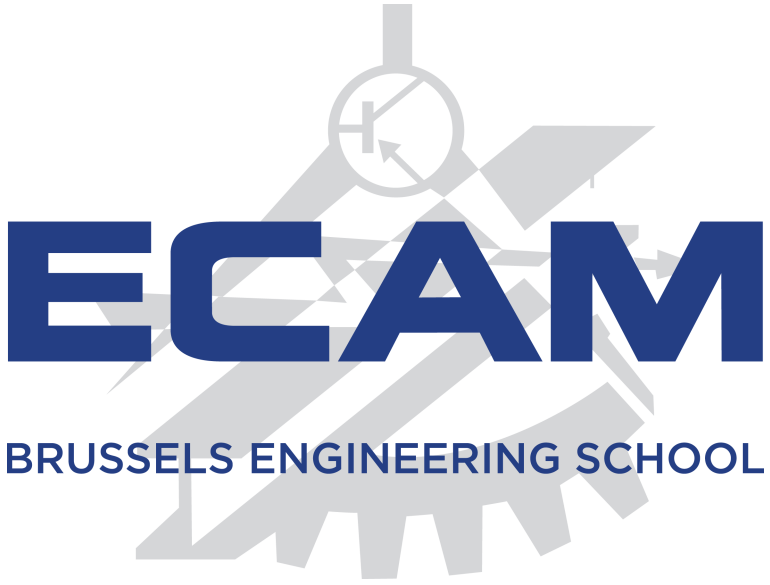
École Centrale des Arts et Métiers



Report - Face and Mood Detection

AI project - Smile

Ducoulombier Théo - 16067
Snyers Harold - 16243

Professor : Quentin Lurkin

21 février 2022

Table des matières

AI project - Smile

This report will give an overview of the project we have developed, a face emotion detection app using a CNN model.

1 Introduction and brief history

When it comes to images and more specifically, object detection, one particular type of neural network has revolutionized the field, namely the CNNs, Convolutional Neural Network. CNNs were introduced in the 1980s by Yann LeCun. A few years later, he released the first important CNN architecture called LeNet-5 which was able to classify digits. However, they discovered that this technique required more and more resources as the image resolution increased. No major breakthrough happened until 2012, during a challenge called ImageNet. This challenge is a competition for large scale visual recognition that started in 2010 and has been a reference for the performance of object detection. The competition works with the ImageNet dataset which contains more than 14 million images and 21.000 categories. In 2012, a first architecture made a breakthrough reducing the classification error from 26% to 15,3%. This last architecture, called AlexNet was highly inspired by the LeNet architecture but was deeper, with more filters by layers. In 2014, researchers of the University of Oxford proposed a new architecture called the vggNet, although that same year it is the GoogleLeNet architecture that won the contest. However, the vggNet is still considered the most preferred choice to extract features from images. We distinguish the vggNet-16 and the vggNet-19 which differ in the number of layers of the model. At this point the classification error decreased to 7,3% and 6,67% for the vggNet and GoogleLeNet respectfully. Finally, in 2015, Microsoft proposed the ResNet model, an extremely deep neural network with 152 layers which won the contest with a 3,57% error, the first model beating the human with 5,1% error [4, 11].

In this paper, we do not propose a model working on the ImageNet or even object classification for that matter. What we propose is a model capable of detecting human emotions considering 7 classes of human emotion possible as we will see later in the document. As we are working with the detection of emotions and thus features in images, CNNs were the most prominent choice although alternative networks or

machine learning models such as SVM (Support Vector Machine) have been used for that task but with significant badder results support vector machine. Also, as we are working on a classification and "object detection" problem as in each emotion being a different object, we decided to work with smaller implementations of some of the models described above. This document will follow first with an introduction to convolution neural networks, how they work and what hyperparameters are most commonly played with. Next, we will quickly compare different models for our case and describe the selected model that we developed and implemented and finally, we will finish by describing the mobile application of the project.

2 Convolutional Neural Network - Rationale

In this section, we will describe first what a Convolution Neural Network is, which will be followed by some of the most renown models that been influential in the field and the recognition of CNN models for Computer Vision.

2.1 What is a Convolutional Neural Network

In short what a convolutional Neural Network does is take an input image and pass it through different types of layers (convolution, pooling, fully connected, ...) to finally get an output. The output of this model can either be numeric or categorical depending on what you are training the model for. In this document, we will only speak about the categorical output. A categorical output is an output that returns a value from a set of possible values. A simple example would be to train a model to distinguish cats and dogs in images. In this last example, the model has two different output values, cat or dog.

CNN where developed based on the visual cortex and thus related to the neuroscience field. Early in the 60s, Scientist discovered that individual neural cells only responded to certain types of edges, for instance only vertical once. This way they were able to discover that neurons where placed in specialized groups and consequently produce visual perception. CNN work in a very similar way.

CNN architecture

A conventional CNN architecture can be divided in four main parts ;

— Convolution Layer

— Pooling Layer

— Flattening

— Full Connection

The first layer as shown in Figure 1.2.1, the Convolution layer is a layer that takes as input the image (IxI) and applies certain filters (FxF) and convolution operations on it such as to extract the features of the image and at the same time reduce the size of the image. Like humans, the CNN will not look for every pixel but will try to extract the useful features such as to recognize what it is seeing. The output of this layer is a feature map or convolved map (OxO). Of course, we don't want to overlook features which is why multiple feature maps are computed in order to preserve the features. This also why we call this layer the feature detector. Most of the time we also add a sub layer called a activation layer to increase non linearity in the model as images are generally highly unlinear.

