Neural Networks for face recognition

Lucas Sort - Schneider Yoann - Jeremy Rambaosolo

December 2020

Contents

1 Building and training an ANN

1.1 Building a balanced database using FDDB

To train and test our neural network we need a balanced database of imagettes with faces and others without faces. The whole dataset was split into three parts with different roles. The training dataset was used to fit the model, the validation dataset used to evaluate the model during the training at each epoch whereas the test dataset was used to evaluate the final performance after training. The first model we designed and used for this part was a 3-layer model with a *sigmoid* as activation function, *Adam* as an optimizer and 0.001 as learning rate. To monitor and halt the training, we used the callback functions `ReduceLROnPlateau` to reduce the learning rate and `EarlyStopping` to stop the training when a metric stops improving. Both of these use the validation dataset to pick the right time to change the training. You can see here the performances obtained with this dataset.

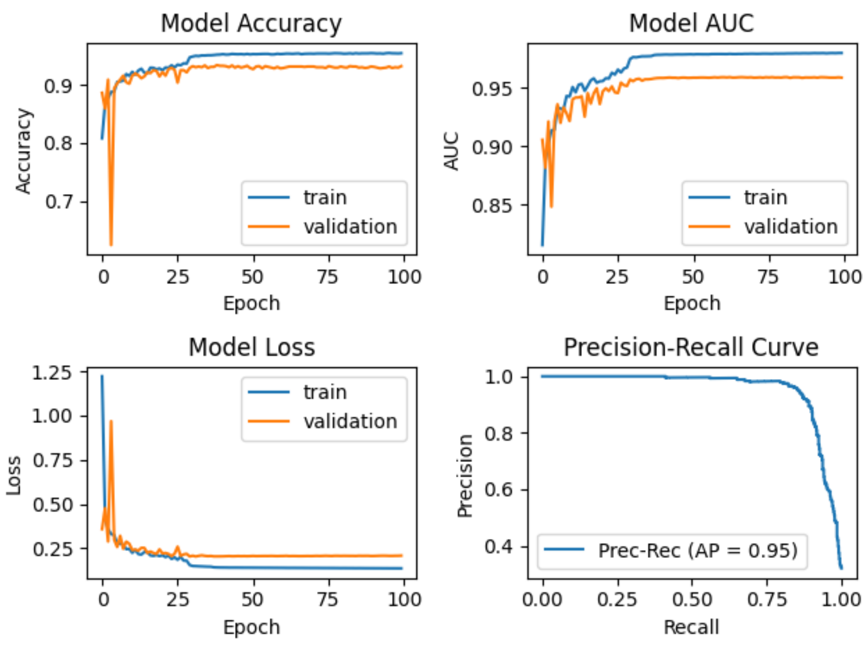


Figure 1: Performances of this model

1.2 Building a bigger dataset using FDDB and WIDER

We built this database from the FDDB database and WIDER database, the FDDB has 5171 faces and WIDER has 393,703 faces with different poses, lightning, expressions and scales. We used these two databases for built our balanced database in order to have a large number of imagettes with faces and thus allow our network of neurons to learn information that will not be too specific. However, in order to don't have a database to hard to learn for our neural network, we made a refinement of these database. We took the faces larger than 24x24 and for the WIDER database, we also took only the faces that were clearly visible. After this refinement, we have approximately 14,000 imagettes with faces in our database. But for training and evaluate a neural network we also need a imagettes without faces. To do this, we randomly extracted imagettes from both databases, making sure we didn't choose the same imagette twice. So finally, we built a database of imagettes with 14,000 positive imagettes (with a face) and 30,000 negative imagettes (without a face). Finally, you can see below the performances obtained with this dataset.

