Facial Keypoints Detection

Detect the location of keypoints on face image

Mattia Capparella, 1746513
Federico Fontana, 1744946

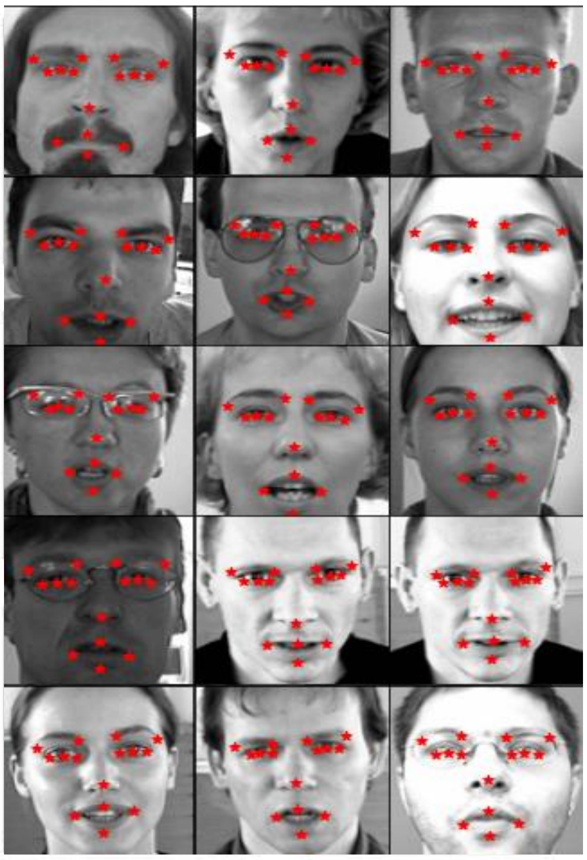
Department of Computer Science,

Course: Biometric Systems, AY 2020/21

Instructor: Maria De Marsico

Introduction

Problem



Facial Key Points (FKPs) Detection
is an important and challenging problem in the fields of computer vision and machine learning. It involves predicting the co-ordinates of the FKPs, e.g., nose tip, center of eyes, etc, for a given face.

This project is about experimenting different models in FKPs Detection that is a critical element in face recognition.

However, there is difficulty to catch keypoints on the face due to complex influences from original images, and there is no guidance to suitable algorithms.

The **problem is** to predict the (x, y) real-valued co-ordinates in the space of image pixels of the FKPs for a given face image. It finds its application in tracking faces in images and videos, analysis of facial expressions, detection of dysmorphic facial signs for medical diagnosis, **face recognition**, etc.

Facial features **vary greatly from one individual to another**, and even for a single individual there is a large amount of variation due to pose, size, position, etc. The problem becomes even more challenging when the face images are taken under
different illumination conditions, viewing angles, etc.

Objective

With this report we want demonstrate that a good preprocessing can increase performance without modifying the model and without slowing down the algorithm too much.

Dataset

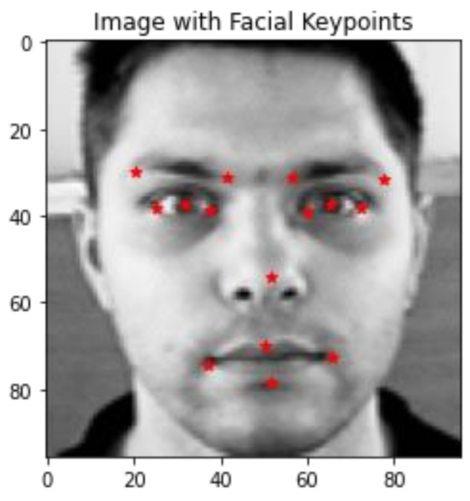
The dataset is taken from the Keggle competition [Facial Keypoints Detection]():

This dataset is composed of 4 files:

- **training.csv**: 7049 *"96x96"* training images. Each row contains the (x,y) coordinates for 15 keypoints, and the image data encoded as a string of pixels.
- **test.csv**: 1783 *"96x96"* test images. Each row contains the *ImageId* and the image encoded a string of pixels.
- **IdLookupTable.csv:** 27124 keypoints to predict. Each row contains a *RowId, ImageId, FeatureName* and finally the *Location,* our target.

