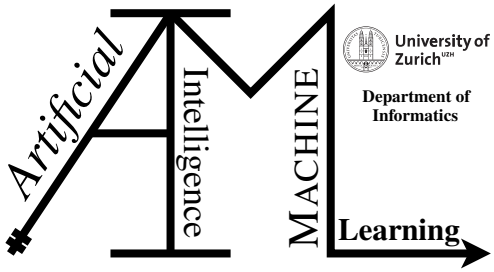
Open-Source Package for Generic Deep-Network-based Face Detection and Recognition in Bob

Master Project

Y. Linghu, X. Zhang
19-771-682, 19-772-235

Submitted on
June 25, 2021

Thesis Supervisor
Prof. Dr. Manuel Günther



Master Project

Author: Y. Linghu, X. Zhang

Project period: January 18, 2021 - June 25, 2021

Artificial Intelligence and Machine Learning Group
Department of Informatics, University of Zurich

Acknowledgements

We thank Prof. Dr. Manuel Günther's supervision on the entire project. We thank Dr. Tiago de Freitas Pereira and IDIAP Research Institute for their help on the coding in this work.

Abstract

This project relies on the system *Bob* and its face verification experiment setup. We develop the generic and specific interfaces for using the pre-trained deep neural face recognition models as the feature extractors and construct the corresponding baselines for each interface. They are tested by different databases, including *LFW, AR_Face, MEDS, MOBIO, morph*. We also develop a face detection package based on the *Tinyface* framework. Those works are now fixing and waiting to integrate into the *Bob* master branch and should be available to use shortly.

Contents

Chapter 1

Introduction

Facial recognition (FR) has a long history and is one of the most prominent combinations of both machine learning and image processing. There are three steps in the general face recognition.

1. Face detection and alignment A face detector locates the faces in a given image and returns the coordinates of a bounding box and/or facial landmarks for each one of them. And the goal of face alignment is to scale and crop face images, in the same way using a set of reference points located at fixed locations in the image.
2. Feature Extraction Traditional methods rely on hand-crafted features, such as edges and texture descriptors, combined with machine learning techniques, like principal component analysis, linear discriminant analysis, or support vector machines. However, with the rapid development of machine learning and deep learning, the above methods have been superseded. Deep neural networks, like convolution neural networks (CNNs), can be trained with very large datasets to learn the best features of the data [Saez-Trigueros et al., 2018].
3. Score Computation FR can be classified into face verification and face identification. In both cases, a set of known subjects are enrolled and stored in the system (or the gallery). Then a new facial image of a subject (the probe) is presented. Algorithms like *cosine distance* or *L2 distance* will be used to compute the one-to-one similarity between the gallery and probe to determine whether the two images are of the same subject, whereas face identification computes one-to-many similarity to determine the specific identity of a probe face.

Most face recognition researches are designed for business purpose, which makes their results non-reproducible. But the reproducibility is a key property of scientific research and essential for the evaluation purpose. To satisfy this need, *Bob* is developed by the biometric security & privacy group at Idiap Research Institute. It is an open-source toolbox, which is designed for signal processing and machine learning[1][Anjos et al., 2012]. The source codes are stored on the GitLab page. *Bob* covers plenty of biometric research works and easy to understand because it uses the Python environment integrated with the C++ library. In particular, for this project, *Bob* provides an excellent environment to compare the facial recognition algorithms because of the reproducibility of the results [Günther et al., 2012]. It provides a complete construction of the face recognition/verification environment, includes Database, Preprocessor, Extractor, and Algorithm. *Bob* contains the traditional face detection and feature extraction (face recognition) methods and is continually introducing the pre-trained deep neural networks for the above steps.

In this project, our main task is to create the interfaces for different neural network frameworks, so that the most common pre-trained face recognition networks could work as the feature extractors in the face verification experiment. We also need to write a package for some face detection pre-trained models as the face annotator. Further, to make them usable, we need to appropri-

[1]https://www.idiap.ch/software/bob/

ately document the source codes of the interfaces, implementing them with sample pre-trained models as baselines. And finally, our work needs to pass the pipeline tests for the Continuous Integration (CI) framework of Idiap Research Institute.

Chapter 2

Preliminary

2.1 bob.bio.base

`bob.bio.base` is the base package for the biometric recognition experiments and we need this package to ensemble the experiment configuration, including the databases, and run them properly. We are mainly working on package `bob.bio.face`, which allows us to run the face recognition experiments with reproducible results. The tools that we need in this package will be explained in Section 2.3. In addition to the face verification, `bob.bio.vein` and `bob.bio.spear` are packages for vein and speaker recognition experiments, respectively.

2.2 Experiment Pipeline

A general experiment pipeline consists of three sub-pipelines: Training, Enrollment, and Scoring. We do not focus on the training pipeline. The data will be preprocessed, that is, be aligned and cropped to remove the noise, and their features will be extracted. Then for some of the data called Reference subject will be registered and their features will be enrolled into the gallery through the Enrollment pipeline, and the rest of the data called Probe subject will be compared with the reference subject in the gallery by computing the similarity of their features through the Scoring pipeline. Finally, an evaluation will be done for the comparing results.

2.3 bob.bio.face

2.3.1 Databases

We use Labeled Faces in the Wild Database [Huang et al., 2007] for the general testing of the new extractor packages and the alignments of baselines. This database is designed for the face verification experiment and it contains over 13,000 images. In particular, 1680 people have more than one image for testing purposes. Just like its name, the images contain at least one person, but not necessarily the frontal pose, some occlusions are possible. They are taken in the wild without restriction (at least partially frontal) and labeled with the person's name.

We use the following databases to test the baselines: *morph, MEDS/MEDS_II, arface, mobio*. The details for those databases are introduced in Chapter 6 (Milestone 4).

2.3.2 Preprocessor

After receiving the input images with annotations, we need to preprocess the inputs to ease the feature extraction and eliminate the possible background noises. There are multiple preprocessors in `bob.bio.face` package, and we focus on using `bob.bio.face.preprocessor.FaceCrop` throughout our project. Figure 2.1 exhibits a face crop example from *Bob*[1]. In particular, we crop the largest face (Figure 2.1 left) into required size, i.e. parameter `cropped_size`, with the annotated eyes aligned to the given eye locations, i.e. parameter `cropped_positions`, as shown in Figure 2.1 right. In other words, it is a geometric normalization based on the eye locations that will be applied on the face hand-labeled eye annotations [2]. The `cropped_positions` should be adjusted with different `cropped_size` to improve the performance of the experiments. The details will be discussed in Chapter 4 (Milestone 2).

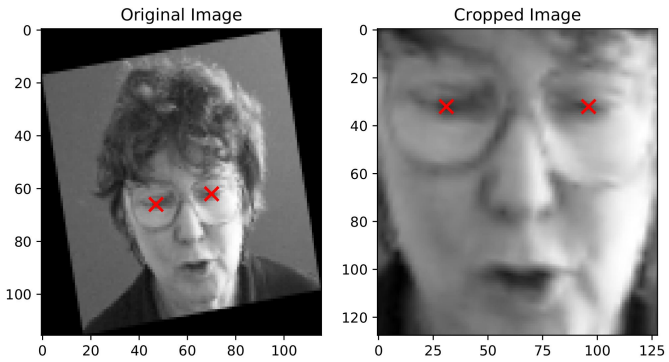


Figure 2.1: Example FaceCrop and Alignment

2.3.3 Extractor

As defined in *Bob*, the extractor grabs the features of the preprocessed images. The dimensionality of the result vectors will be reduced to ease the classification procedures in the next step, that is, calculating the differences between individual vectors. The typical extractors like Discrete Cosine Transform (*DCTBlocks*), Gabor jets in a grid structure (*GridGraph*), and Local Gabor Binary Pattern Histogram Sequences (*LGBPHS*) are introduced and applied in `bob.bio.face` package[2]. To adapt to the deep learning era, the extractors introduced in Chapter 3 (Milestone 1) are aimed to import the pre-trained feature extraction models and weights for multiple interfaces.

