Castilla-La Mancha university

Higher School of Computer Engineering

Final Degree Project

Degree in Computer Engineering

computing

Emotion recognition using Deep Learning techniques

Guillermo López Bermejo

December, 2022

FINAL DEGREE PROJECT

Degree in Computer Engineering

computing

Recognition of emotions through Deep Learning techniques

Author: Guillermo López Bermejo

Tutor: Roberto Sánchez Reolid

Co-Tutor: María Teresa López Bonal

December, 2022

To my family,
for giving me all the facilities to
face this academic stage.

Authorship statement

I, Guillermo López Bermejo, with DNI [redacted], declare that I am the sole author of the final degree project entitled “Emotion recognition through Deep Learning techniques”, that the aforementioned work does not violate the laws in force on intellectual property, and that all The non-original material contained in said work is appropriately attributed to its legitimate authors.

Albacete, December 2, 2022

Signed: Guillermo López Bermejo

Summary

The objective of this TFG is to carry out a study of how Deep Learning could be applied to detect emotions in human faces.
To do this, first of all the performance of different architectures of convolutional neural networks that already exist for our specific problem will be evaluated to subsequently optimize them, and based on the conclusions obtained, design our own architecture more adapted to the problem.

For the development of the project, as we will expand on later, we will use the FER2013 facial expression recognition database, whose state of the art is around 76.82% accuracy using an ensemble of several neural networks.

In this project, an optimization of the Den-seNet201 architecture is proposed as a final model that obtains an accuracy of 69.35%. Furthermore, as we will see during the development of the project, we will explore the optimization of our model to improve other metrics more appropriate for the problem than the accuracy itself.

Thanks

I wanted to thank all the professors who have been part of my academic career during my degree for this work, because without their knowledge the development of this project would not have been possible, and especially to my tutors Maite and Roberto for having known guide me during the development of work. I also wanted to thank my family and friends, because your moral support has been key in keeping me focused and motivated until finish the job.

General index

Index of figures

Table index

1. Introduction

In this first introductory section, both the motivation and objectives for the development of the project and its organization and structure will be explained, as well as a brief explanation of the tools used.

1.1. Motivation and objectives of the project

In recent years, deep learning has burst into the field of machine learning. learning, this is due to its use in tasks as complicated as artificial vision.

This TFG will address the use of these algorithms to address the specific problem of recognizing emotions in people using images. The objective is to create and train a network that will receive images of people expressing different emotions, and that must be able to classify these images. depending on the emotion expressed.

Given that these emotions result from a biological response to a feeling, this type of system could be applied to know, among other things, the degree of satisfaction of a certain target audience for a certain product, or it could be useful for certain studies. psychological during a job interview.

The objective of this project will be, first of all, to carry out a study of the performance of certain existing neural network architectures for this specific problem. After this study, the conclusions obtained will be used to design our own model or optimize existing architecture that obtains the best possible results.

1.2. Project Organization

We can divide the project into the following parts:

1. **Introduction:** In this section, both the motivation and objectives of the present project and the tools used to carry it out will be introduced.

2. **Concepts:** This section will introduce both the principles on which the operation of artificial neural networks is based (from the simple artificial neuron to more complex topologies) as well as other key concepts in the development of the project, such as the basis data and metrics used.

3. **Emotion recognition:** In this section, after a brief introduction to the concept of emotion recognition and the methodology used, we will proceed to evaluate the performance of 4 of the best-known convolutional neural network architectures for emotion recognition. . Finally, using the conclusions drawn from the study, we will try to obtain the best possible architecture for this task.

4. **Conclusions:** In this last section a brief summary of what was learned will be made during the development and the final conclusions obtained.

1.3. Tools

For the development of the project we will use the following tools:

ÿ For the development of the networks, the *Python* programming language will be used . This is undoubtedly the best option for the development of neural networks, due to the large number of tools and libraries it has in the field of computer science. data.

ÿ All the code will be organized in files with an .ipynb *extension,* this corresponds to a special type of python files for the development of *Jupyter notebooks.* These notebooks are widely used in the field of data science, since they allow the code to be organized into two types of cells: *executable cells* and *text cells.*
Executable cells allow us to section the code so that we can execute only the part of the code that we are interested in executing, whether to train a model, display graphs, etc. On the other hand, the text cells are non-executable cells, written in a variation of the *Markdown* markup format that we can insert between the executable cells to explain and comment on the results, show the conclusions obtained, etc.

ÿ The neural networks will be developed using the *Keras library,* which is a library for the development of *deep learning* models built on the basis of the *Tensorflow library,* used for the development of all types of *machine learning models.* These two libraries are, along with *pytorch* to a lesser extent, the most used.

The *Keras* library has a variety of methods for developing all types of neural networks, from simple neural networks whose layers are defined sequentially, to more complex networks with non-sequential connections between layers.
The library also has methods to adjust both the number of neurons and the rest of the parameters that define each layer.

As will be explained in more detail in section 3.2, in our case we will focus on evaluating already existing architectures. Which will allow us, on the one hand, to use architectures with good base performance and, on the other hand, take advantage of the pre-training that these networks already have with other databases.

Specifically, we will evaluate four architectures: VGG [Karen Simonyan, 2015], ResNet [Kaiming He, 2015], DenseNet [Gao Huang, 2016], and Inception [Christian Szegedy, 2015]. These already pre-trained architectures can be invoked using the methods defined in Keras, whose documentation is found in [Chollet, 2014].

2. Concepts

In this section, several concepts common to the entire project will be explained, from all those related to the operation of artificial neural networks to the database and metrics used for both the training and evaluation of the developed models.

2.1. Neural network

2.1.1. Neuron

Neural networks are systems whose operation is based on an abstraction of brain neural networks [Education, 2020].

The most fundamental unit of an artificial neural network is the neuron, this, like a biological neuron, has several input and output channels, the function of the neuron is to receive the inputs coming from the neurons of the previous layer or from the input itself in addition to the bias neuron, each of those inputs will be assigned a weight and the function of the neuron will be to add the result of multiplying each of the inputs by the weight of each input.

In figure 2.1 we can see the representation of an artificial neuron, **X0** and **X1** represent the values that reach the neuron, and **W0, W1** and **W2** the weights assigned to those inputs, the last neuron **(1)** represents the bias, which will be responsible for moving the function **f(x)** on the horizontal axis[?].

These types of transformations in themselves could be useful for classification problems that can be solved using one or more linear functions, but there are many different types of distributions for classification problems, and many of them require some **non-linearity** in the functions to classify. the data correctly. In figure 2.2 we can see two different distributions for a classification problem, the distribution on the left (2.2a) could be solved using a neural network with a linear activation function, while for the distribution on the right (2.2b) it is mandatory to use

some mechanism to add non-linearity to the model.

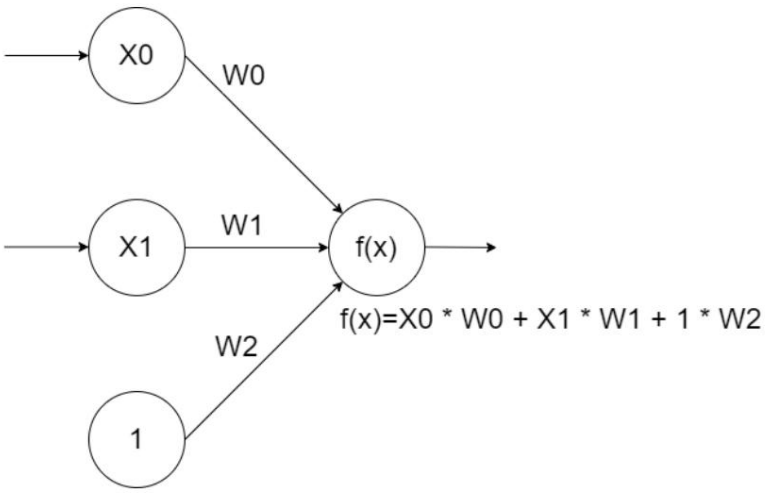
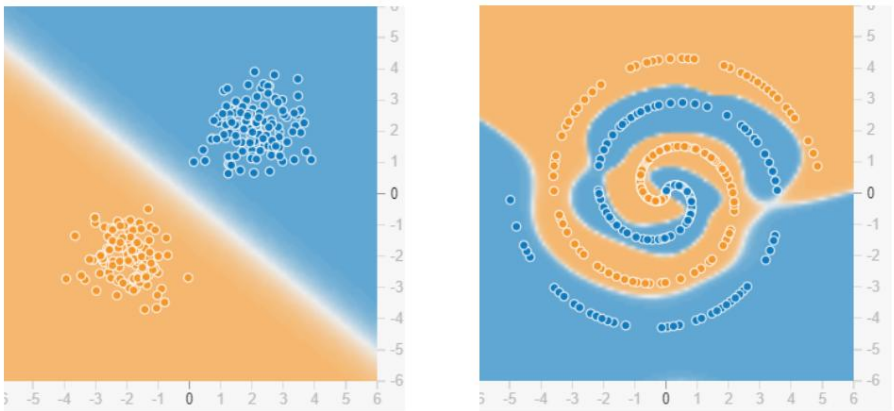


Figure 2.1: Artificial neuron



(a) Linear classification **(b)** Non-linear classification

Figure 2.2: Different distributions in classification problems

To add nonlinearity to the model, a special type of functions called activation functions, these will be applied to the function f(x) as an additional step 2.3.

Some of the most important activation functions are briefly defined below.

ÿ Sigmoid:

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.1}$$

This function (2.1) returns values in the range between 0 and 1, which makes it very suitable for use in the last layer, since it can represent the probability of membership in each of the classes.
Its main drawback is the low values of the derivative in the function in the central part of it, which accentuates the problem of gradient fading.
(annex E.1).

ÿ Tanh:

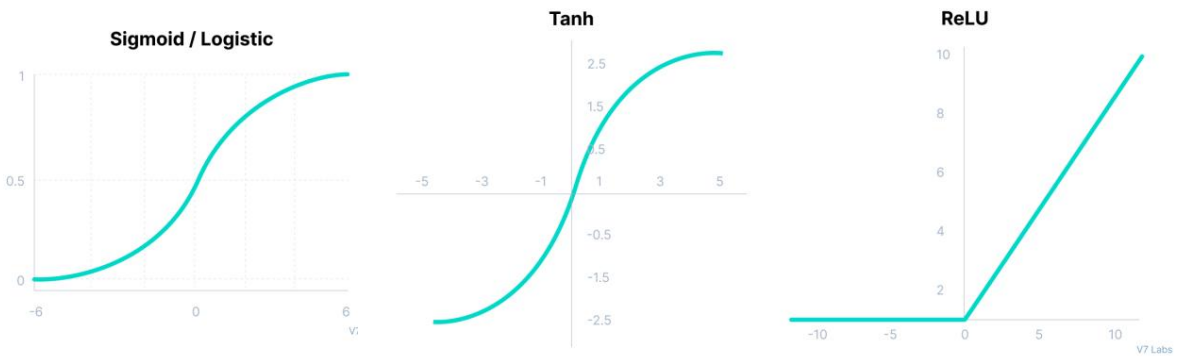
$$f(x) = \frac{e\ x\ddot{y}e\ddot{y}x}{e\ x+e\ddot{y}x} \tag{2.2}$$

This function (2.2) is very similar to the sigmoid, with the difference that it is not bounded between 0 and 1.
It is usually used in hidden layers. Since it is centered at 0, the values of output are differentiated between negative, neutral and positive.
Like the sigmoid function, the value of the derivative at the central values is very low and accentuates the problem of gradient fading.

ÿ ReLU:

$$f(x) = \max(0, x) \tag{23}$$

In this case (function 2.3) only neurons with output greater than 0 will be activated, which improves computational efficiency.
They are used in hidden layers since their linearity accelerates the convergence of the descending gradient.
The drawback of this type of function is the possible appearance of dead neurons which will never be activated, since all negative input values will be transformed into 0, which can harm the learning capacity of the model [Baheti, 2022].



(a) Sigmoid function **(b)** tanhl function **(c)** ReLu function

Figure 2.3: Activation functions

2.1.2. Neural network topology

A neural network is made up of a series of layers, each layer is made up of a number of neurons, there are 3 main types of layers:

- **Input layer:** it is the input layer of the data, and is only connected to the next layer of the network, each of the neurons will represent an input variable in the network

- **Hidden layer:** these are the layers that are between the input layer and the output layer. There are various types of hidden layers, and both the number of these and the number of neurons in each of the layers will depend on the complexity and type of problem to solve.

- **Output layer:** this is the last layer of the network, it can be made up of one or several neurons, and the returned value will be considered the result of the network. In classification problems, each of the neurons in the output layer represents each of the classes of the problem, and the value returned by each of them will represent the probability of belonging to the record characterized by the input variables. the class that represents the neuron, this probability can be simulated using activation functions in the output layer that limit the result between 0 and 1, as is the case of the sigmoid function.

Both the number of layers and neurons, and the way in which they connect with neurons in other layers will define the topology of the network. There is a wide range of neural network topologies. Among them, the most used is the multilayer perceptron (2.4), in which we will have an input layer and an output layer, varying the number of hidden layers and neurons in them.

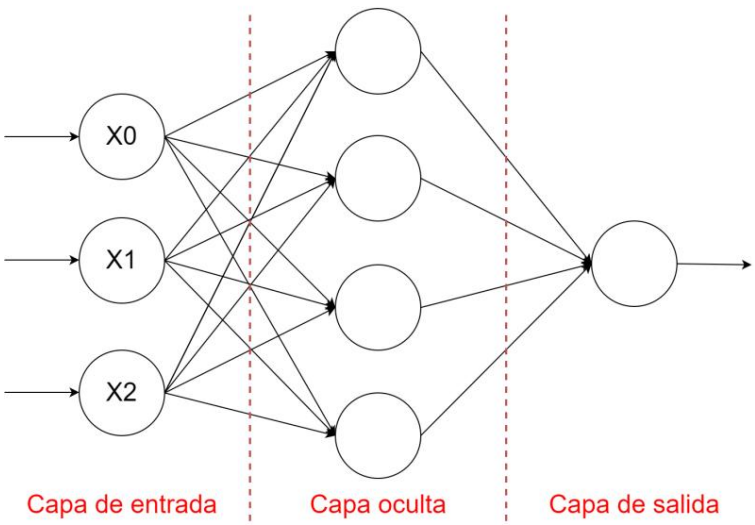


Figure 2.4: Multilayer perceptron

2.1.3. Network learning

The learning of the network occurs in the following way: first the information is propagated from the input layer to the output layer, and returns a result. This result is compared with the real result using a loss function whose result will represent the disparity between the obtained value and the expected one.

Once the loss is calculated, the network will be traversed backwards in a process known as *backpropagation,* with the aim of updating the network weights in a way that minimizes the loss using the gradient descent algorithm.

loss function

The loss function is the one that returns the disparity between the value obtained and the real value, this value will be a function of the weights of the network, so it will be key when calculating the update value of the weights of the network. grid. Among the properties that this function must meet is that of being continuously differentiable, which will allow calculating the derivative of the function with respect to each weight when applying gradient descent 2.5.

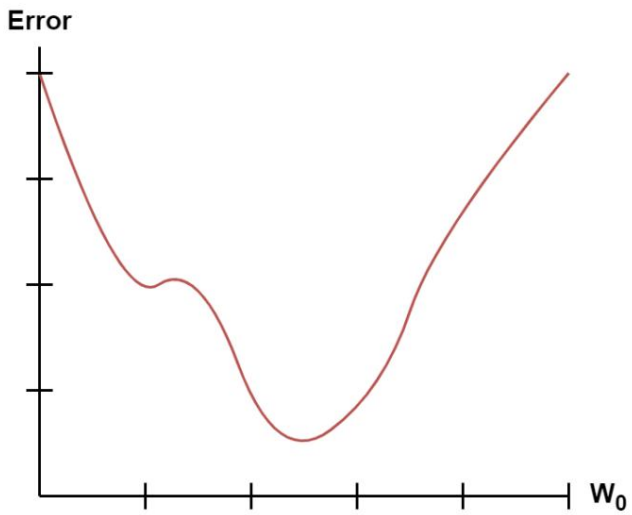


Figure 2.5: Loss function

There are several types of loss functions. Among them we can highlight the error qua-
mean drastic and cross entropy.

ÿ **Mean squared error** [Sammut and Webb, 2010]: is the loss function used in regression problems, and measures the average of the squared errors, that is, the difference between the estimated value and the real value squared (eq. 2.4).

$$NDE = \frac{1}{n} \sum_{i=1}^{n} (\hat{Y}_i - Y_i)^2 \quad (2.4)$$

Where $\hat{\mathbf{y}}$ is the vector of n predictions, and **Y** is the vector of true values.

- **Cross entropy** [ichi.pro, 2021]: Entropy measures the level of uncertainty of all possible outcome variables. The equation used will depend on the value of X, so if **X** is continuous, equation 2.5 will be used, while if it is discrete, the one defined in 2.6 will be used.

$$H(X) = -\int_{x} p(x) \log(p(x)) \quad (2.5)$$

$$H(X) = -\sum_{x} p(x) \log(p(x)) \quad (2.6)$$

With the predicted probabilities being a probability vector of type [0.775, 0.126, 0.039, 0.070] and the actual probabilities being a vector of type [1, 0, 0, 0], the cross entropy loss function 2.7 compares the membership probability of each predicted class with the actual probabilities, penalizing the probabilities based on how far the expected value is from the predicted.

$$L = -\sum_{i=1}^{n} t_i \log(p_i) \quad (2.7)$$

Where **n** is the number of classes, and **t** and **p** are the actual and predicted probability vectors, respectively.

gradient descent

To go through the loss function of the model so that the global minimum that minimizes the error can be found, the gradient descent algorithm is usually used [April, 2021].