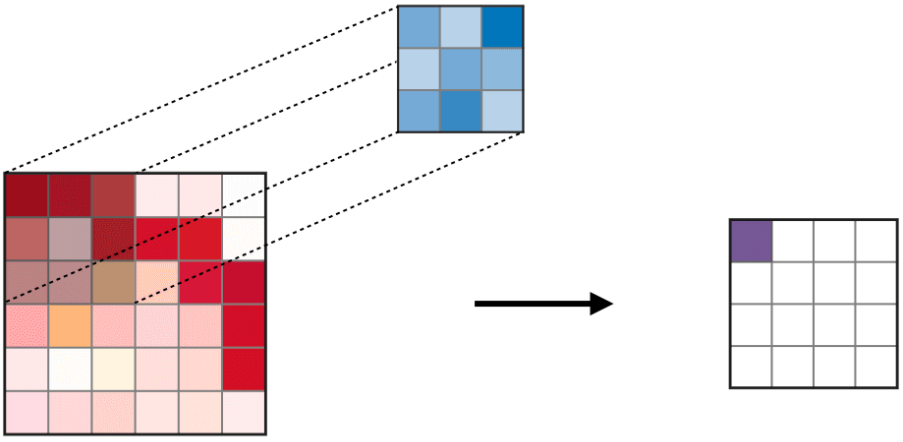


FIGURE 1.2.1 – The convolution layer uses filters that perform convolution operations as it is scanning the input I with respect to its dimension [2].

The next layer, the Pooling Layer, is layer that will integrate spacial invariance (see Figure 1.2.2). This is needed as we want our model to be able to recognize the object it is supposed to recognize no matter where the object is in the image. By doing this we give the model some level of flexibility to be able to find the features it is looking for. Different types of pooling exist, the most used are max pooling and average pooling which as the names suggest take the maximum or average value of the pool. Usually, the one takes a pool size of 2 by 2 but you can modify this of course by your liking. This layer has multiple benefits. First, as mentioned before, introduces spatial invariance. Secondly, it is reduces the size, with a pool size of 2by2, it is reduced by 75%. And additionally it removes also the chance of overfitting as it takes out 75% of the parameters (thus removing the unnecessary information to detect features).

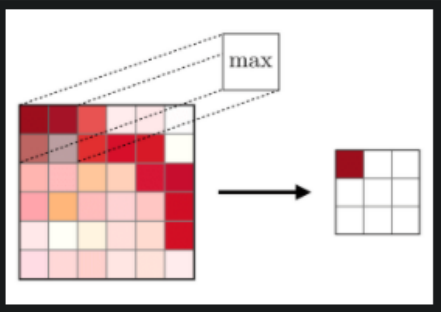
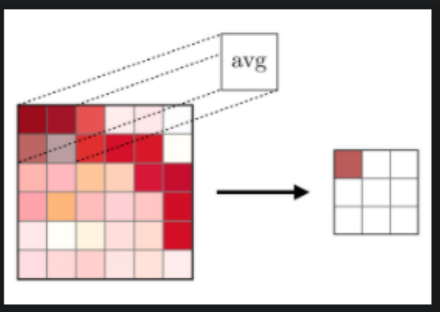
Type	Max pooling	Average pooling
Purpose	Each pooling operation selects the maximum value of the current view	Each pooling operation averages the values of the current view
Illustration		
Comments	• Preserves detected features • Most commonly used	• Downsamples feature map • Used in LeNet

FIGURE 1.2.2 – The pooling layer is a downsampling operation, typically applied after a convolution layer, which does some spatial invariance [2].

The third layer is simply needed to flatten the output of the pooling layer such that the full connection layer can take it as output as illustrated in Figure 1.2.3. The latter is the last layer which is an additional ANN. This input contains all the attributes detected in the image by our model and the ANN will be used to predict the output scores as this model is specialised for this kind of tasks.

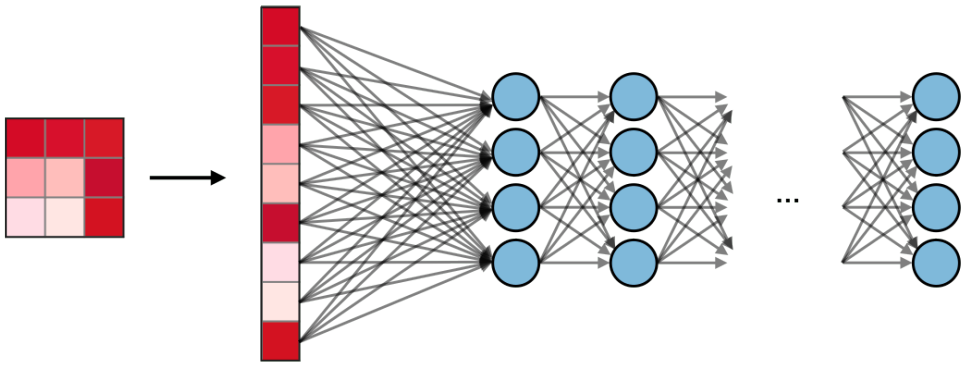


FIGURE 1.2.3 – The fully connected layer operates on a flattened input where each input is connected to all neurons [2]

CCN Hyperparameters

In the convolution layer, a set hyper parameters can be customised such as to change the way we are filtering during our convolutions.

We are able to change the dimension of the filter, which is a two dimensional filter, for instance 2 by 2. This the filter on which the convolution will done on. A second hyperparameter is the stride S which can changed during the convolutional layer and pooling layer. This represents the number of pixels by which the window (of the filter) moves after each operation.

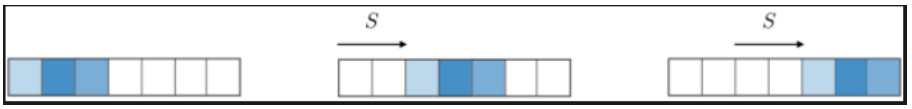


FIGURE 1.2.4 – For a convolutional or a pooling operation, the stride S denotes the number of pixels by which the window moves after each operation[2].

The last hyper parameter is the zero-padding. This parameter denotes the padding P around the image during the filter. We distinguish 3 modes.

