Face Detection

Foundations of Machine Learning
Project 2 - Final Report
DETI

Pedro Gonçalves 88859
Student
pedrog8@ua.pt

Rui Oliveira 89216
Student
ruimigueloliveira@ua.pt

Pétia Georgieva
Course Instructor
petia@ua.pt

***Abstract* - This document is the final report of the second project of the Curricular Unit of Foundations of Machine Learning. It contains the state of the art of the topic facial recognition, the description of the data, its visualization, statistical analysis and pre-processing, the description of the applied Machine Learning algorithm(s), the presentation and discussion of the results and finally the respective conclusions.**

Keywords - Machine Learning; Neural Networks; Deep Learning; Convolutional Neural Network; Image processing;

I. State of the art

Initially, a review of some references that deal with the same topic of facial recognition was made. This review played an important role in shaping the direction of our project.

- Article 1

The first article we studied was *Face Recognition with Python, in Under 25 Lines of Code* by Shantnu Tiwari [1]. In this article, we looked at a surprisingly simple way to get started with facial recognition using Python and the open source OpenCV library.
After installing the OpenCV library and understanding the code, we proceeded to test and evaluate the results on two different photos.
As we can see, the algorithm worked well in the first photo with the ABBA band:

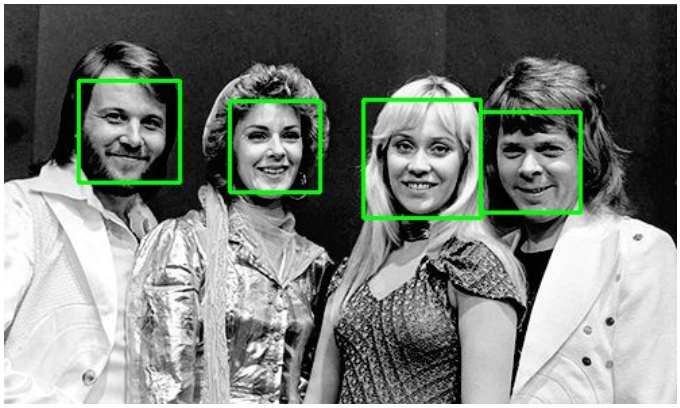

Fig. 1 - ABBA photo results

Testing the second photo the results were not so good:

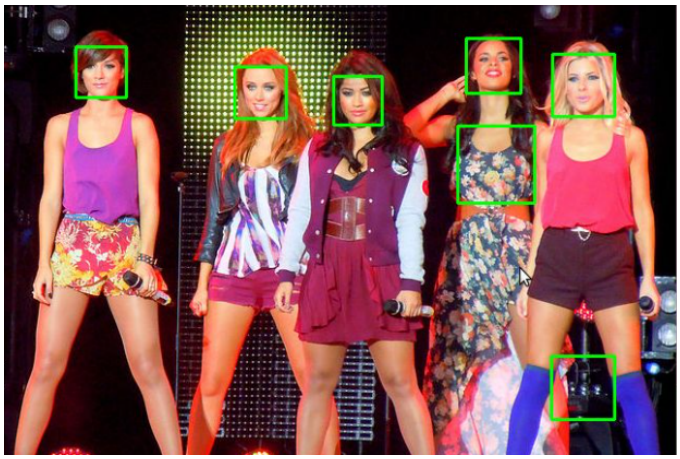

Fig. 2 - Little Mix photo results

Unlike the first photo which was taken up close with a good quality camera, the second one appears to have been taken from afar and possibly at a lower resolution. Thus the algorithm has to be configured on a case-by-case basis to avoid false positives.

We conclude that, despite the results being quite good and the configuration being quite simple, for Machine Learning purposes much of the logic and processing was encapsulated in the OpenCV library, making it impossible to understand the process behind facial recognition and explore all the knowledge and tools we learn in class.

- Article 2

The second article we studied was *How to Perform Face Detection with Deep Learning* by Jason Brownlee [2] whose objective is again the detection of faces in photographs. This tutorial is divided into two parts.