The idea behind this algorithm is to find the derivative of the error function based on each of the synaptic weights of the network. This value of the derivative is known as the gradient, and its inverse, as shown in Figure 2.6, will indicate the direction in which that particular weight must be shifted to minimize the error function. A gradient of 0 tells us that the function is in a local minimum.

Therefore, the gradient will represent the direction that produces the greatest increase in the error from a certain point in space, and its negation (equation 2.8), the direction that minimizes the error.

$$\Delta = -\frac{\partial E}{\partial W_{ij}} \quad (2.8)$$

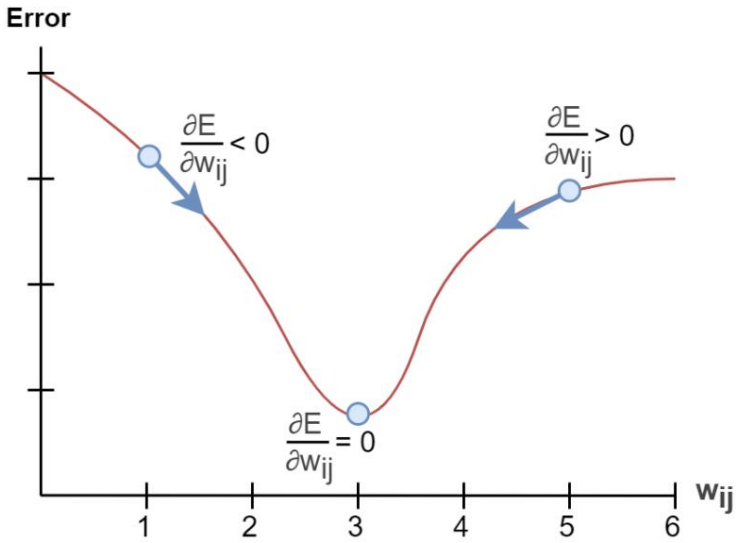


Figure 2.6: Gradient Descent

Backpropagation

Once the necessary concepts have been defined, we can introduce the algorithm that is directly responsible for updating the weights of the neural network, the *backpropagation algorithm.*

The algorithm works as follows: first, a data input is introduced to the network, this input runs through the entire network until it reaches the last layer (output layer), which returns a result. Since we are in the training phase, we can compare the output of the network with the real value, and obtain an error that, as we already mentioned, could be the mean square error in case of regression problems or the cross entropy in the case of classification problems.

The learning process occurs by updating the synaptic weights of the network based on the error obtained after the comparison, and said updating of the weights will depend on the layer in which we are.

The process is shown in Figure 2.7 and, as we can see, it begins in the neuron of the output layer towards the previous layers of the network. This first learning phase is shown in Figure 2.7a, and will cause the update of the synaptic weights that connect the output layer with the immediately preceding layer. If we take the update of the wbc weight as an example, the new value will result from applying equation 2.9.

$$wbc = wbc + ÿ \cdot ÿc \cdot hb \tag{2.9}$$

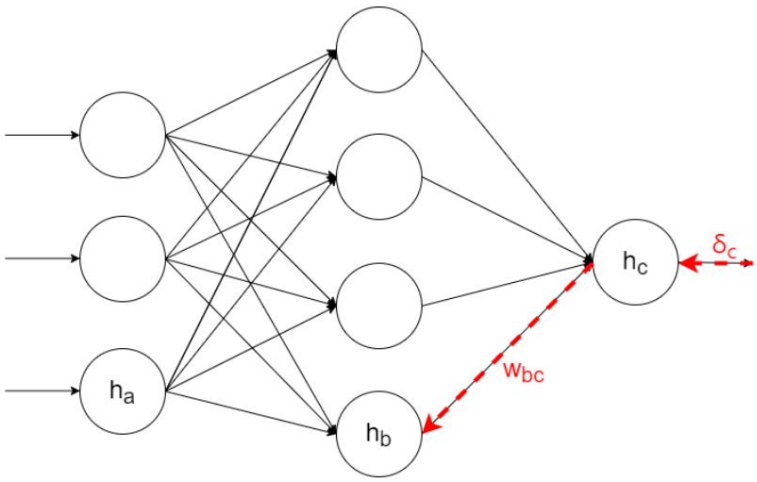
Where ÿ is the learning rate, ÿc is the increase in the gradient in c, and hb is the value in b.

To update the weights of layers prior to the output, since we cannot calculate the error to obtain the gradient, we will calculate the influence that each of the neurons in the layer has had on the final error. Considering figure 2.7b, the new weight for wab would be obtained after applying equation 2.10.

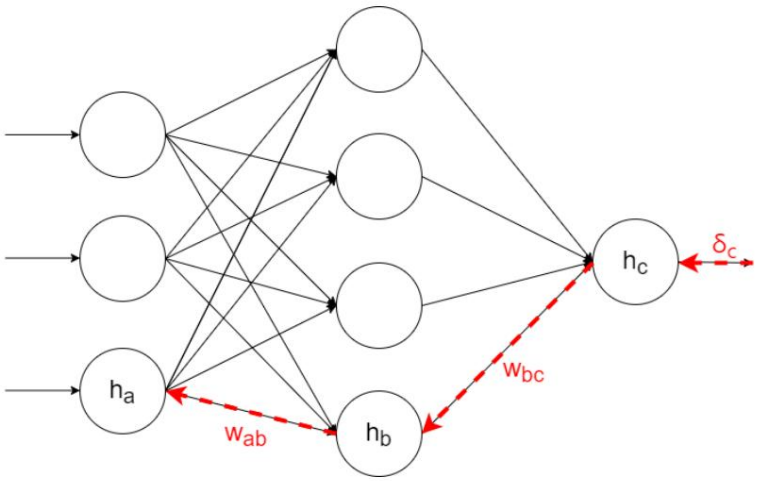
$$\ddot{y}b = (\ddot{y}_{x\ddot{y}s(b)} wbc \cdot \ddot{y}c) \cdot hb \cdot (1 \ddot{y} hb) \quad (2.10)$$

Being s(b) the successors of b in the network.

In this way we would have gone through all the layers of the network as shown in figure 2.7b



(a) First level backtracking



(b) Second level backtracking

Figure 2.7: Backtracking process

Bias and variance

When developing any type of model based on **machine learning,** there are two fundamental concepts when performing model selection: bias and variance [Baro, 2009].

The learning process of a machine learning model consists of training it with part of the data set, this set is known as the training set, and it is from this that the model will extract the necessary patterns during the learning.

Since each of the data is different, an excessively complex model that fits 100% to the training data might not be convenient if what we are looking for is a model capable of learning patterns that allow us to classify new independent data. to the training set, in this case we would say that it is a model that is **overfitting** to the training data and therefore has a high degree of **variance.** This degree of variance is usually proportional to the complexity of the model.

On the other hand, if what we have is a model that does not fit the training data set sufficiently, we say that this model is **generalizing** and therefore it is a model with a high degree of **bias.** In this case the degree of bias is considered inversely proportional to the complexity of the model.

The ideal therefore is to find a degree of compromise between the bias and the variance of the model, all based on both the nature of the problem to be solved and the database.

2.2. Convolutional neural network

Artificial neural networks are an adaptation of neural networks to be used in computer vision problems, which is the discipline that studies the ability of a computer to analyze and understand real-world images.

A convolutional neural network is composed of two main parts: the body and head.

- **Body:** It is the part of the neural network that is responsible for extracting the so-called “Feature Maps” from the image. The set of all feature maps allows the network to “Understand” and differentiate that image from others belonging to other classes. There are many types of feature maps that can be extracted depending on the specific problem, and they can range from simple attributes such as lines to more complex colors, textures and shapes.

- **Head:** Once the necessary information has been extracted from the images, the head of the network classifies images into different classes.

In figure 2.8 we can see a representation of what was explained above, the body of the neural network is responsible for extracting the characteristics of the input image, in this case it is a car from which 4 characteristics are extracted: windows, chassis, headlights and wheels, so that finally the head of the network classifies it as a ”Volkswagen beetle”.

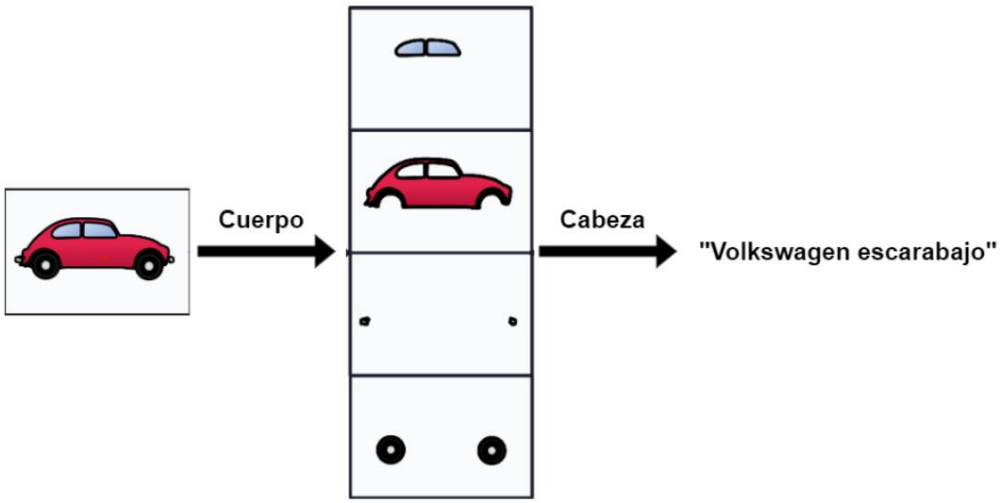


Figure 2.8: Convolutional neural network parts

These types of networks work in the following way:

First, we represent the image as a set of pixels, each of which will have a value depending on the intensity of that pixel in the image. In images with more than one color, several channels will be used, each corresponding to a color. For example, in black and white images only one channel will be needed, while in RGB images 3 channels will be used. Each of these pixels can be an input neuron to our network, so since images are two-dimensional arrays, a neural network for 48x48 black and white images will have 2,304 input neurons.

The two-dimensionality of the input images implies that this pixel input information does not provide sufficient information about the image, since their position in the matrix is also relevant. This is why we must apply a special type of transformation called **convolution.**

This convolution process is divided into three phases:

ÿ **Filtering:** consists of applying a “filter” to the image in question. A filter is nothing more than a matrix smaller than the image. As the filter moves over the image, the input values will be multiplied by those of the filter. There are various types of filters responsible for detecting lines, colors, textures or other attributes of the image.

image, in figure 2.9 we can see how we apply a 2x2 filter (orange matrix) capable of detecting horizontal lines on a 6x6 image, this filter would move until it covers the entire image. The values of these filters will be updated as the network training progresses until obtaining those that generate the most appropriate feature maps.

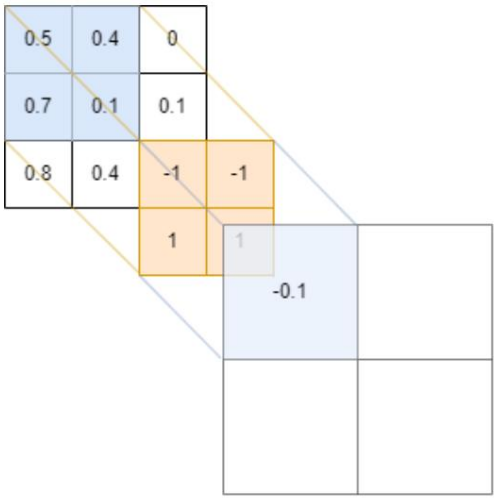


Figure 2.9: Filtering process

ÿ **Detection:** once the image is filtered, an activation function such as the ReLU function (introduced in section 2.1.1) is applied to it, so that all the slightly relevant values are equally insignificant, intensifying the relevant values and obtaining the feature map. As we can see in figure 2.10, negative values become 0 while positive values remain the same.

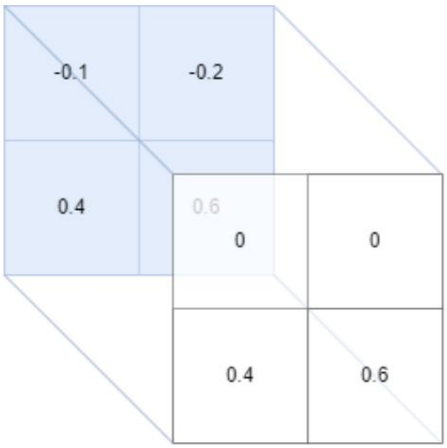


Figure 2.10: Detection process

ÿ **Condensation:** there are several condensation methods, the most common is known as "max pooling", which consists of going through the image in a similar way to what was done during filtering but instead of applying a filter, using the highest value of between the pixels traveled, as shown in figure 2.11.

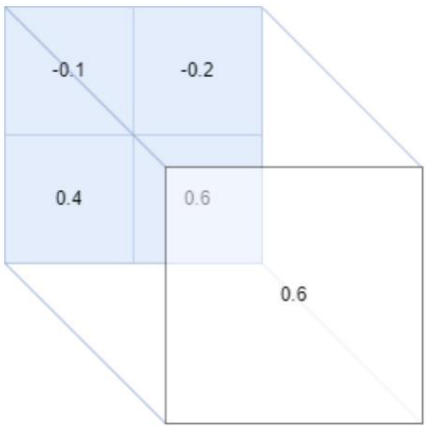


Figure 2.11: Condensation process

As we have seen, convolution has the secondary effect of reducing the resolution of the image. There are different techniques to avoid this phenomenon, such as adding edges to the original image, it all depends on the architectural design.

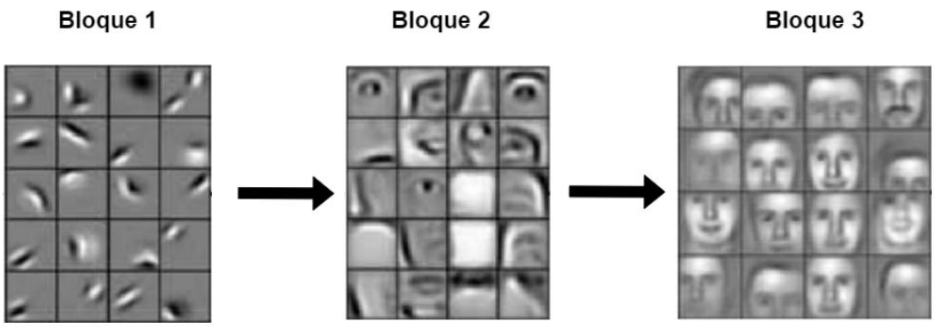


Figure 2.12: Feature Map Levels

Usually convolutional layers are grouped into blocks. For each block, some convolutional layers are grouped that lead to a **max pooling** condensation layer that applies a reduction in the image size, so that more feature blocks can be generated in the next block. Furthermore, this allows the information to be condensed from block to block, since in the case of applying, for

For example, 2x2 filters, each pixel that passes from one block to another contains the information of the previous 4 pixels, this allows generating feature maps with increasingly complex shapes. This can be seen in Figure 2.12, in which the first block of a convolutional network trained with human faces specializes in the detection of simple attributes such as lines and curves, which will be used by the second and third blocks for the detection. detection of more complex attributes such as facial features and full faces. This “hierarchical learning” approach in which the depth of the network determines the obtaining of increasingly significant representations is what is known as **“Deep Learning”.**
[Recuero, 2018].

23. FER2013

For the development of this project, the database for facial expression recognition **FER2013** [April, 2013] will be used, which has 35,887 images with a resolution of 48x48 pixels classified into 7 different emotions: **”Anger”, ”Disgust”, ”Fear”, ”Happiness”, ”Neutrality” , ”Sadness”** and **”Surprise”.**

The distribution of the classes in the database is shown in figure 2.13.

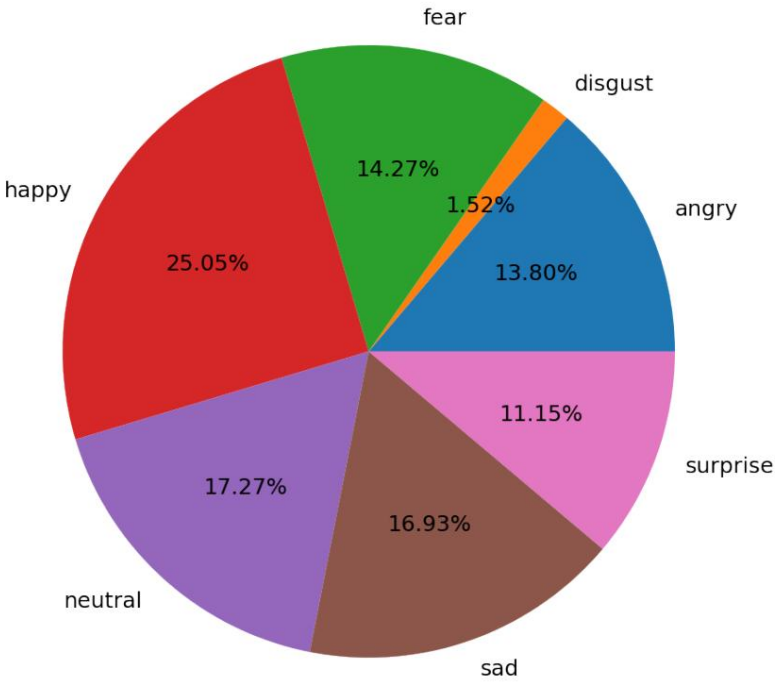


Figure 2.13: Distribution of fer2013 classes

We can see that the distribution of the classes is not balanced, in particular we have a large number of records for the “happy” class (25% of the total), while the “disgust” class only represents 1.52%, this implies that the database is unbalanced.