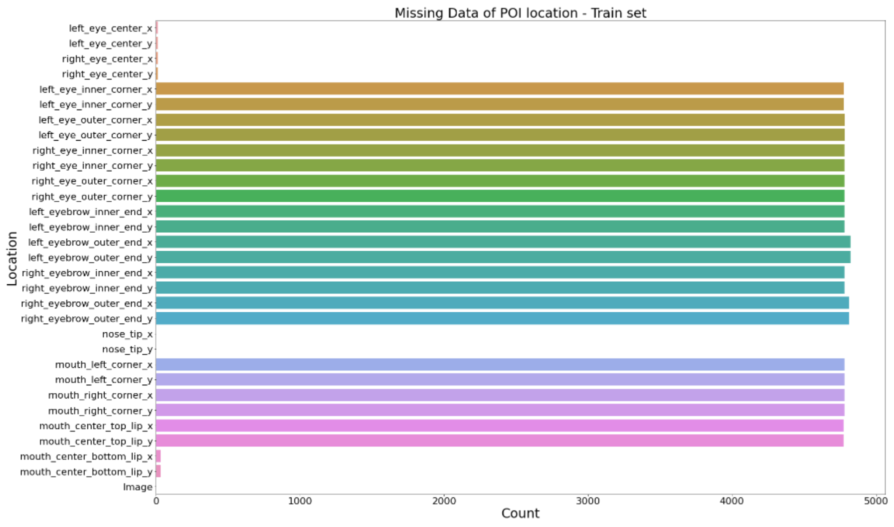
In the dataset there are 15 different keypoints, each one specified by a (x,y) pair of coordinates.
The 15 points represent the following elements:

1. *left_eye_center*
2. *right_eye_center*
3. *left_eye_inner_corner*
4. *left_eye_outer_corner*
5. *right_eye_inner_corner*
6. *right_eye_outer_corner*
7. *left_eyebrow_inner_end*
8. *left_eyebrow_outer_end*
9. *right_eyebrow_inner_end*
10. *right_eyebrow_outer_end*
11. *nose_tip*
12. *mouth_left_corner*
13. *mouth_right_corner*
14. *mouth_center_top_lip*
15. *mouth_center_bottom_lip*



note: *left&right* here refers to the point of view of the subject depicted.

It is important to point out that many *keypoints* in the training dataset are missing:



With the removal of images whose (some) keypoints are missing, we would remain with only 2140 images: a loss of possible valuable information of about 68%.

Tools, libraries, and references

Google Colab

To carry out the training of our models, we have used Google Colab since it offers a free and efficient platform that greatly reduces the *training time,* by providing high-capacity Nvidia GPUs usage.
We made two notebooks: a training one and a demo one used to test each of the models explained further.

Pytorch and Torchvision

To develop our model, a *Convolutional Neural Network* (CNN), we employed the Pytorch framework for its ease of use and the speedup provided by enabling the computations on GPUs.

Pytorch comes with the Torchvision library, a collection that contains many functions for image manipulation, used extensively in our preprocessing phases.

Model selection

To demonstrate valid the idea of applying a *preprocessing phase* we looked for a pre-made model that performed well on the very same task, with no preprocessing involved.
Since FKPs detection is a task performed in a vast number of *online scenarios,* it was fundamental that at testing time the application of such new methodology would have improved the results, while not affecting negatively the time performance.
For these reasons we used LuisOlCo's model, as published on Github (more precisely: the generic CNN, not the LeNet5 one).

LuisOlCo's solution:

The CNN is so formed:

- 4 *convolutional layers*
- 3 *fully connected layers* (note that the last layer has 30 nodes because every FKS is described by a pair of coordinates)

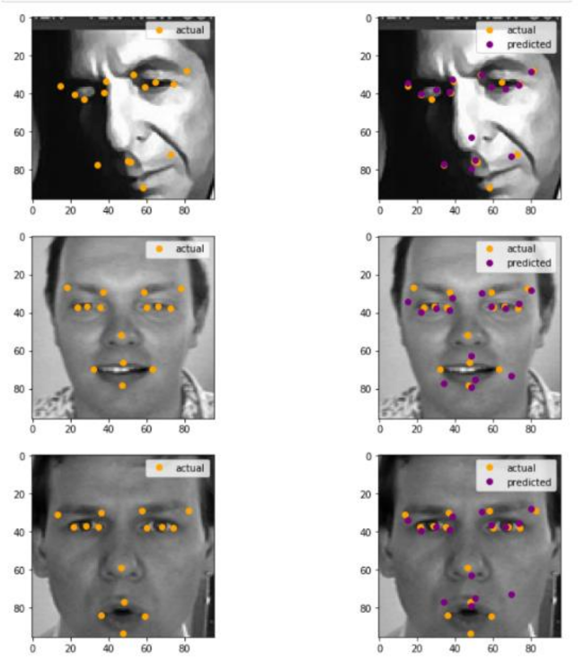
The *error function* being:

- *mean squared error* (MSE)

and the *optimizer:*

- *Adaptive moment estimation* (Adam)

