Pattern Recognition and Machine Learning
Face detection with fully connected artificial neural networks

Émilien FACHE Vincent HACHIN Martin SCHNEIDER

December 2020

In this lab, we will use fully connected neural networks to perform classification on face imagettes. These networks will be implemented with Python thanks to the Tensorflow Keras library. We will be especially focusing on the influence of different training hyperparameters on the performance.

1 Prerequisites

1.1 Building a balanced dataset

In order to train and evaluate our networks, the first thing we need is a balanced dataset of constant size imagettes. The starting point for this is a dataset of annotated face images, like FDDB or WIDER. The annotations on an image allow us to know where faces are to be found within it. Note that for FDDB, annotations take the form of ellipses, but they can be converted into rectangles using the OpenCV function `ellipse2Poly`.

For each image in the dataset, we look for each ground truth face, resize it to the desired standard dimensions $d \times d$ and save it as a positive example. Each time we do that, we also look for another imagette in the same image at a random position and with the same dimensions. We ensure that it does not overlap any of the ground truth rectangles too much using a criterion on the IoU ; if it fails, we just pick a new one until we get a suitable one. We resize it to the dimensions $d \times d$ too and we save it as a negative example.

In practice, it is not always possible to find a negative example in an image because a real face can take too much place. Consequently, we skipped images where at least one face's area was higher than one half of the total image area.

We also realized that even with the IoU criterion, negative imagettes sometimes contained a big part of a face. Thus we have selected a threshold on the IoU that was the lowest possible to try to avoid this situation, but not too low so it remained possible to find negative examples in an image. We ultimately chose the value 0.45.

1.2 Default neural network

As we will play a lot with hyperparameters, we have to use a default network with a fixed architecture and fixed parameter values. In our experiments, we will then make only one or two of them move, letting the other ones unchanged. We chose to use the following network.

```
Dense(256, activation="relu")
Dense(128, activation="relu")
Dense(1, activation="sigmoid")
```

Training Parameters	
Epochs	30
Loss	binary_crossentropy
Optimizer	SGD
Learning Rate	0.01
Batch Size	64

The last layer cannot be modified as we want to perform binary classification. The sigmoid activation function gives a value between 0 and 1, that can be interpreted as a probability of being a face.

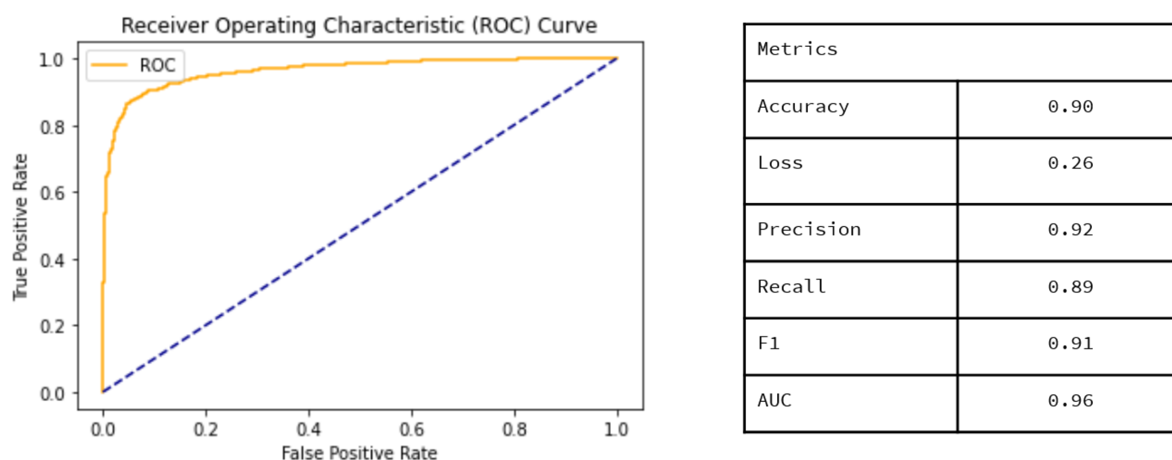
In order to fix the number of epochs, we used a dataset of imagettes from FDDB with size 32×32. We split it into training, validation and test sets, and we used the validation set during the training phase in order to check the evolution of the loss on this set. A number of 30 epochs allowed us to converge in terms of validation loss, without overfitting.