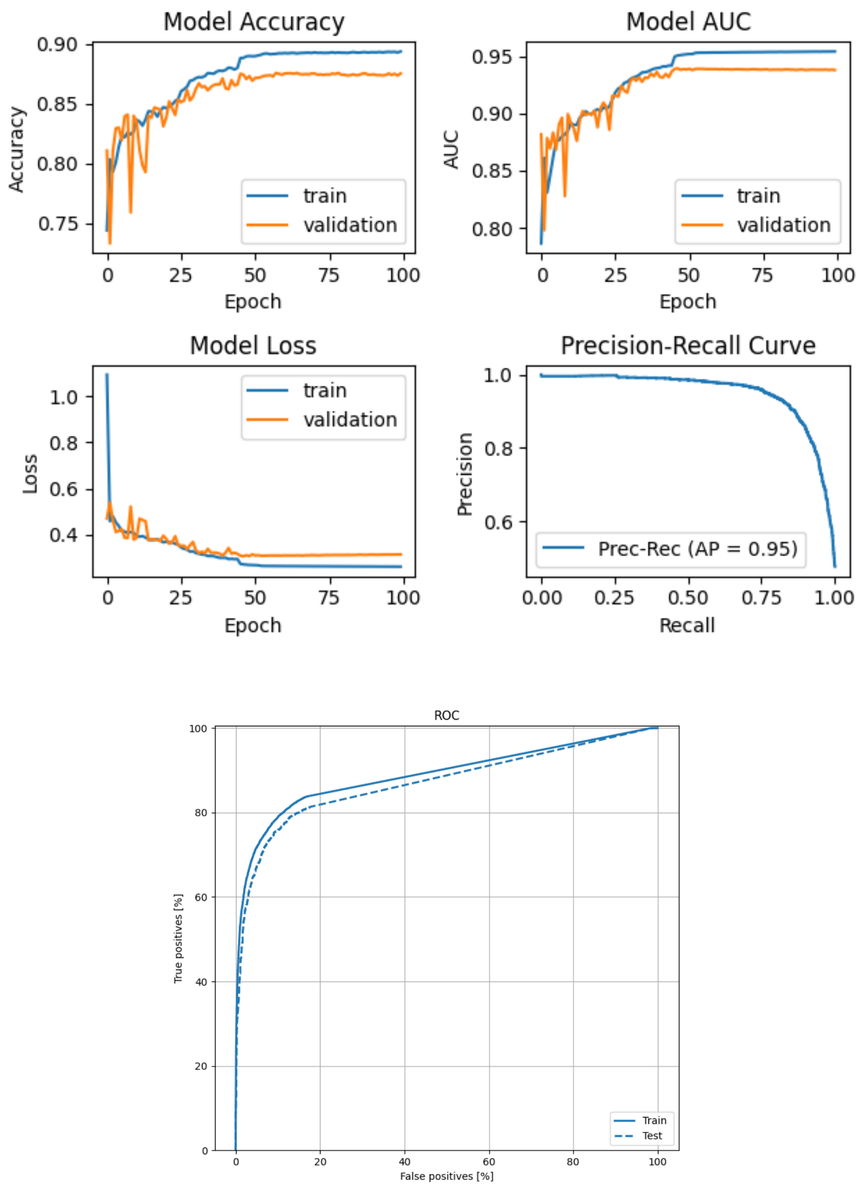


Figure 2: Performances on the dataset

2 Evaluation of the impact of the imagettes

2.1 Face Window Resolution

We evaluated the performance of the model over a range of Face Window resolutions. We used 8-by-8, 16-by-16, 32-by-32 and 64-by-64 imagettes to train 4 different models. To evaluate the model, we used cross-validation to have a better grasp of the model's capability to generalize. We used Keras's `KerasClassifier` class to be able to use the *cross_val_score* function from Scikit-Learn [1].

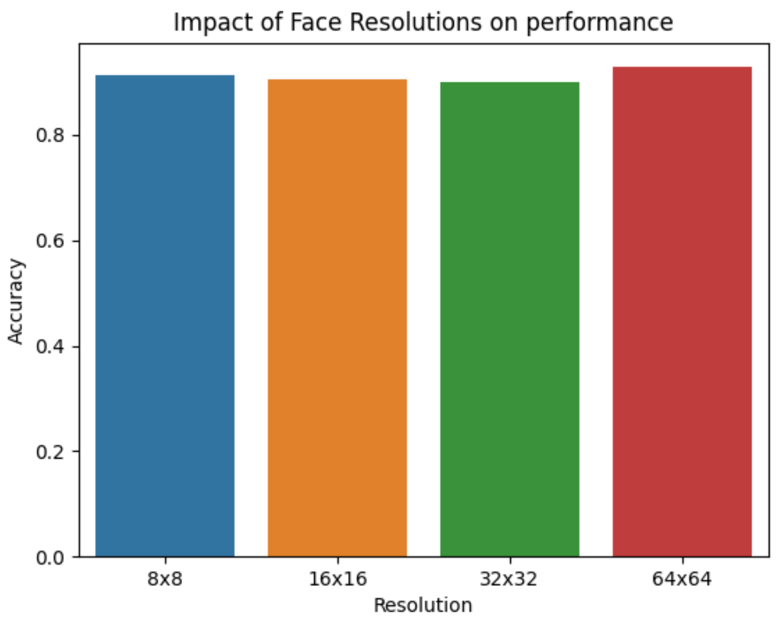


Figure 3: Face Window Resolutions

As seen on the Figure 3, all models perform very well. The difference in accuracy is really small :