2.3.4 Algorithm

After extractions, the algorithm should be applied to compute the score of the experiment, i.e. the similarities of the registered faces. We use `cosine distance` or `scipy.spatial.distance.cosine` as the experiment algorithm throughout this project. It can be called from `bob.bio.base.pipelines.vanilla_biometrics.Distance()` which enrolls the model, i.e. representation

[1] https://www.idiap.ch/software/bob/docs/bob/docs/stable/bob/bob.ip.base/doc/guide.html
[2] https://www.idiap.ch/software/bob/docs/bob/docs/stable/bob/bob.bio.face/doc

of identities, by storing their feature vectors and scores the similarities by computing the distance of the model to the probe by the cosine distance function.

2.3.5 Evaluation

Package `bob.bio.base` contains six evaluation plots. They are designed for different experiments. For example, DIR (detection & identification rate) works well for open-set face identification while for closed-set identification, the CMC is usually employed. In this project, the experiment uses a threshold value to decide whether a similarity score could indicate that the two samples are from the same person. We use the Receiver Operation Characteristic (ROC) plot for the evaluation of the results. It is a True Match Rate (TMR) by False Match Rate (FMR) curve. Thus, for each FMR, we pursue a higher corresponding TMR.

Chapter 3

Milestone 1

We are supposed to get familiar with the *Bob* system, in particular the packages `bob.bio.base` and `bob.bio.face`, and set up the environment. We should be able to run the baselines and get the evaluation plots.

3.1 Setup Working Environment

3.1.1 System and Software

The entire work is done by the Linux or macOS (Intel Core) environment. It is possible to use GPU if the OS has access, but it is not required. The coding is based on the Python (>=3.7) Programming Language, and MiniConda is used to set up the environment.

3.1.2 Required Packages

We created an environment called `bob9b` through MiniConda and installed the following packages to initiate it:

```
$ conda create --name bob9b --override-channels
--channel=http://www.idiap.ch/software/bob/conda/label/beta
--channel=http://www.idiap.ch/software/bob/conda
--channel=defaults distributed
bob.bio.base, bob.buildout, bob.db.arface, bob.db.gbu, bob.db.ijbc,
bob.db.lfw, bob.db.replay, bob.db.replaymobile, bob.db.xm2vts,
bob.extension, bob.ip.base, bob.io.image, bob.ip.facedetect,
bob.ip.gabor, bob.learn.activation, bob.learn.linear,
bob.learn.tensorflow, bob.measure, bob.pipelines, boost, joblib,
matplotlib, pytorch, scikit-image, scikit-learn
```

Further, we `git clone` the `bob.bio.face` package, which is specialized for the facial image verification and identification experiments. In this package, `develop.cfg` includes all dependencies, i.e. necessary *Bob* repositories, for the local package installation, and then use command `buildout` to install the package in development mode and be able to use the libraries in the conda environment `bob9b`. The changes we make in `bob.bio.face` will be explained in Chapter 4. In Chapter 5, we work on package `bob.ip.facedetect`. To co-develop this package

in our environment, we add a few lines (below) to `develop.cfg` and then use command `$ buildout -c develop.cfg`.

```
develop = src/bob.ip.facedetect
eggs = bob.ip.facedetect
[sources]
bob.ip.facedetect = git git@gitlab.idiap.ch:bob/bob.ip.facedetect
```

3.1.3 Problems and Solutions

$ conda list

We tried to install the environment locally on both macOS (Intel Core) and Linux and compared the packages installed, which were not identical. Most of the differences did not impact our project and were ignored.

Not installed in Mac	Not installed in Linux	Installed but Different Version
_libgcc_mutex	bob.db.atnt	bob.learn.tensorflow
astunparse	bob.db.mnist	gast
blinker	gettext	setuptools
cachetools	keras-applications	tensorboard
cudotoolkit	libcxx	tensorflow
dbus	libgfortran	tensrflow-base
expat	libiconv	tensorflow-estimator
fontconfig	llvm-openmp	
glib		
google-auth		
google-auth-oauthlib		
gst-plugins-base		
gstreamer		
ld_impl_linux-64		
libgcc-ng		
libgfortran-ng		
libuuid		
libxcb		
libxml2		
oauthlib		
pcre		
pyasn1		
pyasn1-modules		
pyjwt		
pyqt		
qt		
requests-oauthlib		
rsa		
sip		
tensorboard-plugin-wit		

Table 3.1: conda list results for Linux and MacOS

However, `bob.db.atnt` and `tensorflow` were essential for the setup. In particular, Linux showed an error and asked us to install `bob.db.atnt` package, which was supposed to be in one of the channels. It was easy to solve the problem by adding `bob.db.atnt` in the command `conda create –name bob9b` shown above, or using `conda install` command. If the same problem occurs for other packages, the second method is easier to apply.

And for macOS, the installed `tensorflow` version was lower than `2.0.0`, so we got the following error:

```
ModuleNotFoundError: No module named 'tensorflow.contrib'
```

We solved this problem by using `pip install` instead of calling the package within the conda environment.

```
$ conda remove tensorflow
$ pip3 install tensorflow==2.3.0
```

We could not give an explicit reason for those problems yet, but only the alternative solutions.

Apple M1 chip - Solved

As of the time, we write this report, we haven't successfully created the environment `bob9b` on the Macbook with the Apple M1 chip. One of the reasons was that we could only use `miniforge` to install the package `tensorflow`, but `miniforge` results in installation conflicts with most of *bob* packages. As mentioned above, we had to install `tensorflow` package outside of the conda environment, which was not applicable for Apple M1 chip, and `miniforge` did not support `bob` systems. We also tried to install a virtual Linux system through *Parallel*, but the environment created was not succeeded yet. Our final solution was to use *VSCode* remote ssh for Idiap. Though we still could not run *Bob* experiments locally in macOS with Apple M1 chip, it is possible to build the virtual Linux System to use *Bob*.

3.2 Run Baseline Experiment

3.2.1 Baseline `facenet-sanderberg`

To test the environment, we use `bob.bio.face.config.baseline.facenet_sanderberg` as the baseline test. The baseline test and the rest of the project using the following command line:

```
$ ./bin/bob bio pipelines vanilla-biometrics [DATABASE_NAME]
[BASELINE] -vvv -o [OUTPUT-PATH] -c
```

where we use `lfw_restricted` for `[DATABASE_NAME]` and `facenet-sanderberg` for `[BASELINE]`. We mainly use the aligned LFW database to test our work. In *Bob*, each database can have different protocols, which are designed for different experiment purposes. They are all applicable for this project with slightly different results. In particular, we only use `"view1"` for all the experiments with the LFW database to make the results comparable. This aligned database has the annotation `"eye-centers"` and provides the eye locations in each image. That means the faces are aligned so that their eyes are symmetric horizontally and the midpoint of eyes is horizontally centered. LFW also contains a database with raw images, which will be used in Chapter 5.

`facenet-sanderberg` presents a typical face verification experiment. It uses `FaceCrop()` as the preprocessor, a pre-trained tensorflow deep neural network[1] as an extractor, and `cosine`

[1]https://www.idiap.ch/software/bob/docs/bob/docs/stable/bob/bob.bio.face/doc/baselines.html

`distance` as the algorithm. The performance of this baseline test is good since, for $FMR = 10^{-2}$, the TMR is greater than 96%, which indicates relatively high security of the verification procedure.