1.3 Metrics

In order to evaluate our networks, we used several different metrics. The first one is the accuracy, i.e. the rate of correct predictions, which is also the most common one. We also reused the metrics from lab 1 : precision (penalizing numerous FPs), recall (penalizing numerous FNs) and F1-score (tradeoff between both).

Our last metric is the AUC, which summarizes the information contained in a ROC curve without having to plot it each time. The idea behind ROC curves is much easier here than in lab 1. The probability given as output of the network can be used as a classification score. Making a bias move between the minimum and the maximum of these scores allows us to compute a FPR and a TPR for each value of this bias. The points we obtain give us a ROC curve, and the AUC is by definition the area under this curve. All this can be achieved pretty easily thanks to the functions `sklearn.metrics.roc_curve()` and `sklearn.metrics.roc_auc_score()`.

Evaluating the basic network we presented above gives the following results, showing that it already performs quite well.



Metrics	
Accuracy	0.90
Loss	0.26
Precision	0.92
Recall	0.89
F1	0.91
AUC	0.96

FIGURE 1 – Evaluation of the default network

2 Effect of different parameters on the performance

2.1 Number of layers and number of neurons per layer

The number of layers of a neural network must be mastered. Indeed, too many layers do not necessarily give better performance.

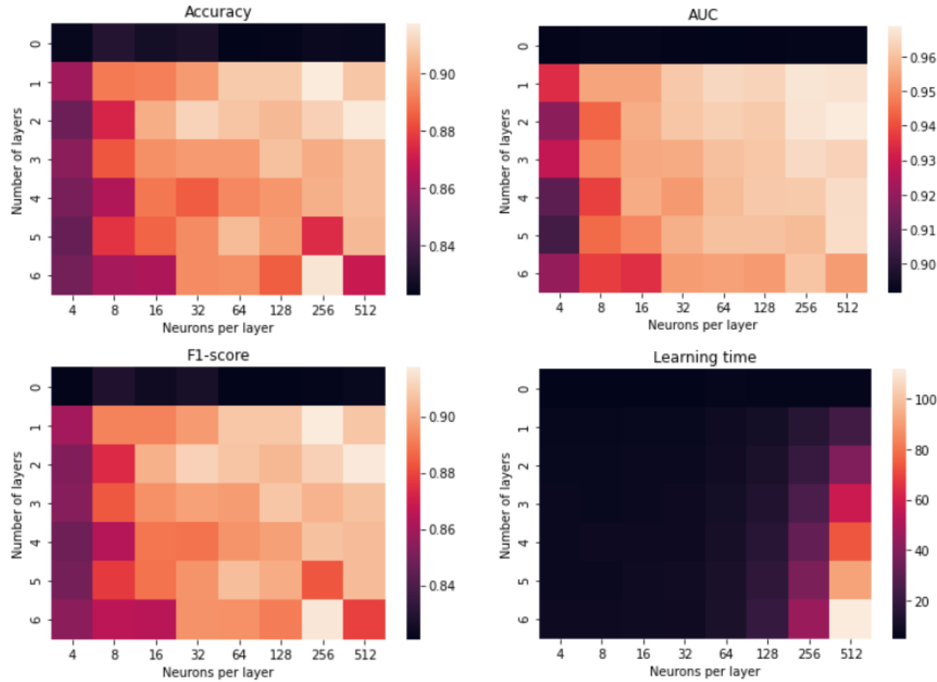


FIGURE 2 – Influence of number of layers and number of units per layer

Here, the performance of the model seems to increase as the number of neurons increases. The best result seems to be obtained with one layer of 256 neurons. The only thing that is certain is that the training time increases with both the number of layers and the number of units per layer. This shows that it is not a good deal to brainlessly add layers and neurons : predictions will not be especially better, but computational cost will dramatically increase.