- 8-by-8 : 91.287%
- 16-by-16 : 90.54%
- 32-by-32 : 89.98%
- 64-by-64 : 92.83%

This means that the detector does not need necessarily a high resolution to perform well. Moreover this means that we can choose between those different resolutions by only considering the computational cost behind.

2.2 Face Orientation

We also evaluated our model on the HeadPose Database to assess its performance on different face orientations. As metric we used the mean squared error between a vector of the prediction by our model (a float between 0 and 1, representing the confidence that the window contains a face) and a vector of *1.0* as all the windows were in fact faces.

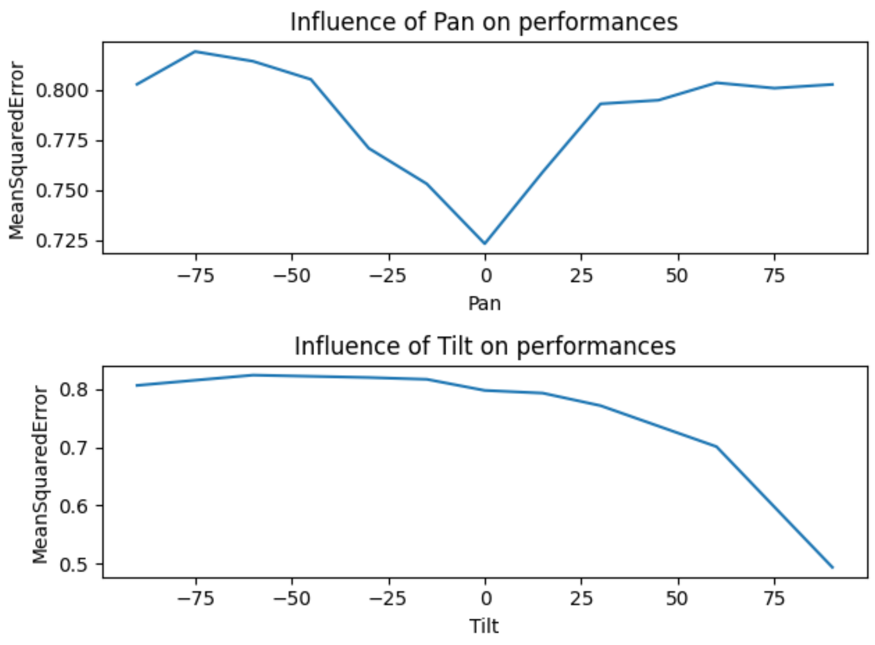


Figure 4: Face Orientation Variation

The first graph gives an information about the easiest pan value for our model. The minimum error is achieved with a pan of 0, which means frontal faces. The second graph gives the performance on multiple tilts. This is really interesting because 0 is not the optimal tilt here. This could mean that our training dataset is not balanced in terms of the tilt of the faces. It may contain more "+90"-tilt faces and thus the model is better at recognizing them.

3 Evaluation of the impact of the different parameters on performance

To evaluate the impact of some hyperparameters of our model, we chose different :

- Loss functions
- Activation functions
- Training and optimization techniques
- Learning rates
- Numbers of hidden layers and numbers of layer units per layer

3.1 Impact of the loss function

The first hyperparameter that we modified is the loss function. We tried 4 different loss functions and tried to observe the impact on the performance of our model. The loss function computes the cost of an error in classifying a sample compared with the ground truth. The loss function considered for this project were : binary crossentropy, Poisson loss, mean squared loss and Kullback Leibler divergence.

