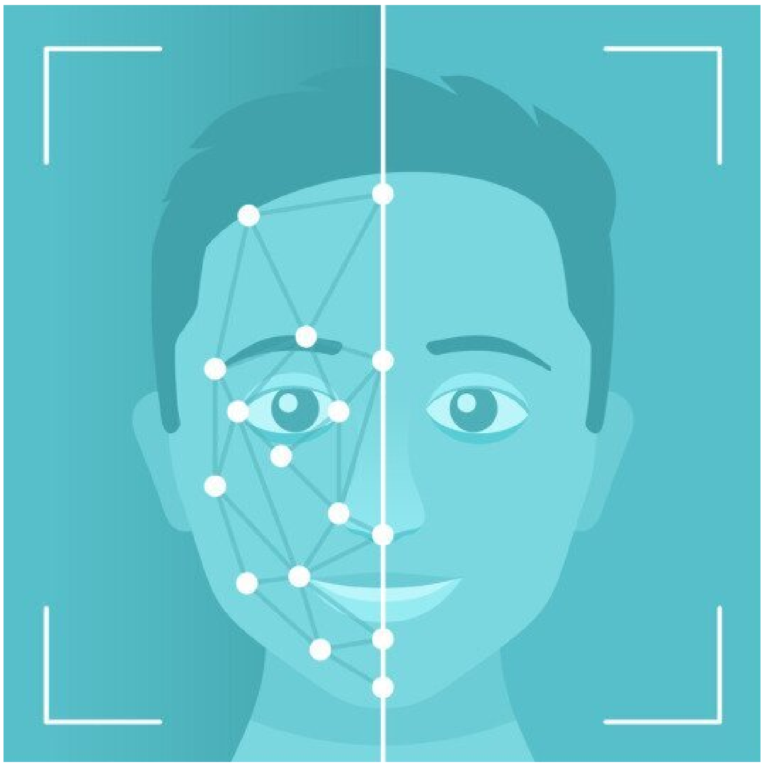
MA422 - INTRODUCTION TO MACHINE LEARNING

# Facial recognition and filter uses



Marwa Kadhem Baptiste Lemaire
Luc Lignian Julien Pain Quentin Rommel

## 1 Introduction

Today we can say that AI is everywhere but the truth is that it is becoming more and more important in our every day lives. And one of the most important method to create AI is machine learning, which is exactly what it sounds like: machines learning on their own.

Deep learning is a specific type of machine learning that is becoming more and more popular because it's especially good at handling large amounts of data. It is a fast-growing type of artificial intelligence, made to work with huge amounts of data. It has the ability to predict outcomes and behaviors by constantly learning from data sets. The sheer size of data involved in Deep learning means that it can find correlations that other types of AI cannot. This will make it increasingly important for businesses in the next few years, as they strive to make better predictions about customer behavior and trends.

Deep learning has finally emerged as the leading edge in artificial intelligence methods. After a long journey from algorithms originally inspired by the brain, such as artificial neural networks, Convolutional Neural Networks (CNNs), and Recurrent Neural Networks (RNNs), deep learning is delivering state-of-the-art performance on many challenging tasks, including image recognition, speech recognition, and machine translation. In recent years, the availability of large amounts of training data and the use of more powerful hardware (such as GPUs) has led to a resurgence in the use of deep learning for solving complex problems.

Along this project we will focus on the uses of algorithms that have been used for facial recognition purposes for some time now. Initially, they were used to identify faces in still images. With the increasing popularity of Snapchat filters, however, deep learning algorithms have been used to identify and track faces in videos as well. This is a particularly challenging task since videos usually contain a lot more variation than photos do. However, with the help of deep learning algorithms, Snapchat has been able to create some really fun filters that make use of people's faces. And this is what we will we try to implement

In our case we will do two programs:

- The first one will detect faces, for that we will use OpenCV. OpenCV (Open source Computer Vision) is a library of programming functions mainly aimed at real-time computer vision. It was started in 2000 by Intel, and is now maintained by Itseez (a subsidiary of Intel). OpenCV includes a wide range of both classical and state-of-the-art algorithms.It also provides a wide range of functions that can be used to process images and videos. We decided not to do the program our self because it would have been very difficult and also because it's not the main goal of our project.
- The second one is the one that we will focus on. The program will be doing the facial recognition. This is where we will put the different landmarks and track them to make them follow our faces.

Most of the people of the group had been educated in machine learning and/or deep learning. Indeed, we studied Deep Learning methods further thanks to Coursera MOOC: Deep Learning Specialization by DeepLearning.AI (`https://www.coursera.org/learn/neural-networks-deep-learning`). The hassle with the chosen project is that the methods seen in the direction are tough to use. That is why we determined to apply them and explain in element precisely how they work.