- valid : $P = 0$
- Same :
  - $P_{start} = \frac{S[\frac{1}{S} - I + F - S]}{2}$
  - $P_{end} = \frac{S[\frac{1}{S} - I + F - S]}{2}$
  - output size mathematically convenient
- Full :
  - $P_{start} \in [0, F-1]$
  - $P_{end} = F - 1$

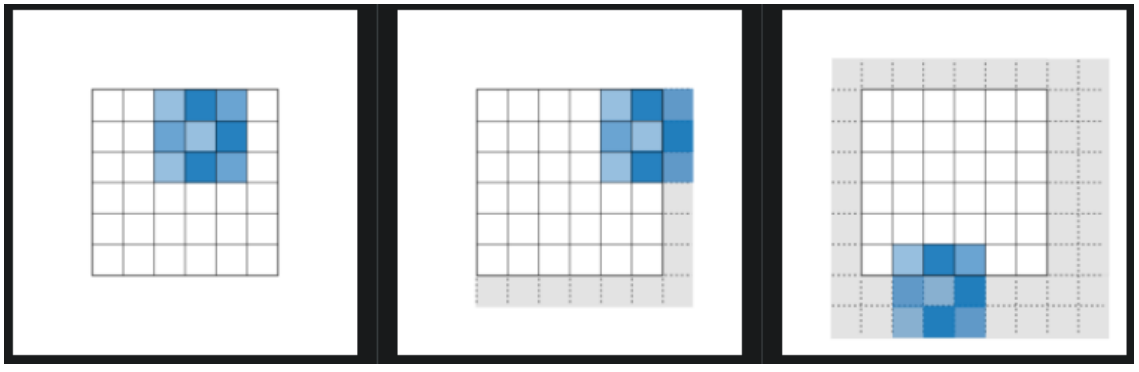


FIGURE 1.2.5 – Zero-padding denotes the process of adding P zeroes to each side of the boundaries [2].

Tuning Hyperparameters

When modifying parameters, we must also make sure of parameter compatibility in the convolution layer.

By noting I the length of the input volume size, F the length of the filter, P the amount of zero padding, S the stride, then the output size O of the feature map along that dimension is given by :

$$O = \frac{I - F + P_{start} + P_{end}}{S} + 1$$

In order to compute the complexity of the model, it is often useful to determine the amount of parameters in the model.

Commonly used activation functions

Rectified Linear Unit

Since a few years back (2011), the ReLu has been found to enable better training in Neural Network, compared to widely used activation functions prior in 2011, for instance the logistic sigmoid [1].

ELU

$$f(x) = \begin{cases} x & if x > 0 \\ \alpha\left(e^x - 1\right) & otherwise \end{cases} \tag{1.1}$$

which can also be written as :

$$f(x) = \max\left(\alpha\left(e^x - 1\right), x\right) with \alpha \ll 1 \tag{1.2}$$

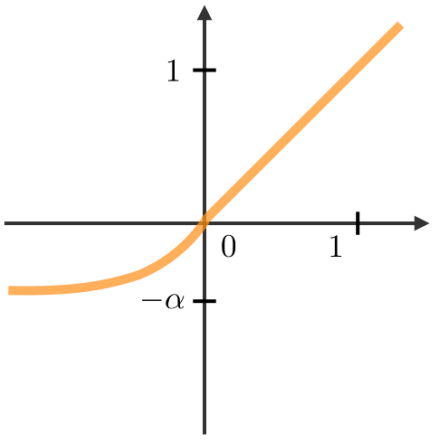


FIGURE 1.2.6 – Elu

— Differentiable everywhere

1. Xavier Glorot, Antoine Bordes and Yoshua Bengio https ://en.wikipedia.org/wiki/Yoshua_Bengio (2011). *Deep sparse rectifier neural networks* http ://jmlr.org/proceedings/papers/v15/glorot11a/glorot11a.pdf (PDF). AISTATS. Rectifier and softplus activation functions. The second one is a smooth version of the first.

ReLu

$$f(x) = \max(0, x) \tag{1.3}$$

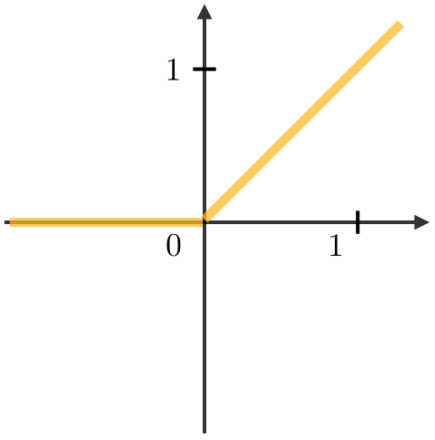


FIGURE 1.2.7 – ReLu

Non-linearity complexities biologically interpretable

Leaky ReLU

$$f(x) = \begin{cases} x & if x > 0 \\ \alpha x & otherwise. \end{cases} \tag{1.4}$$

Which can also be written as :

$$f(x) = \max(\alpha x, x) with \alpha \ll 1 \tag{1.5}$$

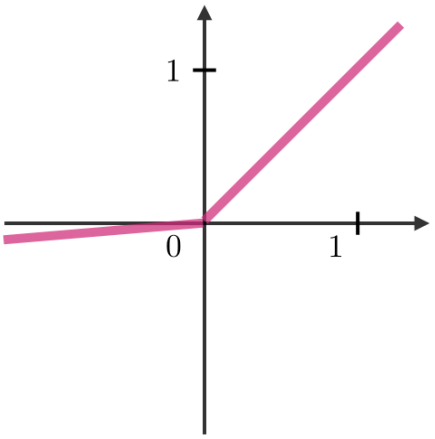


FIGURE 1.2.8 – Leaky ReLu

— Addresses dying ReLU issue for negative values
— Leaky ReLUs allow a small, non-zero gradient when the unit is not active.

PReLU

$$f(x) = \begin{cases} x & if x > 0 \\ \alpha x & otherwise. \end{cases}$$

— Parametric ReLUs take Leaky ReLu idea even further by making the coefficient of leakage into a parameter that is learned along with the other neural network parameters.

— Motivation behind PReLU was to overcome shortcomings of ReLU(dying ReLU problem) and LeakyReLU(inconsistent predictions for negative input values). So the authors of the paper behind PReLU thought why not let the a in ax for x<0 (in LeakyReLU) get learned!!