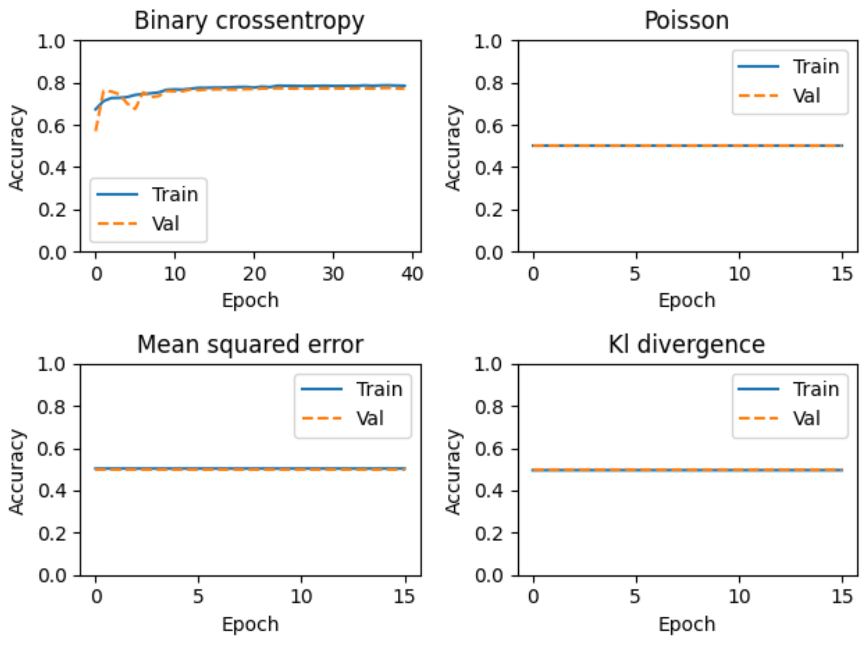


Figure 5: Impact of the loss function

The figure above show the impact of each loss function in the performance of our model. We can observe that the model have a good performance only with the binary crossentropy loss function. With the others loss functions the model performs poorly and seems to learn nothing during the training. These results are due to the fact that only the binary crossentropy is adapted to problems with two classes, others are not.

3.2 Impact of the activation function

The second hyperparameter that we modified is the activation function. The activation function is the function used by a neuron to output a value according to the values it received from the previous layer. For this project, we decided to compare four popular activation functions: rectified linear unit (ReLU), exponential linear unit (Elu), the hyperbolic tangent (Tanh) and a simple linear function.

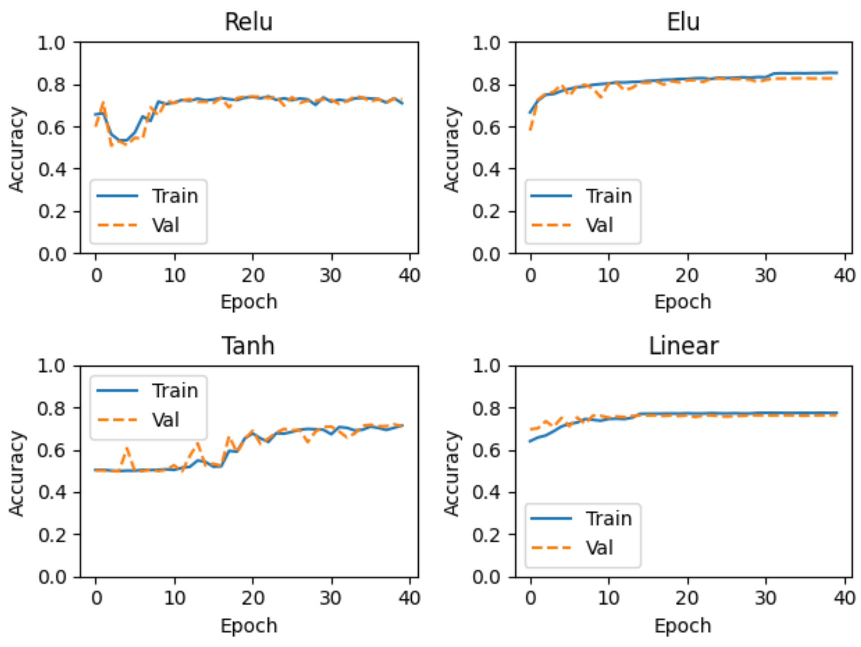


Figure 6: Impact of the activation function

The figure above show the impact of each activation function in the performance of our model. We can observe that for our model, the activation function that give the best result is the Elu. But in contrast with the loss function, all the activation functions considered here produced good results. In fact if you compare the efficiency / computational cost ratio on various problems you will see that ReLU is often the best function because it is very easy to compute (its a simple linear function) and it performs good.