A rather naive idea one can have is that an increase in the number of neurons increases precision at the expense of recall. Indeed, we could think that a larger number of neurons would allow us to capture more details, thus reducing the number of detection errors but also rejecting more faces. However, this hypothesis is not verified in practice, and after many tests, we could not find any relevant link between the number of layers / neurons per layer and performance.

2.2 Activation function

An activation function is a function that transforms the signal entering a neuron into an output signal. We chose `relu` as default one because it is the most commonly used and it has the very interesting property of avoiding the vanishing gradient problem in the learning phase. The vanishing gradient is the problem linked to the cancellation of the gradient of the cost function during the learning phase, thus preventing a good update of the weights of the connections between neurons.

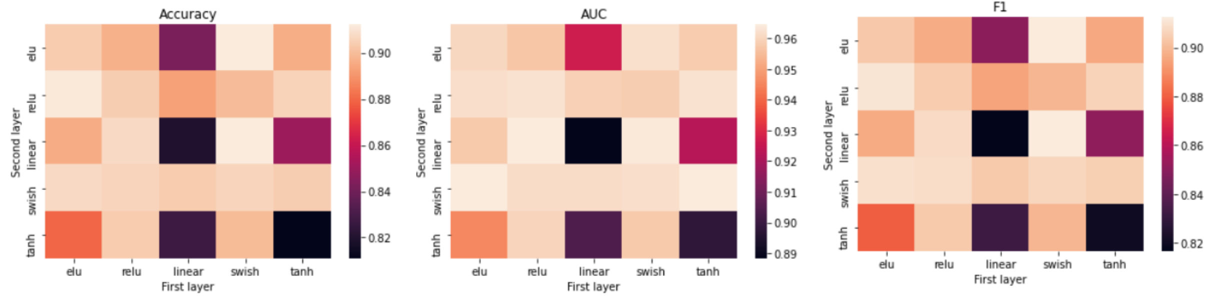


FIGURE 3 – Influence of the activation function

We tested 5 activation functions (`relu`, `elu`, `swish`, `tanh`, linear) on 2 hidden layers. The third layer always has a `sigmoid` activation in order to get a probability. If the model is modified for the recognition of new additional shapes (hand, car, eyes, etc.) the `softmax` function would be used.

In these heatmaps, the `swish-relu` combination seems to be the best. However, for another training the best combination was `relu`-linear, so it remains difficult to say which combination is really the best. Nevertheless, some combinations do not seem to fit here, like `tanh-tanh`.

Finally, considering our experiments and various existing articles, the function that seems to be the most suitable is the `swish` function. Indeed, the performance of classification algorithms such as ImageNet or Inception-ResNet-v2 was allowed by substituting the `relu` function by `swish` (`https://medium.com/@neuralnets/swish-activation-function-by-google-53e1ea86f820`).

2.3 Loss function

The loss function is a function that evaluates the difference between the predictions made by the neural network and the actual classes. The goal is to minimize this function so that the model performs as well as possible. Its minimization is done by adjusting the different weights of the neural network (see Optimizer).