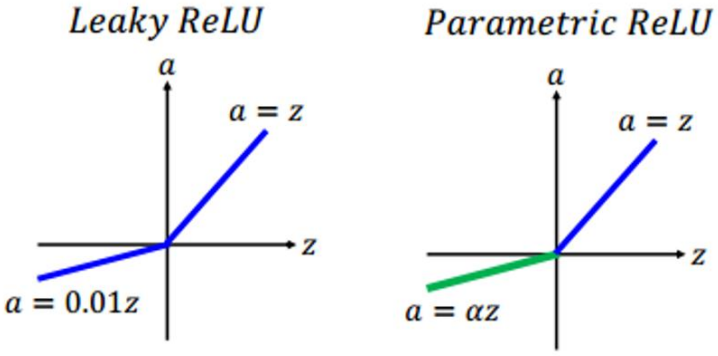


FIGURE 1.2.9 – PReLu vs Leaky ReLu

Softmax

Finally, at the end of our model we add a last step that can be seen as a generalized logistic function, the Softmax function used in a Dense layer as activation function. It is used often used for multi-class classifications and turns a vector of K real values into a vector of K real values that sum to 1. In fact, it converts the scores of the last layers to a normalized probability distribution that matches the classification problem the model is trying to solve. Combined with the Dense layer, it can for example transform the scores output by the model into the the number of classification classes it needs to predict to.

2.2 Short description of some CNN models

In the introduction, we had an overview of the most important dates and architectures that have shaped the CNNs we know today. In this section, we will try to give a short description of three of these models that have been used our final model. We will go over the vggNet, the ResNet and the Xception.

VggNet

As we mentioned earlier, the vggNet was a model proposed at the ImageNet challenge in 2014 and was the runner-up behind GoogleLeNet. While it was not the winner, the model architecture was much appreciated by the community. Contrary to previous architectures like the AlexNet, instead of using like large-sized filters, it uses several 3x3 kernals-size filters consecutively. The architecture exists with 16 layers or 19 layers. An example vggNet-16 is illustrated in Figure 1.2.13. The vggNet-19 architecture has one more layer for the convolution number 3 to 5.

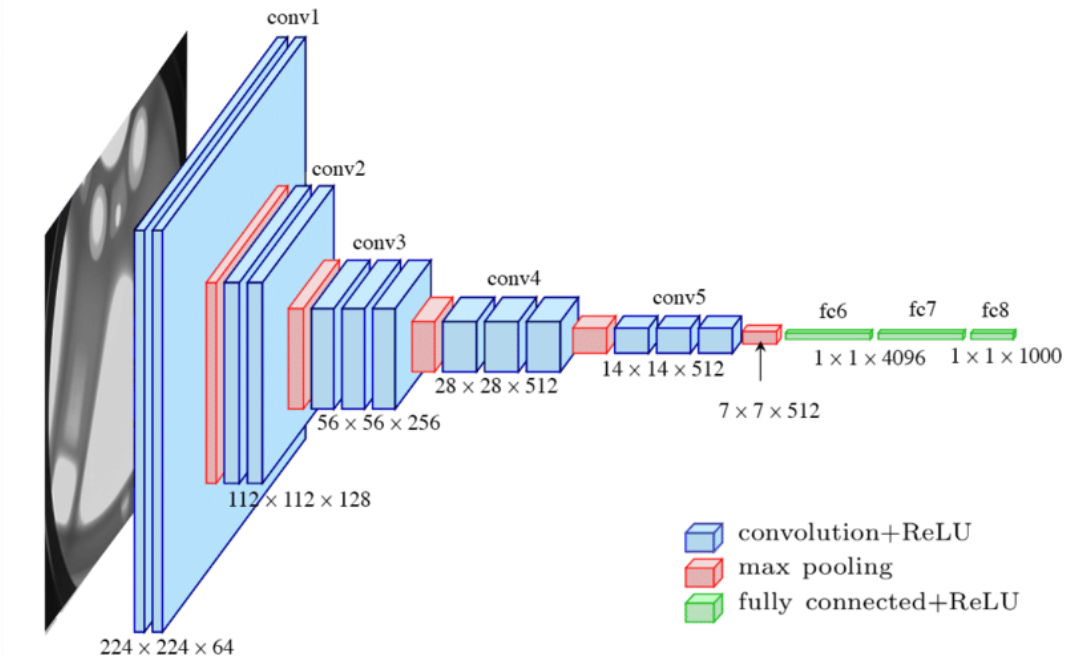


FIGURE 1.2.10 – VggNet-16 at ImageNet Challenge (2014)

One major drawback, however, from this architecture is that it is very slow to train and the weights of the network take a lot of place when saved.

ResNet

The ResNet architecture is an architecture that enabled to bias the problem vanishing or exploding gradient when using extremely deep neural networks. The ResNet uses residual blocks and skip connections for increasing the count of hidden layers to 152 without worrying about the vanishing gradient problem [11]. In traditional neural networks, each layer feed into the next layer, but this not the case with residual blocks. Additionally to having each layer feeding into each layer, each layer feeds also into layer 2-3 hops away as shown in Figure 1.2.11. We will however not dive deep into why residual blocks can counter pass this degradation problem we have when adding more layers.

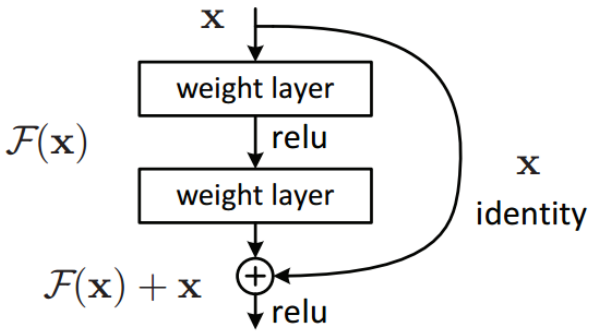


FIGURE 1.2.11 – Residual block architecture

Nevertheless, thanks to this residual block, they were able to add much more layers. It is known that the accuracy increases with the number of layers but until this, increasing the number of layers resulted into exploding gradients. The ResNet proposed at the ImageNet challenge was a 152 layers architecture but other configuration with slightly less accurate were also developed such as the 50 layers and 101 layers as shown in Figure 1.2.13. Even the 152 layer architecture is has a lower complexity than the vggNet.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

ures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of block

FIGURE 1.2.12 – ResNet architectures

Xception

The Xception model was slightly inspired by the ResNet and mostly by the Inception module V3, GoogleLeNet third version[3]. The authors proposed a CNN architecture based entirely on depthwise separable convolution layers. The model is based on a hypothesis underlying the Inception architecture but only strong which is why they named the architecture Xception, for "Extreme Inception".