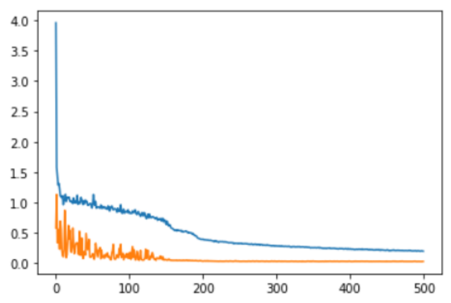
As shown by the *train/val* loss plot and the *scatter plot,* the results obtained are quite accurate, but the *train loss* has a "*slow"* and the *validation loss* seems suffering from hysteresis and the best results are obtained after a great number of iterations.



```
End of epoch 490:
Training error = [0.20617914362179673]  Validation error = [0.03
1164655382370406]
End of epoch 500:
Training error = [0.19917828310532068]  Validation error = [0.03
191014285932485]
```

```
plt.plot(train_error_list)
plt.plot(valid_error_list)
```

```
[<matplotlib.lines.Line2D at 0x7f2df339cc50>]
```

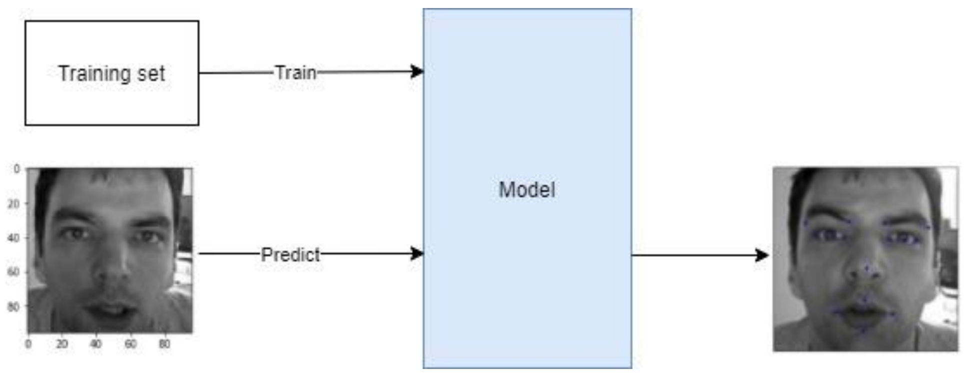


Our contribution:

We focused on both scenarios of the *preprocessing phase*:

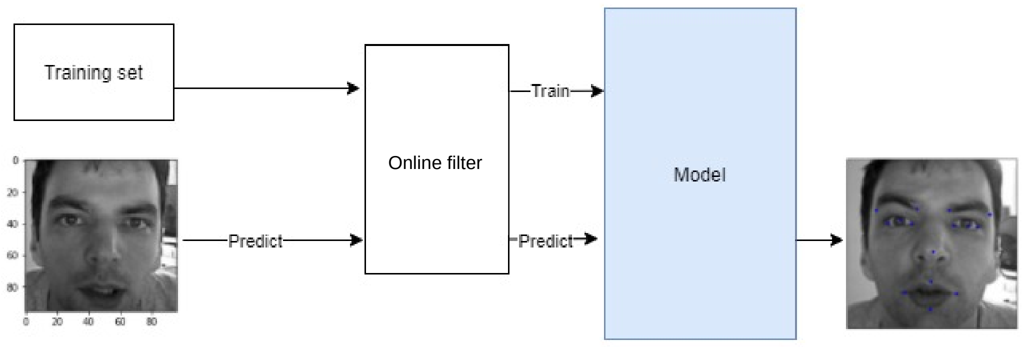
- *Training*: we can use augmentation to increases dataset size and introducing natural image distortion to make the model generalize better.
- *Real-time*: the speed is fundamental, so that we can apply our transformations to (a sequence of) images even *online*.

Baseline Model



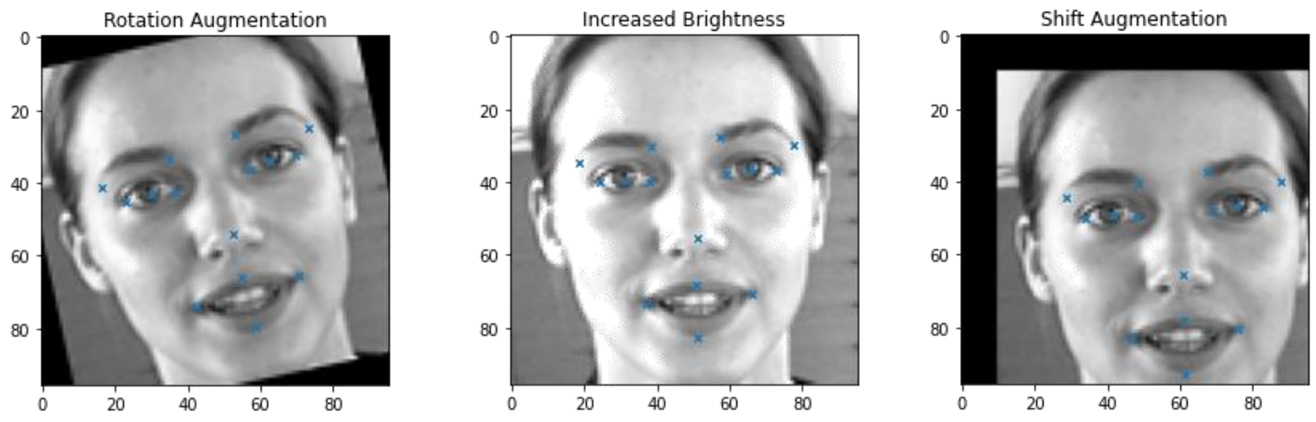
This model trains and makes its predictions without preprocessing. We will use this model to make comparison against ours.