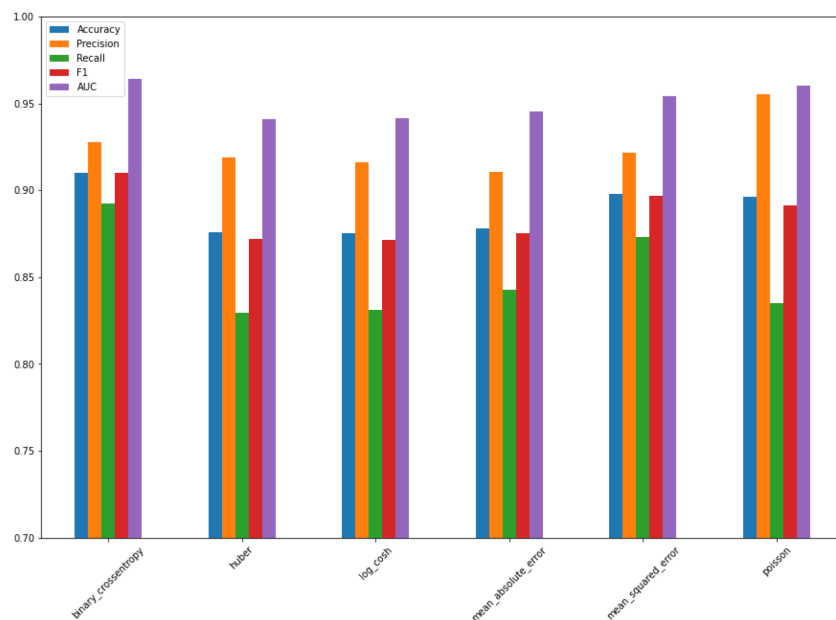


FIGURE 4 – Influence of the loss function

We have compared 6 loss functions (binary crossentropy, huber, logcosh, mean absolute error, mean squared error and poisson). We can see that the binary crossentropy gives the best results here. We could expect this because it is the most commonly used function for a binary classification task. Indeed, the logarithmic term compensates the almost stationary zones of the sigmoid function for points far from 0, thus allowing a faster convergence during training.

2.4 Optimizer

Optimization is the technique used to update the parameters of a neural network and make them converge to optimal values. In other words, optimizing is the process of reducing the prediction error, which is given by the loss function. There are several different optimization techniques, that are all based on the idea of gradient descent. This idea consists in computing the gradient of the error (derivatives with respect to the parameters that can be updated) and then updating the parameters in the direction that will decrease the error.

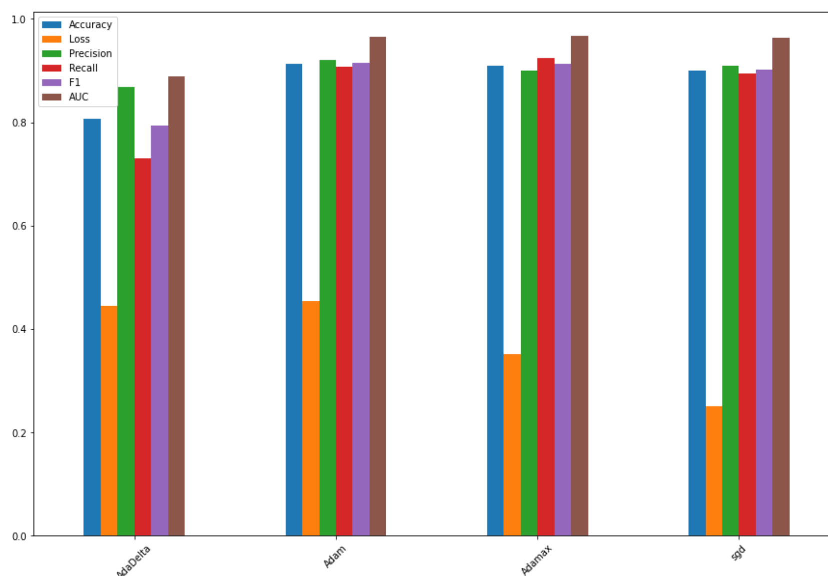


FIGURE 5 – Influence of the optimizer

We compared 4 optimizers (Adam, Adamax, SGD and AdaDelta). We quickly notice that AdaDelta performs worse than the other ones. Adam, Adamax and SGD have the best performance, the choice of one or the other seems to depend on the type of performance we want. Adamax will make more errors than Adam and SGD but will have a smaller recall. Adam and SGD really seem to be equivalent in this case. Adam is actually the most popular optimizer at the time of writing this report, even if new powerful optimizers are emerging (RAdam or LookAhead).

2.5 Dropout

Dropout is a regularization technique to reduce overfitting in neural networks. At each training epoch, a certain proportion of the units of a layer will randomly not be taken into account. This prevents neurons being too specialized. We tried to add some dropout to both hidden layers of our default network (same rate for both).