The first involves the detection of faces with OpenCV, which we have already looked into in the first reference and concluded that it encapsulates much of the Machine Learning logic and its norms/rules. For that reason, this part was discarded.

In the second part of the tutorial, Deep Learning is already used for face detection, more specifically "Multi-Task Cascaded Convolutional Neural Network," or MTCNN for short.

The MTCNN is popular because it achieved state-of-the-art results on a range of benchmark datasets, and because it is capable of also recognizing other facial features such as eyes and mouth, called landmark detection.

The network uses a cascade structure with three networks; first the image is rescaled to a range of different sizes, then the first model proposes candidate facial regions, the second model filters the bounding boxes, and the third model proposes facial landmarks.

The image below provides a helpful summary of the three stages from top-to-bottom and the output of each stage left-to-right.

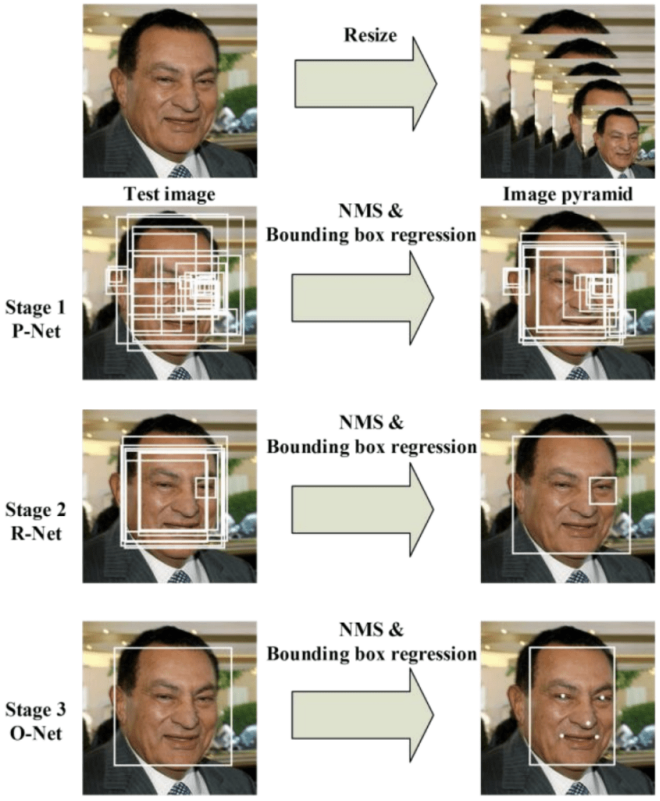

Fig. 3 - Pipeline for the Multi-Task Cascaded Convolutional Neural NetworkTaken

After installing the mtcnn library via pip, we proceeded to the tests.

Fig. 4 - College Students Photograph With Bounding Boxes Drawn for Each Detected Face Using MTCNN

We can see that eyes, nose, and mouth are detected well on each face, although the mouth on the right face could be better detected, with the points looking a little lower than the corners of the mouth.

We conclude that for a "normal" user to have access to a top-performing pre-trained model can be a great benefit.
However, we are looking to develop a little more the issue of training the model and its features and later improve it with our dataset.

- Article 3

In the third article we explore Marcelo Rovai's *Real-Time Face Recognition: An End-To-End Project* [3] that shows how to use PiCam to recognize faces in real-time.

(The Pi camera module is a portable lightweight camera that supports Raspberry Pi. It communicates with Pi using the MIPI camera serial interface protocol. It is normally used in image processing, machine learning or in surveillance projects.)

In this tutorial we are shown how OpenCV allows automatic tracking of faces from a webcam.

One of the most common ways of detecting faces is the “Haar Cascade classifier”. This classifier is a machine learning based approach where the cascade function is trained from a lot of positive and negative images. It is then used to detect objects in other images.