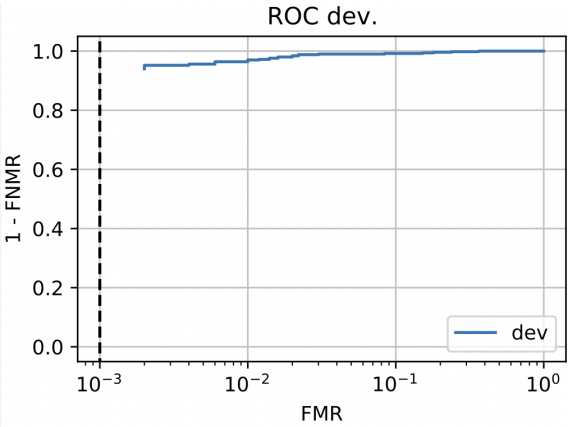


Figure 3.1: Resulting ROC plot of `facenet-sanderberg` experiment

3.2.2 Parallel Execution and GPU

The above baseline test takes about 10 minutes to run locally. Depends on the pre-trained models (especially for Annotators (Chapter 5)), some experiments may take more than an hour. It is possible to speed up this procedure by adding code at the end of the command line above. We could do it locally, add `-dask-client local-parallel`, or through the remote ssh in any console (we use *VScode* in this project), add `-dask-client sge`. The former requires the OS to have GPU available. For convenience, we use the Idiap remote ssh instead.

3.2.3 Problems and Solutions

LFW

While we ran the baseline in February 2021, the resulting FMR started with 10^{-6}, which is not possible since LFW only contains thousands of peoples and 10^{-6} requires more samples. This problem was fixed by Dr. Tiago Pereira. Further, for the protocols that have the evaluation set, like `"view2"` in LFW, add `-g eval` to the command line above can run the experiment on the evaluation set.

Parallel Execution

`-dask-client local-parallel` resulted in an `ImportError`. Dr. Tiago Pereira suggested to use `conda upgrade`, as shown below.

```
ImportError: Error while finding loader for
'bob.pipelines.config.distributed.local_parallel'
(<class 'ModuleNotFoundError'>: No module named
'bob.pipelines.config')

# Solution
$ conda clean -a
$ conda upgrade bob.pipelines -c
https://www.idiap.ch/software/bob/conda/label/beta/
```

Chapter 4

Milestone 2

A general experiment can be split into three stages: Preprocessing, Extracting, and applying the algorithm. In this chapter, we focus on the first and second stages. We are supposed to create the frameworks to use the pre-trained face recognition neural networks in *Bob* as the feature extractor. In this way, users can use their pre-trained models to capture features in the experiment. All the codes we created are viewed and modified by Dr. Tiago Pereira before merging to the master branch. We will explain the differences when necessary.

4.1 Embeddings/Extractors

4.1.1 Framework

We create three specialized interfaces for the frameworks `TensorFlow`, `PyTorch`, and `MxNet`, and a generic interface for other models. Each class consists of five functions,

```
def __init__(self, **kwargs)
def _load_model(self)
def transform(self, X)
def __getstate__(self)
def _more_tags(self)
```

The initialization function mainly includes the parameter `weights` (called `checkpoint_path` in the modified version), sometimes the parameters `config` and `use_gpu` are included. `weights` is the path to the binary file that contains the trained weights/parameters. `config` refers to the path to the text file that contains network configuration, but it is not required for all frameworks. Similarly, each framework has its method to initiate GPU and we make the initiation possible but do not provide the parameter that decides how many GPUs that need to work.

Method `_load_model` reads the pre-trained model, and method `transform` normalizes the input images `X`, forwards them into the model, and returns the extracted features. Initialized parameters `preprocessor` and `memory_demanding` are added by Dr. Pereira. The former is a function that will transform the data before being forwarded. The default is dividing by 255. The default of `memory_demanding` is `False`, which indicates that there is enough memory to forward a lot of data once in the function `transform`. If it is `True`, the `transform` method will run one sample at a time. The last two functions are used to form the pipeline and the same for all the frameworks. Table 4.1 shows all the supported models for each interface.

FrameWork	Supported Models
MxNet	MxNet
PyTorch	PyTorch (Load Model OR Call from Library)
TensorFlow	TensorFlow
OpenCV (Generic)	Caffe, TensorFlow, Torch, Darknet, DLDT, ONNX

Table 4.1: Supported pre-trained model

MxNet A MxNet model requires to have the pre-trained weights, `.params`, and network configuration file, `.json`. The class `MxNetModel` initializes three parameters,

```
weights : str or None
    PATH/To/WEIGHTS, default=None

config : str or None
    PATH/TO/CONFIG, default=None

use_gpu : True or False.
    If gpu is available, set True, default=False
```

While using *MxNet* to read the model, both `weights` and `config` are required. We define a default model, Arcface Insightface[1] [Deng et al., 2019] in case that either `weights` or `config` is None. In general cases, the cropped images should be normalized (Here, divided by 255) before passing into the model. Most experiments could improve the performance from 50% to >95% at $10^{-2}FMR$ through this step. This is common sense, but not true for all the face recognition models (extractors). The default model we use exhibits an opposite result [Deng et al., 2019]. The inputs `X` received from the preprocessor are in format `numpy.array`. We only need to convert them into format `mxnet.ndarrary`, and switch the outputs back to `numpy.array` for the next step. If `X` is divided by 255, then the TMR is 10% at 10^{-2}FMR as shown in Figure 4.1. This rule is applicable for all face recognition models in Arcface Insightface [Deng et al., 2019].

```
X = check_array(X, allow_nd=True)
X = mx.nd.array(X)
return self.model(X,).asnumpy()
```

PyTorch There are two ways to import the pre-trained networks in the PyTorch framework. First, class `PyTorchLoadedModel` looks similar to the `MxNetModel`, which requires both `weights (.pth)` and `config` (.py) documents. If the GPU is available, then in `_load_model` function, it will be initiated. The default model we use is an AFFFE model [Günther et al., 2017]. This time the inputs should be a `tensor` to fit in the model, and the normalization has a positive impact on the performance, see the code below. The result of the default model will be shown in the next section.

[1] https://github.com/deepinsight/insightface

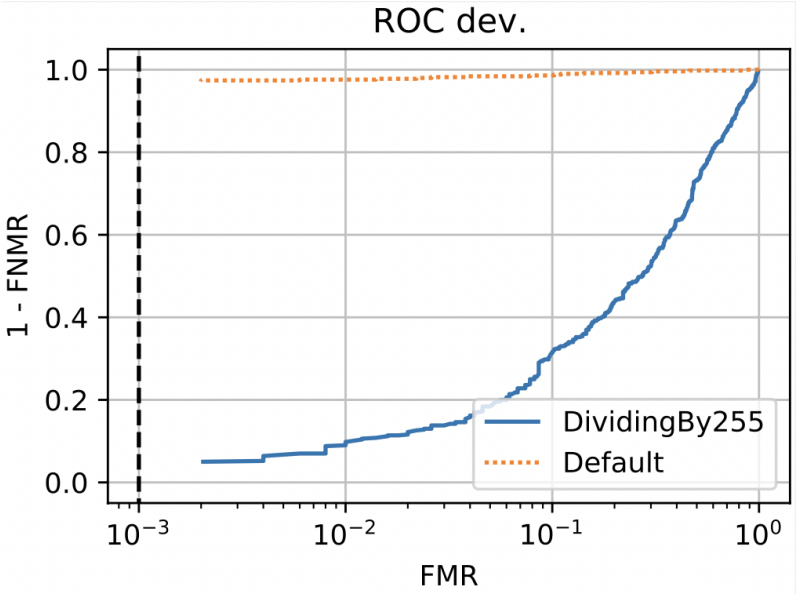


Figure 4.1: Arcface InsightFace [Deng et al., 2019]: Comparison of example normalization