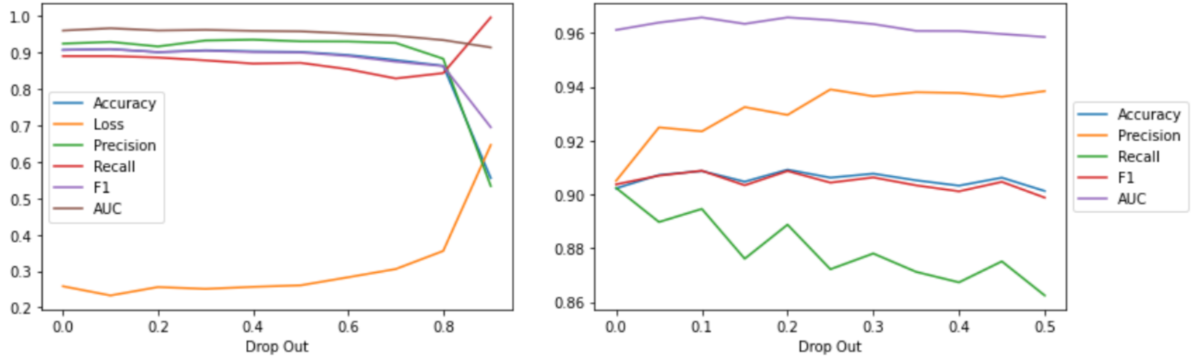


FIGURE 6 – Influence of dropout (2 different ranges)

The best results seem to be reached for a dropout of 0.1. Beyond 0.8, the results become random which is easily understandable since we have very little neuron left for training. There seems to be a link between precision, recall and dropout. Indeed, the precision increases and the recall decreases while the dropout increases. Unfortunately, we could not manage to find a convincing explanation for this.

2.6 Batch size

The batch size is the number of examples that are "shown" to the model before the weights of the model are recalculated. Batch size has an influence in the optimization because the smaller it is, the more limited the learning is.

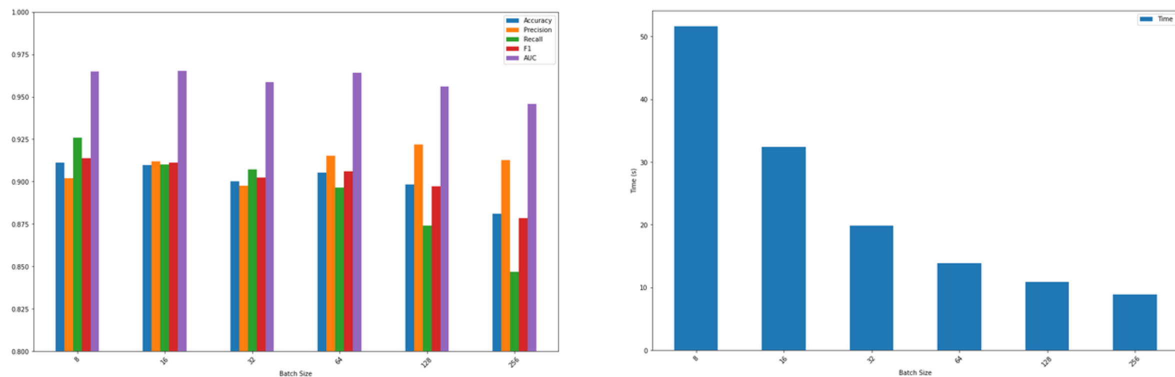


FIGURE 7 – Influence of batch size

Note that the batch size is linked to the training time : the smaller the batch size, the longer the training. Recall also dramatically decreases when the batch size increases. The batch size that seems to best reconcile computation time and performance is size 64.

2.7 Imagette resolution

We are now interested in the influence of imagette size. Thus we built different datasets with imagettes from FDDB, one for each resolution in $\{8\times 8, 16\times 16, 32\times 32, 64\times 64\}$. We then used these 4 datasets for training and evaluation.