The best way to show the results is not from a report like this, as the results are real-time, i.e. detection is done in multiple frames and not in a single frame. Therefore, you are encouraged to visit the indicated website to see the results in a dynamic way.
However, a picture is presented for demonstration purposes:

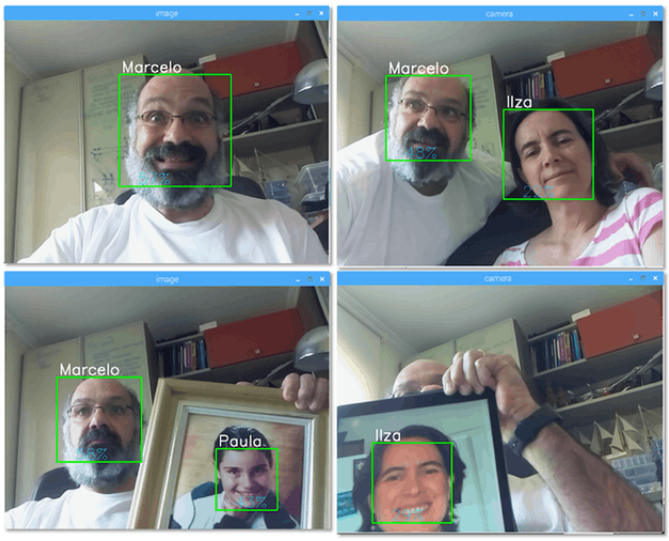

Fig. 5 - Real-Time WebCam face detection

- Article 4

In this blog entitled *Face recognition with OpenCV, Python, and deep learning* by Adrian Rosebrock [4] another already trained network is used to detect human faces with deep learning. This network has already been trained to create 128-d embeddings on a dataset of ~3 million images.

In order to perform face recognition with Python and OpenCV, we need to install two additional libraries:

The *dlib* library contains the implementation of “deep metric learning” which is used to construct our face embeddings used for the current recognition process.

The *face_recognition* library wraps around dlib's facial recognition functionality, making it easier to work with.

The results are excellent as expected from a trained photo network of its size. This project is able to recognize faces both in photographs and videos or even using the webcam (real-time).

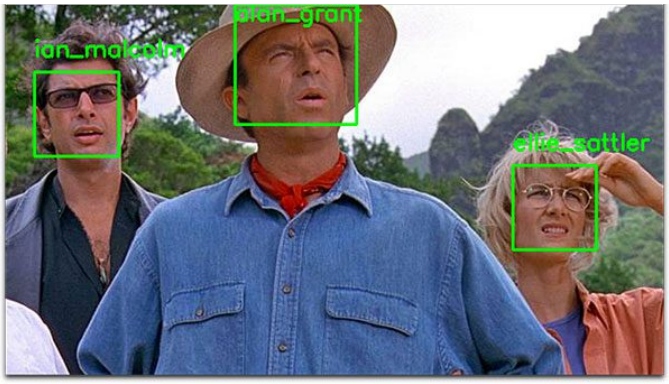

Fig. 6 - Face recognition with OpenCV and Python

The following link redirects to the facial recognition demo in a scene from the Jurassic Park movie:
https://www.youtube.com/watch?v=ZweuO_qcfrw

The following link redirects to the real-time facial recognition demonstration using a webcam
https://www.youtube.com/watch?v=dCKl4oGP69s

- Article 5

In the fifth and final article called Face Recognition with Python by Data Flair [5] there is already more flexibility compared to the other four previously presented. Despite using external libraries such as *face_recognition* and OpenCV, it is already possible to train the network using our own photographs and later classify them as having a face or not.
Therefore, from the moment the dataset is prepared, the model is trained and the appropriate tests are carried out, the results are as follows.