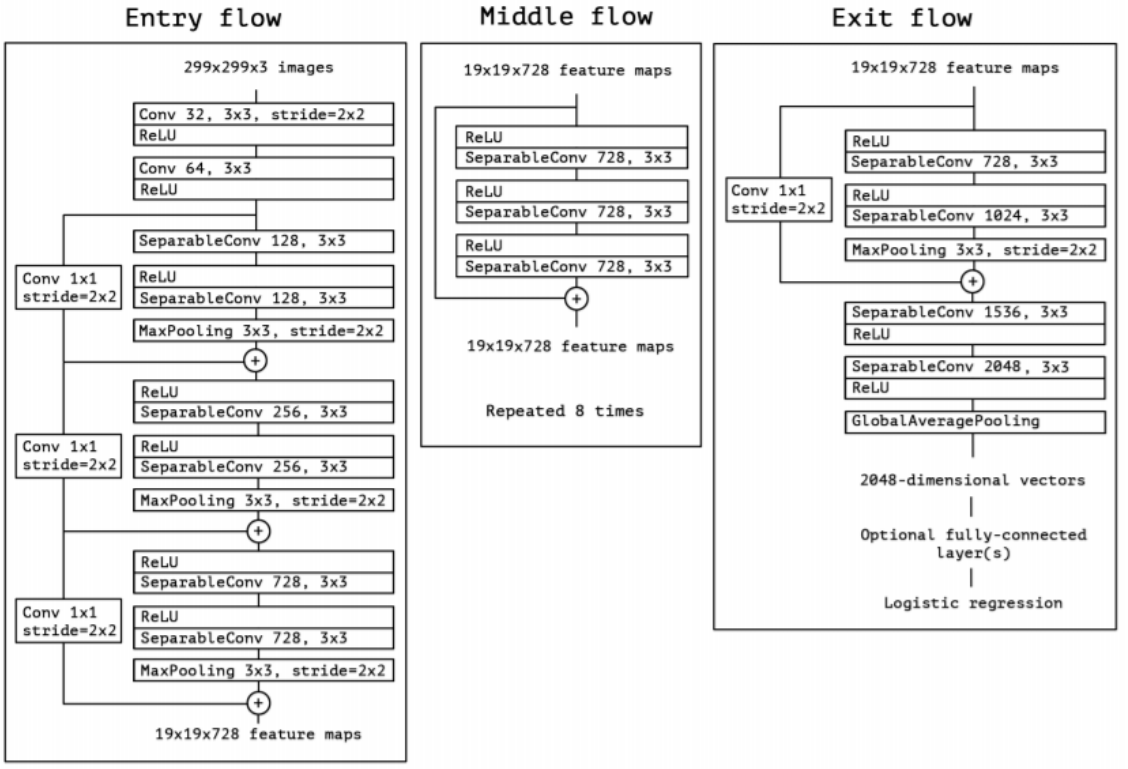


FIGURE 1.2.13 – Xception architecture

3 Emotion Detection Model

In the previous section, we went over the basic convolution neural network architecture and its different modules and parameters and concluded with some important CNN models that have had much traction in the field. These previous models were specifically developed for object detection but we thought that we could try to adapt these models to our project. The goal of the project is to be able to detect the emotions a person. Emotion recognition, also known as affective computing is a process of detecting human emotions from facial expressions. We know that 90% of a person communication is non verbal. The brain is able to detect emotions and an AI should be able to do the same by learning each facial expression meaning.

In machine learning, a well known problem is quality of the data, as garbage in means garbage out. But this is not the only difficulty with affective computing. Indeed, it runs deeper than labeling the data, we are still not sure how to label the data in the first place. One can label data based on a category or based on a dimension. Categorical labeling argues that each emotion falls into a set of classes. This idea comes from the pioneer of this approach, a Swedish anatomist who stated the simple idea that there is a finite set of human emotions. This idea was then developed and we distinguish seven different emotions ; happiness, sadness, surprise, fear, anger, disgust, and contempt. The other labeling technique, called dimensional

labeling assumes that emotions exist on a spectrum and can't be defined concretely. One approach for dimensional labeling is using pleasure and arousal as dimensions [1]. In this document, we decided to work with categorical labeling.

3.1 Data

For the purpose of the project, we used FER2013 dataset, a public dataset containing 34034 different labeled images. 80% of the dataset is used for training and the 20% remaining percents for validation and tests. In Figure