Regarding the **state of the art** of models trained with this dataset, we find a maximum hit rate or accuracy (2.4.1) of 76.82% [Paperswithcode, 2022].

As we will see, this imbalance in the distribution of classes can lead to certain metrics not adequately reflecting the performance of the models, so one of the objectives of the study will be the optimization of a model that maximizes other more metrics. suitable for measuring performance in unbalanced databases.

2.4. Metrics used

Our model will return an array whose length will be equal to the number of classes (7 in this case), each of the values of that array corresponds to the probability of membership of the input image to each of the 7 classes. To make the final prediction we must consider only the highest value of the array.

For the choice of metrics we must consider that it is a problem of classification.
multiclass sification in which all classes have equal importance.

To define the formulas we will use the following nomenclatures [Fawcett, 2005]:

ÿ True positives **(TP):** for a given class, the number of database records corresponding to that class correctly classified as that given class by the model. ÿ True negatives **(TR):** for a given class, the number of database records

not corresponding to that class correctly classified as another class (not necessarily the correct one) by the model.

ÿ False positives **(FP):** for a given class, number of database records not corresponding to that class incorrectly classified as that given class by the model.

ÿ False Negatives **(FN):** For a given class, the number of database records corresponding to that class incorrectly classified as another class by the model.

ÿ Total positives (P): total number of real positives for a given class (VP + FN).

ÿ Total negatives (N): total number of real negatives for a given class (VN + FP).

At the same time, as seen in section 2.3, we must consider that the database is not completely balanced, so the classes that appear the most could overshadow those that appear the least for certain metrics.

2.4.1. Accuracy

The accuracy **(ACC)** or success rate is the metric that measures the number of times the model is correct in its predictions.

$$ACC = \frac{VP + VN}{P + N} \quad (2.11)$$

As we can see in formula 2.11, the weight that the true negatives (VN) contribute to the metric implies that it is not suitable for unbalanced data sets, since the successes in the majority class will overshadow the failures of the minority ones, so Although it can be a good indicator of the virtues of the model, we must also consider other metrics.

2.4.2. Metrics for imbalanced data

As we have introduced previously, metrics such as accuracy are not good indicators to measure the efficiency of models on unbalanced databases, since, for example, a hypothetical classifier that worked on a database with 990 samples of the class 1 and only 10 of class 2, it would obtain a success rate of 99%, being a bad model whose only job would be to classify all the records as belonging to class 1 regardless of the input data.

Metrics for multiclass classification

Since our emotion classifier must classify the input records into 1 of 7 classes, we are faced with a multi-class classification problem. Since these types of metrics are calculated based on each of the classes, it is necessary to use an appropriate method to average the results obtained and obtain a single value that represents the value of the metric for the model. There are various methods to average the metrics of a multiclass classifier, the two most representative being the macromean and the micromean.

- **Macromedia:** It consists of adding the confusion matrices obtained for each of the classes to later calculate the metrics on it.
- **Micromedia:** It consists of obtaining the confusion matrices for each of the classes and calculating the average of all, this makes it more suitable for an unbalanced data set since the score obtained for the class with fewer records will have the same relevance than that obtained for the classes with more records.

In our case, since we are faced with a multi-class classification problem of 7 classes, all of them with the same importance, we will choose to use micromedia .

confusion matrix

The confusion matrix [Fawcett, 2005] is the basis of the rest of the metrics. It is a two-dimensional matrix in which each column represents the number of predictions corresponding to each class while each row represents the real value. With i being the .

rows and j being the columns, each cell Cij represents the number of records in the database. predicted data as class *j* that actually belongs to class i. In our case, as shown in equation 2.12, to make the matrix more interpretable, this value will be divided by the total number of real records that belong to that class, so that we will obtain a value between 0 and 1.

$$Cij = \frac{Cij}{T\ otalj} \tag{2.12}$$

Recall

The Recall (R), also known as sensitivity or true positive rate [Fawcett, 2005] , represents the fraction of positive cases that have been detected, that is, of all the cases that exist for a given emotion, how many of them have been classified as such, as can be seen in equation 2.13.

$$R = \frac{VP}{VP + FN} \tag{2.13}$$

Precision

The precision or positive predictive value [Fawcett, 2005] measures the fraction of the cases classified as positive that were really so, that is, of all the cases classified as a certain emotion, what fraction of them really are.

$$R = \frac{VP}{VP + FP} \tag{2.14}$$

F1-score

The F1-score [Taha AA, 2015] is used to measure the balance or harmonic mean between recall and precision, as shown in equation 2.15.

$$F1 = \frac{2 \cdot R \cdot P}{R + P} \tag{2.15}$$

ROC/AUC curve

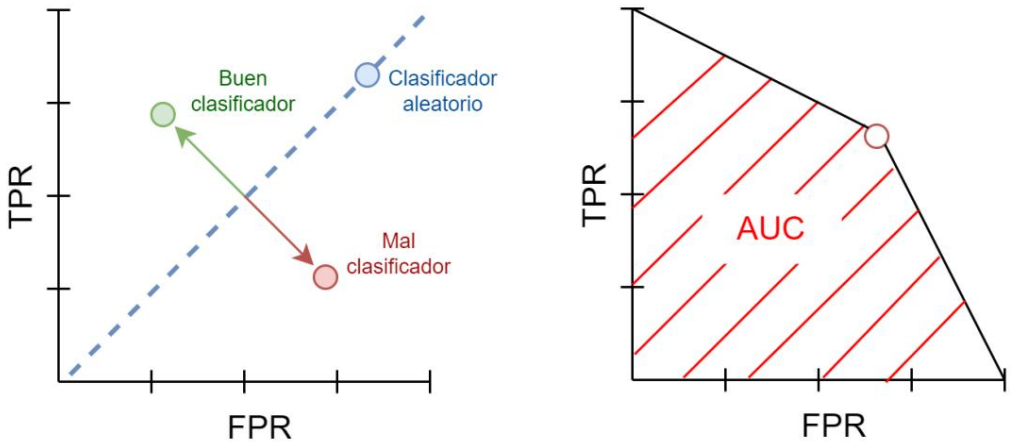
The ROC curve [Fawcett, 2005] is defined as the curve that we obtain if we represent on an axis of Cartesian coordinates both the rate of true positives or sensitivity on the Y axis and the rate of false positives (1 - specificity) on the X axis. .

Sensitivity indicates the ability of our system to classify truly positive cases as positive, while specificity indicates the ability to classify

as negative the really negative cases. Therefore the ROC curve obtained (also known as ROC space), will indicate the balance between the rate of true positives (presumed benefits), and the false positive rate (presumed costs).

Figure 2.14a shows the graph with the Cartesian axes corresponding to the ROC curve. A perfect classifier would place its ROC score in the upper left corner (coordinates (0,1)), and would be characterized by 100% sensitivity and specificity, while any random classifier would be placed at any point along the diagonal that joins the point (0,0) with (1,1).

Therefore, classifiers that place their score above the diagonal that represents randomness will be considered good classifiers, while those that are below will be considered poor classifiers (2.14a).



(a) ROC classifiers **(b)** AUC

AUC obtained	Classifier quality
0.5	Without discrimination
0.5 - 0.7	Poor discrimination
0.7 - 0.8	Acceptable discrimination
0.8 - 0.9	Excellent discrimination
0.9 - 1	Outstanding discrimination

Table 2.1: AUC ranges

When using this information for model comparison, it is common to use the area under the curve (AUC), which as its name indicates, represents the measurement

of the area under the curve generated in the ROC space (2.14b).

The interpretation of this metric when evaluating models depends on the field of application of the model, since the cost of an error on the part of the classifier in a medical context with a model used for disease prediction is not the same as in a context business, with the first one having to obtain a better score to be considered an appropriate model. For our model, we will consider a series of ranges, shown in table 2.1, extracted from [David W. Hosmer Jr., 2013], these ranges cover from a minimum score of 0.5, this being the one that any model that classifies the inputs would obtain randomly (blue dashed line in figure 2.14a), up to scores between 0.97 and 1, considering these as outstanding classifiers.

3. Recognition of emotions

Before starting the study, it is necessary to introduce what we can consider a emotion as well as the problems and limitations associated with its recognition through artificial intelligence systems.

An emotion is considered a mental process that reflects the evaluation attitude. of an individual in different situations [?]. The main difference between an emotion and a feeling, is that while emotions are more related to biological reactions to stimuli, feelings are mental perceptions of those stimuli, so that we can consider joy, sadness, fear or anger as emotions, while love, happiness or hate are more related to emotions. feelings [Chen, 2019].

The recognition of both emotions and feelings of people for part of intelligent systems has been the object of study due to the large number of applications in areas such as marketing since these emotions and feelings could indicate the degree of customer satisfaction with a certain product or service, the same way they could be used to analyze the skills of a certain person in a job interview. There are various patterns that allow us to recognize these feelings and emotions, whether analyzing the person's own language, their voice, their facial expressions...

To do this, certain assumptions must be made and limitations known. Among these assumptions is that of assuming, on the one hand, that emotions are classifiable and, on the other hand, that this classification is universalizable. In relation to the classification of emotions, there are different models for this, the most widespread being the one concluded by the study led by psychologist Paul Ekman in 1972, who concluded that there are six basic emotions that people express universally: surprise, sadness, fear, happiness, anger and disgust [Ekman, 1992]. This type of classifications, despite not being a definitive solution to the problem, since not all cultures express emotions in the same way, they can give rise to approximations that can be useful in certain areas that do not require perfect recognition and classification.

In this project we will focus on emotion recognition through the analysis of facial features. This is viable since, as we have mentioned, the

Emotions are characterized by producing biological responses in people, among which are the different facial expressions that can be compiled in an image database such as FER2013, which in turn are classified into 7 classes very similar to those proposed. by Ekman, so a convolutional neural network trained with these data could be capable of making an approach to emotion recognition.

3.1. Methodology

To find the best model for emotion recognition, we will follow the following methodology:

Firstly, we will evaluate the performance of several existing convolutional neural network architectures for the classification of the FER2013 database, to do so we will follow the following methodology for each of the architectures:

ÿ In the first phase, we will evaluate the performance of each of the architectures separately to determine the best possible hyperparameter configuration.

To do this, a search algorithm based on **Bayesian optimization** A.1 will be used to test the different combinations of hyperparameters and check which is the most optimal for each of the architectures.

Since the search algorithm must train the model for each possible combination of hyperparameters, we will use the **holdout validation method,** which, as we can see in Figure 3.1, after extracting the +test set from the data, uses a only fixed division between the **training set,** which as its name indicates is the data on which the network will carry out its learning through backtracking, and the **validation set** on which it will be evaluated, formed by the data set that does not has been used for learning. In this way, unlike methods that we will see later such as cross-validation, it will only be necessary to train the model once for each possible configuration.

To perform holdout validation, and since our database is considerably large, we can afford to reserve 20% of the total data as a test set. Which will only be used in the last validation to ensure the consistency of the model.

Of the remaining 80% of the database, we will reserve 10% (of the total) as a validation set, and the rest (70% of the total) will be the set used to train the models (3.1).

Since in holdout validation the validation set remains fixed, it is possible that the division of the database in training/validation could be, by chance, excessively kind or detrimental to the model in question, so after obtaining the best model, we will carry out a second validation on

the three best models, this time using the **cross-validation method.**

In this case, as we can see in figure 3.2, after extracting 20% of the database for the test, we will divide the rest of the database into 8 sets that we will use to train the model 8 times. For each training, we will use one of the 8 sets as validation, and the rest for training, and we will obtain the final metrics from the average of the 8 trainings. In this way we will ensure that the results obtained in holdout validation were not accidental.

ÿ Once the cross-validation has been carried out and the best model among the three has been identified consistently, we will carry out a final evaluation of the model, this time introducing the reserved 20% that was not used in the previous phases.

To obtain consistent results, we will repeat this training 10 times, randomizing the distribution between the training and validation sets and finally obtaining the average.

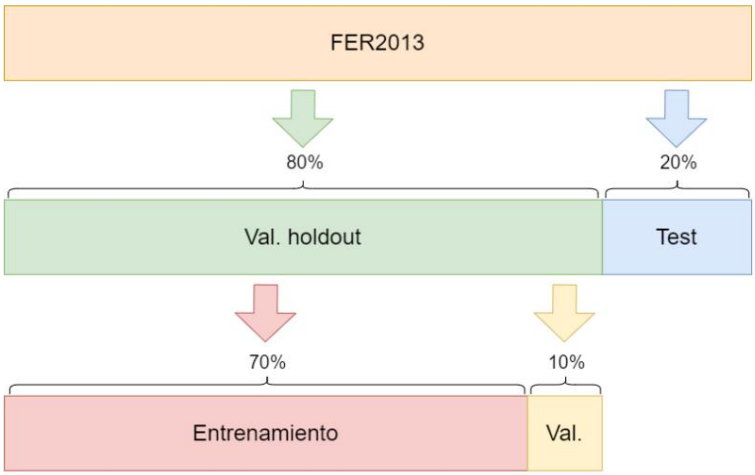


Figure 3.1: Holdout validation

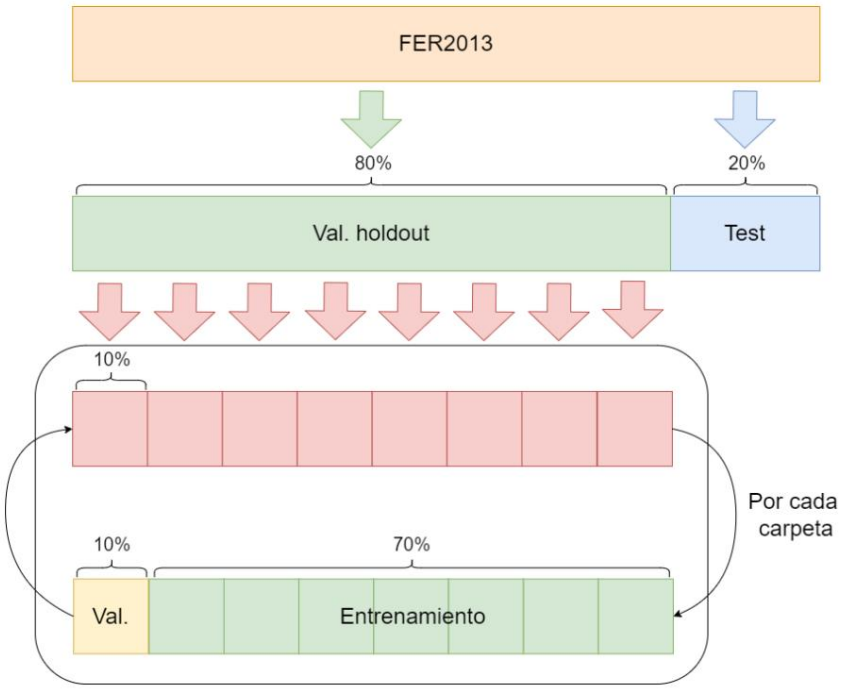


Figure 3.2: Cross validation

3.2. Existing architectures

In this part of the project we will dedicate ourselves to carrying out a study of the performance of widely known convolutional neural network architectures, all of which were developed to classify the **ImageNet data set,** which contains more than 14 millions with 224x224 resolution of images corresponding to 1000 classes of all types, from images of people to animals and all kinds of objects.

To train these models we will use a technique called transfer learning. This technique consists of using a pre-trained neural network instead of training it from zero with random weights. In our case we will use models whose base has been pre-trained with ImageNet, this, despite the fact that the ImageNet classes are not the same as those of FER2013, will allow the model to take advantage of many of the simplest features (those of the first layers and convolutional blocks) for the FER2013 classification.

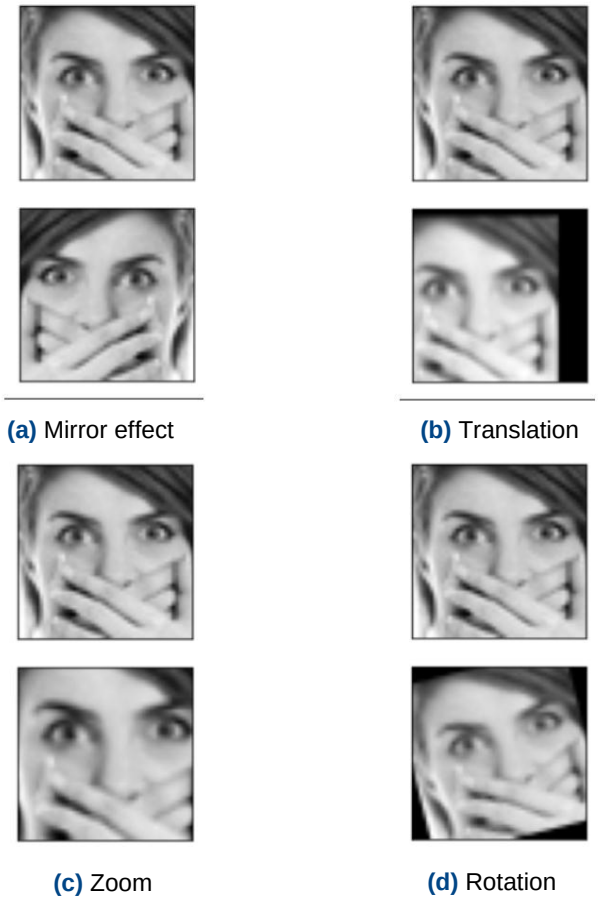
As we mentioned previously, first we will perform a Bayesian search to optimize the hyperparameters for each of the architectures, the hyperparameters to optimize will be the following:

- **Learning rate:** It is the learning rate of the model, we will consider the rates 0.01, 0.001 and 0.0001 within the search space.
- **Data Augmentation:** Data augmentation is transformations that are performed on the images with the aim of modifying them slightly and thus preventing overfitting During training, we will evaluate whether or not to add this technique during data pre-processing.