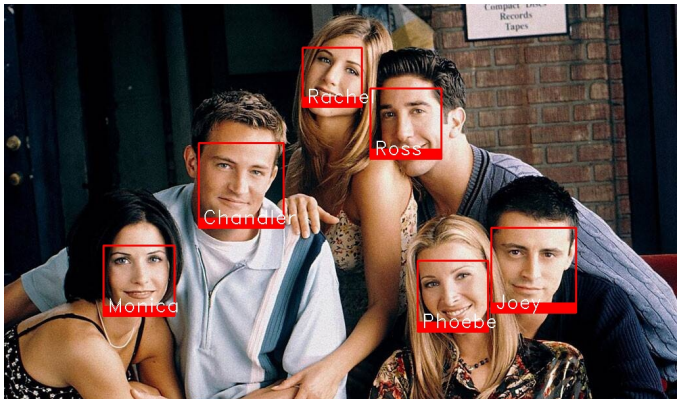

Fig. 7 - Face recognition with OpenCV and Python

From the OpenCV libraries, a rectangle and the person's name around their face are also displayed. This is a common feature in projects that use OpenCV in their implementation.

II. Data Description, Visualization and Statistical analysis

- Objective

Compared to all the articles presented, our project stands out for delving a little deeper into the entire experience of configuring and manipulating data in Machine Learning. By this we mean that although we use some libraries that allow us to abstract a little from the lower layer of computation in the Convolutional Neural Networks, we are still able to apply all the important knowledge taught in the classes. In this way, topics such as data pre-processing, data quality, choice of data size, division between training data and test data, image processing, classification models, batch size, number of epochs, accuracy, loss... were not abstracted by the layer of libraries that, although practical and powerful, take away from us this experience that consequently gives us knowledge of how "everything" works internally.
In short, we propose to make a face detector from images from our own dataset and consequent trained network. It is worth noting the use of keras libraries that facilitate the creation of the CNN model. In the end it will be possible to send an image to our program and it will decide if it contains human faces or not. Furthermore, if an image is classified as having human faces, the location of the faces is pointed out on the canvas with a red rectangle around them. It is important to mention that a pre-trained MTCNN model is used in order to locate the faces in the image.

- Features

Regarding the features of our project, these consist of a dataset of images with human faces and another where they don't exist. From there, we chose to label each of these photographs with a key that tells us whether or not it has a human face. This then allows the CNN model to verify that the results are accurate. From here, we divide the dataset into training and testing (more on this in the data preprocessing chapter).
The content (RGB) of each image is stored in a list that contains the pixels of the same and that serves as the input layer of the CNN model.

Example of **some** of the content of **one** RGB image:

```
[[212 206 206]
  [212 206 206]
  [212 206 206]
  ...
  [ 52  70 106]
  [ 54  72 108]
  [ 55  73 109]]
```

Another list with the same dimension is created but with labels that indicate whether or not each image has a face in it.
Example:

```
[[ 0 1 0 1 1 0 1 0 0 1 0 0 0 0 1 1 0 1 0 1 0 1 1 0 1 1 1 0 … ]]
```

So the '1' represents that a photo contains human faces and the '0' indicates the opposite.

- Statistical analysis

Here is an example of some statistical analysis of the dataset in question:

```
Number of training examples = 1000
Number of testing examples = 1000
Images train shape: (1000, 256, 256, 3)
Labels train shape: (1000, 1)
Images test shape: (1000, 256, 256, 3)
Labels test shape: (1000, 1)
```

In this specific test, the dataset contains 1000 images for training and 1000 for testing (the amount of images with faces and without faces is practically the same as you can see in the data preprocessing chapter). Here we can also verify that, as expected, the number of labels is equal to the number of images. We also see that the resolution of the images is 256x256 pixels.

III. Data preprocessing

We start our program by receiving 2 directories with data, one directory with images with faces, and one directory with images without faces. After receiving the input from the user we start our preprocessing of data.

- Image formatting

We created a function that receives a folder named, for example “dir_name” and a pixel size, which the default is 256. The first thing this function does is create two directories, one called “dir_name_train” which is the directory for the training images, and one called “dir_name_test” which is for the testing images.
Then, we proceed to loop through all the files inside the folder passed as argument from the user and convert each image to a squared image. If the image is not already a square, we add black bars on the top and bottom, or on the sides, so that the image doesn’t become unformatted.
Finally, the user can choose the resolution of the image, or go with our default, which is 256 x 256, just having to keep in mind the better the resolution, the more time consuming the algorithm becomes.