3.3 Impact of the optimizer

The next hyperparameter that we modified is the optimizer. The optimizer is the optimization algorithm used by the neural network to learn the best weights and biases for a training set and so increase his performances. For this project, we chose 4 different optimizers: the stochastic gradient descent algorithm (SGD), the Adamax algorithm, the Adagrad algorithm and the Adam algorithm. The Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments. The Adamax algorithm is a variant of Adam based on the infinity norm, and the Adagrad is an optimizer with parameter-specific learning rates, which are adapted relative to how frequently a parameter gets updated during training.

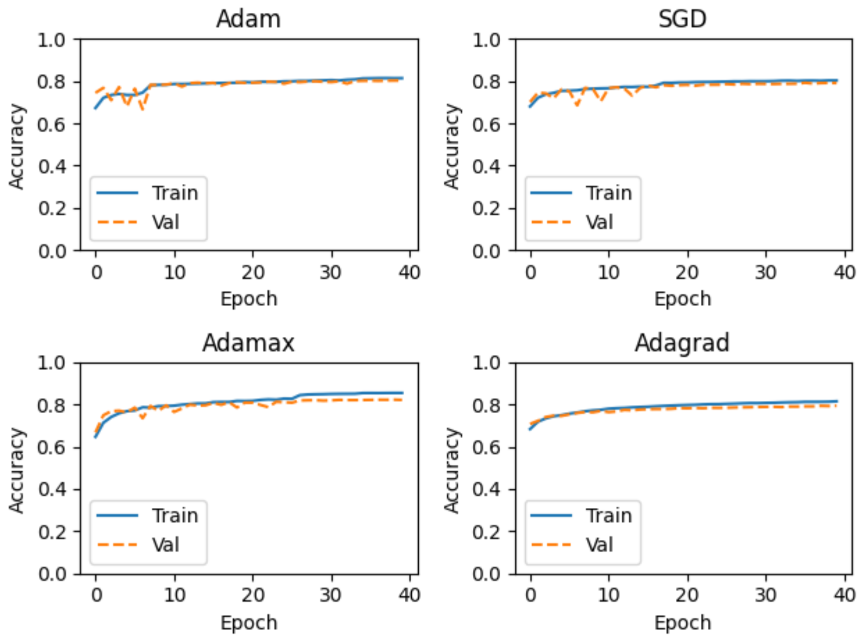


Figure 7: Impact of the `optimizer`

3.4 Impact of the learning rate

Finally, the last hyperparameter that we changed is the learning rate of the optimizer. Here we used the Adam optimizer where we used four different learning rates: 0.0001, 0.001, 0.01 and 0.1. The learning rate determines the intensity of change towards the best model during the training process.

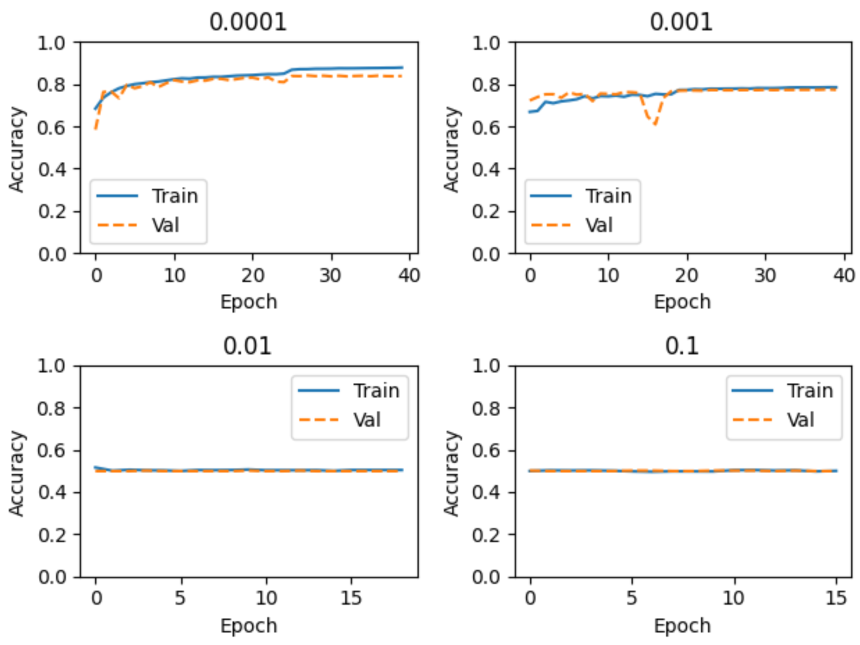


Figure 8: Impact of the `learning rate`

Thus we can see that the best choice for the learning rate in our problem is 0.0001. We can also use a changing learning rate thought the training to have even better performances. This can be easily implemented with callbacks in keras. With this method, the learning rate is divided by a given factor each number of time (that you decide) that the model does not improve on the validation set. Thus it helps the model in reaching the minimum more precisely (and not oscillating and overshooting around it)