  There are many types of transformations to augment the data, for our models we will use four: mirror effect (figure 3.3a), translations (figure 3.3b), zoom (figure 3.3c) and rotations (figure 3.3d). These four transformations will allow us obtain variations of the images without modifying the originals excessively.
- **Version:** some of these architectures are presented in different versions that introduce variations in both the depth and structure of the network, so For each of the architectures we will consider different versions within of the search space.
- **Batch size:** represents the number of images that will traverse the network before performing each of the updates of the weights using the average of the gradient obtained, we will consider the batch sizes of 32, 64, 128 and 256 images within of the search space.

There is one last hyperparameter to optimize, it is the epochs: an epoch represents that all the entries in the training database have passed through the neural network and have been used to update their weights, after each epoch the metrics are evaluated obtained on the validation set.

The training of neural networks is divided into several epochs, in our case we will use a technique called **Early Stopping** so that the network stops training when this stops reflecting learning, in our case it will be considered that the network has stopped



(a) Mirror effect **(b)** Translation

(c) Zoom **(d)** Rotation

Figure 3.3: Data Augmentation

learn when 6 epochs pass without the evaluation of the *F1-score* obtained on the validation set increasing by 0.001 units, so it will not be necessary set the number of epochs or include it in the search algorithm.

Since the search space is relatively large, we will use the validation known as holdout validation.

3.2.1. Definitions

VGG

Visual Geometry Group (VGG) [Karen Simonyan, 2015] is a classic Convolutional Neural Network architecture. It is a neural network with a considerable depth that varies in its two versions, the **VGG16** version has 16 layers while **VGG19** has 19.

The architecture of these two versions is shown in figure 3.4.

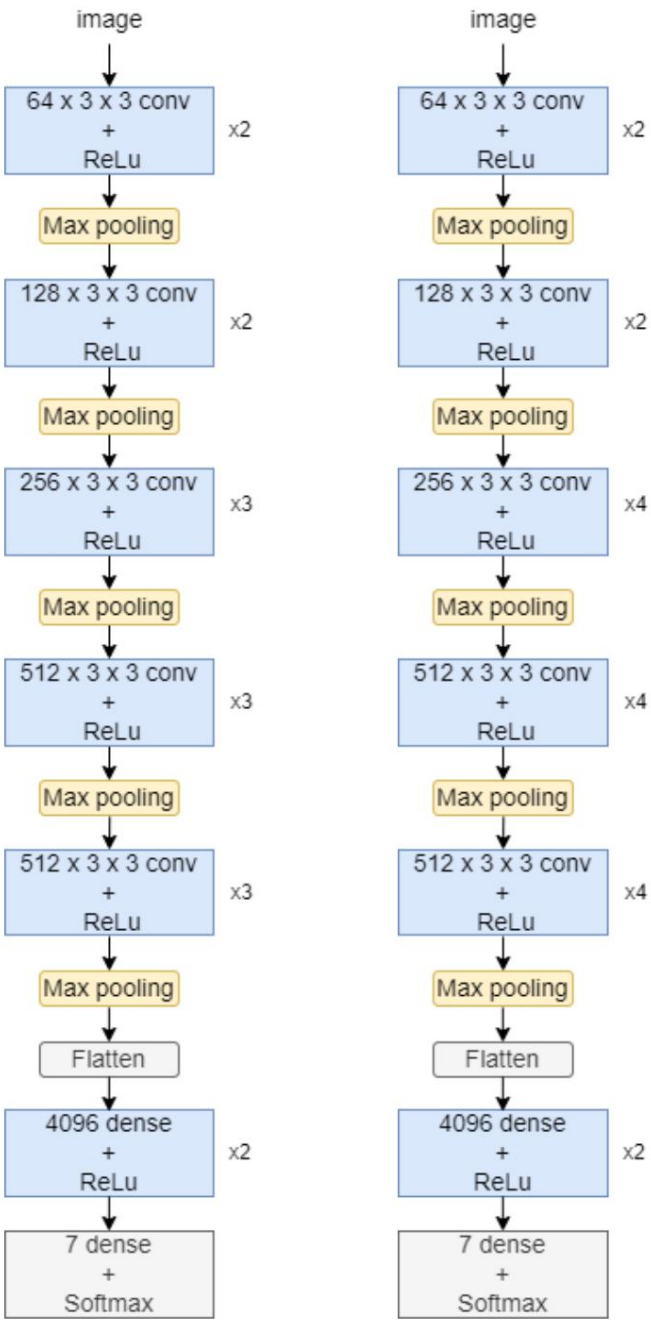
Both networks are made up of 5 blocks, made up of convolutional layers with 3x3 size filters with Relu activation function followed by a **max pooling layer.**

This max pooling layer will fulfill the maximum condensation function introduced in section 2.2, also reducing the dimensions of the images by half.

The convolutional layers of the first block generate 64 feature maps, this number is multiplied by two in each of the next three blocks. This is possible without excessively increasing the needs since, as already mentioned, each max pooling layer reduces the dimensions of the images by half.

The difference between both versions lies in the last 3 blocks of the network body, since in the VGG16 version they are made up of 3 sets of **convolutional layers + ReLU** while in VGG19 they are made up of 4.

The body of the network is made up of two fully connected dense layers of 4096 neurons each followed by the classification layer with 1000 neurons, but since in our specific case we will only have to classify for 7 different classes, we must specialize the network by reducing the last class to 7 neurons.



(a) VGG16 architecture **(b)** VGG19 architecture

Figure 3.4: VGG Architectures

ResNet

When designing the architecture of neural networks, the following phenomenon is observed: the success rate of the network tends to increase as layers and blocks are stacked in the network that increase the depth of the network, these blocks generate maps of increasingly complex characteristics until beyond a certain threshold, the error begins to increase due to the **problem of gradient fading** (annex E.1).

The name ResNet comes from residual network [Kaiming He, 2015], and they are networks whose architecture has what we call **residual blocks,** which are blocks that incorporate "jumps" between layers, so, if we consider *f(x)* as the output of the last layer of the block and *x* the input of the block, the output function of the block will correspond not to *f(x)* as in a traditional block, but to *f(x)* + x, as shown in figure 3.5. This improves the flow of the gradient allowing more layers to be stacked without increasing the error.

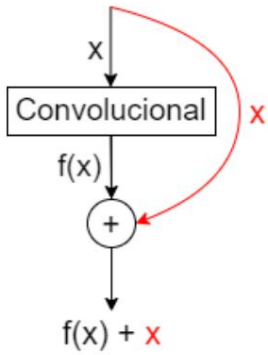


Figure 3.5: Residual block

In figure 3.6 we can see a comparison of the error obtained according to the number of layers for two different architectures, architecture 3.6a is a flat neural network without residual layers while architecture 3.6b uses residual layers, in the case From the flat neural network (3.6a) we observe that the 34-layer version has obtained a greater error than the 18-layer version, so it is no longer convenient to stack more layers, while in the case of the network with residual layers (3.6b) we can observe the opposite phenomenon: the version with 34 layers obtains a lower error than the one with 18, since the residual layers delay the appearance of the gradient descent problem as layers are stacked.

There are several versions of residual neural networks, which generally vary in depending on the number of layers and the type of residual blocks used:

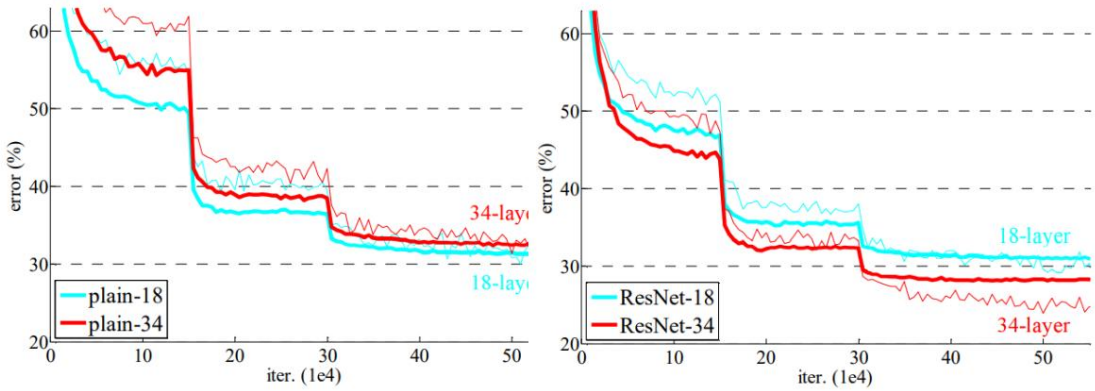
Depending on the type of blocks, we can differentiate between the **ResNetV1** versions and the more current **ResNetV2.** In both versions the same layers are used, ordered in different ways, the biggest difference is that while in ResNetV1 the last ReLU non-linearity function is applied after the addition of weights, in ResNetV1 it is done before

(diagrams in annex B.1).

This should mean that in the ResNetV2 version the gradient flow should improve.
change since there is no ReLU function that ignores negative values.

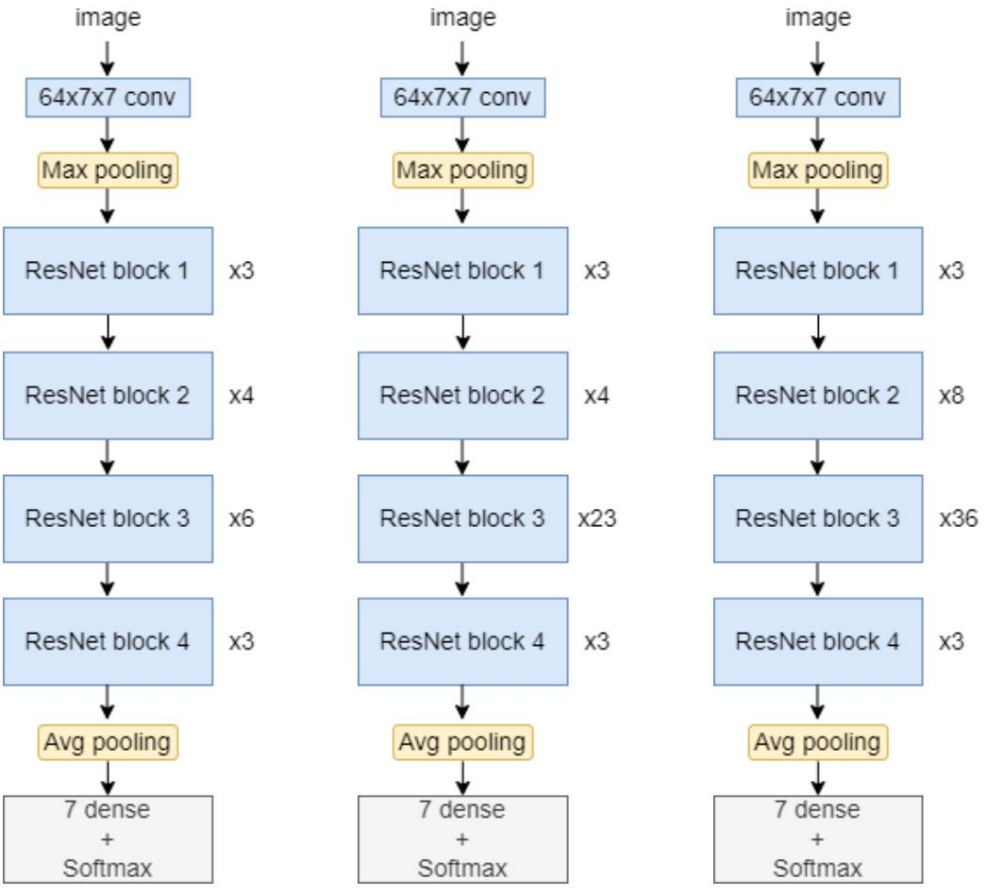
For the search, we will evaluate the V2 version of the **ResNet50, ResNet101** and **ResNet152 architectures,** these architectures are formed by the combination of the four blocks shown in Annex B.2, and will differ only in the number of said blocks that make up the architecture. , with ResNet50 being the shallowest version and ResNet152 being the deepest.

The diagram of these three architectures is shown in Figure 3.7.



(a) Planar architectures **(b)** ResNetl architecture

Figure 3.6: Error according to layers (extracted from [Kaiming He, 2015])



(a) ResNet50 block **(b)** ResNet101 block **(c)** ResNet152 block

Figure 3.7: ResNet Architectures

DenseNet

The DenseNet architecture [Gao Huang, 2016] is based on maximizing the flow of information through the network by reusing the concept of residual networks to connect each of the convolutional outputs with the rest of the outputs of its same block (see block 3.8).
For this to be possible, the feature maps that flow within the same block must have the same dimensions.

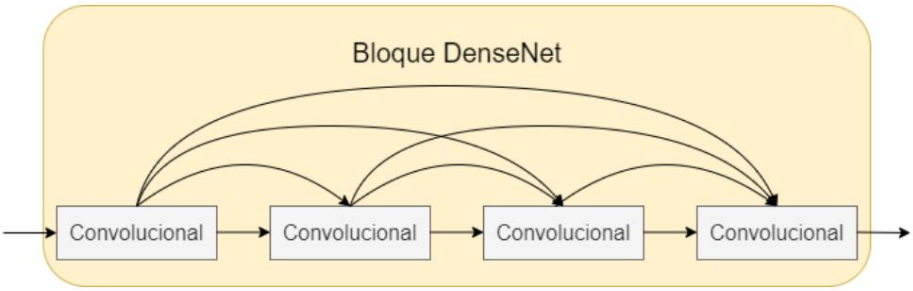


Figure 3.8: DenseNet Block

Each block is in turn formed by sub-blocks of convolutional layers, these,
As we can see in Figure 3.9, they are formed by two batch normalization sequences C.1 - ReLU - convolutional, the first convolutional layer extracts 128 feature maps with 1x1 filters, so the function of these filters is only regular he
number of feature maps that lead to the second sequence, which obtains
32 feature maps with 3x3 filters.

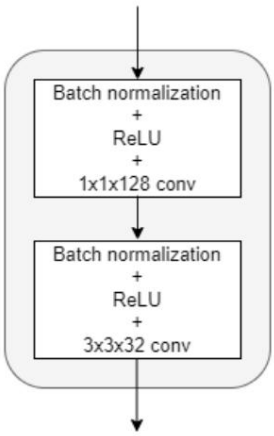


Figure 3.9: Convolutional block

In this case and unlike the ResNet architecture, at the end of each block the maps of features instead of being added, they are concatenated, so layer l will receive l inputs coming from the previous l layers, and their own feature maps will pass to the L ÿ l next layers, so a network with L layers will have $\frac{L(L+1)}{2}$ connections.

After each DenseNet block, a **transition block** is applied to the next block whose The objective is to reduce the dimensions of the feature maps by half, as we can see in Figure 3.10, this block is formed by the same *convolutional normalization - ReLU -* sequence that leads to a 2x2 average pooling layer that reduces the dimensions in half.

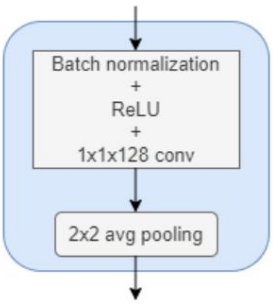


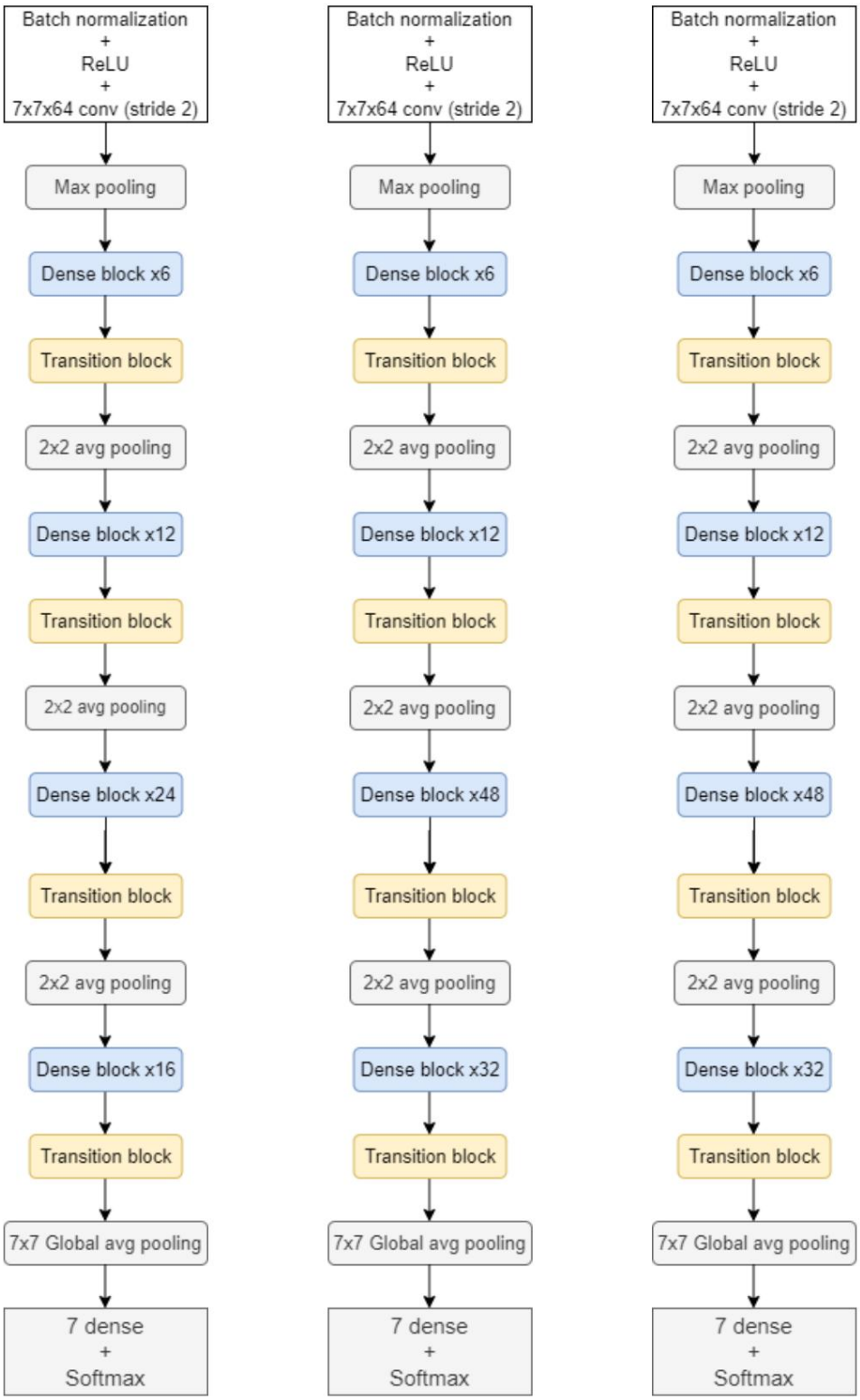
Figure 3.10: Transition block

The reuse of filters in this type of networks reduces their tendency to learn redundant filters, this implies that great performance can be obtained from using layers with few filters that will add the generated filters to the one in this case, since the classifier will make a prediction based on all the maps generated in the network (and not just the last ones), the collective knowledge of the network is considered.