- Data division

The division of data is also automatically made by our function, we copy 75% of the data set to the training folder, which will be used to train the algorithm, and the other 25% to the testing folder, which will be used to validate the algorithm.

- Avoid oversampling and undersampling

Let us first clarify the concepts of oversampling and undersampling. The first one occurs when we use an excessive size of data, in this case to train the model, the second is the opposite, meaning that we’re using a non sufficient size of data for the purpose.
To avoid both problems, we applied a filter to the data, making it only accept around 1000 images, even if we have a dataset with 20000 images. This way both the folders (“faces” and “no_faces”) will have a similar number of training and testing images, making the algorithm much more efficient and accurate.

- Normalization of data

At last, when all data is processed and organized, we have to normalize it. In order to do that, we go through each image and normalize the value of each pixel. Since the RGB scale goes from 0 to 255, we divide each value of it by 255, so that we have values between 0 and 1. Then we proceed to train our model with the now normalized values.
This makes the whole process of training way faster, since we work with smaller numbers.

IV. Description of the applied Machine Learning algorithm(s)

Moving on to the applied Machine Learning algorithms, we decided to use the Keras library.
Keras is a high-level neural networks API written in Python, and is capable of running on top of several lower-level frameworks like TensorFlow and CNTK. It is also a relatively simple way to work with Deep Learning using minimal code, allowing for rapid prototyping.

- Model description

The first thing we did was to build a function to describe our model, “FaceDetectionModel”.

```
def FaceDetectionModel(input_shape):
    """
    Implementation of the FaceDetectionModel.

    Arguments:
    input_shape -- shape of the images of the dataset

    Returns:
    model -- a Model() instance in Keras
    """
    X_input = Input(input_shape)

    # Zero-Padding: pads the border of X_input with zeroes
    X = ZeroPadding2D((3, 3))(X_input)

    X = Conv2D(32, (7,7), strides=(1,1), name = 'conv0')(X)

    X = BatchNormalization(axis = 3, name = 'bn0')(X)

    X = Activation('relu')(X)
    max_pool_size = (2, 2)
    X = MaxPooling2D(max_pool_size,
name='max_pool')(X)

    X = Flatten()(X)
    X = Dense(1, activation='sigmoid', name='fc')(X)

    # Create model.
    model = Model(inputs = X_input, outputs = X,
name='FaceDetectionModel')

    return model
```

After creating the model structure, we have to create the model and compile it with the correct parameters.

```
facedetectionmodel =
FaceDetectionModel(imgs_train_norm.shape[1:])

facedetectionmodel.compile(optimizer='adam',
loss=BinaryCrossentropy(), metrics=["accuracy"])
```

In the first two lines we simply instantiate the model.
In the last two we compile the model. This compile function receives 3 arguments, the first one being the optimization algorithm, second one the loss function and the last one the metrics. This function can receive extra parameters but they are not necessary and we opted for not using any.

- Optimization algorithm

For the optimization we chose "Adam". Adam is a stochastic gradient descent method based on adaptive estimation of first-order and second-order moments.
This method is computationally efficient and has little memory requirement, which is great for training a simple model with small data.

- Loss function

Since our model was based on a problem of 1's or 0's the choice of the loss function was not hard. BinaryCrossentropy() is a cross-entropy loss for binary classification applications, which computes the loss between the labels and predictions.

- Metrics

Let's start by explaining what a metric is. Basically it's a function that is used to judge the performance of our model. In our case we wanted a model that could accurately classify an image as having a face or not, so for the metrics argument we simply chose "accuracy" since it was our top priority.

