($EER = {\sim}0.25$).

- The best model for a verification task is `vggFETriplet`, since the $EER = {\sim}0.16$. Note that a valid second choice could be `featNetTriplet` with $EER = {\sim}0.2$, this result is amazing since the network has been trained from scratch.

- The best model for an identification task is `featNetTriplet`, with

  - $EER = {\sim}0.3$ at $t = 1.5$

- $DIR(t = 1.5, 1) = {\sim}0.7$
- $CMS(rank = 1) = 0.8$

- Triplet training performs way better than pair training. This is probably due to the fact that in the first one the training phase on a single step tries to minimize distance between similar objects while maximizing the distance between different objects.

7 Demo

The demo consists of 3 python scripts on CLI:

- "Enrollment.py" to perform enrollment of a subject in a specific "db" (actually it is a simple csv file) specifying the input image and the associated identity

- "Verification.py" to perform a verification operation by specifying the input image and the claimed identity. At the end of this operation it will print the outcome: *"Identity verified, you are user#ID"* or *"Access denied"*.

- "Identification.py" to perform an identification operation by specifying the input image. At the end of this operation it will print the outcome: *"You are user#ID"* or *"No match in dataset"*.

For simplicity in the demo I decided to use only one model, specifically featNetTriplet (this can easily be changed with another), because is the most versatile and above all the file containing the pytorch model is only a few kilobytes.

Instructions for using this demo are in "README.md" file.

8 Future work

Since time was limited and I didn't have any colleagues to work with, I had to choose what to do and what to not do; here I list several things that could be interesting to test:

- Train and test all 4 feature extractors on a mixed dataset (e.g. *Ubiris + Utiris + MICHE*)

- Plug different feature extractors: could be a different pretrained network (e.g. densenet), or a custom network to train from scratch

References

[1] M.S. Hosseini, B.N. Araabi, and H. Soltanian-Zadeh. "Pigment Melanin: Pattern for Iris Recognition". In: *Instrumentation and Measurement, IEEE Transactions on* 59.4 (Apr. 2010), pp. 792–804. ISSN: 0018-9456. DOI: 10. 1109/TIM.2009.2037996.

[2] Gregory Koch, Richard Zemel, Ruslan Salakhutdinov, et al. "Siamese neural networks for one-shot image recognition". In: *ICML deep learning workshop*. Vol. 2. Lille. 2015, p. 0.

[3] Caiyong Wang et al. "Towards Complete and Accurate Iris Segmentation Using Deep Multi-Task Attention Network for Non-Cooperative Iris Recognition". In: *IEEE Transactions on Information Forensics and Security* 15 (2020), pp. 2944–2959. DOI: 10.1109/TIFS.2020.2980791.