Another advantage of this type of networks is the improvement of the gradient flow with respect to ResNet networks, this is because as we saw, ResNet networks add the maps of characteristics at the end of each residual block, which implies that the output of the block corresponds to f(x) + x, with f being the transformation carried out by the block, which worsens the gradient flow, while the output of DenseNet blocks corresponds to f([x0, x1...xlÿ1]) with [x0, x1...xlÿ1] being the concatenation of the previous layers of the network.

Additionally, dense connections have a regularizing effect that reduces overfitting when using small databases.

There are several versions of the DenseNet architecture, in our case we will consider three of them: **DenseNet121, DenseNet169,** and **DenseNet201,** which will vary only in the number of convolutional blocks per DenseNet block. As we can see in the figure B.1 will only differ in the number of convolutional blocks in the last two DenseNet blocks, DenseNet121 being the architecture with fewer layers and DenseNet201 the architecture with more layers.



(a) DenseNet121 **(b)** DenseNet169 **(c)** DenseNet201

Figure 3.11: DenseNet Architectures

inception

The concept behind the Inception architecture [Christian Szegedy, 2015] is based on leaving aside the increase in the depth of the networks and choosing to increase their "width", applying filters of different sizes on the same input to learn the different patterns.

As we can see in figure 3.12, the first version of Inception has modules that apply 1x1, 3x3 and 5x5 filters on the same input, in addition to a 3x3 max pooling layer that in this case will not apply dimension reduction in any way. that the outputs of these 4 layers can be concatenated. 1x1 convolutions will also be applied to the 3x3 and 5x5 convolutional layers and to the max pooling layer in order to reduce the generated feature maps.

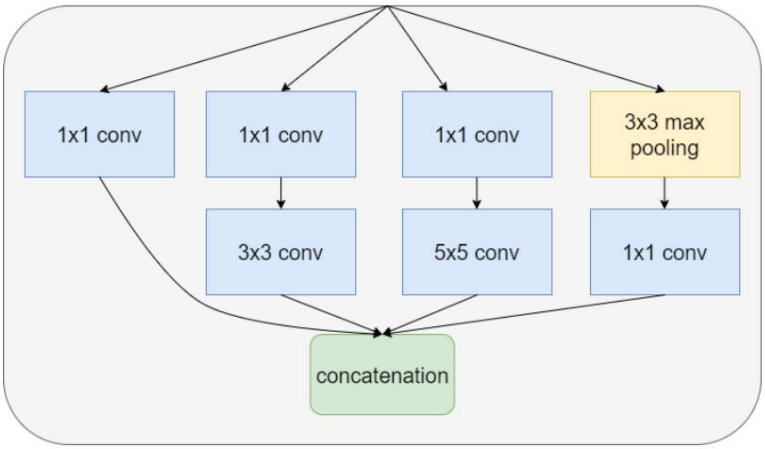


Figure 3.12: InceptionV1

These modules are stacked to form what is known as the **GoogleLeNet architecture,** which is made up of 9 of these modules that lead to a global average pooling layer. In addition, the network has auxiliary classifiers throughout the network structure in addition to the final classifier whose objective will be to optimize the network training phase. These auxiliary classifiers will be used only during network training, being eliminated for the inference phase.

The second version of Inception, known as **InceptionV2** [Christian Szegedy, 2015], combines three different types of blocks. This architecture proposes to improve computational performance by dividing larger filters into smaller filters.

In this case, the filters that apply the 3x3 and 5x5 convolutional layers are divided into smaller but equivalent filters, so the 5x5 filters are replaced by 3x3 filters, and these in turn by 3x1 and 1x3 filters, furthermore, following Inception's philosophy of not increasing the depth of the network excessively, the new added layers expand in

width and not depth (see annex B.3).

InceptionV3 [Christian Szegedy, 2015] is an improved version of the InceptionV2 architecture without making drastic changes to it. Among the improvements, the use of the RMSProp optimizer stands out, the incorporation of larger convolutional filters (7x7) and the regularization of the model through the use of batch normalization in auxiliary classifiers and smoothing of the final prediction.

The fourth version of the Inception architecture **(InceptionV4)** [Christian Szegedy, 2016] It tries to modify previous versions by providing more uniform and simple modules. Firstly, the initial sequence of layers of the network prior to the first module is modified, We can see the diagram of said sequence in annex B.4.

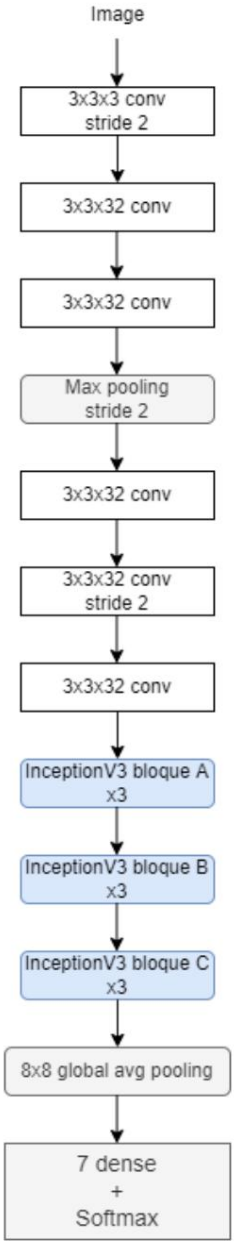
This architecture uses three modules to build the network (see annex B.5). This type of architecture is also characterized by introducing implicit reduction blocks, unlike previous versions in which this reduction was applied within the own Inception blocks. We can see the two types of reduction blocks used in Annex B.8.

The **Inception-Resnet architecture[Christian** Szegedy, 2016] tries to reuse the concepts introduced in InceptionV4 but also adding the virtues of residual layers to improve the gradient flow in the same way as in ResNet.

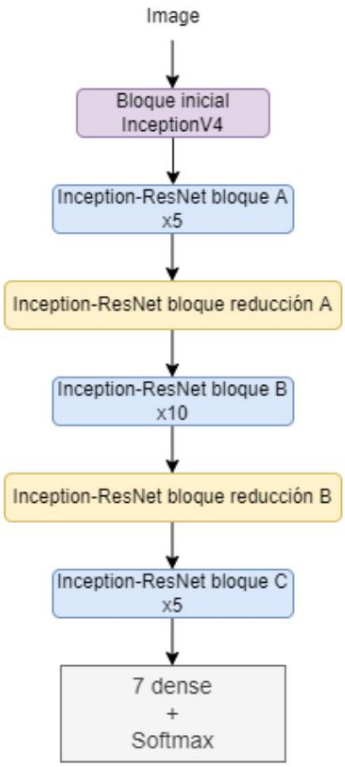
This architecture, like InceptionV4, uses three different block types (see annex B.7). In these, a final 1x1 convolutional layer is applied between the concatenation layer of the Inception block and the layer that adds the result of the block with the output. residual, this is because the feature map addition operation requires that both exits have the same depth. This need to unify the dimensions of the outputs also causes the total elimination of max pooling layers to reduce the dimensionality within the Inception blocks, the reduction blocks (shown in annex (B.8) being in this case the only ones to apply said reduction.

For the search we will consider in the search space two of the most popular versions. modern versions of the Inception architecture: **InceptionV3** and **Inception-ResNet** (3.13).

The blocks used by these architectures will be those previously defined.



(a) InceptionV3



(b) Inception-ResNet

Figure 3.13: InceptionV3 and Inception-ResNet architectures

3.2.2. Search

In this section we are going to execute the Bayesian search algorithm for the optimization of the architectures. As previously mentioned, the hyperparameters corresponding to the type of optimizer, the learning rate, the use or not of augmentation will be optimized.
of data and the batch-size, in addition to the version of the architecture, the latter being the only one hyperparameter that will vary between architectures.

VGG

For the search, in addition to the hyperparameters common to all searches,
We will consider the **VGG16** and VGG19 versions .

Optimizer	Version	L.R.	Increase data	batch size	Accuracy	Precision	Recall	F1
VGG16	Adam	0.0001	False	256	0.6447	0.6338	0.5999	0.6086
VGG16	Adam	0.0001	True	256	0.6524	0.6399	0.5940	0.6023
VGG19	Adam	0.0001	True	256	0.6319	0.6272	0.5808	0.5899
VGG19	Adam	0.0001	False	256	0.6285	0.6063	0.5886	0.5892
VGG16	Adam	0.0001	True	32	0.6701	0.5929	0.5850	0.5652

Table 3.1: Best VGG models

As we can see in table 3.1, certain hyperparameters such as the optimizer
Adam with a *learning rate* of 0.0001 and a relatively high *batch size* of 256 seem
very positively affect the model, others, such as versioning and data augmentation
They seem to affect to a lesser extent.

We can see that the model with the best *F1-score* in this first search is the VGG16 version with Adam optimizer, learning rate of 0.0001, without
data increase and with batch size of 256.

Regarding the accuracy metric, we observe that it is not strictly proportional to
the *F1-score* obtained, since as we can see in the table, the fifth model in terms of
The best *F1-score* obtained is the first in terms of accuracy. This uniqueness arises from
of the imbalance of the data, which, as we already mentioned, means that we must consider
F1-score as the most suitable metric for comparison.

After this first search, we proceed to execute cross validation on the 3 methods.
best models, along with the mean and standard deviation of each of the metrics.

Validation of the first model:

Optimizer	Version	L.R.	Batch size	data increase
VGG16	Adam	0.0001	False	256

Table 3.2: First VGG model

Accuracy validation	folder	Precision	Recall		F1
0	0.6478	0.6151	0.6016	0.5982	
1	0.6512	0.6335	0.5916	0.5977	
2	0.6088	0.6004	0.562	0.5644	
3	0.6252	0.6331	0.6196	0.6173	
4	0.6266	0.595	0.5903	0.5863	
5	0.6237	0.6421	0.578	0.5894	
6	0.6452	0.6448	0.5973	0.6075	
7	0.648	0.6405	0.5969	0.6062	
Half	**0.6345**	**0.6255**	**0.5921**	**0.5958**	
Half (%)	**63.45%**	**62.55%**	**59.21%**	**59.58%**	
Typical deviation	**0.0155**	**0.0195**	**0.0169**	**0.0162**	

Table 3.3: Cross validation first VGG model

As we can see, we obtain an average final *F1-score* of 0.5958 and an accuracy final average of 0.6345, very consistent results considering those obtained during the search.

Validation of the second model:

Optimizer	Version	LR	Batch size	data increase
VGG16	Adam	0.0001	True	256

Table 3.4: Second VGG model

Accuracy validation	folder	Precision	Recall		F1
0	0.5865	0.6015	0.5701	0.5718	
1	0.6041	0.6024	0.5795	0.5776	
2	0.5977	0.5921	0.5579	0.5632	
3	0.5762	0.5687	0.5462	0.548	
4	0.603	0.548	0.5683	0.5481	
5	0.5914	0.5604	0.5475	0.5456	
6	0.6009	0.5894	0.5711	0.5724	
7	0.5889	0.5948	0.5564	0.5622	
Half	**0.5936**	**0.5822**	**0.5621**	**0.5611**	
Half (%)	**59.36%**	**58.22%**	**56.21%**	**56.11%**	
Typical deviation	**0.0096**	**0.0204**	**0.0120**	**0.0125**	

Table 3.5: Cross validation second VGG model

As we can see, we obtain an average *F1-score* of 0.5611, a figure considerably lower than that obtained in the first best model, and an average final accuracy of 0.5936, also lower than that obtained in the first model. This is interesting considering Keep in mind that as we observed during the search (results in table 3.1), the second The best model in terms of *F1-score* obtained better accuracy, so we deduce that this good score was due more to the random goodness of the sets used for the search than to real superiority.

Validation of the third model:

Optimizer	Version	L.R.	Batch size	data increase
VGG16	Adam	0.0001	True	256

Table 3.6: Third VGG model

Accuracy validation	folder	Precision	Recall		F1
0	0.5946	0.6147	0.5777	0.5859	
1	0.5831	0.5917	0.5831	0.5757	
2	0.611	0.5958	0.5635	0.571	
3	0.5782	0.5647	0.5237	0.5273	
4	0.6091	0.5830	0.557	0.5609	
5	0.5981	0.5967	0.5667	0.5696	
6	0.6115	0.5937	0.5524	0.5606	
7	0.6073	0.5859	0.5662	0.5680	
Half	**0.5991**	**0.5908**	**0.5613**	**0.5649**	
Half (%)	**59.91%**	**59.08%**	**56.13%**	**56.49%**	
Typical deviation	**0.0130**	**0.0142**	**0.0182**	**0.0172**	

Table 3.7: Cross validation third VGG model

In this case we obtain an *F1-score* of 0.5649 and an accuracy of 0.5991. This places to this model as the second best obtained, above the second best model obtained during the search.

Best model obtained:

Version	Optimizer	LR	Batch size	data increase
VGG16	Adam	0.0001	False	256

Table 3.8: Best VGG model

In this case, the best model obtained in the search corresponds to the best model obtained during cross-validation. In table 3.9 we can see the result of running the model 10 times, training it with the complete training set (80%) and validating it with the test set (20%). For each iteration, as introduced In section 3.1, the distribution of the data set is randomized between the sets of training and validation.

Iteration	Accuracy	Precision	Recall	F1
0	0.6461	0.6102	0.6321	0.6121
1	0.6133	0.5679	0.6136	0.5783
2	0.6167	0.5779	0.6299	0.5857
3	0.6095	0.5695	0.5962	0.572
4	0.6241	0.5874	0.6321	0.5999
5	0.6275	0.6068	0.6419	0.6107
6	0.6244	0.5909	0.6352	0.5997
7	0.6174	0.5962	0.5996	0.5876
8	0.6266	0.5836	0.6171	0.5917
9	0.5972	0.5528	0.5706	0.5511
Half	**0.6203**	**0.5843**	**0.6168**	**0.5889**
Half (%)	**62.03%**	**58.43%**	**61.68%**	**58.89%**
Standard deviation	**0.0129**	**0.0179**	**0.0223**	**0.0185**

Table 3.9: Final VGG model iterations

As we can see in table 3.9, we obtain an average *F1-score* of 0.5889 and an accuracy of 0.6203, very consistent results taking into account those obtained in cross validation. The row of the results table marked in yellow corresponds to the

best model obtained, which in this case was obtained in iteration 0.

After choosing the optimal model we can analyze the performance of the classifier for each of the 7 classes, for this we will use two additional metrics, **the confusion matrix** (introduced in section 2.4.2) and the **ROC curves** (introduced in section 2.4 .2).
For this evaluation we will use the best model obtained from among the 10 iterations of the final validation (model marked in yellow in results table 3.9).

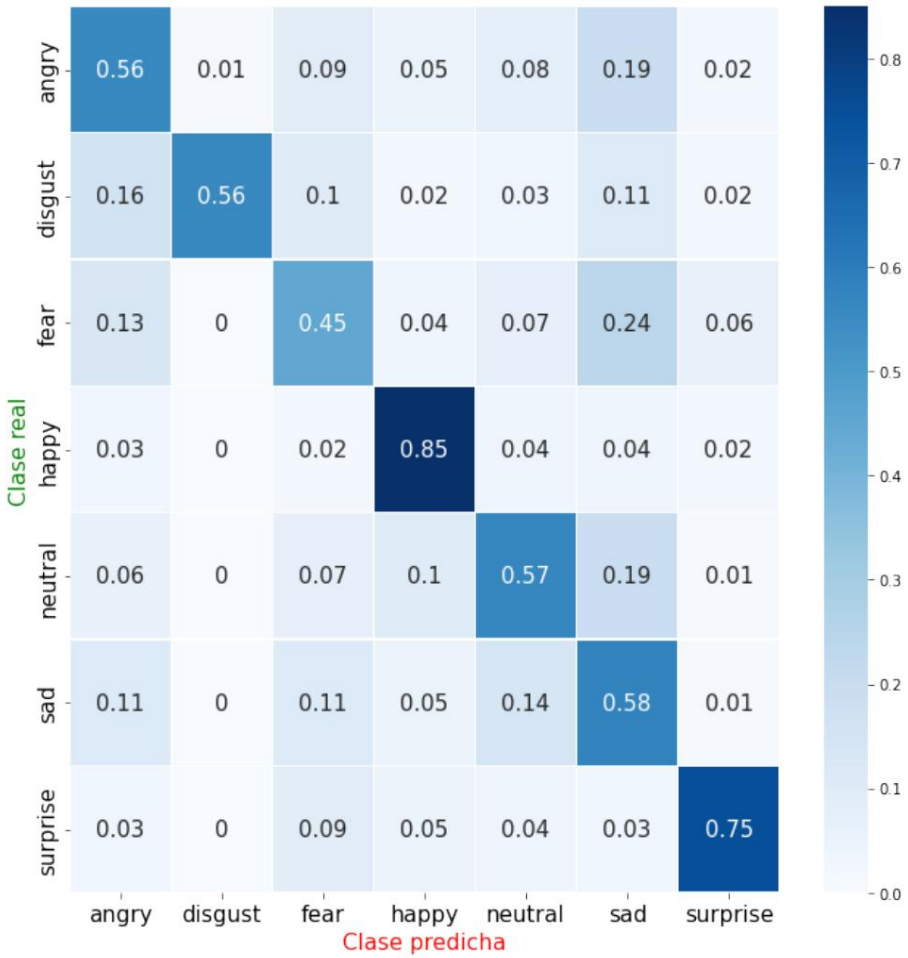


Figure 3.14: VGG confusion matrix

Figure 3.14 shows the confusion matrix obtained, the vertical axis represents the true values of the classes while the horizontal axis represents the values predicted by the classifier.