Online Augmentation Model



In here the modifications we perform are the following:

- Rotations (counter/clockwise)
- Changes in brightness (increase/decrease)
- Shift
- Random noise



note: each *pad pixel* of the image is randomly colored with a certain probability P_{noise} independent from the other pixels of the image, set as $P_{noise} = 0.025$ in our experiments.

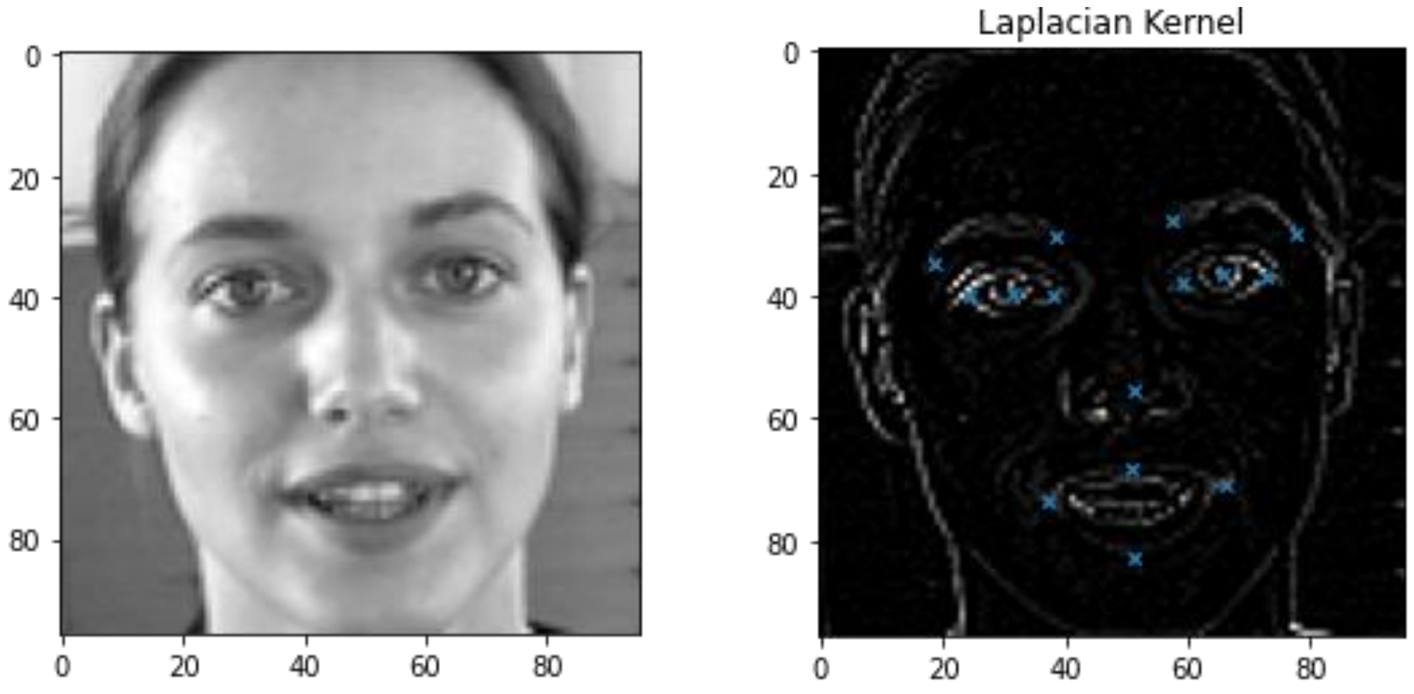
This model does not use pre-augmented datasets but applies instead a filter just before the beginning of every *training epoch* using different probabilities to transform each image in the *batch dataset* “*in a unique way*” to train a model able to generalize better.

Real-time Laplacian Filtered Model

In here the modification we perform is only one:

- Application of the *Laplacian Operator*

The idea behind this transformation is that since we are looking for FDKs that lie on *"natural edges"* of the face, such as eyes or lips contour, eyebrows and so on… we could remove all the "non relevant information" from the original image and keep only the lines/edges on top of which we can find the points we are interested in.
To do so, the Laplacian Operator seemed to be the perfect way to get what we wanted:



Since the *execution time* is a critical problem, let's do some math:
we want use this on *real-time scenarios*, so the transformation time must be less than:

$$30\,frames : 1\,s = 1\,frame : \frac{1}{30}\,s$$

i.e. to transform an image as it arrives, we must perform it in less than $30\ \mathrm{ms}$, but considering that the model takes $15\ ms$ to predict one image we have:

$$30\,ms - 15\,ms = 15\,ms$$

$i.e.$ less than $15\ ms$ to transform an image.