- Model summary

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 256, 256, 3)]     0

 zero_padding2d (ZeroPadding  (None, 262, 262, 3)      0
 2D)

 conv0 (Conv2D)              (None, 256, 256, 32)      4736

 bn0 (BatchNormalization)    (None, 256, 256, 32)      128

 activation (Activation)     (None, 256, 256, 32)      0

 max_pool (MaxPooling2D)     (None, 128, 128, 32)      0

 flatten (Flatten)           (None, 524288)            0

 fc (Dense)                  (None, 1)                 524289

=================================================================
Total params: 529,153
Trainable params: 529,089
Non-trainable params: 64
```

Fig. 8 - Model summary

In the picture above we can see some details about the data we are using and the results of each step of the model. For example, the number of trainable and non-trainable parameters.

- Model training

Let's dive into how we trained our model.
For that we used the keras' fit method, which trains the model for a fixed number of epochs.

```
history = facedetectionmodel.fit(
   imgs_train_norm, label_train_T,
   batch_size = 64,
   epochs = 20,
   validation_data = (imgs_test_norm, label_test_T))
```

This method can receive up to 19 arguments but we only found necessary to use 5 of them.

- imgs_train_norm: Our image training data.
- label_train_T: The labels of our images (1 or 0)
- batch_size: Number of samples per gradient update, we opted to use 32.
- epochs: Number of epochs to train the model. In other words, is the number of iterations over the entire data provided, we chose to use 15, but if the accuracy is low, we train the model more times.
- validation_data: This parameter compares the training values with the test data (imgs_test_norm and label_test_T) and gives us some info on the validation accuracy and loss.

- Model evaluation

Finally, we use the Keras's evaluate method to evaluate the results of our training. This method returns the loss value and metrics value for the model in test mode.