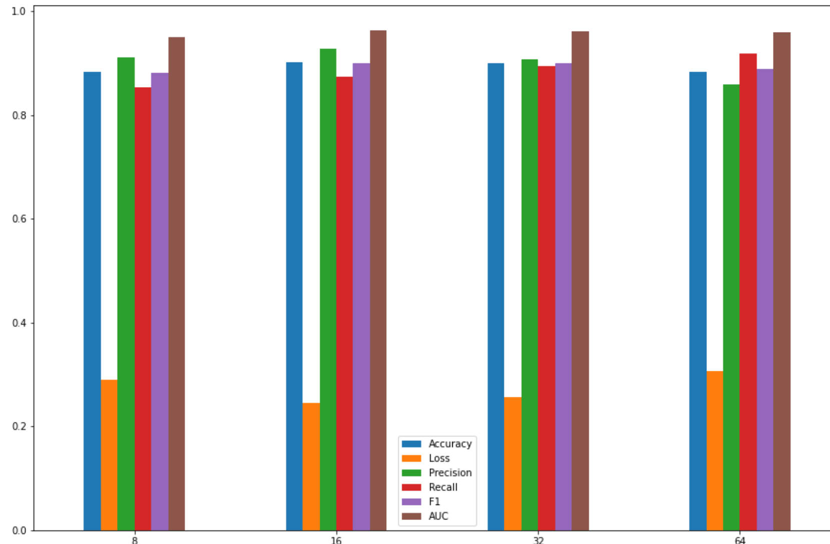


FIGURE 8 – Influence of imagette resolution

Contrary to what one might think, the size of the imagettes does not seem to have any influence on the performance. We can just notice a small increase in recall as the resolution increases.

2.8 Face orientation

In order to analyze models, accuracy on oriented faces can be computed. To do so, there is a dataset called Head Pose : *The head pose database is a benchmark of 2790 monocular face images of 15 persons with variations of pan and tilt angles from -90 to +90 degrees* (`http://www-prima.inrialpes.fr`). By extracting imagettes from this dataset, and associating for each imagette a tilt angle and a pan angle, the accuracy depending on the orientation can be computed. The network we used was trained on FDDB 32×32, and we only used positive imagettes from Head Pose for the evaluation.

We used two different approaches for the data representation. First approach is to sum up the orientation as the sum of absolute value of both pan and tilt angle. Therefore we only have one dimension left and we can easily plot a graph.

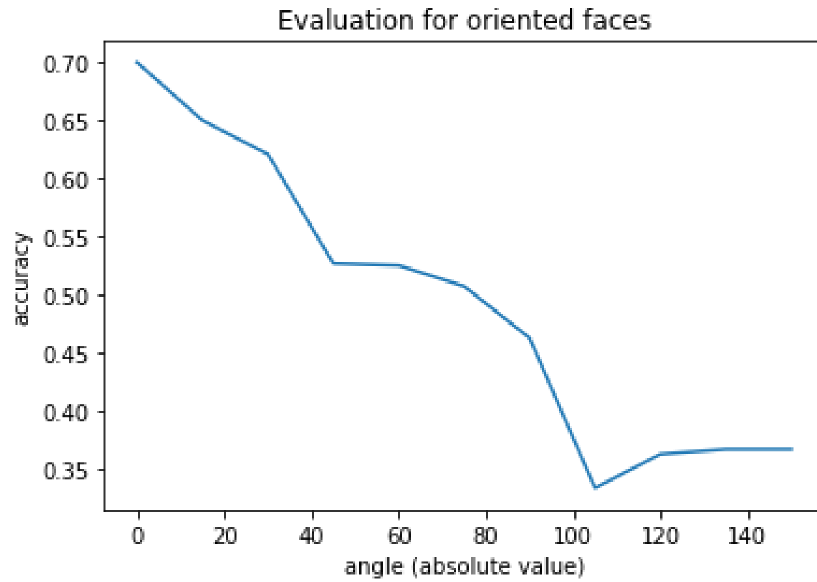


FIGURE 9 – Influence of head pose (sum of absolute values)

On this graph we can see how accuracy is decreasing over the angle.