3.5 Impact of the architecture

We first tried to assess the impact of the number of hidden layers. To do so we used a generic architecture composed of:

- An input layer
- Multiple hidden layers with *ReLU* as activation function
- An output layer with *sigmoid* as activation function

The model with 8 hidden layers seems to perform better than the others and its accuracy converges faster. It means that our problem needs a certain level of complexity and abstraction to achieve the best results. The first two models seem to still be learning after 100 epochs, maybe they would have caught up accuracy-wise to the last one after some time.

Then we tried to assess the impact of the number of units per layer. To do so we used the same generic architecture with 3 hidden layers. The parameters this time is the number of units per layer, which is the same for all layers except the output layer.

Thus, we can see that the more the network has neurons the more it performs. We also didn't notice overfitting during the processes

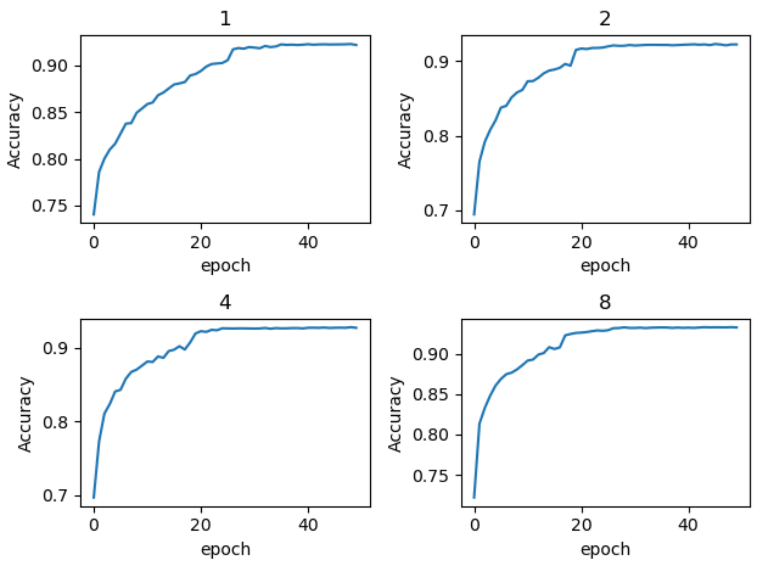


Figure 9: Impact of the number of hidden layers on the performance

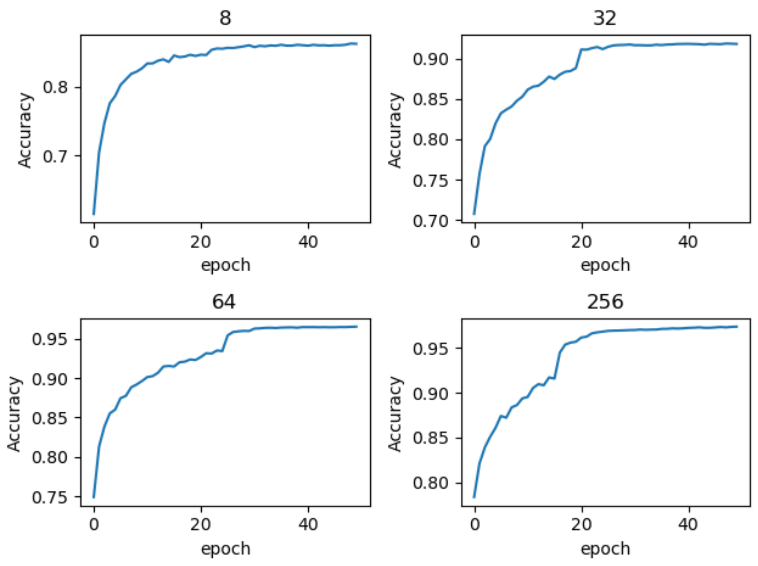


Figure 10: Impact of the number of units per layer on the performance

4 Designing a sliding window face detector

4.1 Image face detection

There are multiple steps in our algorithm. First the image is computed in gray scale to get rid of the third dimension. Then we used a sliding window with the same dimension as the imagettes used to train the Deep Learning model, here we used 64-by-64. This sliding window begins at the top left corner of the image and boxes are computed for prediction. The model predicts a confidence float, between 0 and 1, which means how