```
preds = facedetectionmodel.evaluate(x = imgs_test_norm, y
= label_test_T)
```

If the accuracy returned by this evaluate method is higher than 95% we consider that the model is good to go, but obviously the higher the accuracy the better. If 95% is not enough we can train the model further to achieve better results.

- Model structure

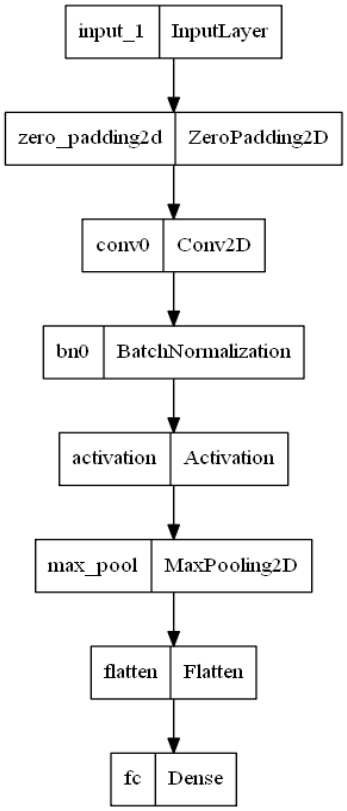

Fig. 9 - Structure of our model

In the image above we can see the steps our model takes before being ready to execute.

V. Presentation and Discussion of Results

In this section two models are tested to train our dataset. The first is related to Article 1 and the second is related to the model we created and adapted for our own dataset.

- *Model 1 Results*

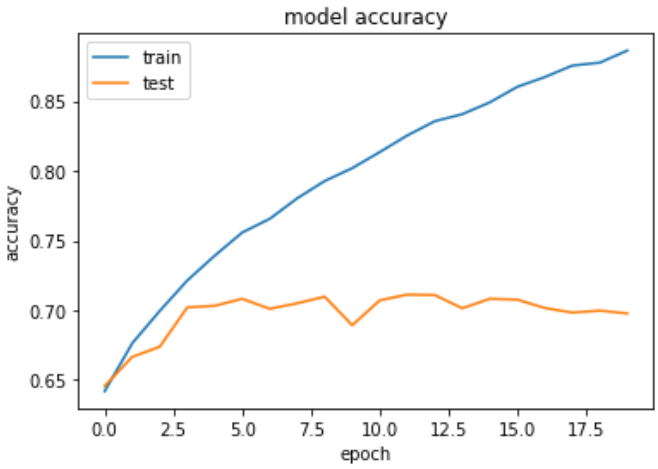

Fig. 10 - Article 1 Model accuracy results

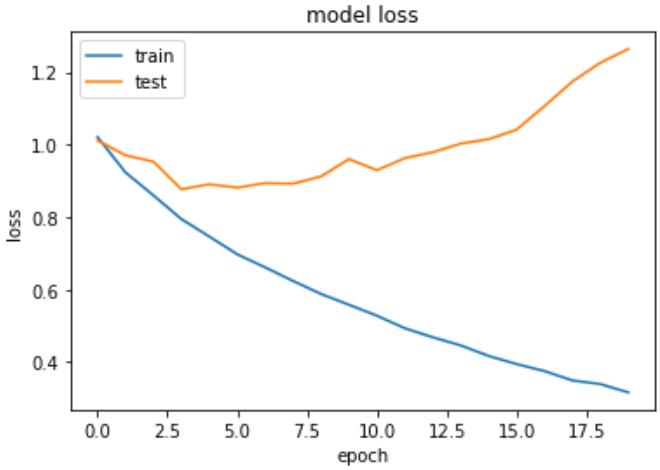

Fig. 11 - Article 1 Model loss results

- *Our Model Results*

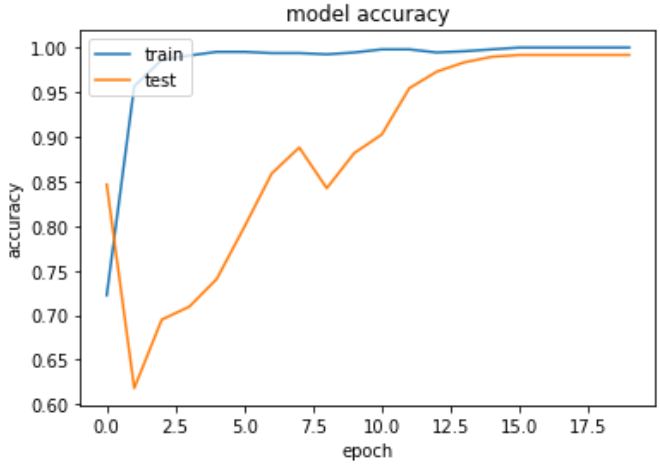

Fig. 12 - Model accuracy

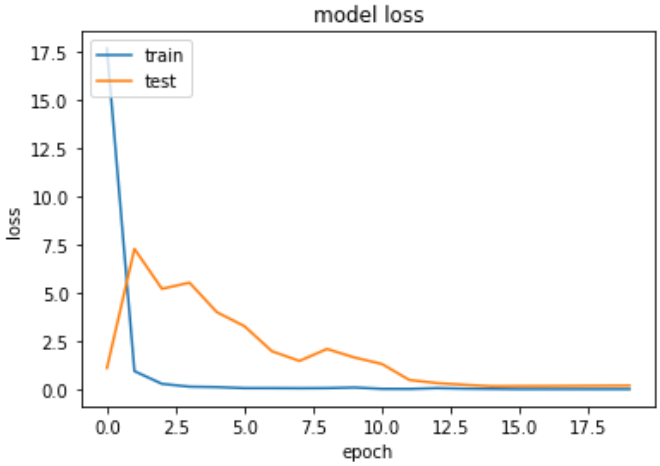

Fig. 13- Model loss

- *Comparison*

After training and evaluating the first model with our testing data we reached interesting results, the model accuracy wasn't improving for the testing data, peaking at around 70% on epoch 12. The model loss was also getting worse with the epochs for the testing data.

Our results were more satisfactory, with an accuracy of around 93%, depending of course on the number of training images, and the epochs of the training model. In the graphics above we notice that our model accuracy increases visibly as we run the model more epochs, both for the training and testing data. In addition, we can see the model loss decreasing through the epochs and stabilizing around epoch 13 for the testing data.

After concluding that our model obtained better results with a higher precision and accuracy, we proceeded to the tests with images outside the dataset and the results are found in the next section.

- Results

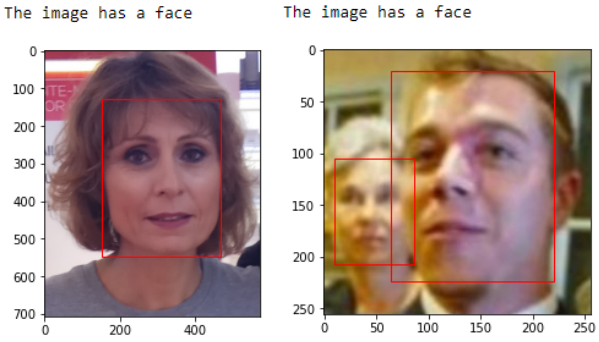

Fig. 14 & 15 - Images with recognized human faces

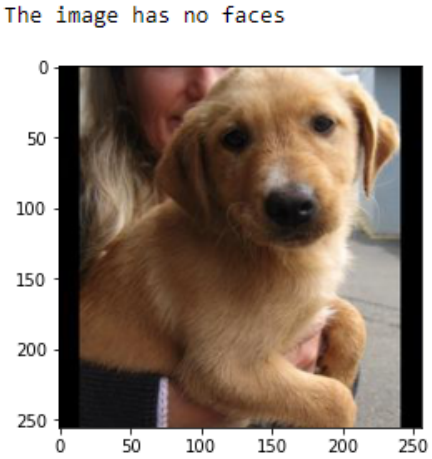

Fig. 16 - Image without human faces

In the first two images we can see that our algorithm detected there was a face in the picture, and so we also applied the MTCNN model to detect the location of the faces.
In the third image, there is no human face, so our algorithm prints "The image has no faces" and the MTCNN model isn't applied.