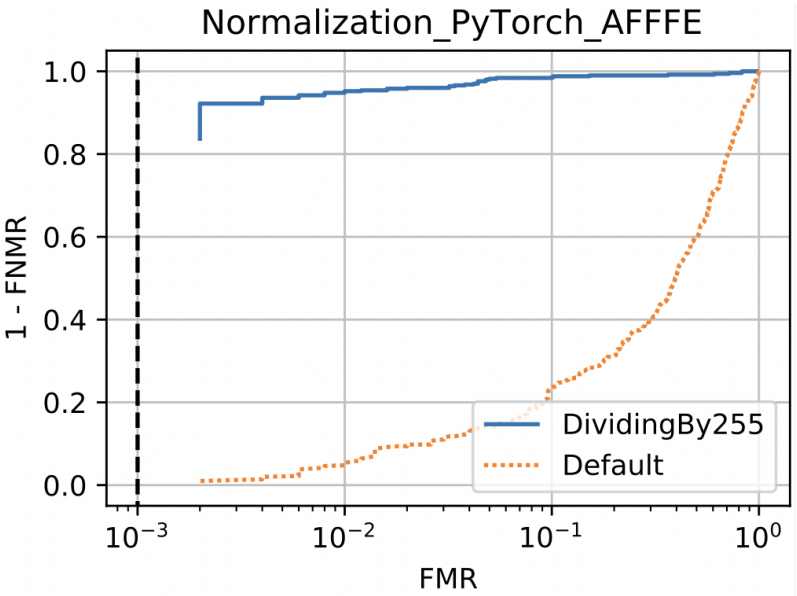


Figure 4.2: PyTorchAFFFE [Günther et al., 2017]: Comparison of example normalization

```
X = check_array(X, allow_nd=True)
X = torch.Tensor(X)
X = X/255
return self.model(X).detach().numpy()
```

The second class `PyTorchLibraryModel` is used when the pre-trained model is saved in the `pytorch` Library. We test this class using pre-trained InceptionResNetV1 model from `facenet_pytorch` Library[1]. That is, we need to install the required library, which contains the weights and structure, before using it. For some other libraries, there is no need to install the library but download the weights from the library instead. When users using `PyTorchLibraryModel`, they need to initialize the parameter `model` in the configuration to define which pre-trained/downloaded model to use, see example below.

```
from facenet_pytorch import InceptionResnetV1
model = InceptionResnetV1(pretrained='vggface2').eval()
extractor_transformer = PyTorchLibraryModel(model=model)
```

TensorFlow For a general `tensorflow` model, we use method `tf.keras.models.load_model` to implement it. The parameter `self.weights` (generally called `filename`), which contains one of the following: *String*, *pathlib.Path* object, the path to the saved model, or *h5py.File* object. So we ask users to define the directory to the folder that contains those elements as the only parameter for the class `TensorFlowModel`. The default model we use is InceptionResNet [2]. The input `X` would be converted into a `tensor` with the channel_last format through `to_channels_last` command, and the normalization has a positive impact on the performance. The result of the default model will be shown in the next section.

```
X = check_array(X, allow_nd=True)
X = tf.convert_to_tensor(X)
X = to_channels_last(X)
X = X/255
return self.model.predict(X)
```

Generic OpenCV When the given model does not fit any of the interfaces above, like .caffemodel and .onnx, we use `OpenCV` to create a generic interface. This class, `OpenCVModel`, supports six types of models, as listed in Table 4.1. To be compatible with multiple interfaces, the initialized parameters contain both `weights` and `config`, but only when both are not specified, the default model[1] will be called. The command `cv2.dnn.readNet()` can deal with the case that `config` is None. A normalization is followed to improve the performance as usual. Dividing by 255 is not ideal for our default model. The modified version chooses to convert inputs into RGB format, but the performance on the LFW database does not improve a lot.

[1]https://github.com/timesler/facenet-pytorch

[2]https://www.idiap.ch/software/bob/docs/bob/docs/stable/bob/bob.bio.face/doc/baselines.html#deep-learning-baselines

[1]https://www.robots.ox.ac.uk/ vgg/software/vgg_face/

```
cv2.dnn.readNet(self.weights,self.config) #Import Network

# Old version
X = np.array(X)
X = X/255

# New version
caffe_average_img = [129.1863, 104.7624, 93.5940]
X[:, :, :, 0] -= caffe_average_img[0]
X[:, :, :, 1] -= caffe_average_img[1]
X[:, :, :, 2] -= caffe_average_img[2]
# To BGR
X = X[:, ::-1, :, :].astype("float32")

# forward
self.model.setInput(X)
return self.model.forward()
```

In the modified version, the above four classes are set as the base class and the default models are moved into their subclass. For example, the base class for MxNet is `MxNetTransformer()` and it has a subclass `ArcFaceInsightFace_LResNet100()` which imports an ArcFace MxNet model. Further, within the same file, the template, or baseline we use in Section 4.2, is defined.

4.1.2 Problems and Solutions

Default Models We got stuck on where to put the default model. It was ideal to let users change the default without changing the class in the embeddings. However, no default defined in the `__init__` function will make testing of the embedding difficult. The details of testing will be discussed in chapter 7.2. So we have to give up the possibilities for changing the default. This problem was solved by Pereira in the modified version.

4.2 Baselines

4.2.1 Construct Baseline

The baseline in this report refers to the complete face verification experiment. Listing 4.1 is an example baseline implementing MxNet Interface. In this experiment, we call the database, `lfw_restricted`, in the Terminal command line as shown in Section 3.2.1, and all the other components are defined in a `.py` file. First, in line 11 to line 21, we check whether the database is hand-annotated or given a fixed position of eyes. We will explain how to use the `annotator` to annotate the database in the next section. Second, we define the preprocessing step, i.e. `FaceCrop()` only in this project. Line 23 defines `cropped_positions`, which are the coordinates in the cropped image. The annotated points should be put to those two coordinates[1]. There are multiple choices for the `cropped_positions`, and it depends on whether those coordinates are provided

[1] https://www.idiap.ch/software/bob/docs/bob/bob.bio.face/stable/implemented.html#bob.bio.face.preprocessor.FaceCrop

in the annotations, for example, mouth location and nose location. We mainly focus on eye locations, i.e. `{'leye', 'reye'}`, since the locations of eyes are relatively fixed in each face. Line 24 instantiates the `FaceCrop()`, in particular, the `cropped_image_size` should follow the required input shape of the pre-trained network in the extractor. Third, we instantiate the extractor. If the users want to pass their own pre-trained network, then they have to define `weights`, i.e. path to the parameter weights, and `config`, i.e. path to the `.json` configuration file, respectively. Otherwise, the extractor will use the default model automatically. Then, the algorithm is defined in line 33. The preprocessor and extractor are combined like a transformer pipeline in lines 36 to 40. And finally, we pass the transformer and algorithm to the `VanillaBiometricsPipe` for the complete experiment.

4.2.2 Result ROC plots

For each of the five classes we created in Section 4.1.1 (except case 2 in PyTorch), we run a corresponding baseline experiment. The example MxNet model requires a (112×112) input image, (224×224) for the example PyTorch model, (160×160) for the example TensorFlow model, (224×224) for the example Caffe model using Generic interface. Though the `cropped_positions` vary while cropped size changes, the relative position of the eyes on the cropped image does not change. As the default, we use ratio (0.64, 0.38), that is, x-measure of `leye` is $0.64 * width$ (second integer in `cropped_image_size`), same for `reye`. We do not fix a relative position for y-measure, but we choose to use some y-measures that are above the middle but not too extreme. The choices of x-y measures depend on the network. The cropped images should look similar to the training samples of the network. Figure 4.2 shows the resulting ROC plot for each experiment. It seems that only the vgg_face Caffe model does not have a good performance, which is reasonable since it is relatively out-of-date. The other three experiments have at least 90% TMR at 10^{-2} FMR.