Performance evaluation

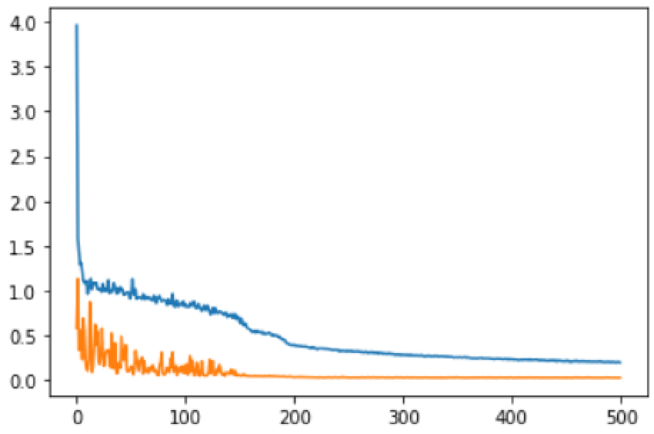
In this section we are going to analyze the performance achieved by every single model

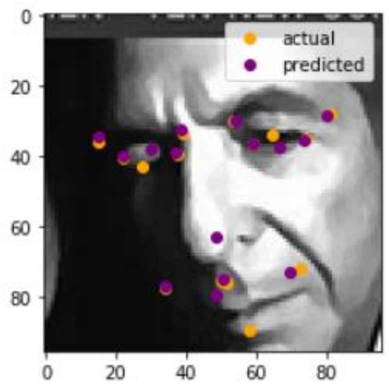
Baseline Model

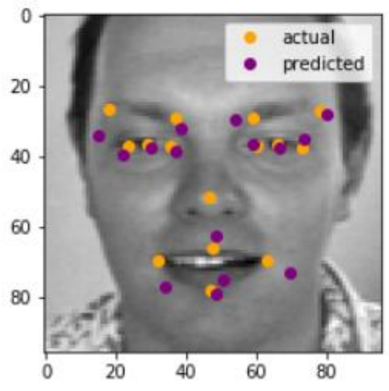
```
End of epoch 500:
Training error = [0.19917828310532068]  Validation error = [0.03
191014285932485]
```

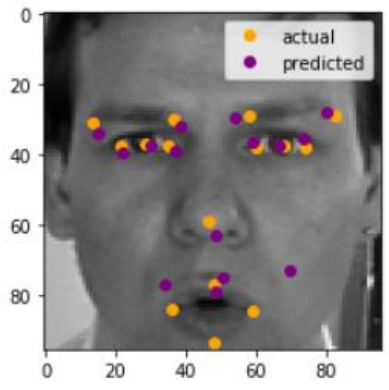
```
plt.plot(train_error_list)
plt.plot(valid_error_list)
```

```
[<matplotlib.lines.Line2D at 0x7f2df339cc50>]
```