## 2 Explication CNN

### 2.1 Basics

We learnt this semester how to use Deep Neural Networks (DNN). Few decades ago, a new Deep Learning architecture appeared. It became particularly efficient for Image processing several years ago after the apparition of GPUs and their high calculation rate. Indeed, where a regular DNN takes a vector of values as input, a Convolutional Neural Network can take an multi-dimensional array .

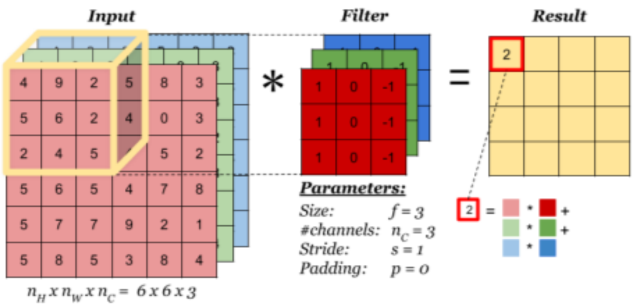


**Figure 1:** Diagram of a Convolution multiplication

The way information moves through the network is slightly different from Regular NN. We don't see a layer as a vector of neurons but as a Volume. The Figure 1 above explains how an array is computed in a CNN. We multiply the input matrix with a filter. On the diagram, We use a RGB image as input. The filter is then a 3D-array. But we can also use 2D-arrays for the input and for the filter. The filter shape is defined by the size and the number of channels. The size represents the height and the width, the number of channels is the depth of the filter. A cell represent one neuron. So the total number of parameters can easily explode compared to a regular DNN. As a single layer is composed of multiple stacked matrices, a Convolution layer could be seen as a cuboid. These volume tends to shrink in size due to the convolution as observed but also increase in depth due to the number of filters $f$ stacked.

### 2.2 Stride

The stride is the movement of the filter onto the input array. In the Figure 1, we compute the very top left "block". For the next computation, we shift the orange cube on the right. We repeat until the cube reach the right side of the matrix. Then, we shift the cube on the next row and put it back on the left side. We keep doing that until the cube reach the bottom right corner of the input. The Figure 2 shows how the stride works.

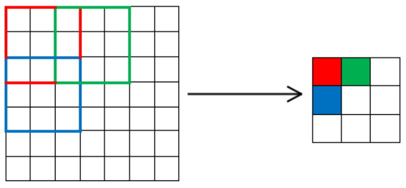


**Figure 2:** Stride diagram

## 2.3 Padding

In the previous section, we understood how a convolution multiplication is computed. But we can see that some cells are less important than others. For example, the top left cell will be computed only one time where a central cell will be used many times. To prevent this phenomenon, we use Padding. This method involves expanding the input array. We add layers of zeros around the input so that the filter uses each cell the same amount of time. This method also prevent shrinking as a Convolution layer can reduce the output size.

## 2.4 Pooling

A method to reduce the size of the hidden layer is Pooling. It usually divides the input into same-sized array and from those array applies a filter to return a single cell. If Average pooling is used, the filter takes the mean value of the array and return this value. If Max pooling is used, the filter return the maximal value. The Figure 4 explains more visually this method.

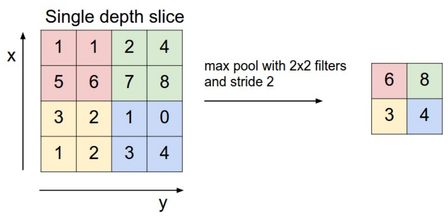


**Figure 3:** Pooling diagram

## 3 Our Solution

### 3.1 Dataset

To implement our solution, we first have to find a dataset that will allow us to detect faces. We found a dataset including 7049 gray scaled face images with 15 landmark points. These points are places around the eyes, the eyebrows, the nose and the mouth. Before training our model, we have to process these images with their corresponding points. We found this dataset on the website Kaggle (`https://www.kaggle.com/c/facial-keypoints-detection`).
First, we process the input : the images. To normalize the input, we divide the image array of 96x96 pixels by 255. Therefore, we end up with pixel values between 0 and 1. To have more data, we tried to double the dataset by flipping vertically every images. However, during the training, the loss was too high to have an accurate model. So we decided to only train the model on the 7049 images.
Then, we processed the output. The output is composed of 30 float, corresponding to 15 points (x and y). First, we shifted the values given in the dataset. Indeed, the given coordinates are placed on the image with coordinates varying between 0 and 96. The X and Y axis are placed on the top left corner of the pictures. Therefore, we scaled the axis from -1 to 1 and shifted them to the center of the picture. Finally, the outputs vary between -1 and 1 for both axis.
Furthermore, we had some issues about missing data. Indeed, 11 points have around 4780 missing coordinates for the 7049 images. We decided to use the Pandas function to fill the missing values $pandas.fillna()$ with the "ffill" method. The algorithm takes the last seen valid value and replace it for every missing values in the corresponding column. This method allows us to use the 7049 images but with less accuracy as more than the half of the output values are not manually placed. Finally, we end up with the following input/output for our model :