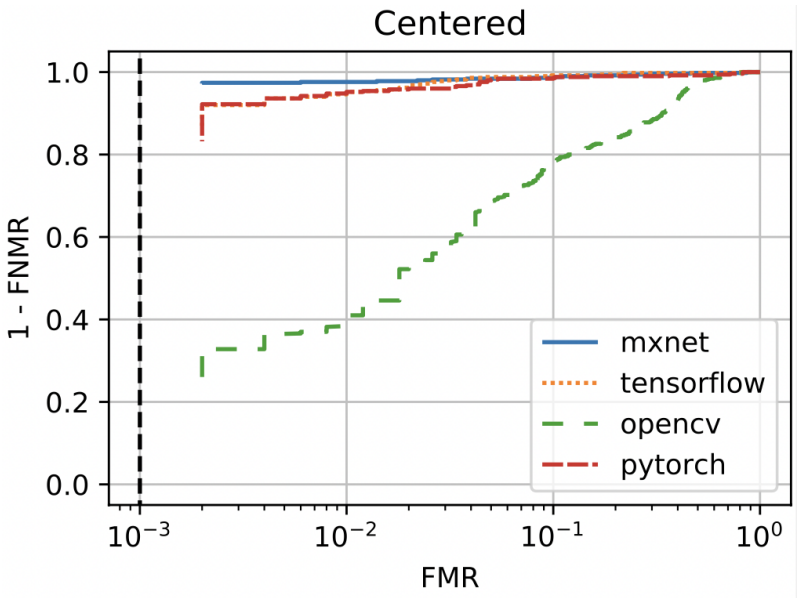


Figure 4.3: Resulting ROC plot of 4 baselines

```python
import bob.bio.base
from bob.bio.base.pipelines.vanilla_biometrics import Distance
from bob.bio.base.pipelines.vanilla_biometrics import VanillaBiometricsPipeline
from bob.bio.face.preprocessor import FaceCrop
from bob.bio.face.embeddings.MxNetModel import MxNetModel
from bob.pipelines import wrap
import scipy.spatial
from sklearn.pipeline import make_pipeline

# Annotator & Preprocessor
memory_demanding = False
if "database" in locals():
    annotation_type = database.annotation_type
    fixed_positions = database.fixed_positions
    memory_demanding = (
        database.memory_demanding if hasattr(database, "memory_demanding") else False
    )

else:
    annotation_type = None
    fixed_positions = None

cropped_positions={'leye':(49,72), 'reye':(49,38)}
preprocessor_transformer = FaceCrop(cropped_image_size=(112,112), cropped_positions=
    cropped_positions, color_channel='rgb',fixed_positions=fixed_positions)

# Extractor
weights = None # PATH/TO/WEIGHTS
config = None # PATH/TO/CONFIG

extractor_transformer = MxNetModel(weights=weights,config=config)

# Algorithm
algorithm = Distance(distance_function = scipy.spatial.distance.cosine,
    is_distance_function = True)

# Chain the Transformers together
transformer = make_pipeline(
    wrap(["sample"], preprocessor_transformer,transform_extra_arguments=
    transform_extra_arguments),
    wrap(["sample"], extractor_transformer)
    # Add more transformers here if needed
)

# Assemble the Vanilla Biometric pipeline and execute
pipeline = VanillaBiometricsPipeline(transformer, algorithm)
transformer = pipeline.transformer
```

Listing 4.1: MxNet Baseline

4.2.3 Problems and Solutions

Performance of Baseline For each baseline test, we ran at least 20 times with different eye locations. It is necessary to choose `cropped_positions` that makes the face horizontally centered

in the image, for all `cropped_image_size`. Ideally, the centered eye positions should result in the best performance, but an exception exists. We found this during experiments. Table 4.2 and Figure 4.3 are for reference only. It contains the best eye locations for the baselines. None of them has exactly centered eyes. Notice that `pytorch-pipe` and `opencv-pipe` have the same `cropped_image_size` but different `cropped_positions`. We also tried to use the default `preprocessor` method defined by Dr. Pereira instead of dividing by 255, but the performance is worse. This might be caused by the low quality of embedding models, as the performance is restricted by the out-of-date property.

Table 4.2: Eye locations relative to different image sizes (optimal)

Baseline	image size	Ratio of height	Ratio of leye	Ratio of reye
mxnet-pipe	(112, 112)	0.44	0.64	0.34
pytorch-pipe-v1	(224, 224)	0.49	0.64	0.43
pytorch-pipe-v2	(224, 224)	0.49	0.64	0.43
tensorflow-pipe	(160, 160)	0.50	0.63	0.38
opencv-pipe	(224, 224)	0.44	0.51	0.34

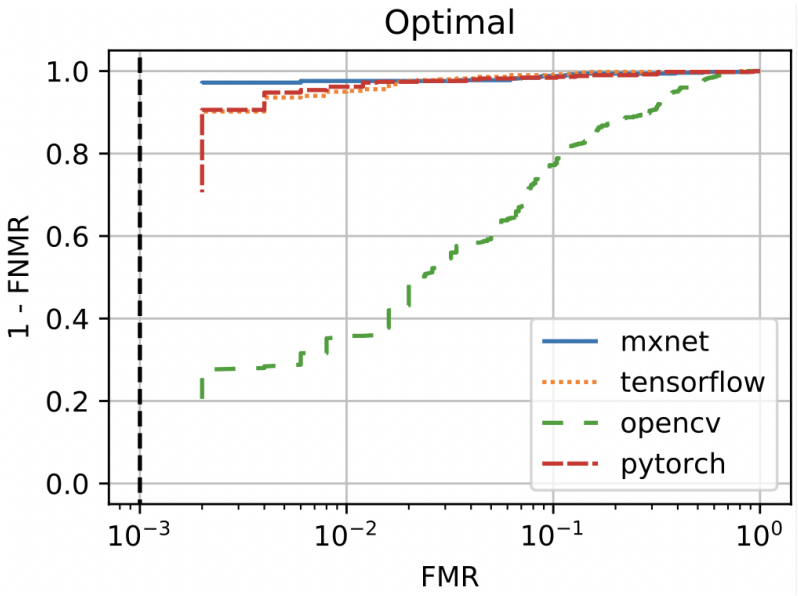


Figure 4.4: Resulting ROC plot of 4 baselines with Optimal eye positions

Chapter 5

Milestone 3

We are supposed to implement several face detection algorithms, including `MTCNN`, `TinyFace`, `Faster-RCNN`, and `HyperFace`. `MTCNN` has been introduced in package `bob.ip.facedetect` and we cannot figure out the compatible source codes for `Faster-RCNN` and `HyperFace`. Thus, we mainly develop the `TinyFace` inside `bob.ip.facedetect` and make it usable in `bob.bio.face`.

5.1 Face Annotator

Depending on the databases, sometimes we need to annotate the faces before proceeding to the preprocessors. In other words, we need to know where the face is before crop the image according to its location. And if there are multiple faces in one image, the cropping should be based on the largest one. For example, LFW database `all_images_aligned_with_funneling` has annotations `"eye-centers"`, and database `all_images` needs an explicit annotator. In the former case, eyes have been annotated and the images are aligned so that faces are centered. The latter is the raw images with possible skewed eyes. In general, annotations of an image include the locations of faces, usually the top-left and bottom-right coordinates of the bounding boxes, of eyes, of mouth, and nose, but not all of them are required as the annotation results. The annotator will be used when there is no alignment applied on the LFW database and it helps locate the faces.

5.1.1 bob.ip.facedetect

`bob.ip.facedetect` is a package that includes classifiers and functions to detect whether the given image contains a face[1]. If so, the information of the detected face(s) should be saved and passed into the next stage (preprocessor). For example, `bob.ip.facedetect.mtcnn.MTCNN` is an annotator that detects face using Multi-task Cascaded Convolutional Networks (MTCNN) [Zhang et al., 2016]. This class reads in the pre-trained MTCNN model and passes the image into the model. It returns a dictionary that includes eight keys: `topleft, bottomright, reye, leye, nose, mouthright, mouthleft, quality`. The annotator will be used when the line 20 and line 21 in Listing 4.1 are both None. To deal with this case, `bob.ip.facedetect.mtcnn.MTCNN` is called and re-initiated in `bob.bio.face` as `BobIpMTCNN`. And we only need to initiate this annotator within the `FaceCrop()`.

[1] https://www.idiap.ch/software/bob/docs/bob/docs/stable/bob/bob.ip.facedetect/doc/index.html

```
from bob.bio.face.annotator import BobIpMTCNN
annotator_transformer = BobIpMTCNN()
preprocessor_transformer = FaceCrop(cropped_image_size=(112,112),
cropped_positions={'leye':(49,72), 'reye':(49,38)},
color_channel='rgb', annotator=annotator_transformer)
```

5.2 Face Detection Model: TinyFace

We define a new annotator, `tinyface.TinyFacesDetector`, in `bob.ip.facedetect`. It is a model introduced by Peiyun Hu and Deva Ramanan to find the small faces in an image[1] [Hu and Ramanan, 2017]. The original model is in `.matlab` format. Then from github[2], we converted the original model into the `mxnet` format.