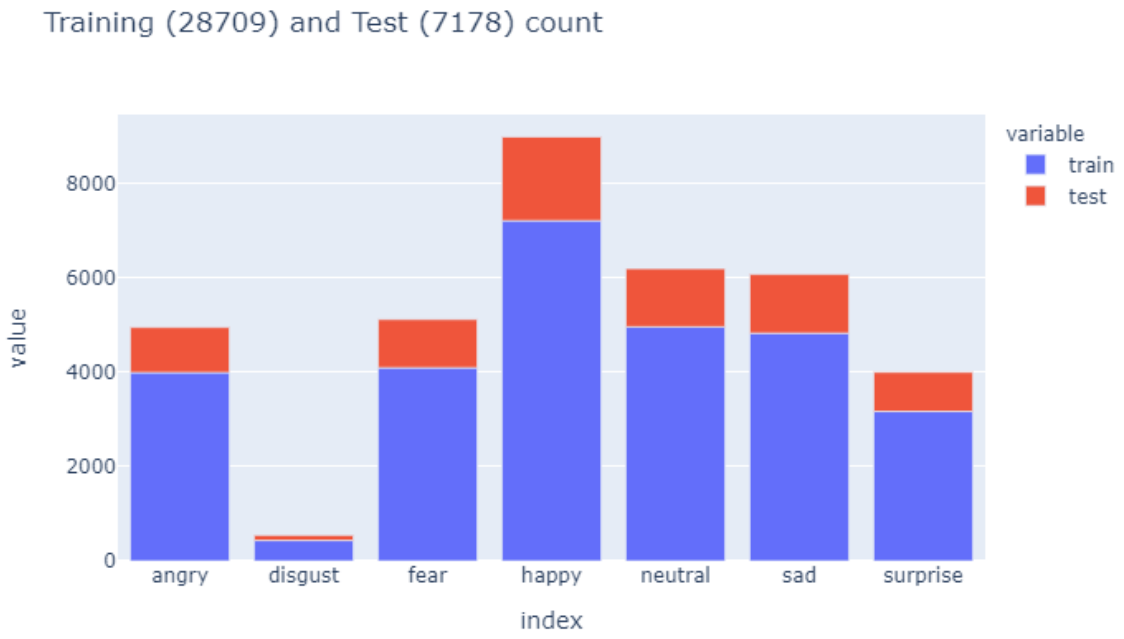


FIGURE 1.3.14 – FER2013 dataset count

FIGURE 1.3.15 – FER2013 dataset image sample for each emotion

If we look back at the Figure with the dataset count, we are able to see that the data is not evenly distributed. This will give our data a biased result, at least for the disgust emotion. However, keeping this in mind, we can still evaluate our model by comparing ourselves against other users or comparing our models against each other. Another dataset much more complete with 291.650 images with 8 classes (neutral added), called the AffectNet. However, we were not able to use it the data is even

less evenly distributed and the our computer power is not strong enough to train the data on it. To put into perspective, during the training with the FER2013, it took on average 25s for one epoch (a training cycle). For this dataset, it took more than 6 hours.

3.2 Comparison

For the purpose of the comparison to find the model that best suites our project, with the best accuracy, we developed or implemented different models. First We developed a very simple model with the following architecture :

- First convolution layer made of 2 convolutions (3x3, 64)
- First Pooling layer (2x2, max pool, stride 2)
- Second convolution layer made of 2 convolutions (3x3, 128) and 2 convolutions (3x3, 256)
- Second Pooling layer (2x2, max pool, stride 2)
- Fully connection layer : 1000-d fc
- output layer with softmax activation function

Each convolution is activated with a ReLu function as it is the most common choice. However, we did try to compare it with a PReLu activation layer but the resulting accuracy was lower, 4%, so we choose to keep going with a simple ReLu. We also implemented a vggNet architecture. We tried a simpler version and then the vggNet-16 and and vggNet-19 as illustrated in section 2.2. From the resulting tests, the vggNet-16 outperformed the others on accuracy. Another model we implemented, is the Xception architecture as described in section 2.2 [7]. For this architecture we didn't change anything, although we did play around with the filters amounts for each layer of the different flows (entry, middle and exit). At last, we implemented the ResNet architecture as described in section 2.2 with different amounts and sets of layers as it is illustrated in Figure 1.2.12 with the following sets of layers :

- resnet_30 : [2, 5, 5, 2],
- resnet_50 : [3, 4, 6, 3],
- resnet_101 : [3, 4, 23, 3],
- resnet_152 : [3, 8, 36, 3]

The resNet_30 was found online and was our starter model to which we going to compare our model [5]. This model used the architecture illustrated in Figure 1.2.12 under the 18-layer and 34-layer for the convolution layers from 2 to 5. All the other architectures used exactly the ones as described in their respective layers. However, tests showed explosing gradients when adding the max pooling at the beginning and the full connection of 1000 at end. We assumed it was due the different input shapes and we decided to remove these two elements from our network.

In the table below, we compare each models parameter size and training speed for each training cycle during the training. From this table we are able to already depict some interesting information from these different models. Indeed, although the ResNet_50 has more than 6 times the number of layers of the simplest model, it still has twice less parameters. Not surprisingly however, the ResNet models take at least twice more time per epoch on average which of course means that their training time should take much more time. Another characteristic we can see from this table is the huge amount of parameters for the vggNet models compared to the other models. This of course will have an impact on memory size disk available.

TABLE 1.1 – Model Parameter Count (448 steps per epoch)

model	total params	Trainable parameters	s/epoch	# layers
simple model	22,386,375	22,386,247	20s	8
vggNet-16	37,753,511	37,726,807	30s	16
vggNet-19	43,068,327	43,039,063	33s	19
Xception	17,796,383	17,749,215	38s	41
ResNet_30	11,816,715	11,804,297	59s	30
ResNet_50	10,669,115	10,631,337	67s	50
ResNet_101	15,504,187	15,422,889	108s	101
ResNet_152	19,491,899	19,372,201	155s	152

Finally, something we are not able to see in this table but that we acquired from tests. The vggNet models are the slowest in terms of getting to the final accuracy compared to the others that reach the final value they will reach more quickly.

Now that we have quickly described each model architecture and parameter size, we would like to give an overview of the best performance of each model so that we can afterwards conclude on our chosen model. In Figure 1.3.16, we provided the top 10 tests of models we were able to train using a computer with the following configurations : GeForce RTX 3080 Laptop GPU computeCapability : 8.6, coreClock : 1.545GHz coreCount : 48 deviceMemorySize : 16.00GiB deviceMemoryBandwidth : 417.29GiB/s. Additionally, we implemented these models using the Tensorflow library in Python.