We are very satisfied with our results. However, our model suffers a bit from overfitting, probably because the data we used to train and test is not as much as we'd like (only about 900 images for each case).

Our final results for the "test_imgs" were as such:

- Accuracy: 0.925
- Precision: 0.944
- Recall: 0.894
- F1 Score: 0.919

This means some of the images can be labeled as having a face even though they don't., especially when the image has some similarities with an image of a face. It also means that some pictures of faces might be labeled as not having a face, mainly when the face occupies a small number of pixels of the image.

VI. Conclusions

Throughout this project, we acquired a lot of knowledge on several topics of Machine Learning. As far as convolutional neural networks are concerned, we have come to some interesting conclusions. Despite having a very high accuracy in this type of face recognition problems and automatically detecting the main features, CNN does not encode object orientation and a lot of training data is needed which makes the training time quite extensive.

However, we believe that we made the right decision regarding the choice of the model and mainly because we did not use a pre-trained model. This allowed us to better understand how Convolutional Neural Networks work.

In the future our main goal is to increase the size of the dataset and train the model more exhaustively. From there, we also intend to make the trained model available online.

VII. Contributions

We consider that both elements of the group participated equally in the development of the project.

VIII. REFERENCES

[1] Face Recognition with Python, in Under 25 Lines of Code
https://realpython.com/face-recognition-with-python/

[2] How to Perform Face Detection with Deep Learning
https://machinelearningmastery.com/how-to-perform-face-detection-with-classical-and-deep-learning-methods-in-python-with-keras/

[3] Real-Time Face Recognition: An End-To-End Project | by Marcelo Rovai | Towards Data Science
https://towardsdatascience.com/real-time-face-recognition-an-end-to-end-project-b738bb0f7348

[4] Face recognition with OpenCV, Python, and deep learning - PyImageSearch
https://www.pyimagesearch.com/2018/06/18/face-recognition-with-opencv-python-and-deep-learning/

[5] Face Recognition with Python [source code included] - DataFlair (data-flair.training)
https://data-flair.training/blogs/python-face-recognition/