5.2.1 bob.ip.facedetect.tinyface.TinyFacesDetector

This package is originally designed by chinakook[2] with a MIT License and we modify it to fit in the *Bob* system. With MIT License, we are allowed to obtain a copy of the model and modify the code. So the model has been saved in the Idiap data for further use[3]. First, we add the `checkpoint_path` in `__init__`, which allows to download the model automatically through the link[3]. Second, we assume the input images are *Bob* format (and possibly RGB), the following codes are added before the original code:

```
# In case the input raw_img is not in three-channel format, convert
it into RGB.
from bob.ip.color import gray_to_rgb
if len(raw_img.shape) == 2:
    raw_img = gray_to_rgb(raw_img)
assert img.shape[0] == 3, img.shape

# The original code expects raw_img is BGR, convert it.
from bob.io.image import to_matplotlib
raw_img = to_matplotlib(raw_img)
raw_img = raw_img[..., ::-1]
```

This class contains one parameter `prob_thresh`, which is supposed to be a `float` and it represents a trade-off ratio between false positives and missed detections. The pre-trained tinyface model is initialized in the `__init__` function, and all faces should be detected in function `detect` and it returns a list of annotations which contains the `topleft, bottomright, reye, leye`. Notice that the tinyface model can only detect the face and provide a bounding box, and does not capture further details of the faces. Since the eye locations are relatively fixed in human faces, we estimate `reye` and `leye` using the bounding box coordinates. Through experiments, the ratio $(0.37, 0.3), (0.37, 0.7)$ works best, where the first entry is the ratio to the height of the detected face, and the second entry refers to the width of each eye.

[1] https://github.com/peiyunh/tiny
[2] https://github.com/chinakook/hr101_mxnet
[3] https://www.idiap.ch/software/bob/data/bob/bob.ip.facedetect/master/tinyface_detector.tar.gz

5.2.2 Call in Configuration in bob.bio.face

To use this face detector as an annotator in package `bob.bio.face`, we define a class `bob.bio.face.annotator.BobIpTinyface`. It initializes `bob.ip.facedetect.tinyface.TinyFacesDetector`, passes the RGB image that is in Bob format into function `annotate` and returns the annotation results for the largest face. `prob_thresh` is a parameter in class `TinyFacesDetector` and to make it usable while calling `BobIpTinyface()`, we add a class property for it:

```
@property
def prob_thresh(self):
    return self.tinyface.prob_thresh
```

Adding following command in the configuration to apply `BobIpTinyface`:

```
from bob.bio.face.annotator import BobIpTinyface
annotator_transformer = BobIpTinyface(prob_thresh=0.51)
preprocessor_transformer = FaceCrop(cropped_image_size=(112,112),
cropped_positions={'leye':(49,72), 'reye':(49,38)},
color_channel='rgb', annotator=annotator_transformer)
```

Figure 5.1 compares the results for three experiments with similar setup: LFW Database, `FaceCrop()`,`MxNetModel()`,`Distance()`. Listing 5.1 exhibits an example to apply an annotator to a general database. The aligned case uses the aligned database `all_images_aligned_with_funneling`, thus no annotator applied, and the other use raw database `all_images` with annotator `BobIpMTCNN` and `BobIpTinyface`. It seems that the experiment with an aligned database performs best, followed by the annotator `BobIpTinyface`, and the worst is the annotator `BobIpMTCNN`. The differences between them can be ignored. It is not surprising to suppose that aligned by true eyes should give the most secure result, and if there is no face with weird postures, then aligned by the detected eyes and estimated eyes would not result in a large difference. The good news is, in the LFW database, the worst case we get is 98% TMR in 10^{-2} FMR.

5.2.3 Problems and Solutions

Swapped x-y Coordinates returns by class TinyFacesDetector

In the entire *Bob* system, the images are represented in (y, x) format, instead of (x, y) in the general quadrants. The only exception is in the class `bob.ip.facedetect.tinyface.TinyFacesDetector`. Although the length and width of the detected faces do not differ a lot, the impacts on the estimated eye locations resulted in an obvious difference in output. Before we noticed this problem, the initial TMR for setup in Figure 5.2 was at most 80%. This problem has been fixed.

```
# Annotator & Preprocessor
memory_demanding = False
if "database" in locals():
    annotation_type = database.annotation_type
    fixed_positions = database.fixed_positions
    memory_demanding = (
        database.memory_demanding if hasattr(database, "memory_demanding") else False
    )
else:
    annotation_type = None
    fixed_positions = None

if annotation_type = None:
    annotator_transformer = BobIpTinyface(prob_thresh=0.51)
    fixed_positions = None
else:
    annotator_transformer = annotation_type
    fixed_positions = fixed_positions

cropped_positions={'leye':(49,72), 'reye':(49,38)}
preprocessor_transformer = FaceCrop(cropped_image_size=(112,112), cropped_positions=
    cropped_positions, color_channel='rgb',fixed_positions=fixed_positions,annotator=
    annotator_transformer)
```

Listing 5.1: Applying Annotator Example

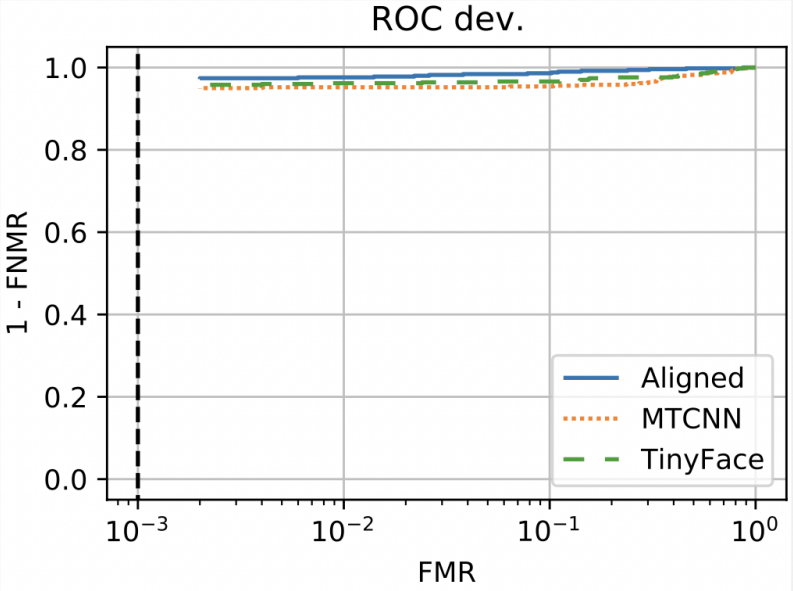


Figure 5.1: Comparing ROC plots of experiments without annotator, using MTCNN, and TinyFace (Arcface Insightface MxNet embedding)

Chapter 6

Milestone 4

It is necessary to check whether the extractors in Chapter 4 (Milestone 2) work for different databases, instead of designed only for the LFW database. We choose four databases available in Idiap Resource[1] and use the default protocol. Fortunately, all of them contain the annotation `"eye-centers"`, so we do not have to define the annotator in `FaceCrop()`.

6.1 Database AR_Face

AR_Face database is created by Aleix M Martinez and Robert Benavente[2]. It contains more than 4,000 images for 126 people. Each person has more than 26 images, includes different facial expressions, illumination conditions, and occlusions, but there is no restriction for the other features of the person. Those pictures are taken under control and have a time span. In the experiment, we chose to use the raw colored images instead of the gray version. We use protocol `"all"` for this database[3].