The second approach is to keep information of both tilt and pan angle, and to plot a heatmap.

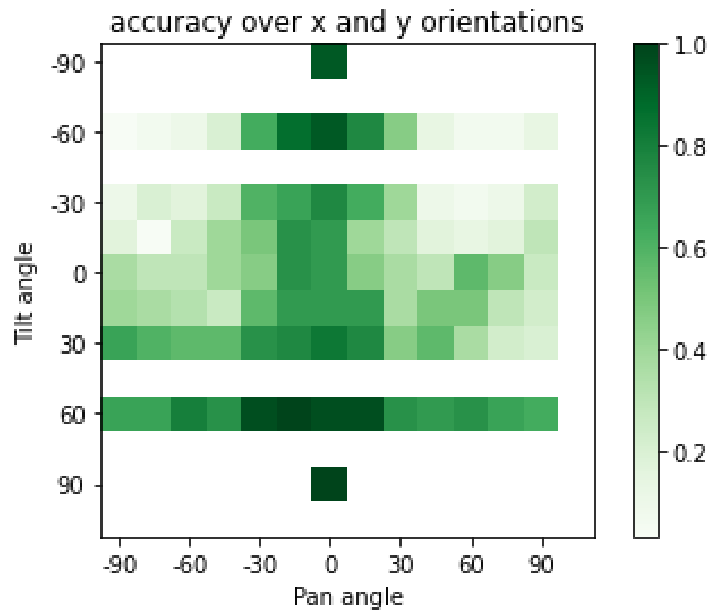


FIGURE 10 – Influence of head pose

The results seem a bit more inconsistent on this heatmap, even though the accuracy globally decreases when angles are increasing (on absolute value).

3 Neural network used as a sliding window detector

3.1 Implementation

We really wanted to test our model on real images, like the webcam's input. However, these images are not imagettes. We implemented an algorithm to use our neural network as a sliding window detector, reusing code from `https://www.pyimagesearch.com/2015/03/23/sliding-windows-for-object-detection-with-python-and-opencv/`. The first step for this algorithm is to have an iterator over all boxes, or Regions Of Interest (ROI). We will then consider a box as an imagette and see if it contains a face or not. To deal with the face size issue (a face is not necessarily 32×32 pixels, although imagettes are), we firstly constructed an "image pyramid" which consists of rescaled instances of the original image. We can then slide a window inside every image of the pyramid, and only keep boxes that are considered as a face by our model.

Now, we have a list of boxes that contain a face according to our model. A common issue about these sliding windows is the redundancy. Indeed, several imagettes will be kept for every face, but we only need to keep the most relevant one for each face. To do so, we used a popular algorithm : non-maxima-suppression. This algorithm is very simple.

1. Select the most accurate box (the one with the largest probability of being a face according to the model).
2. Remove every box that overlaps this box too much ("too much" is a parameter of the algorithm).
3. Repeat with the remaining boxes until every box has been processed.

3.2 Results

The results in terms of computational cost were pretty impressive. We managed to have a face detector on a webcam running in real time with about 5 to 20 frames per second, depending on the step of the sliding window.

In terms of accuracy, it was not that impressive. A lot of ROIs were detected although they were absolutely not faces (like sometimes the completely white background). To solve this issue, we increased the threshold for which a box would be considered as a face (about 0.9999). With this value, results are more accurate. Even though the detector is pretty poor, we can at least say that the algorithm is a face detector…