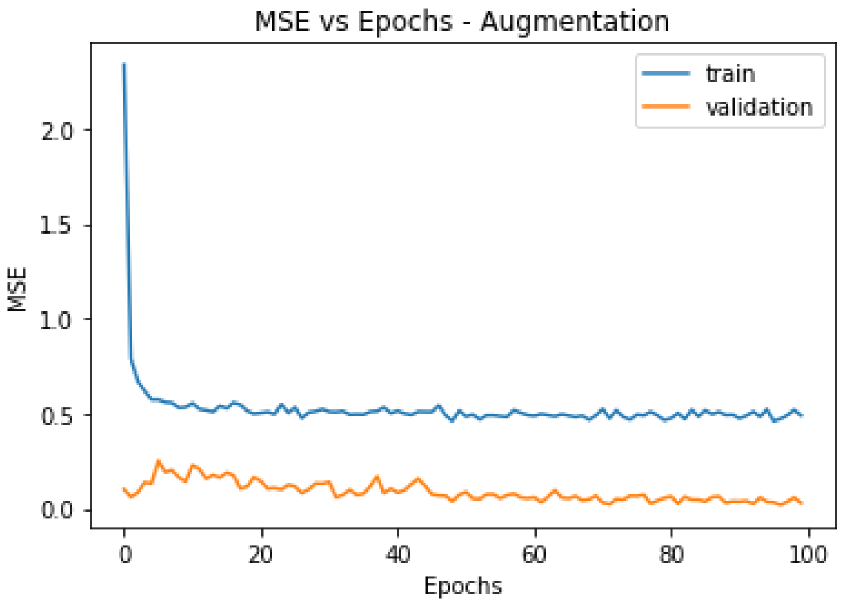






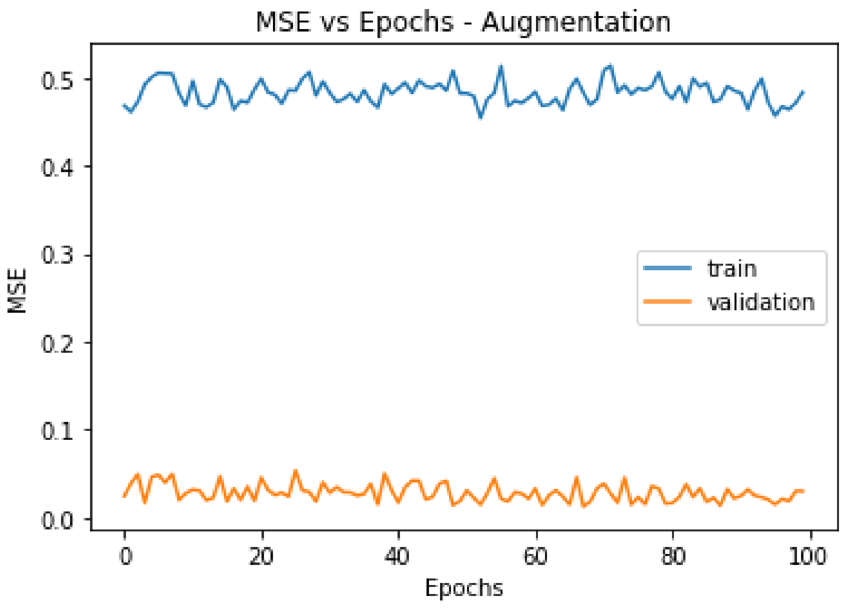
As we already said, the results obtained with this model of are quite good, but the problem is the fact that the *train loss* descend is slow and, most importantly, the *validation loss descent* suffers from hysteresis

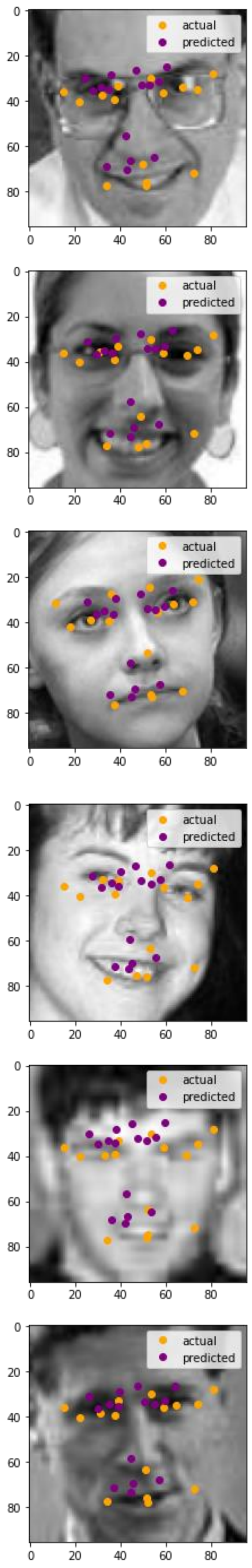
Real-time Augmentation Model



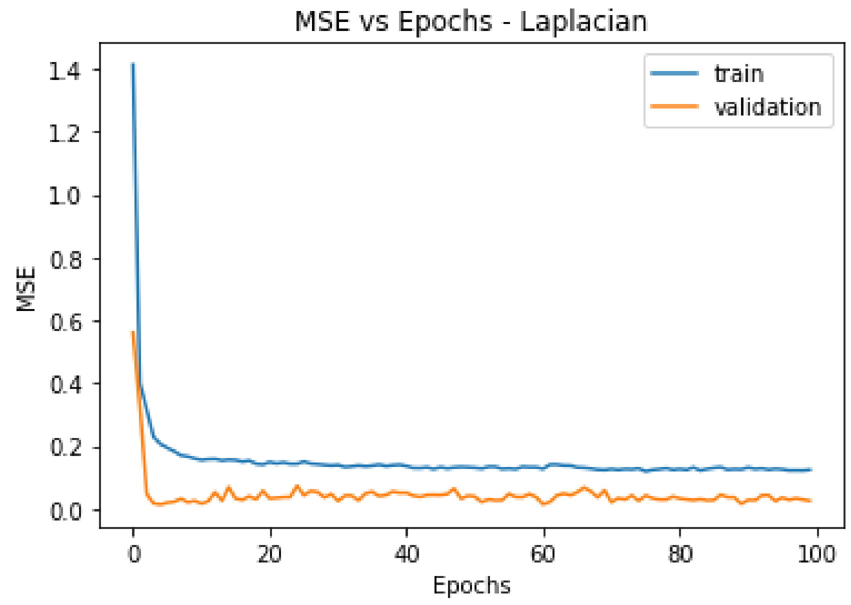
By applying the *on-line augmentation,* we have a faster descent for the *train loss* and a "*smoother" validation loss w.r.t.* the *base model,* but as we can see from the *scatter plot,* the results are not so promising: the predicted points seem to follow a right pattern disposition, but they are quite distant from the actual positions.