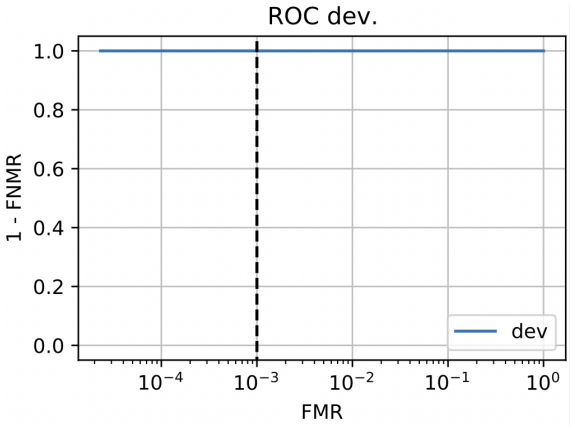


Figure 6.1: Resulting ROC plot of experiment using AR_Face database with Arcface Insightface MxNet embedding

[1]https://www.idiap.ch/software/bob/
[2]http://www2.ece.ohio-state.edu/~aleix/ARdatabase.html
[3]https://gitlab.idiap.ch/bob/bob.bio.face/-/blob/master/bob/bio/face/config/database/arface.py

6.2 Database MEDS/MEDS_II

The *MEDS_II* database is developed by NIST. The original database contains 518 identities, but they are classified by gender and race, which results in the extremely unbalanced distribution of images. The *Idiap* chose to use a subset of the database, i.e. 383 identities (White and Black men only), who consist of more than one sample. We use the default protocol `"verification_fold1"` for the experiment, `"verification_fold2"` and `"verification_fold3"` are available[1].

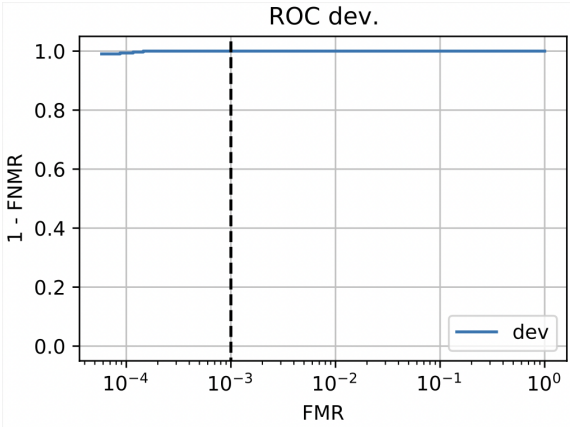


Figure 6.2: Resulting ROC plot of experiment using MEDS database with ArcFace InsightFace MxNet embedding

6.3 Database mobio

The *MOBIO* database is not designed for face verification only. It is a video database that contains 152 identities. The data is collected from mobile phones (NOKIA N93i) and laptop computers (standard 2008 MacBook). The Idiap chooses to grasp the image from the video for face recognition experiments. Those samples have time spans. We use the default protocol `"mobile0-male-female"` for the experiment[2].

6.4 Database morph

The MORPH dataset is relatively old, but it contains more samples than the above databases. This is a quite large database. It takes us more than 5 hours to run the `mxnet-pipe` with `morph` and the resulting score file is $4GB$ large. There are 13,000 identities with 55,000 samples. Those identities are classified by gender and ethnicity. The interface in *Bob* contains three verification protocols, which means that the distributions of identities are different in each protocol. We use the default protocol `"verification_fold1"` for the experiment[3].

[1]https://gitlab.idiap.ch/bob/bob.bio.face/-/blob/master/bob/bio/face/database/meds.py
[2]https://gitlab.idiap.ch/bob/bob.bio.face/-/blob/master/bob/bio/face/database/mobio.py
[3]https://gitlab.idiap.ch/bob/bob.bio.face/-/blob/master/bob/bio/face/database/morph.py

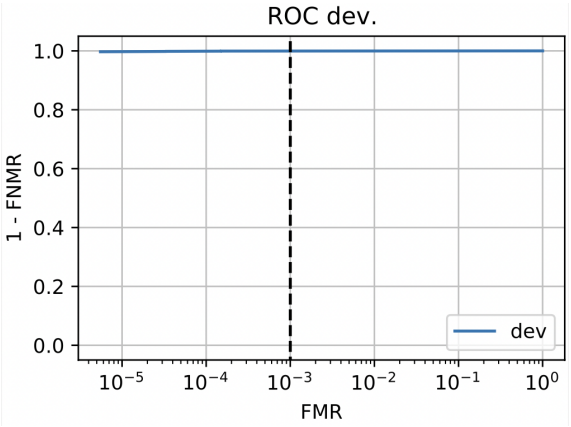


Figure 6.3: Resulting ROC plot of experiment using mobio database with arcface insightface mxnet embedding

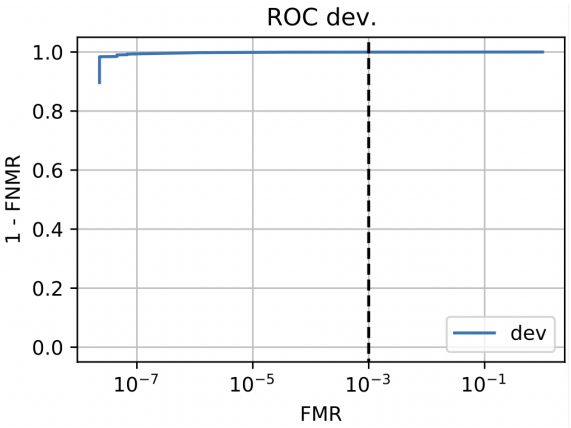


Figure 6.4: Resulting ROC plot of experiment using morph database with arcface insightface mxnet embedding

It seems that the general performance of the databases in this chapter is better than the LFW database. This makes sense since LFW has more variation and difficulties for verification. The nature of each database decides its performance in the same experiment environment.

6.5 Problems and Solutions

No CPU Allocated

Before we noticed the fact that those databases have annotations, we applied `BobIpTinyface` as the annotator. Usually, the experiment could not be run and be killed in the middle. We supposed the problem happened because the neural network in the annotator took too much memory. Then the system killed the experiment automatically. The experiments without an annotator ran smoothly.

Image format for AR_Face

The images saved in *AR_Face* are `.ppm` format. However, in `bob.bio.face.config.database.arface`, the read in format is `.png`, which results in the `"Could not found"` error. The problem was reported by Prof. Dr. Manuel Günther and we are still waiting for the fixes now.

Chapter 7

Milestone 5

All the newly introduced packages are needed to be tested to make sure that they can work properly, and be well-documented to make them easy to understand and use. Further, we need to integrate the new packages into the *Bob* ecosystem including the Continuous Integration system.

7.1 Tests

7.1.1 bob.ip.facedetect

Based on Chapter 5.1, annotators are used to capturing the faces in the given image. There are two different situations: an image with one face and an image with multi-faces. Thus, the annotator should return the bounding box and landmarks for each face, respectively. Here we define 3 tests for *TinyFace* annotator: Returns the correct coordinates for the testing image; Annotates one face correctly; Detects multiple faces in one image.

7.1.2 bob.bio.face

In this project, we need to test separately for each extractor we defined above. Before extracting the image, it should be preprocessed first. So we implement the `Facecrop` function from `test_preprocessors.py`. Before writing a test, we need to run the preprocessor and extractor first to get the results as the reference. We use methods `HDF5File, test_utils` from `bob.io.base` so that we can convert the result of the `feature` into a `.hdf5` file:

```
from bob.io.base import HDF5File, test_utils

outfile = HDF5File("bob/bio/face/test/data/pytorch_v2.hdf5", 'w')
outfile.set('pytorch_v2.hdf5', feature)
```

For each extractor, we need to make sure whether the extractor's format is right. Then we compare the features resulting from the image extracted with the reference.

Next, we focus on tests for five baselines. In the original test file, we call method "`run_baseline`" to test:

```
@pytest.mark.slow
@is_library_available("opencv-python")
def test_opencv_pipe():
    run_baseline("opencv-pipe", target_scores=None)
```

7.1.3 Problems and Solutions

Swapped x-y measures in test_baselines.py

Both the original tests and our newly added baseline tests failed without clear reasons. Sometimes the error message suggested that there was a swap for eye positions. We checked the baseline and the methods originally defined in `test_baselines.py`. It comes out that the annotation for eyes has swapped x-y measures while generating samples in method `get_fake_sample_set`. So we switched the x-y measures to fix the bug.