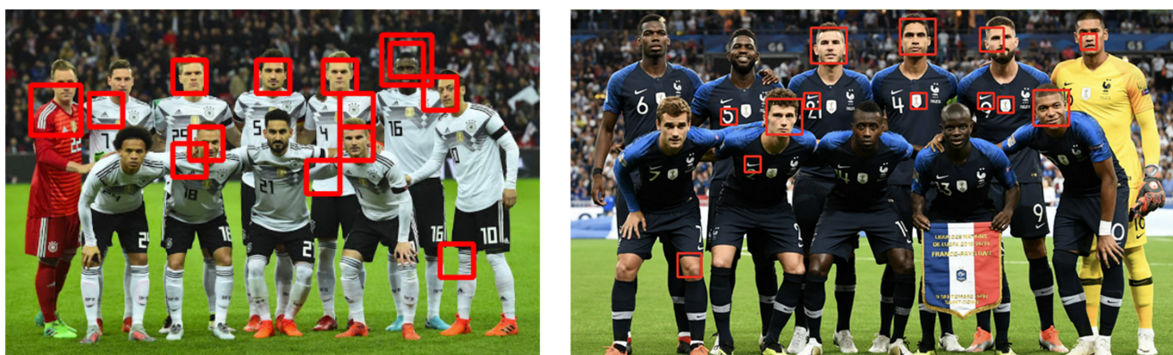


FIGURE 11 – Face detection with our sliding window

As you can see in these two pictures, a lot of faces are not recognized and regions that are obviously not faces are detected as faces. By increasing even more the threshold, the few

other remaining faces would not be detected so the threshold value of 0.9999 is a limit, and this detector is one of the best we can provide with this method and this model.

Note that on the second picture, it seems that white faces are more recognized than black ones. It may either come from bias in the dataset we used, or from a too small contrast between black faces and the background. We decided not to analyze further this phenomenon.

Besides these pictures, you can find attached a nice video of our real-time detector running on a webcam!

4 Generating faces with an autoencoder

As a last experiment, we tried to verify the ability of a trained autoencoder to generate data "from scratch". This potential has not been very developed in the research community yet, and it has been highlighted by a Grenoble PhD student and a researcher from the LJK with whom one of us could work as an intern. The principle of this generative process is the following.

We first train an autoencoder on positive imagettes of constant size (e.g. : FDDB 64×64). We did not go into considerations of sparsity for the loss function, we only used a classic mean squared error loss. Then we start from an imagette made of random gaussian noise. The idea is to inject this imagette into the autoencoder, add a little gaussian term to the output and reinject it into the autoencoder. After a certain number of reinjections, our imagette is supposed to converge into something that looks like a face.

This process worked pretty well with simple images like the handwritten digits from the MNIST dataset. However, it already took a long time to find an architecture that gives convincing synthetic data. In this lab, the application is much more complex, this is why we do not expect outstanding results. We were all the same satisfied to sometimes find a satisfactory shape.

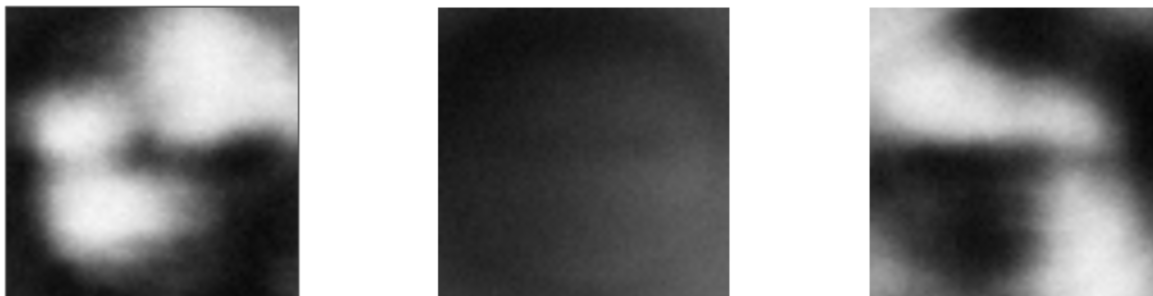


Figure 12 – Generated faces

Note that we had to use a different architecture for each of the resulting imagettes above. In this context of face generation, one autoencoder almost always converges to the same attractor. Consequently, we are still far from a good sampling method in this case, a method that could explore the whole space of face imagettes to also generate diversity.