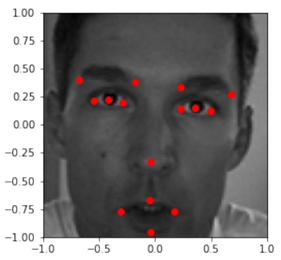


**Figure 4:** Dataset sample

### 3.2 Initializer

As we saw during the lectures, the weights and biases of the neural network are not all set to 0. when the network is initialized. If so, back propagation would be useless as the new values are calculated by using the previous values. If both weights and biases are equals to zeros, the new values would also be equal to zero. Therefore, Initialization is a very important step. Neural networks are sensitive to the initial neurons parameters.
For this model, we use a simple but effective method, the Xavier initialization[1] (Glorot init.). This method initializes the biases to 0., and the weights are calculated as follow:

$$W_{ij} \sim U[-limit, limit]$$

The weight of every neurons for each layers is randomly selected in a uniform range going from $-limit$ to $limit$. The limit is calculated with the following equation:

$$limit = \sqrt{\frac{6}{fan\_in + fan\_out}}$$

$fan\_in$ and $fan\_out$ respectively represent the number of neurons in the previous layer and the number of neurons in the next layer. This calculation means the bigger the model is the less the weight will be initialized. Therefore, the model is more unlikely to quickly diverge as the weights are small.

### 3.3 Loss function

The loss function measures how well the model is doing. This function computes a real number representing how far the predicted value is from the expected one. This number is then used to update the neurons parameters of the network, as we will see in the next paragraph. For a regression problem, a good loss function is the Mean Squared Error :

$$MSE(Y, \hat{Y}) = \frac{1}{n}\sum_{i=0}^{n}(Y_i - \hat{Y}_i)^2$$

This metrics get the average squared difference between the true value and the predict one.
Now that we know the loss function, we can see how the neurons parameters are updated thanks to this function.

### 3.4 Optimizer

Now that we saw how the weights are initialized, we need to see how they are updated. During lectures, we saw that the weights

[1]Xavier Glorot et al. (2010) "Understanding the difficulty of training deep feedforward neural networks" in Journal of Machine Learning Research 9:249-256

are modified thanks to back-propagation and gradient descent algorithm with the following equation:

$$\theta_{ij} = \theta_{ij} + \alpha \frac{\partial J(\theta)}{\partial \theta_{ij}}$$

$\theta_{ij}$ represent or the weight or the bias of the neuron $j$ in layer $i$. We use the same equation for both parameters. $\frac{\partial J(\theta)}{\partial \theta_{ij}}$ is the derivative of the loss function. This number is the most important because it represents how far the weight is from its optimal value. $\alpha$ is the learning rate. This equation is an application of gradient descent, and more precisely "fixed pitch gradient descent" as $\alpha$ is a fixed value. Another version of gradient descent is "optimal gradient descent". In this variant, the learning rate $\alpha$ is no more a constant value through the training. It has a different value for each weight and epoch.
The challenge now is to find the optimal learning rate. For that, we will use one of the most known and used optimizer: Adam (Adaptive Moment Estimation) optimizer[2].
It is defined as follow:

$$\begin{aligned}
g_{t,i} &= \nabla_\theta J(\theta_{t,i}) \\
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1,i} &= \theta_{t,i} - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t
\end{aligned}$$

$t$ is the epoch number. For the different parameters, I used the predefined ones :

- $\alpha = 0.001$
- $\beta_1 = 0.9$
- $\beta_2 = 0.999$
- $\epsilon = 10^{-8}$

Optimizer are used to make the model converges faster toward the solution.