7.2 Documentation and Continuous Integration (CI)

Documentation is necessary to explain our work and make code usable for other users. The entire webpage of *Bob* is generated by *Sphinx*. In *Bob*, we need to take care of two pages, one for the explanations in text and one for the source codes. The latter was done by

`.. automodule ::` and/or `.. autosummary::`

In particular, Chapter 4 and 5 introduce new packages which require documentation to use.

After done with Tests and Documentation, the source codes and those changes should be integrated into the Continuous Integration (CI) framework of Idiap. Specifically, we need to commit all changes, push them into the GitLab page, and make sure that all pipeline tests turn green. Then our work could be merged into the master branch.

7.2.1 bob.ip.facedetect.tinyface

As mentioned in Chapter 5, we introduce a class `TinyFacesDetector`, which is a modified version of the code from GitHub, to read in the `tinyface` model and pass the images to detect the faces. To exhibit the performance of the class, we refer to the page for `MTCNN`. We provide an example code[1] on how to use `bob.ip.facedetect.tinyface.TinyFacesDetector` to detect multiple faces in one image. In the example page for `MTCNN`, the code is executable and the resulting image is generated automatically while compiling. It does not work for us since `mxnet` is not in the default *Bob* environment. As the documentation should pass the Continuous Integration (CI) of *Bob*, we move the `import mxnet` inside the function, instead of as the global variable, and avoid running the function with `mxnet` in the CI pipeline. Thus, the resulting image with bounding boxes here (Figure 7.1) is uploaded separately.

Basic explanation of parameters and functions are added within the class `TinyFacesDetector`. We follow the original format of `bob.ip.facedetect` to use only

`.. automodule:: bob.ip.facedetect.tinyface`

in `py_api.rst`, which is the summarization page for all available modules in package `bob.ip.facedetect`, to generate a readable layout of the class, which can explain the parameters and available functions.

[1] https://www.idiap.ch/software/bob/docs/bob/docs/stable/bob/bob.ip.facedetect/doc/tinyface.html

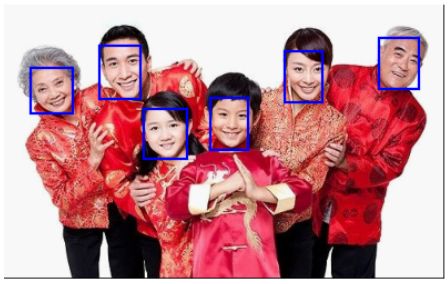


Figure 7.1: Multi-Face Detection results using TinyFace

7.2.2 bob.bio.face.embeddings

Similarly, we create an explanation page, i.e. `deeplearningextractor.rst`, and modify the source code page, i.e. `implemented.rst`, which has the same role as `py_api.rst` above. In Milestone 2, we have introduced five embeddings. For the source code page, we add a short description of each class and its parameters before initialization `__init__(self, **kwargs)`. Among four functions, only `transform(self, X)` was publicly available, since this is the only one that accepts the inputs other than `self`. The description, accepted inputs, and expected outputs are added at the beginning of the function. In `implemented.rst`, we have given the entire path of each class for the `autosummary` and `automodule`, since they are not registered in `init.py` of embeddings. We follow the convention for other embeddings. Again, importing of new packages like `mxnet` and `cv2` are moved inside the functions to avoid the failed pipelines for CI.

`deeplearningextractor.rst` explains how to import and use the pre-trained feature-extraction models in a face verification experiment. It contains four sections: general introduction to the extractor and its derivation, using the pre-trained networks to extract features, named embeddings; step by step instruction for using each embedding in the experiment configuration; baselines for each embedding and its resulting ROC plot; special case suggestion. The last section is designed in case that none of the above interfaces are compatible with the user's model. We suggest the users use MMDNN, i.e. the acronym of Model Management and Deep Neural Network[1], to convert the model into the available interfaces. If the converted version is too old to fit in the *Bob* environment, `.onnx` should be a good choice and `GenericOpenCV` is designed to deal with it. So we suggest the users use `ONNX` as their alternative.

We also modify some other details in `bob.bio.face` page. First, we list the baselines mentioned in Chapters 4 and 5 in `baselines.rst`. Second, the baseline for `pytorch-pipe-v1` applied the `AFFFE` model which is not from GitHub, so we add a reference for it in `references.rst`. Further, since an extra webpage `deeplearningextractor.rst` is created, its name is added into the `index.rst` for consistency. There might be some differences after the modification by Dr. Pereira.

[1]https://github.com/microsoft/MMdnn

7.2.3 Problems and Solutions

Incompatible Packages

The incompatible packages are the main reasons for the failed pipeline test in CI. In our project, it failed because of `mxnet` and `cv2`. Both of them are not in the default environment of the CI framework, so it is impossible to run the files that are imported them globally. As mentioned above, moving the importing command inside the function is a good idea and should be applied in all cases.

Chapter 8

Conclusion

Bob has a relatively complete and understandable construction of face verification experiments. We develop some add-ons for that structure. First, we create four `.py` files with five classes to read in the pre-trained neural networks and forward the images to extract features. For models in `mxnet, tensorflow, pytorch` format, it is a good choice to call `MxNetModel(), TensorFlowModel(), PyTorchLoadedModel(), PyTorchLibraryModel(),`respectively. For the model in other formats, call `OpenCVModel()` if it is able to read by `opencv`, or try to convert them into format `.ONNX` first. Second, we try to extend the face annotator package. By modifying the source code from GitHub [1], we introduce the pre-trained *TinyFace* models for face detection. This model is designed for detecting the small faces in the image and only provides the bounding box coordinates, so we estimate the eye locations to stabilize the performance. To test the above changes, we create the baseline tests for each extractor interface and one for the annotator `BobIpTinyface()`. Users should be able to use them by simply calling them in the configuration file of the experiment. We provide a detailed explanation for each package on the *Bob* website. All the baselines have a good performance on database LFW except for VGGFace, whose performance is quite low. Packages are tested on the other databases for generality. All the above-mentioned changes are pushed into the `bob.ip.facedetect` and `bob.bio.face` packages and are waiting for CI before they become available to the public.

[1] https://github.com/chinakook/hr101_mxnet

List of Figures

List of Tables

List of Listings

Bibliography

Anjos, A., El-Shafey, L., Wallace, R., Günther, M., McCool, C., and Marcel, S. (2012). Bob: a free signal processing and machine learning toolbox for researchers. In *Proceedings of the 20th ACM international conference on Multimedia*, pages 1449–1452.

Deng, J., Guo, J., Niannan, X., and Zafeiriou, S. (2019). ArcFace: Additive angular margin loss for deep face recognition. In *CVPR*.

Günther, M., Rozsa, A., and Boult, T. E. (2017). AFFACT: Alignment-free facial attribute classification technique. In *2017 IEEE International Joint Conference on Biometrics (IJCB)*, pages 90–99.

Günther, M., Wallace, R., and Marcel, S. (2012). An open source framework for standardized comparisons of face recognition algorithms. In *ECCV'12 Proceedings of the 12th international conference on Computer Vision - Volume Part III*, pages 547–556.

Hu, P. and Ramanan, D. (2017). Finding tiny faces. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1522–1530.

Huang, G. B., Ramesh, M., Berg, T., and Learned-Miller, E. (2007). Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst.

Saez-Trigueros, D., Meng, L., and Hartnett, M. (2018). Face recognition: From traditional to deep learning methods. *arXiv preprint arXiv:1811.00116*.

Zhang, K., Zhang, Z., Li, Z., and Qiao, Y. (2016). Joint face detection and alignment using multitask cascaded convolutional networks. *IEEE Signal Processing Letters*, 23(10):1499–1503.