The matrix is represented in a heatmap format, so the closer the cell value is to the optimal value of 1, the darker the color of the intersecting cell will be.

As we can see, a clear diagonal is perceived in the matrix, this diagonal represents the intersection between the value of the real class and the correctly predicted one and is

indicative of the good performance of the model.

As we can see, the “happy” class obtains the best result of all (0.85), this was expected since, as shown in 2.3, it is the majority class in the database (25.05% of the total). Also noteworthy is the good result obtained for the “surpri-se” class, which despite not being a majority class (it only represents 11.15% of the total), obtains a score of 0.75. As for the rest of the classes, they all move around 0.57 except for the “fear” class, the latter obtaining a score below 0.5.

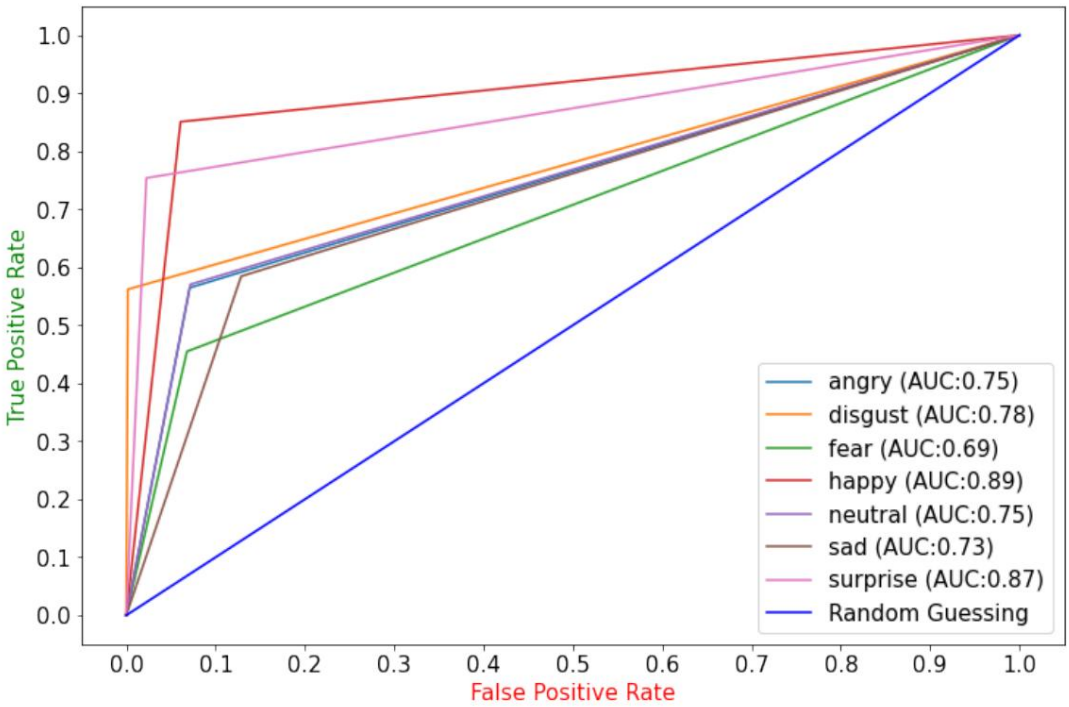


Figure 3.15: VGG ROC curves

Regarding the ROC curves (figure 3.15), we can see that the “fear” minority class obtains the lowest rate of false positives (the one closest to 0), this, together with the bad result obtained for this same class in the matrix of confusion, reflects the model's tendency not to classify images as belonging to the "fear" class.

The average AUC score of all classes for this classifier is 0.7882, which
It positions it, according to Table 2.1, in the acceptable discriminator range.

ResNet

For the search, in addition to the hyperparameters common to the other searches, we will consider three versions of the V2 version of ResNet, each characterized by the depth of the network: **ResNet50V2, ResNet101V2, ResNet152V2.**

The 5 best models obtained are shown in table 3.10.

Version	Optimizer	L.R.	Increase data	batch size	Accuracy	Precision	Recall	F1
ResNet152V2	Adam	0.0001	True	256	0.6057	0.5880	0.5357	0.5416
ResNet152V2	Adam	0.0001	False	256	0.5848	0.5619	0.5396	0.5409
ResNet50V2	Adam	0.0001	True	256	0.6010	0.5734	0.5282	0.5391
ResNet50V2	SGD	0.01	True	256	0.5776	0.5345	0.5234	0.5190
ResNet50V2	Adam	0.0001	False	256	0.5662	0.5317	0.5210	0.5163

Table 3.10: Best ResNet models

As can be seen in results table 3.10, the best model corresponds to deeper model ResNet152V2, with an *F1-score* of 0.5416 and an accuracy of 0.6057. We can also see that the difference in terms of the *F1-score* between the first and second best model is minimal, with only 0.0007 units of difference. By which in this case takes on special importance the execution of the second validation to ensure consistent results.

We can see that again the dominant hyperparameters are the optimizer *Adam* with a batch size of 256, it is also observed that the deepest architecture (ResNet152V2) is the most optimal for the problem, followed by ResNet50V2, being Resnet101V2 obtains the worst results. We also observed for the first time the use of the *SGD* optimizer in one of the best models obtained (in this case the fourth), which As we can see, it benefits from a higher learning rate (0.01).

Now we move on to execute cross-validation with 8 folders on the three best models obtained.

Validation of the first model:

Version	Optimizer	LR	Batch size	data increase
ResNet152V2	Adam	0.0001	True	256

Table 3.11: First ResNet model

Accuracy validation	folder	Precision	Recall		F1
0	0.5954	0.5779	0.5416	0.5468	
1	0.6141	0.5942	0.5628	0.5682	
2	0.6169	0.576	0.5497	0.5545	
3	0.6186	0.6143	0.5885	0.5923	
4	0.6085	0.5855	0.557	0.5594	
5	0.6037	0.5818	0.5452	0.5461	
6	0.6001	0.5762	0.5452	0.553	
7	0.5934	0.533	0.5247	0.5209	
Half	**0.6063**	**0.5799**	**0.5518**	**0.5552**	
Half (%)	**60.63%**	**57.99%**	**55.18%**	**55.52%**	
Typical deviation	**0.0097**	**0.0229**	**0.0186**	**0.0203**	

Table 3.12: Cross-validation of the first ResNet model

As we can see in table 3.12, we obtain an *F1-score* of 0.5552, and a accuracy of 0.6063, metrics somewhat higher than those obtained for this model during the search. It is also worth highlighting that the standard deviation obtained, especially in precision and *F1-score* (0.0229 and 0.0203 respectively) is somewhat high if we compare it with other models, which may be indicative of a low consistency of the model when it comes to varying the partitions.

Validation of the second model:

Version	Optimizer	LR	Batch size	data increase
ResNet152V2	Adam	0.0001	False	256

Table 3.13: Third VGG model

Accuracy validation folder	Precision	Recall		F1
0	0.5913	0.5759	0.5486	0.548
1	0.6091	0.617	0.5829	0.5879
2	0.5929	0.5822	0.5441	0.5533
3	0.5913	0.5883	0.56	0.5665
4	0.6105	0.5919	0.5777	0.5769
5	0.6087	0.5796	0.5757	0.5681
6	0.612	0.6065	0.5565	0.5698
7	0.5964	0.5989	0.5622	0.5671
Half	**0.6015**	**0.5925**	**0.5635**	**0.5672**
Half (%)	**60.15%**	**59.25%**	**56.35%**	**56.72%**
Typical deviation	**0.0093**	**0.0142**	**0.0141**	**0.0125**

Table 3.14: Cross-validation second ResNet model

In the validation of this second model (table 3.14) we can observe that it obtains an average *F1-score* of 0.5672, higher than that obtained in the first model of 0.5552, and a average accuracy of 0.6063, which is slightly lower than that obtained in the first model (0.6015). Furthermore, in this case we find lower values of standard deviation than the obtained during the validation of the first model, which indicates greater consistency of the model and makes it a better candidate as a more optimal representative of the architecture.

Validation of the third model:

Version	Optimizer	L.R.	Batch size data increase	
ResNet50V2	Adam	0.0001	True	256

Table 3.15: Third VGG model

Accuracy validation folder	Precision	Recall		F1
0	0.5868	0.5687	0.5417	0.5437
1	0.6049	0.5692	0.5575	0.5498
2	0.6124	0.5766	0.5432	0.5455
3	0.5904	0.5681	0.5338	0.5378
4	0.6261	0.606	0.5753	0.5809
5	0.6054	0.6069	0.5567	0.5694
6	0.604	0.5754	0.5454	0.5481
7	0.6154	0.5982	0.5668	0.5705
Half	**0.6057**	**0.5836**	**0.5525**	**0.5557**
Half (%)	**60.57%**	**58.36%**	**55.25%**	**55.57%**
Typical deviation	**0.0128**	**0.0171**	**0.0140**	**0.0156**

Table 3.16: Cross-validation third ResNet model

As we observed in 3.16, in this case we obtain an *F1-score* of 0.5557 and an average accuracy of 0.6057, values very similar to those obtained in the first validation.
with the difference that in this case, the standard deviations obtained and consequently the consistency of the model are somewhat more reasonable, which makes this the second best model obtained.

As we have been able to observe during the validation, in this case the superiority of first best model with respect to the next two obtained in the search was due, among other things, to the goodness of the partitions used as training sets and the validation of the models, this explains the high standard deviation obtained for this model.

For this reason, the second model obtained during the Bayesian search is considered the most optimal representative for the specific architecture and problem.

Best model obtained:

Version	Optimizer	LR	Batch size	data increase
ResNet152V2	Adam	0.0001	False	256

Table 3.17: Best ResNet model

Iteration	Accuracy	Precision	Recall	F1
0	0.6009	0.5617	0.5742	0.5592
1	0.6092	0.5731	0.6095	0.5804
2	0.6134	0.5723	0.6135	0.5799
3	0.6028	0.5635	0.6153	0.5724
4	0.5978	0.5783	0.5983	0.5747
5	0.5979	0.5888	0.6036	0.5884
6	0.5861	0.5784	0.5939	0.5726
7	0.6087	0.568	0.588	0.5684
8	0.6014	0.5522	0.6013	0.5622
9	0.5861	0.559	0.5692	0.5532
Half	**0.6004**	**0.5695**	**0.5967**	**0.5711**
Half (%)	**60.04%**	**56.95%**	**59.67%**	**57.11%**
Standard deviation	**0.0091**	**0.0108**	**0.0157**	**0.0107**

Table 3.18: Final Resnet model iterations

After training the model using the full training set, we can Note that we obtain an average *F1-score* of 0.5711, somewhat higher than that obtained during

compared to the previous phases of the study, and an accuracy of 0.6004, with reasonable standard deviations and consistency.

We now move on to analyze the confusion matrix and ROC-AUC curves obtained for each of the classes for the best model after the final 10 iterations shown in 3.18 (model highlighted in yellow).

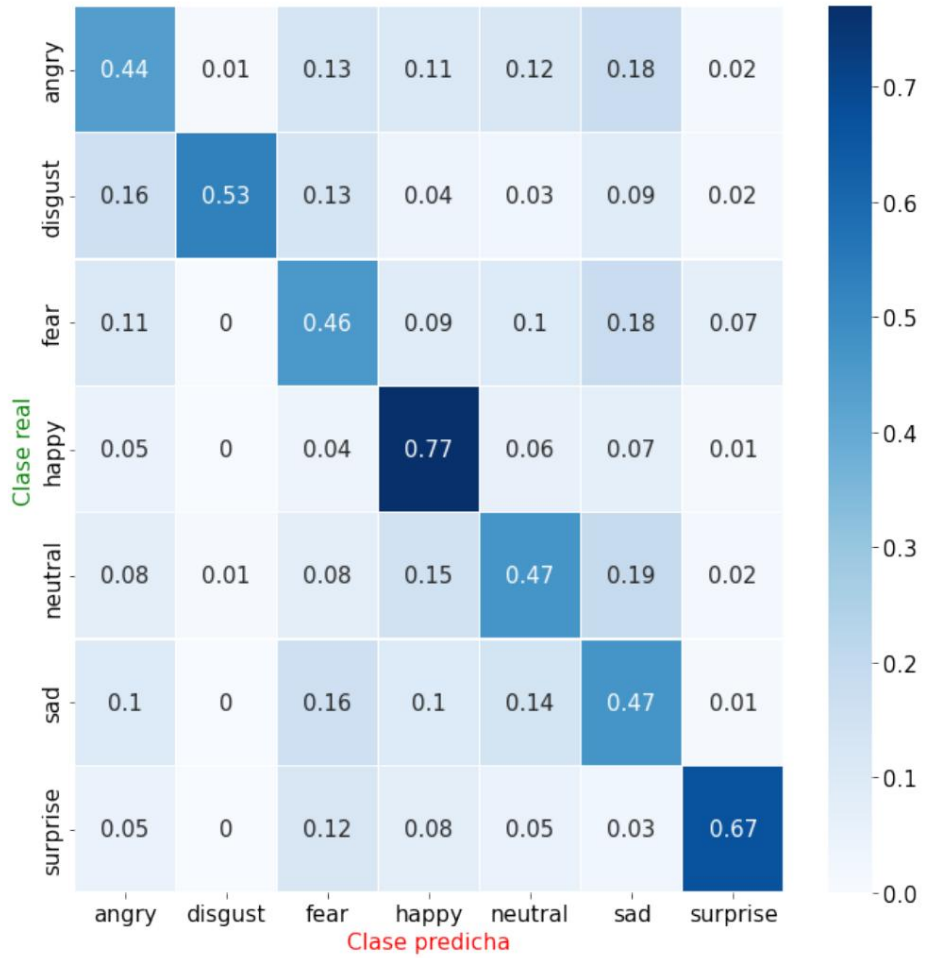


Figure 3.16: ResNet confusion matrix

Figure 3.16 shows the confusion matrix obtained. In this case, as expected, worse results are observed than those obtained in VGG. We observe that again the best value (0.77) corresponds to the “happy” class and the worst (0.46) to the “fear” class.

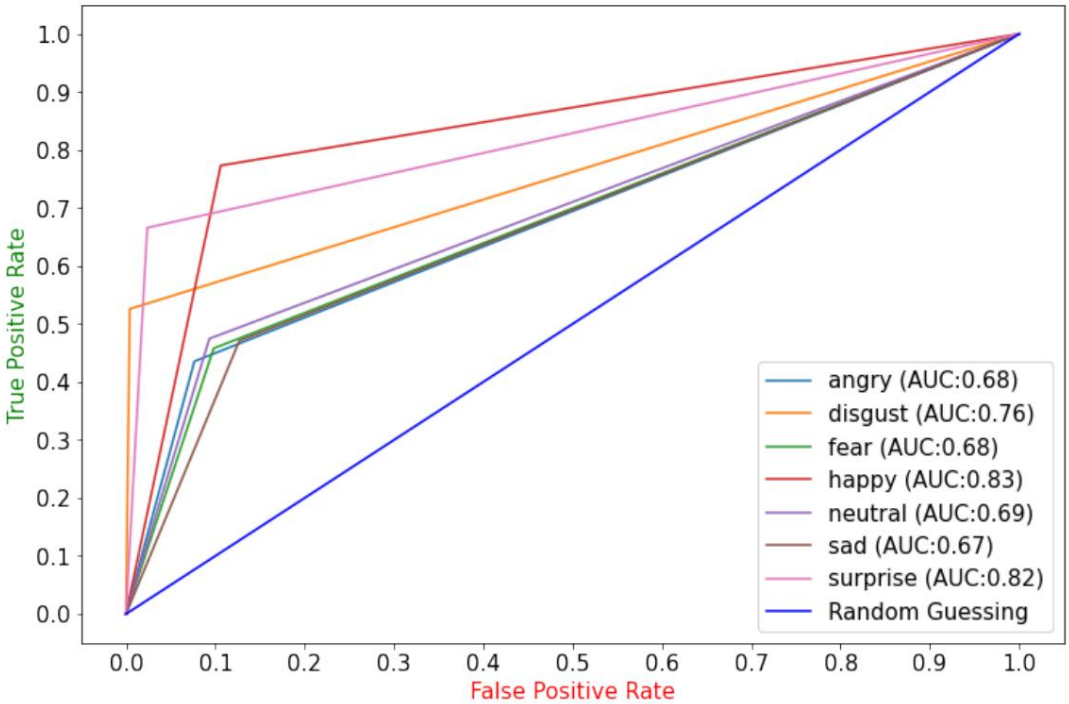


Figure 3.17: ResNet ROC curves

Figure 3.17 shows the ROC curves obtained for each class. The model obtains an average AUC score of 0.7445, which despite being lower than that obtained for the
previous model VGG, according to table 2.1 can be considered as an acceptable discriminator.