From these tests, we were able to see that the best accuracy was obtained by the ResNet model and more particularly, the ResNet_50. Contrary to what we had thought, the Xception model performed the worst of the 4 main models. However, it is likely due to the small dataset size. Indeed, as the author, Chollet [3], mentioned in his paper, the Xception was able to outperform even more the Inception V3 and ResNet-152 with a bigger dataset than the one used in the ImageNet challenge, which already contains more than 20 million images. The vggNet did perform alright but

Name	model	dataset	acc_train	acc_test	loss_train	loss_test	batch_si...	epochs
test 5	ResNet	fer13	0.7748	0.6805	0.6121	0.9313	32	70
test 2	ResNet	fer13	0.899	0.6749	0.3124	1.242	80	80
test 1	ResNet	fer13	0.916	0.6713	0.2352	1.353	128	100
test 3	ResNet	fer13	0.8787	0.6673	0.3242	1.1704	128	80
test 3	model1	fer13	0.7232	0.662	0.7344	0.9713	64	80
test 2	model1	fer13	0.8616	0.6565	0.3868	1.229	32	80
test 4	ResNet	fer13	0.7946	0.6555	0.5707	1.043	96	80
test 3	vgg_net	fer13	0.7175	0.6546	0.7615	0.9776	64	80
test 5	model1	fer13	0.7309	0.6535	0.7217	0.9585	128	100
test 4	model1	fer13	0.7024	0.649	0.7897	0.9735	128	80
test 2	Xception	fer13	0.8186	0.6423	0.4851	1.1673	96	80

FIGURE 1.3.16 – Test Accuracy and loss for the different models.

we decided to not pursue our training with it as their parameter count is too big for our case. Finally, we also discovered that increasing the number of layers to 101 and 152 brought back explosing gradients. As our 50 layer model already performed really good, we increasing the size of the model was not recommended. In the next section, we will describe our model more accurately to showcase the differences as well from the original ResNet_50.

3.3 Solution

As we said just earlier, our final model is a ResNet_50 adapted to our project. Not only did he outperformed the others on accuracy, it also much lighter in size to the number of parameters it needs to retain. As our model should run on a mobile application, having a light-weight model was one of our main priorities.

Secondly, we have found online very few models that outperformed our model on a 1%-2% scale but this was mostly due to the preprocessing stage of our training. Indeed, as we investigated how they outperformed us, we were able to see that they reduced the data augmentation. We decided to increase the range of this data augmentation so that our model would perform better in the real world using a mobile camera. For instance, we went with a rotation angle of 30° while the others were mostly using a 10° rotation. The preprocessing stage is important for the training as it enable us to augment our images to our liking.

We already mentioned that our model was very close to the ResNet_50 proposed with the ResNet_152 during the ImageNet challenge. We will try to describe shortly the different steps of the architecture of the model here follows. In the next table, we provided the model's architecture.

TABLE 1.2 – ResNet_50 architecture

layer name	output size	[kernel_size, filters]	note
conv1	48×48	$\begin{bmatrix} 3 \times 3, 8 \end{bmatrix}$ stride 2	
conv2	48×48	$\begin{bmatrix} 1 \times 1, 32 \\ 3 \times 3, 32 \\ 1 \times 1, 128 \end{bmatrix} \times 3$	stride 1 on last conv
conv3	24×24	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 32 \\ 1 \times 1, 256 \end{bmatrix} \times 4$	stride 2 on last conv
conv4	12×12	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 32 \\ 1 \times 1, 512 \end{bmatrix} \times 6$	stride 2 on last conv
conv5	6×6	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 3$	stride 2 on last conv
	1×1	average pool (2×2), 512-d fc, softmax	

The first difference with the original architecture is the output size due to a different input size. Because of this we also decided to change filter size of the first convolution layer to 8. Secondly, during our tests, the max pool step in the second convolution layer made arise explosing gradients, so we decided to remove this step from our model. This means that the output size stays the same as the first convolution layer. We could have put the stride to 2 for the second convolution layer but the result were better with a stride of 1. The next difference lies in the filters' size where we start at 32 filters instead of 64 filters. By doing so, we observed a more steady increase in accuracy. Finally, for the last part, we only changed the average pool size from 7×7 to 2×2 and the full connection layer from 1024 to 512 units as increasing these made the gradient explosing again. This model final accuracy is 68,05% with a loss of 0,9313 on the test set.

The training phase was also improved thanks to callback parameters such as reducing the learning rate or early stopping which are continuously monitored. The early stopping parameter will stop the training when the validation loss doesn't improve for a number of sequential epochs.

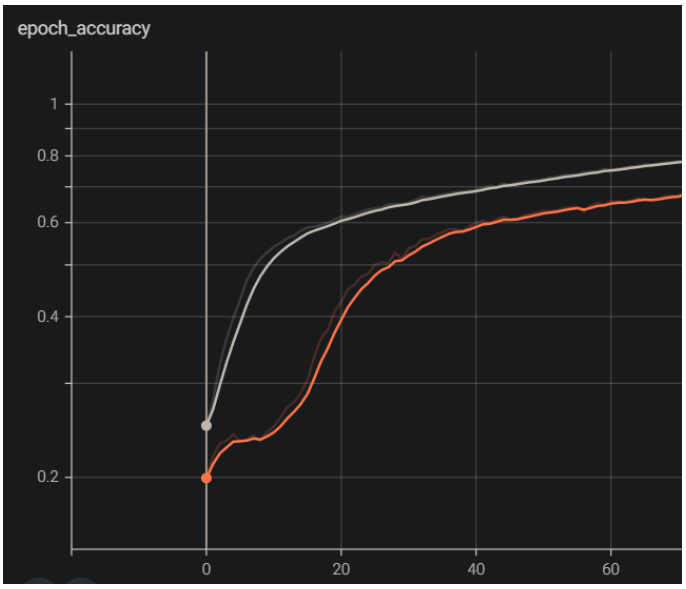


FIGURE 1.3.17 – Train and validation accuracy training

4 Emotion Detection Application

This section is about the mobile application which integrate the different models. The goal of the android app is to achieve the recognition of the faces and their emotions to be able to replace all the faces by the corresponding emoticons.

4.1 Development

The application has been developed with React Native[8] a JavaScript framework for native mobile application. The framework is really easy to use and you only need to code one time to deploy an application compatible with Android and Apple devices. An other major advantage is Expo[6] a platform to help debugging the application by letting the developers to test the application by scanning a QR code.

To exploit our models we use the TensorFlow[10] library adapted for Javascript and React Native. The library also include a module to yield tensors representing the camera stream so we can easily process, display and choose the frame-rate displayed to the user.