strong it believes that there is a face at this spot. If this confidence number is higher than a preset value, the bounding boxes is stored in memory and will be used later. When the image has been entirely parsed, we downscale it. We repeat this process as long as we can compute a window in the image. The scale factor has a direct impact on computation time because it is related to the number of image scale we will have to go through. At the end of this step, we have a list of bounding boxes. However there are overlapping boxes that needs to be cleaned out. To remove them we use a Non-Maximum Suppression algorithm [2]. When facing multiple overlapping bounding boxes with sufficient overlap, this algorithm keeps only the largest one area-wise.

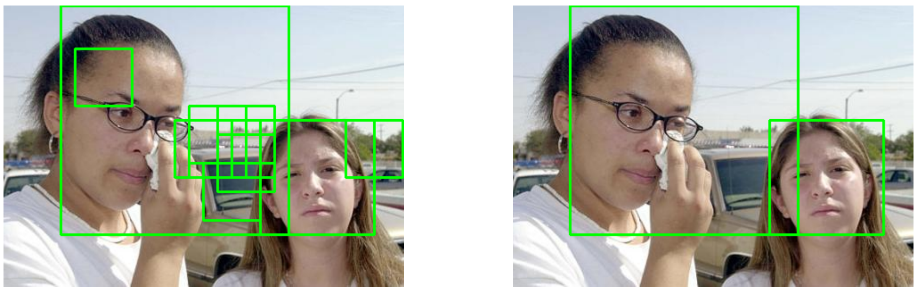
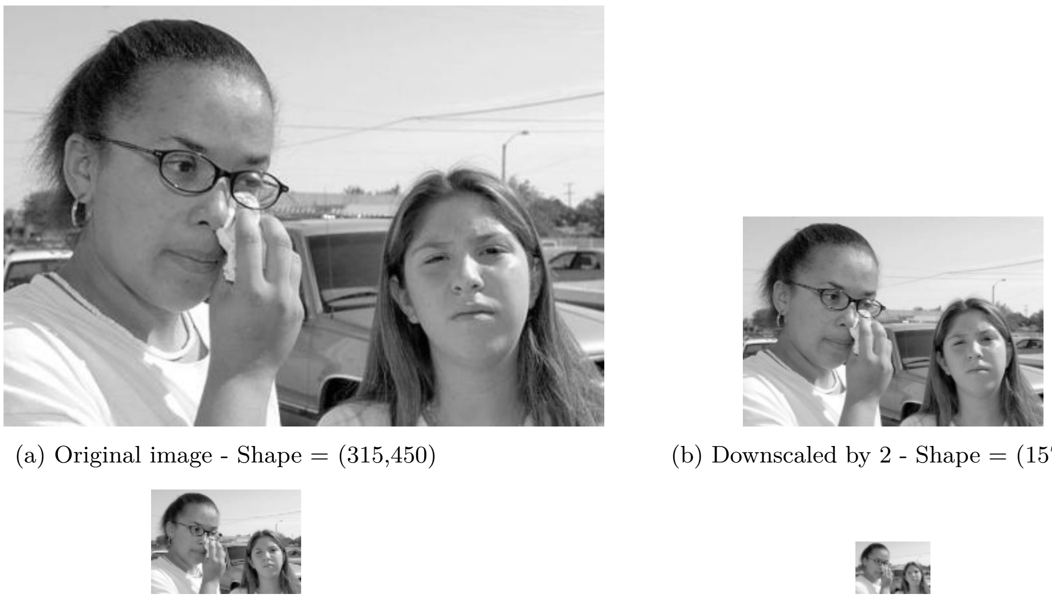

(a) Bounding boxes before NMS (b) Bounding boxes after NMS

Figure 11: Face detection on an image from FDDB



(a) Original image - Shape = (315,450)

(b) Downscaled by 2 - Shape = (157,225)

(c) Downscaled by 4 - Shape = (78,112)

(d) Downscaled by 8 - Shape = (39,56)

Figure 12: Scales of the image that the algorithm go through

On figure 12d, the image is too small for a window (of size 64-by-64) to be computed. The algorithm thus stops.

There are some tricks though and some traps that should be avoided. The biggest problem we encountered was recovering the position of a face detected in a low scale. A transformation is done to recover the bounding box coordinates in the original scale, just like the bounding boxes width and height that are becoming bigger and bigger as the image goes down in size. To solve this, we have to multiply the box shapes and the coordinates by the scale factor at each scale.