DenseNet

In this section we will apply optimization search to the DenseNet architecture. In
In this case we will consider the **DenseNet121, DenseNet169** and DenseNet201 versions .

Version	Optimizer	L.R.	Increase data	batch size	Accuracy	Precision	Recall	F1
DenseNet201	Adam	0.0001	False	256	0.6447	0.6338	0.5999	0.5636
DenseNet201	Adam	0.0001	True	256	0.6524	0.6399	0.5391	0.5422
DenseNet121	Adam	0.0001	True	256	0.0.6319	0.6272	0.5808	0.5380
DenseNet121	Adam	0.0001	False	256	0.6285	0.6063	0.5886	0.5371
DenseNet121	Adam	0.0001	True	32	0.6701	0.5929	0.5850	0.5368

Table 3.19: Best DenseNet models

As we can see in table 3.19, the two best models correspond to the
version with and without data augmentation of the deeper architecture DenseNet201,
After this are again the versions with and without version data increase
of lower depth DenseNet121, as for the rest of the hyperparameters, it is observed
new the superiority of the *Adam* optimizer with learning rate of 0.0001.

On this occasion, the first model has obtained an *F1-score* of 0.5636, much higher
to that obtained by the second model (0.5422), and an accuracy of 0.6447, slightly lower
to that of the second model.

After this first search we moved on to perform the cross-validation on the
three best models to obtain a more consistent comparison.

Validation of the first model:

Version	Optimizer	L.R.	Batch size data increase	
DenseNet201	Adam	0.0001	False	256

Table 3.20: First DenseNet model

Accuracy validation folder	Precision	Recall		F1
0	0.6258	0.6235	0.5947	0.6027
1	0.645	0.6095	0.6047	0.6006
2	0.6300	0.598	0.578	0.5785
3	0.5837	0.5848	0.5458	0.5508
4	0.6422	0.6118	0.6099	0.6045
5	0.638	0.6368	0.6036	0.6088
6	0.6388	0.6183	0.5983	0.6019
7	0.6268	0.5953	0.5923	0.5854
Half	**0.6287**	**0.6098**	**0.5909**	**0.5917**
Half (%)	**62.87%**	**60.98%**	**59.09%**	**59.17%**
Typical deviation	**0.0195**	**0.0168**	**0.0206**	**0.0194**

Table 3.21: Cross-validation of the first DenseNet model

As we can see in table 3.21, this first model obtains an average *F1-score* of 0.5917 and an average accuracy of 0.6287, some inconsistency can be observed between training with different folders, especially with the third of them, which gives lead to a somewhat high standard deviation but within reason.

Validation of the second model:

Version	Optimizer	LR	Batch size	data increase
DenseNet210	Adam	0.0001	True	256

Table 3.22: Second DenseNet model

Accuracy validation folder	Precision	Recall		F1
0	0.6361	0.5902	0.5884	0.5805
1	0.6436	0.6316	0.6072	0.6088
2	0.6088	0.5836	0.5573	0.5613
3	0.6367	0.6245	0.6006	0.6019
4	0.64	0.6236	0.606	0.6080
5	0.6361	0.5902	0.5884	0.5805
6	0.6157	0.5954	0.5343	0.5483
7	0.6544	0.6403	0.5993	0.6046
Half	**0.6348**	**0.6130**	**0.5856**	**0.5884**
Half (%)	**63.48%**	**61.30%**	**58.56%**	**58.84%**
Typical deviation	**0.0152**	**0.0208**	**0.0261**	**0.0229**

Table 3.23: Cross-validation second DenseNet model

In this case, as we can see in table 3.23, we obtain an average *F1-score* of 0.5884, somewhat lower than that obtained in the first model. However, despite this, Again, the accuracy is higher in this second model (0.6348 compared to 0.6287). Also we observe an increase in the standard deviations of precision, recall and *F1-score* due to largely due to its poor performance with the second and sixth folder, so this model It is somewhat more inconsistent than the first.

Validation of the third model:

Version	Optimizer	L.R.	Batch size data increase	
DenseNet121	Adam	0.0001	True	256

Table 3.24: Third DenseNet model

Accuracy validation folder	Precision	Recall		F1
0	0.6074	0.5794	0.5519	0.5502
1	0.6294	0.6231	0.5712	0.5793
2	0.6303	0.585	0.5521	0.553
3	0.6177	0.6007	0.5611	0.5681
4	0.6219	0.6136	0.5664	0.5761
5	0.6154	0.6151	0.5648	0.5778
6	0.6377	0.633	0.5858	0.5947
7	0.604	0.5523	0.5519	0.5451
Half	**0.6205**	**0.6003**	**0.5631**	**0.5680**
Half (%)	**62.05%**	**60.03%**	**56.31%**	**56.80%**
Typical deviation	**0.0117**	**0.0266**	**0.0118**	**0.0172**

Table 3.25: Cross-validation third DenseNet model

As we see in table 3.25, in this third model, although somewhat more consistent, tes, we obtain the worst scores *(F1-score* of 0.5680 and accuracy of 0.6205).

In this case, the best model obtained in the search is consistent with the best one obtained after cross-validation, this being the version with the greatest depth. Despite this, It should be noted that these three models have obtained a higher *F1-score* in the cross-validation than in the search, so we can deduce that in this case a partition of the data set that did not benefit this particular architecture.

Best model obtained:

Version	Optimizer	LR	Batch size	data increase
DenseNet201	Adam	0.0001	False	256

Table 3.26: Best DenseNet model

Iteration	Accuracy	Precision	Recall	F1
0	0.6452	0.6252	0.6454	0.6264
1	0.641	0.6032	0.6383	0.6098
2	0.6489	0.619	0.6519	0.6241
3	0.6421	0.6142	0.644	0.6185
4	0.5879	0.5543	0.6006	0.5627
5	0.6562	0.6325	0.6578	0.6358
6	0.6441	0.6215	0.647	0.6213
7	0.6477	0.6082	0.6224	0.6026
8	0.6446	0.611	0.6323	0.6142
9	0.6321	0.6007	0.6267	0.6023
Half	**0.6390**	**0.6090**	**0.6366**	**0.6118**
Half (%)	**63.90%**	**60.90%**	**63.66%**	**61.18%**
Standard deviation	**0.0190**	**0.0216**	**0.0168**	**0.0202**

Table 3.27: Final DenseNet model iterations

Table 3.19 shows the result of the 10 iterations for validation including the test set. As we can see, we have obtained an average *F1-score* of 0.6118 with an accuracy of 0.6390, results very consistent with those obtained during validation crossed and the most promising taking into account the rest of the architectures evaluated.

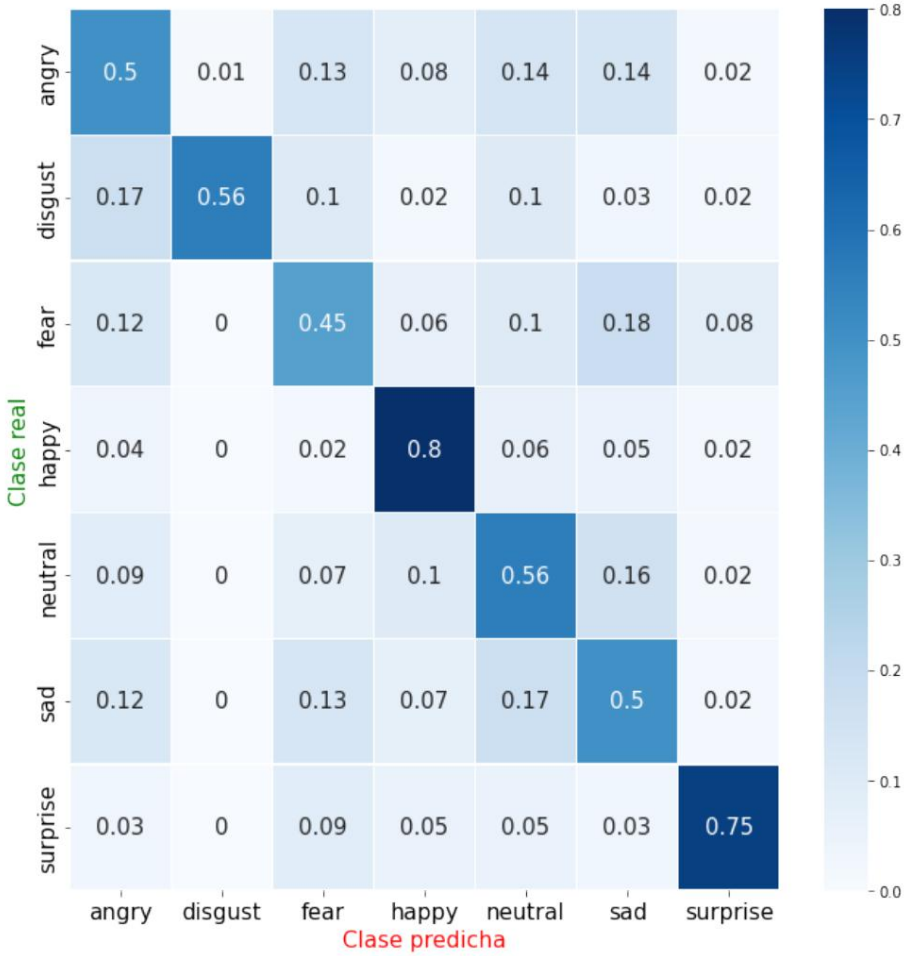


Figure 3.18: DenseNet confusion matrix

Figure 3.18 shows the confusion matrix obtained for the best model after the final 10 iterations. If we look at the diagonal, we can highlight a maximum score of 0.8 for the “happy” class and 0.75 for the “surprise” class. The rest of the scores oscillate around a score of 0.5, with the lowest again being the one obtained for the “fear” class with 0.45.

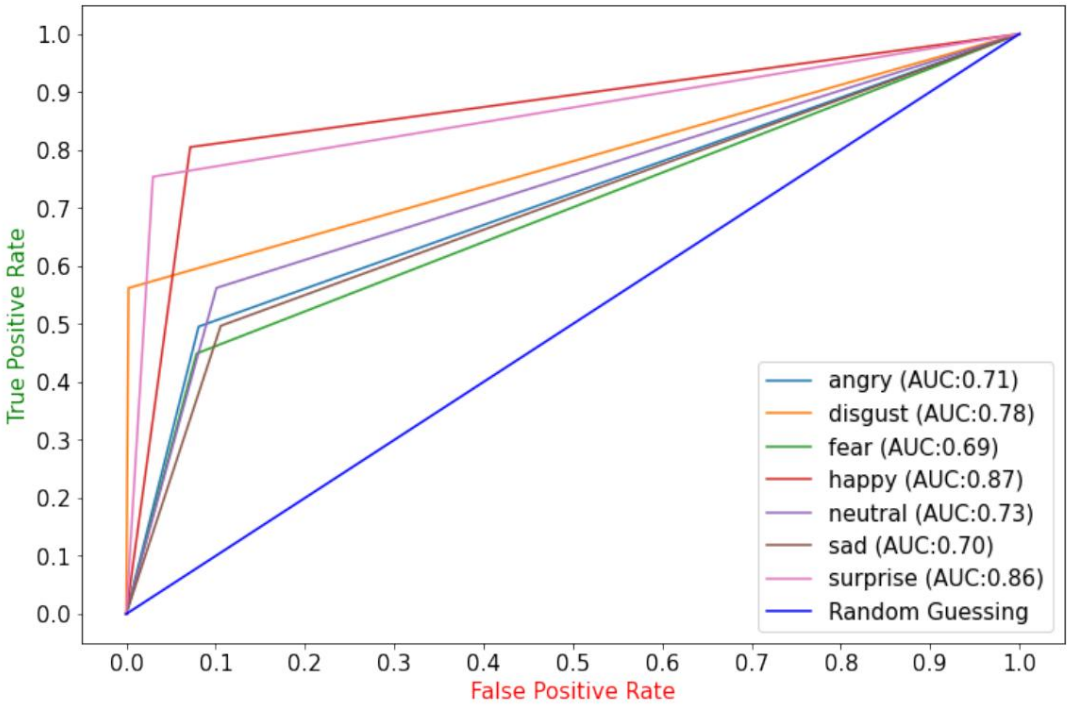


Figure 3.19: DenseNet ROC curves

Figure 3.19 shows the ROC curves obtained for each class. Finally, it obtains an average AUC score of 0.7716, which is again in the range of “acceptable discriminator” according to table 2.1.

inception

To optimize the Inception architecture we will assess, in addition to the hyperparameters, common parameters, the **InceptionV3** and **InceptionResNetV2** versions of the architecture.

Version	Optimizer	L.R.	Increase data	batch size	Accuracy	Precision	Recall	F1
InceptionV3	Adam	0.0001	True	256	0.6121	0.6022	0.5661	0.5707
IncepResnetV2	Adam	0.0001	True	256	0.6029	0.5561	0.5734	0.5521
IncepResnetV2	SGD	0.01	True	32	0.6019	0.5501	0.5699	0.5417
IncepResnetV2	Adam	0.0001	False	32	0.5885	0.5353	0.5401	0.5295
InceptionV3	Adam	0.0001	True	32	0.5701	0.5229	0.5309	0.5282

Table 3.28: Best Inception models

As we can see in table 3.28, we obtain that the first best model corresponds to the InceptionV3 version with Adam optimizer, learning rate of 0.0001, increase of data and batch size of 256, obtaining an *F1-score* of 0.5707 and an accuracy of 0.6121, followed by the same configuration but in this case for the InceptionV3 version.

As peculiarities, in this case we can once again find the favored SGD optimizer with a relatively high learning rate of 0.01 as the third best model. Also We can observe that the increase in data seems to favor this model to a greater extent if we take into account the rest of the architectures analyzed, since 4 of the 5 best models have it. We can also find that a small batch size seems favor the model in specific cases.

After this first search we proceed to cross-validate the three best models obtained to ensure the consistency of the results.

Validation of the first model:

Version	Optimizer	LR	Batch size	data increase
InceptionV3	Adam	0.0001	True	256

Table 3.29: First Inception model

Accuracy validation folder	Precision	Recall		F1
0	0.6113	0.5973	0.5573	0.5665
1	0.6211	0.6054	0.5955	0.5905
2	0.6096	0.5746	0.5533	0.555
3	0.6102	0.6061	0.5715	0.5785
4	0.5578	0.5108	0.4961	0.4937
5	0.6076	0.5919	0.5539	0.5647
6	0.6185	0.6011	0.5747	0.5811
7	0.6243	0.5943	0.5733	0.5739
Half	**0.6076**	**0.5852**	**0.5595**	**0.5630**
Half (%)	**60.76%**	**58.52%**	**55.95%**	**56.30%**
Typical deviation	**0.0210**	**0.0317**	**0.0292**	**0.0301**

Table 3.30: Cross-validation of the first Inception model

In table 3.30 we show the validation data of the first best model obtained. This obtains an *F1-score* of 0.5630 and an accuracy of 0.6076, very consistent with those obtained during the search, however, we can observe a considerable standard deviation that in two of the four metrics evaluated exceeds 0.03. This is because, as we can see, the model is very inconsistent in function of the data partition, in particular it highlights how harmed the model is using the fourth partition, with which it obtains an *F1-score* of 0.4937, very far from the mean of 0.56. Therefore we must treat this inconsistency as a negative point of the model to take into account.

Validation of the second model:

Version	Optimizer	LR	Batch size	data increase
InceptionResnetV2	Adam	0.0001	True	256

Table 3.31: Second Inception model

Accuracy validation folder	Precision	Recall		F1
0	0.6024	0.599	0.581	0.5826
1	0.6333	0.6218	0.6097	0.6107
2	0.6255	0.5985	0.5793	0.5759
3	0.6333	0.6338	0.608	0.6123
4	0.6261	0.6099	0.5864	0.5907
5	0.6293	0.5985	0.5855	0.5829
6	0.6076	0.5855	0.5679	0.5686
7	0.6157	0.5773	0.5652	0.5591
Half	**0.6217**	**0.6030**	**0.5854**	**0.5853**
Half (%)	**62.17%**	**60.30%**	**58.54%**	**58.53%**
Typical deviation	**0.0118**	**0.0184**	**0.0164**	**0.0188**

Table 3.32: Cross-validation second Inception model

In this second validation, whose results are shown in table (3.34), the model obtains an *F1-score* of 0.5853 and an accuracy of 0.6217, much higher scores to those obtained in the first model with, in addition, a much more reasonable standard deviation below 0.02 in all metrics, which makes it a much more consistent and a better candidate as an optimal representative of the architecture applied to recognition of emotions.

Validation of the third model:

Version	LR Optimizer	Batch size	data increase	
InceptionResNetV2	SGD	0.01	True	32

Table 3.33: Third Inception model

Accuracy validation folder	Precision	Recall		F1
0	0.6055	0.5441	0.5327	0.5147
1	0.6119	0.5344	0.5322	0.5069
2	0.6297	0.5602	0.5514	0.5313
3	0.6358	0.5603	0.5639	0.5378
4	0.623	0.5433	0.5405	0.5179
5	0.1385	0.0199	0.1416	0.0342
6	0.1385	0.0196	0.1378	0.0337
7	0.1382	0.0196	0.1403	0.0335
Half	**0.4401**	**0.3502**	**0.3925**	**0.3388**
Half (%)	**44.01%**	**35.02%**	**39.25%**	**33.88%**
Typical deviation	**0.2500**	**0.2738**	**0.2095**	**0.2527**

Table 3.34: Cross-validation third InceptionV3 model

In this third validation, whose results are shown in table 3.33, we obtain an *F1-score* of 0.3388 and an accuracy of 0.4401, this being the worst of the 3 models with a difference. If we look at the results folder by folder, we can see that the model obtains relatively good results and close to those obtained in the search in folders 0 to 4. However, these results worsen drastically in folders 5 to 7, in which the accuracy of 0.13 suggests that the model is limited to randomly classifying the images. This highlights the low consistency provided by the pattern formed by the SGD optimizer with learning rate of 0.01 for this particular model and data set.