Since the previous model trains for 500 epochs, we tried adding 100 epochs more, but the results did not get any improvement.



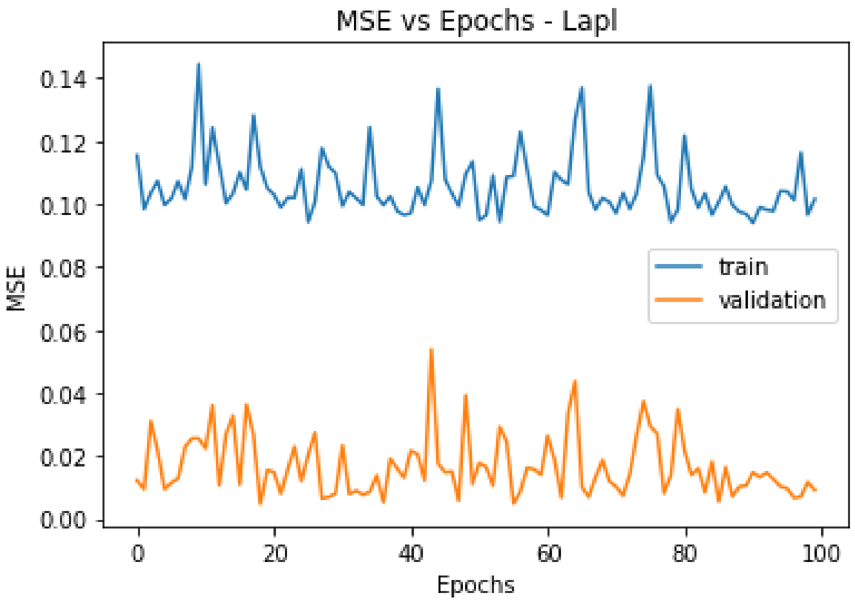


Real-time Laplacian Filtered Model

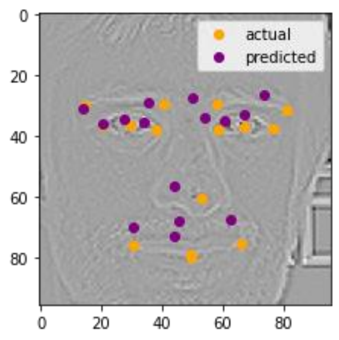


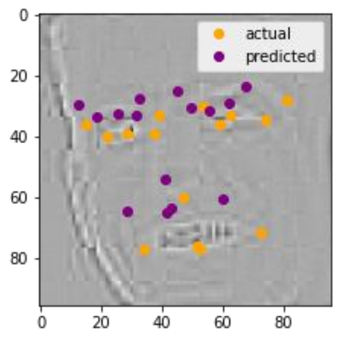
By simply applying the Laplacian transformation, we got a faster *train loss descent* and a more stable *validation loss*, while training only for 100 *epochs.* The problem was that even with a lower and more stable *loss*, the predicted coordinates for many points still resulted quite distant from their actual location.

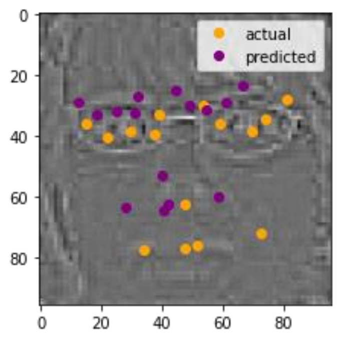
But as soon as we trained the same model for 100 epochs more (arriving at 200 epochs), not only both the *train* and *validation losses* got reduced even more, but the predicted positions started looking way better.

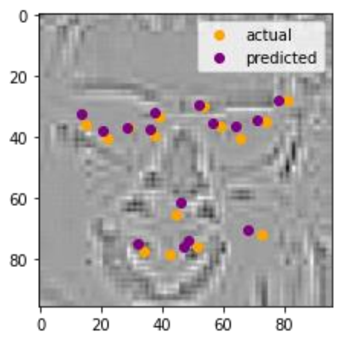


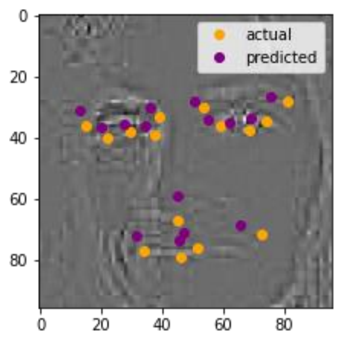
note: here the spikes do not oscillate as previous model: the range of oscillation is much shorter