## 3.5 Model

Now that we saw the different method to create a Convolutional Neural Network, we can use them to create our model. To create this model, we took the Lenet-5 architecture and add much more layers to have a more complete model. Lenet-5 is known to be efficient for pattern recognition. So this model should be efficient enough for our solution. The model architecture is specified in the Table 1
To train the model, we use a train/validation split. Therefore, the model trains on a training set and test its accuracy on a validation set. As the model doesn't train on the validation set, the validation MSE will help us to know if the model performs well on images it never saw before and so if it can generalize. We made a 90/10 split. So 90% of the whole dataset is for the training and 10% for the validation. We end up with a 6344/705 split.

[2]Diederik P. Kingma et al. (2015) "ADAM: A method for stochastic optimization" for ICLR 2015

**Table 1:** Architecture of our CNN

| Layer Number | Layer Name | Layer Shape | # Params |
|---|---|---|---|
| 1 | $Input_1$ | (96,96,1) | 0 |
| 2 | $Conv2D_1$ | (96,96,32) | 320 |
| 3 | $BatchNorm_1$ | (96,96,32) | 128 |
| 4 | $Conv2D_2$ | (96,96,32) | 9248 |
| 5 | $BatchNorm_2$ | (96,96,32) | 128 |
| 6 | $MaxPool2D_1$ | (48,48,32) | 0 |
| 7 | $Conv2D_3$ | (48,48,64) | 18496 |
| 8 | $BatchNorm_3$ | (48,48,64) | 256 |
| 9 | $Conv2D_4$ | (48,48,64) | 36928 |
| 10 | $BatchNorm_4$ | (48,48,64) | 256 |
| 11 | $MaxPool2D_2$ | (24,24,64) | 0 |
| 12 | $Conv2D_5$ | (24,24,96) | 55392 |
| 13 | $BatchNorm_5$ | (24,24,96) | 384 |
| 14 | $Conv2D_6$ | (24,24,96) | 83040 |
| 15 | $BatchNorm_6$ | (24,24,96) | 384 |
| 16 | $MaxPool2D_3$ | (12,12,96) | 0 |
| 17 | $Conv2D_7$ | (12,12,128) | 110720 |
| 18 | $BatchNorm_7$ | (12,12,128) | 512 |
| 19 | $Conv2D_8$ | (12,12,128) | 147584 |
| 20 | $BatchNorm_8$ | (12,12,128) | 512 |
| 21 | $MaxPool2D_4$ | (6,6,256) | 0 |
| 22 | $Conv2D_9$ | (6,6,256) | 295168 |
| 23 | $BatchNorm_9$ | (6,6,256) | 1024 |
| 24 | $Conv2D_10$ | (6,6,256) | 590080 |
| 25 | $BatchNorm_10$ | (6,6,256) | 1024 |
| 26 | $MaxPool2D_5$ | (3,3,256) | 0 |
| 27 | $Conv2D_11$ | (3,3,512) | 1180160 |
| 28 | $BatchNorm_11$ | (3,3,512) | 2048 |
| 29 | $Conv2D_12$ | (3,3,512) | 2359808 |
| 30 | $BatchNorm_12$ | (3,3,512) | 2048 |
| 31 | $Flatten_1$ | (4608) | 0 |
| 32 | $Dense_1$ | (512) | 2359808 |
| 33 | $Dropout_1$ | (512) | 0 |
| 34 | $Dense_2$ | (30) | 15390 |
| 35 | Activation | (30) | 0 |

Each Dense and Convolutional layers use the Leaky Rectified Linear Unit (Leaky ReLU) activation. It is define as follow:

$$LeakyReLU(x) = max(a \times x, x), \text{with } a = 0.1$$

This model has a total of 7,268,670 parameters.
This number of parameters forced us to use a framework as Tensorflow. For the training, we used a callback function named EarlyStopping. Before training the model, we set the EarlyStopping on the validation loss with a patience of 40 epochs. This means that if the model doesn't improve its validation loss, it automatically stops the training as we supposed it reached a local minimum. Finally, the model trains for 20 minutes. We set the initial number of epochs to 200, but the EarlyStopping stops the training at epochs 59. The model converged quickly to a satisfying solution. We can see on Figure5, at epoch 30, the validation MSE explodes. The model then overfits the training set. The EarlyStopping callbacks prevented it.