The model for the face detection is Blazeface [9]. The model has been chosen because it is lightweight and adapted for mobile application. The overall accuracy of the model is 97,95%. With such precision and optimization to perform with a mobile CPU with great efficiency, we decided to use BlazeFace because with our limited time and hardware, we couldn't train and create a model exploitable in real time on a mobile device.

The application is made with 2 layers. The first one with the camera stream and the second with the emoticons to be displayed over the faces on the stream. The emoticons are also rotated to correspond to the faces.

4.2 Workflow

The workflow of the application can be seen on the Figure 1.4.18. The workflow can be divide in two parts :

— The initialization

— The loop

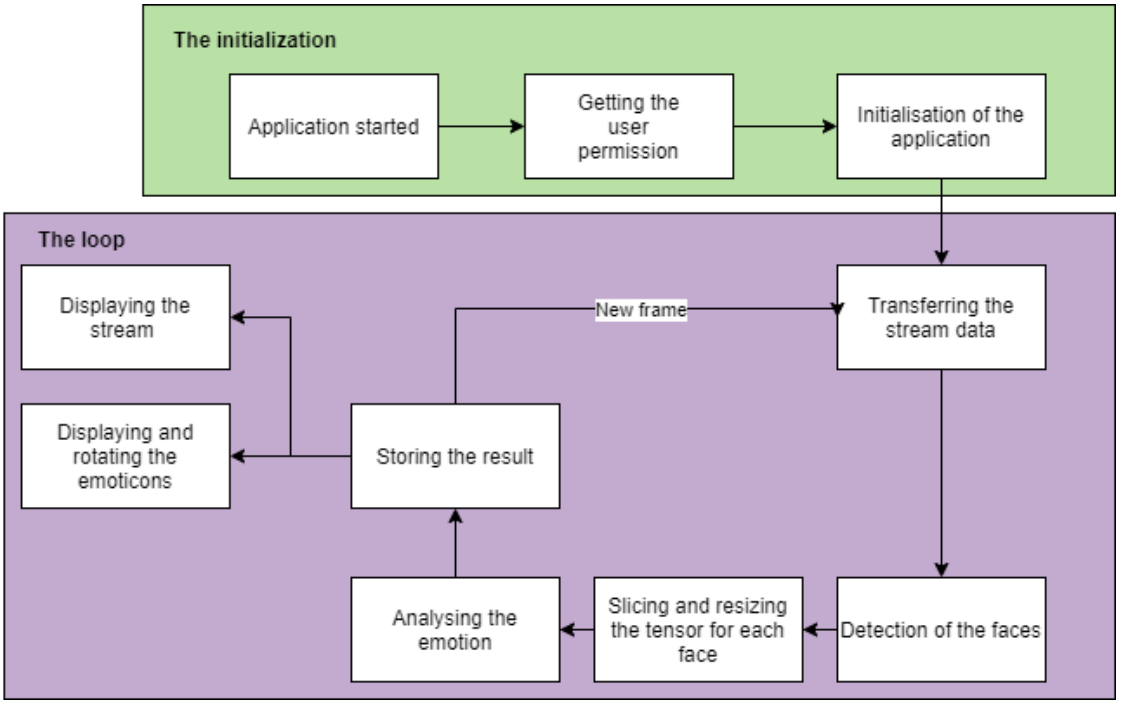


FIGURE 1.4.18 – Workflow of the application

As The application is started, there is a need for the user to allow the application to use the camera. When it is done the application will load the models and begin the camera stream.

After the initialization, we can enter the loop.

— The first step of the loop is the acquisition of the stream data from the camera.

— The second step is the detection of the faces on the frame from the stream. The coordinates of the faces are stored inside a JSON (position of the corner , eyes, nose, ears and mouth).

— The third step begin with a slicing of each face from the frame. Then the faces are resized to correspond with the tensor input of our model.

— The Fourth step is the analysis of the faces by the model and the recuperation of the data.

— The fifth step is the storing of the results.

— The final step of the loop before going back to the acquisition of a frame is the displaying step. On the first layer the frame analyzed and on the second layer the corresponding emoticon for each face with the good rotation are displayed to the user.

5 Conclusion

To conclude this report, we are satisfied with the result of our models but there is still margin for improvement by having a bigger dataset and balance between the emotions. A way to improve the dataset would be to blend the fer2013 and affectNet to have a well balanced one the only problem will thus be in the need of better hardware. Better hardware could also help to achieve better result by enhancing the processing power and the efficiency of the training. We only had a 3080 RTX mobile version which is a already a good graphic card but in terms of machine learning it is really trying to nail with a shoe. The application can also be upgraded to be more stable, we can also add new features such as a button to take picture or a button to switch for the front camera.

Bibliographie

[1] Algorithmia (2018). Introduction to facial emotion recognition. https://algorithmia.com/blog/introduction-to-emotion-recognition. (Accessed on 06/06/2021).

[2] Amidi, A. (2021). afshinea/stanford-cs-230-deep-learning CNN. original-date : 2018-11-27T05 :12 :37Z.

[3] Chollet, F. (2017). Xception : Deep learning with depthwise separable convolutions.

[4] Das, S. (2019). CNN Architectures : LeNet, AlexNet, VGG, GoogLeNet, ResNet and more. https://medium.com/analytics-vidhya/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5, accessed online March 2021.

[5] Dorian (2020). Building a ResNet in Keras.

[6] Expo (2021). Expo platform. https://expo.io/, accessed online april 2021.

[7] Fabien, M. (2019). Xception model and depthwise separable convolutions - . https://maelfabien.github.io/deeplearning/xception/#. (Accessed on 06/06/2021).

[8] Facebook (2021). React native. https://reactnative.dev/, accessed online april 2021.

[9] Google (2021a). Blazeface. https://github.com/tensorflow/tfjs-models/tree/master/blazeface, accessed online april 2021.

[10] Google (2021b). Tensorflow js. https://js.tensorflow.org/api_react_native/0.5.0/, accessed online april 2021.

[11] Sharma, P. (2020). 7 Popular Image Classification Models in ImageNet Challenge (ILSVRC) Competition History. https://machinelearningknowledge.ai/popular-image-classification-models-in-imagenet-challenge-ilsvrc-competition-history/, accessed online April 2021.