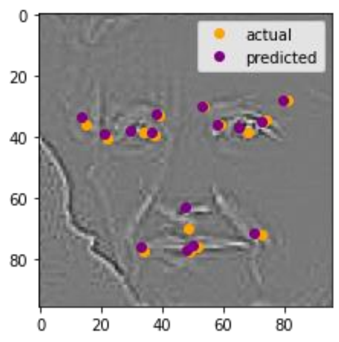








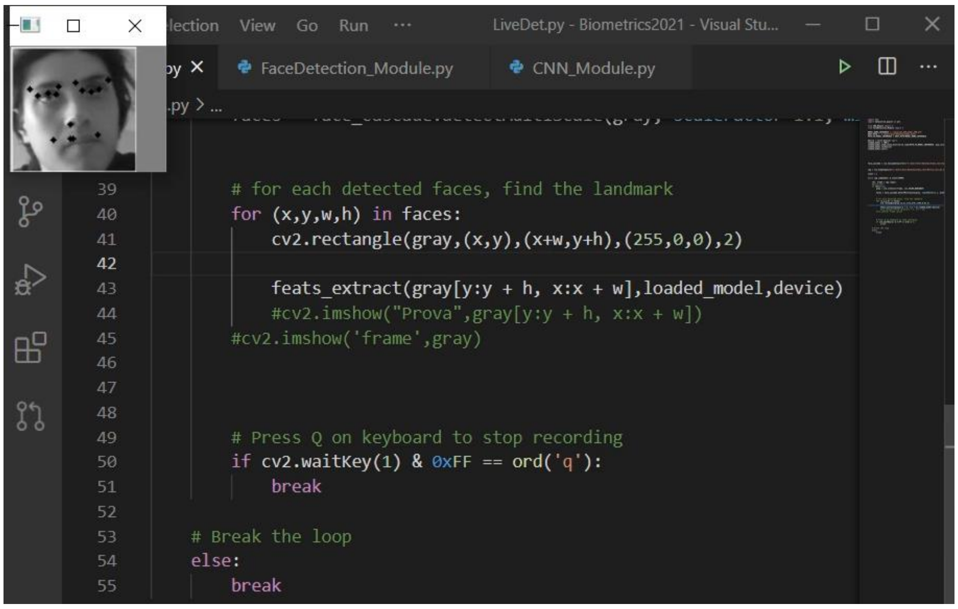




Live Detector (LD)

As a final touch, we implemented a toy *“live” features extractor* to test how our solution would have performed in a *Real Scenario.*
Here it is shown a snippet where the *feats_extract* function is called: the function takes in in input what’s the camera is recording, the detected (by a pre-made *Haar Cascade Face Detector*) cropped region of the face localization an our model.
The coordinates of the FKPs are computed and drawn directly on the cropped image.

For the sake of simplicity, we have not tested it as a “*live detector”*, but we implemented a simple snippet to record a video from the integrated pc camera, and pass it to the *Live Detector* at a later time (with small changes it is possible to pass directly to the *LD* what’s the camera is recording in live session).



```
        # for each detected faces, find the landmark
        for (x,y,w,h) in faces:
            cv2.rectangle(gray,(x,y),(x+w,y+h),(255,0,0),2)

            feats_extract(gray[y:y + h, x:x + w],loaded_model,device)
            #cv2.imshow("Prova",gray[y:y + h, x:x + w])
        #cv2.imshow('frame',gray)



        # Press Q on keyboard to stop recording
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    # Break the loop
    else:
        break
```

Further consideration

We have tested the LD in a couple of different scenarios, testing its robustness to pose variation, and we found out the “tracking systems” does not perform so well: the FKPs related to the most deformable part of the face (such as the lips) result often offside: further investigation must be taken to assess if the problem is due to either the model or the *LD* system.
Moreover, analyzing accurately the labelling of the train dataset, it is possible to point out some inconsistences with the *spots* marked as “actual location” (of the features) and the actual feature: this could also have been the cause of more error.

Conclusions

Given these results, we can conclude that:

- It is possible applying some real-time filter increasing the performance without degrading the *time performance.*
- The augmentation alone performs better than the base model, but it does not achieve desirable results for a reliable application in real life scenarios.
- Best results are obtained when *Laplacian Operator* is applied to the *training dataset* and to the incoming images during testing time.
- With such model the *training phase* is reduced by a significant amount of time, while obtaining reliable, stable results.

References

1. Connor Shorten et al., 2019,
   A survey on Image Data Augmentation for Deep Learning,
   https://www.researchgate.net/publication/334279066_A_survey_on_Image_Data_Augmentation_for_Deep_Learning
2. Naimish et al., 2017,
   Facial Key Points Detection using Deep Convolutional Neural Network - NaimishNet,
   https://arxiv.org/pdf/1710.00977.pdf
3. Rucha Waghulde, Luis Oliveros Colón and Siddharth Mandgi, 2019,
   Facial-Keypoints-Detection,
   https://github.com/ruchawaghulde/Facial-Keypoints-Detection

Resources

- Google Colab’s notebook: Biometric Project

- GitHub (private) repo: Biometrics2021