4.2 Real-time face detection

We used our algorithm for image face detection to create a Real-time face detection system using the built-in camera. We used OpenCV to recover each frame of the captured video, compute the bounding boxes with the algorithm described in the subsection above and show the frame with the bounding boxes. The detection is as nice as with images, the only downside is the speed of the detection. Even with a high scale and step size (to reduce the number of windows to go through), the prediction by the model takes a lot of time, around 0.03 seconds per window and it goes through a lot of window just for one frame. All in all each frame needs more than 8 seconds to compute the detections. That problem renders this method unusable for real-time face detection.

5 Others training and evaluating methods

5.1 The Keras Tuner

Another method to train and evaluate our neural network is to do a hypertuning or hyperparameter tuning. This process consists in choosing the optimal set of hyperparameters for our neural network. To make this, we used the library: Keras Tuner. We built a hypermodel where we define the tuned parameters : the number of layers, the number of units per layer, the loss function, the learning rate and the activation function. To perform hypertuning, we used the hyperband tuner and chose the function to optimize, here, the validation accuracy. The hyperband tuning algorithm uses adaptive resource allocation and early-stopping to quickly converge on a high-performing model. This is done using a sports championship style bracket. The algorithm trains a large number of models for a few epochs and carries forward only the top-performing half of models to the next round [3]

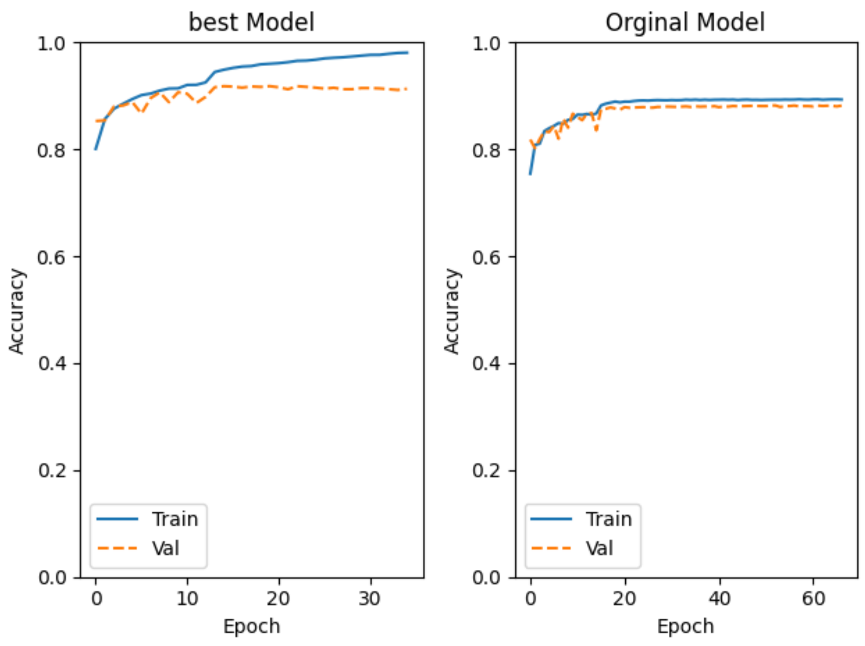


Figure 13: Comparison between tuning model and our model

The figure above show the difference of the performances in the training process between our initial model and the model with the optimal set of hyperparameters according to Keras Tuner. We can see that the tuning model has better performance that our initial model as excepted. But unlike our initial model, where the difference in performance between training and validation samples is not significant, with the tuning model we can observe this difference at the end of the training where the accuracy on the training dataset is 0.9 whereas the accuracy of the validation dataset is 0.8. So the tuning model unlike our original model is more sensitive to overfitting, probably because of its greater complexity. We don't describe the tuning model here because it is very complex, but to give a comparison between our model and the tuning model, our model has 3 layers, one input layer and one output layer. The tuning model has 10 hidden layers and all the layers, except the output layer, have an activation function that is different.

Bibliography

[1] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher,

M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[2] Tomasz Malisiewicz, Abhinav Gupta, and Alexei A. Efros. Ensemble of exemplar-svms for object detection and beyond. In *ICCV*, 2011.

[3] Introduction to the Keras Tuner. `https://www.tensorflow.org/tutorials/keras/keras_tuner`.