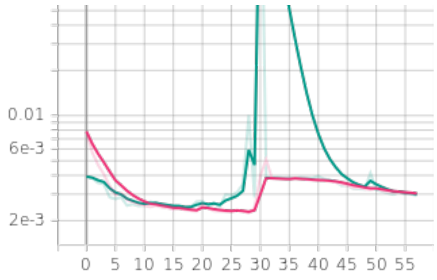


**Figure 5:** MSE through epochs (train set in purple, validation set in blue)

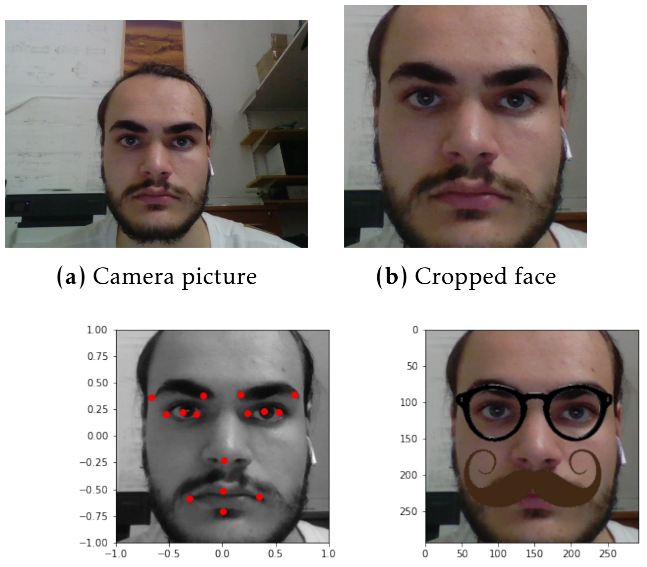
If we had a such quick training, it's because the model was big enough to be efficient so it didn't have to take so many epochs to train and the epochs were quick thanks to the GPUs. Indeed, we run the model with Google Colab. Google lets programmers use their server to train their neural networks on GPUs, which are faster than CPUs. Thanks to that, the training was quick and we end up with a accurate model : we have a validation MSE of 0.0024 at best.

## 4 Application

Now that we have created the Model that place the different landmarks, we can use it to create filters such as SnapChat. The first step is to retrieve the camera of our laptop and detect faces. To do this, we use the library OpenCV (Open Computer Vision). This library has been made by Intel to do Image Processing. The method *cv2.VideoCapture* automatically connect Python to the laptop's webcam. We obtain images as on the Figure 6a. Now that we have images, we have to find where the faces are on the picture. To do that, we will use a facial recognition algorithm implemented in OpenCV : Haar-Cascade (`https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html`). This algorithm detects faces and returns a rectangle with the coordinates of the top left corner and its height and width for each faces detected. To use our model, we then just have to separate this rectangle from the whole image. We end with the picture in Figure 6b.
As we can see, this type of image is very similar to the images used to train the model. Therefore, our model should perform pretty well for our application. The next step is to provide this images to the model. So we have to convert this image to gray scale. *OpenCV* has a method to make this transformation.

After using the model, we predict the different landmark on our test face. We can see on Figure 6c that the model succeed to generalize and so make good prediction on images different from its training distribution.
Now that we can detect eyes and mouths on images, we can create filters. As we only have 15 landmarks points, we can only implement basic filters. We then will try to put a mustache and glasses to the faces. First, we download two PNG images for the respective elements. This type of images handle transparency. After, we open them in Python. We resize them to fit the center of the two eyes for the glasses and the left and right corner of the mouth for the mustache. Finally, we places the two filters to the corresponding coordinates and combine the three images together.

We finally end up with the image in Figure 6d. We can apply those filters on one image, we now implement this solution for live image capture. You can visualize the result thanks to the video present in the folder attached with this report.



**(a)** Camera picture **(b)** Cropped face

**(c)** Prediction on test image **(d)** Test image with filters

**Figure 6:** The four steps to apply the two filters

The final result is satisfying according to our dataset. Indeed, we had many missing values and a lot of different image quality for the training. Even though our model doesn't have the best accuracy, it is sufficient for our solution. We can still see in the video that the two filters move a lot. Indeed, between two predictions, the coordinates of the points can move even if the face doesn't. A solution to this problem would be to implement an algorithm that check the previous values of the predictions and calculate new points according to the actual and previous prediction in order to reduce this noise. Furthermore,too create more complexes filters, Snapchat has to use a much more bigger CNN. Indeed, they can predict 96 landmark points on faces. Those points really draw the face where ours only show the mouth and eyes..

## 5 Conclusion

Finally, we may characterize this study as a valuable learning experience in the field of AI, specifically Deep Learning.

This study introduced us to the world of facial recognition and explained how it works in further depth. Using OpenCV, we were able to build and simplify the next task, which was the use of filters. We were able to comprehend the main functionality of several contemporary applications using filters, such as Snapchat or Messenger, thanks to a program that allowed us to apply filters to points gathered on the face.

Even if we did not use the different methods seen in class due to their pace and accuracy, our project was more than enriching for everyone's knowledge, allowing us to see it more in depth.