Best model obtained:

Version	Optimizer	LR	Batch size	data increase
InceptionResnetV2	Adam	0.0001	True	256

Table 3.35: Best Inception model

On this occasion, again the best model obtained during the search was not really the most suitable. This is due, as already mentioned, to the inconsistent nature of holdout validation with a single folder, which in this specific case was detrimental.
to the true best model (second best in the search), which got better and
more consistent metrics during the cross-validation process.

Iteration	Accuracy	Precision	Recall	F1
0	0.6422	0.6124	0.6326	0.6126
1	0.6364	0.6111	0.6173	0.6058
2	0.6468	0.6203	0.6267	0.6157
3	0.6491	0.6212	0.6538	0.6238
4	0.6328	0.6219	0.646	0.6207
5	0.6211	0.606	0.6409	0.6119
6	0.633	0.6075	0.6316	0.6064
7	0.5972	0.5716	0.5942	0.5694
8	0.6369	0.6034	0.6288	0.6026
9	0.6368	0.6169	0.6246	0.6141
Half	**0.6332**	**0.6092**	**0.6296**	**0.6083**
Half (%)	**63.32%**	**60.92%**	**62.96%**	**60.83%**
Standard deviation	**0.0149**	**0.0148**	**0.0164**	**0.0152**

Table 3.36: Final Inception model iterations

Table 3.36 shows the 10 iterations of model training using
the final training and test sets, in this case we observe a great improvement with
regarding the results obtained during the search and cross-validation, obtaining
This has a mean *F1-score* and accuracy of 0.6083 and 0.6332 respectively. This improvement can
due to the benefit that this specific model obtains from the incorporation of a greater

number of images to the training set.

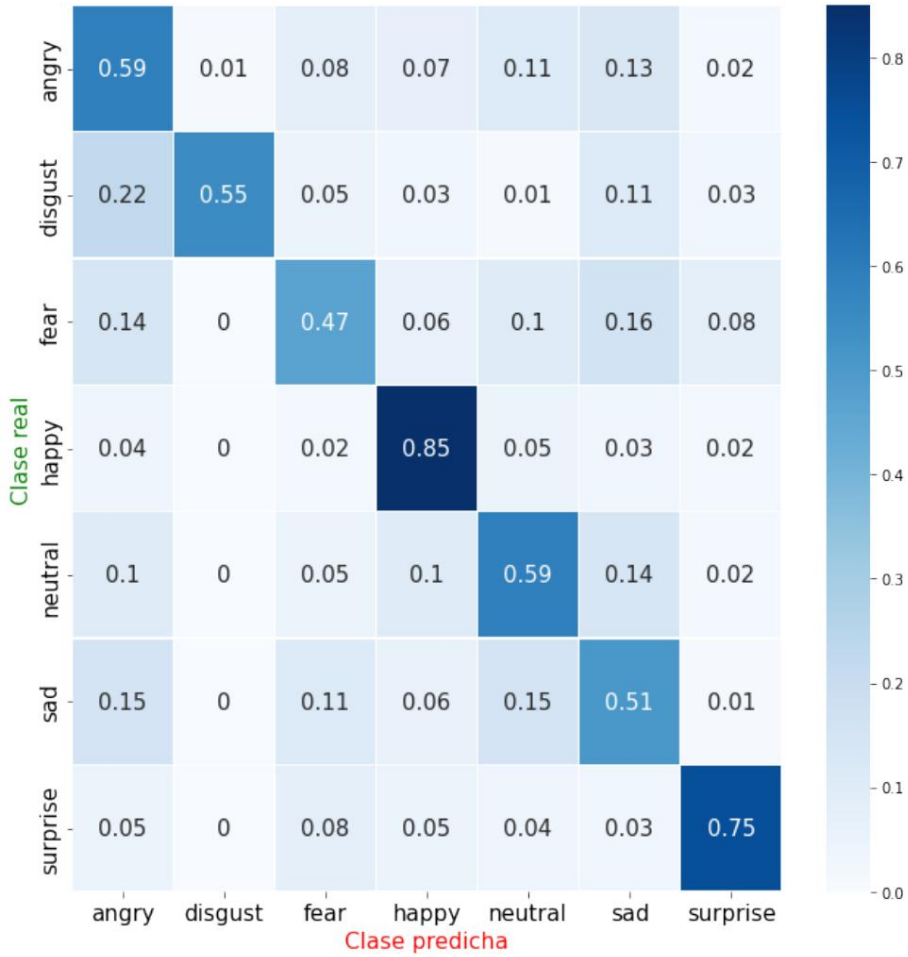


Figure 3.20: Inception confusion matrix

Figure 3.20 shows the confusion matrix obtained for the best model obtained after the 10 final iterations of the architecture (row highlighted in yellow in table 3.36). It is worth highlighting the score of 0.47 obtained for the “fear” class, which, although still low, is the best value obtained so far for this class. The rest of the values are very similar to those obtained for DenseNet.

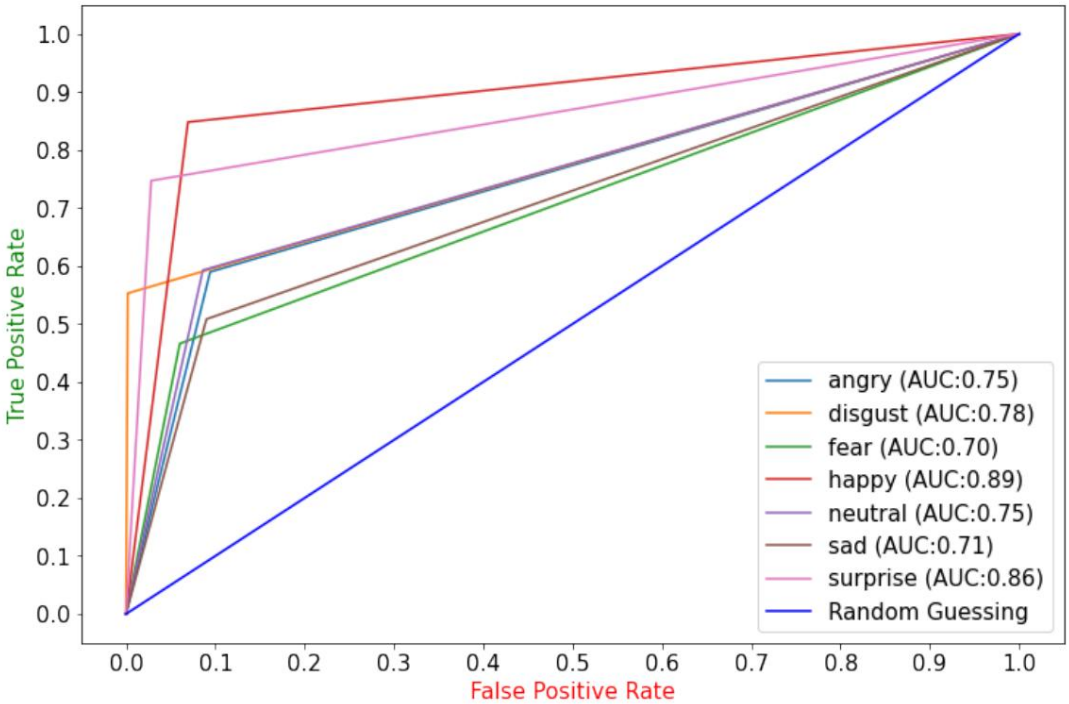


Figure 3.21: Inception ROC curves

Regarding the ROC curves (figure 3.21), we did not find any singularity to highlight, the average AUC score obtained being 0.7911, which classifies it according to the ranges shown in table 2.1 as an acceptable discriminator.

3.2.3. Comparison

Version	Optimizer	L.R.	Increase data	batch size	Accuracy	Precision	Recall	F1
DenseNet201	Adam	0.0001	False	256	0.6390	0.6090	0.6366	0.6118
IncepResnetV2	Adam	0.0001	True	256	0.6332	0.6092	0.6296	0.6083
VGG16	Adam	0.0001	False	256	0.6202	0.5843	0.6168	0.5889
ResNet152V2	Adam	0.0001	False	256	0.6004	0.5695	0.5967	0.5711

Table 3.37: Best final models

In table 3.39 we show the best models obtained ordered from highest to lowest. score obtained for the *F1-score* metric in its final version. The scores correspond to the average of those obtained for the final model trained with 80% of the base of Complete data in 10 iterations. As we can see, the best model corresponds to the DenseNet201 architecture, with an *F1-score* of 0.6118, followed by InceptionResNetV2, which obtained a slightly worse *F1-score* (0.6083). The third best model is It is about the VGG16 architecture, which despite its simplicity and lack of depth compared to the rest of the architectures, obtained an F1-score of 0.5889. Finally The worst model is the optimization of the ResNet architecture, specifically Res-Net152V2, which obtained a score of 0.5711.

As for the rest of the metrics, there is no notable imbalance between the precision and recall, these metrics being, as expected, very proportional to *F1-score* obtained, in terms of accuracy, we can observe that it decreases proportionally to the *F1-score* obtained by the models.

All the analyzed architectures seem to benefit from the pattern formed by the Adam optimizer with learning rate of 0.0001 and batch-size of 256. Regarding the increase in data, appears to have only benefited the Inception model.

Figures 3.22, 3.23, 3.24, 3.25 show the learning curves of the best iteration of each of the models.

These curves are represented in the form of graphs in which the horizontal axis represents the training epochs, while the vertical axis represents the *F1-score.* obtained. In these graphs, the orange line shows how the *F1-score* increases obtained on the training set while the blue one shows the one obtained on the validation set.

At first glance we can see that each of the networks reached its optimal value after running a different number of epochs. Thus, we can observe that while the network VGG reached its optimal value after a relatively small number of 12 epochs (3.22), the red Inception caught up after running 28 (3.25). As for Resnet and DenseNet, they executed, respectively, 18 and 20 epochs (figures 3.23 and 3.24).

As we can see, they all share a common pattern: both curves count

with an increasing trend, but while the learning curve on the training data (blue line) grows until reaching scores very close to 1 (perfect score), the validation curve presents an increasingly less pronounced growth. This is because while the learning curve on the training set shows the score obtained on the same data with which the network was trained, the curve on the validation set does so on data that was never used for the training. The further apart these two curves are, the greater the tendency the model will have to overfit the training data, and the less ability it will have to obtain good results when used for inference from new data.

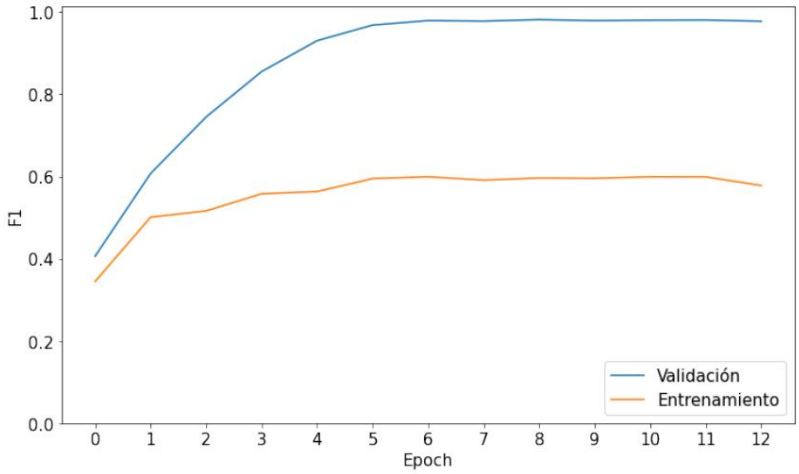


Figure 3.22: F1-score VGG curves

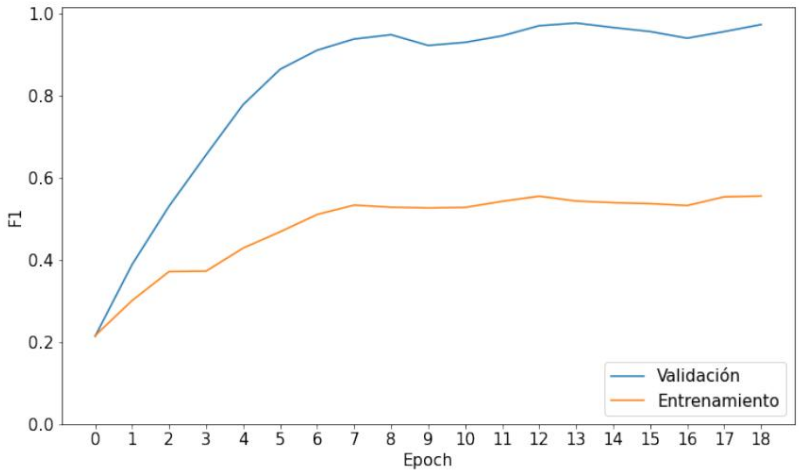


Figure 3.23: ResNet F1-score curves

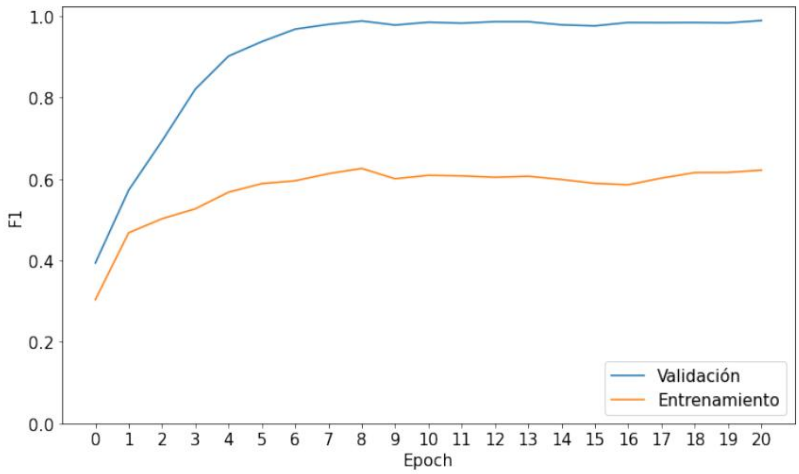


Figure 3.24: DenseNet F1-score curves

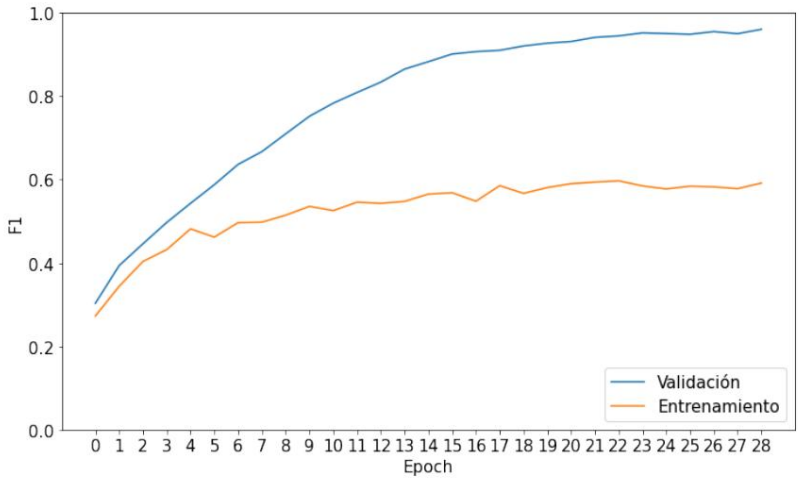


Figure 3.25: F1-score Inception curves

Conclusions

After carrying out the study we can extract the following conclusions from it:

ÿ The architectures that obtained the best results are relatively deep architectures with residual layers, with DenseNet201 being the architecture that obtained the best results.

ÿ The models appear to benefit from the reduction of a relatively small learning rate of 0.0001. In addition to the use of the Adam optimizer compared to the SGD, which as

We commented in D.2, it applies an adaptive learning rate to training, so as a possible improvement it would be interesting to study the use of even lower learning rates with different learning rate reduction techniques.

- The applied data augmentation does not seem to increase the *F1-score* obtained in any of the cases, with the exception of the Inception-ResNetV2 architecture.
- All models have benefited from the largest batch-size evaluated during the search (256), which leads us to deduce that the models could benefit from even larger batch-sizes.
- Observing the learning curves, all models show overfitting, so certain techniques for reducing it could benefit these models.

3.3. Own Architecture

As we have observed, the pre-trained models analyzed are capable of returning good results in emotion recognition with FER2013, which is why when designing an architecture that improves on the previous ones, the most efficient thing is to create optimized versions of the same for the specific problem.

3.3.1. VGG16 modification

For this first iteration, the modification of the **VGG16 network will be experimented with,** this is because, despite not being the best model obtained as a conclusion of the study, it did obtain good results considering its simplicity compared to the rest of the models. . This simplicity is what will allow us to make small modifications to the body of the network to try to improve its results.

The idea is to try to combine this architecture with one of the concepts that support the **DenseNet architecture,** since it was this that obtained the best results after the study.

As we already saw in 2.2, during the training of a deep learning model for computer vision, the first blocks with convolutional layers of the model tend to learn filters for the detection of simple shapes such as straight lines and curves. As the depth of the network increases, the layers tend to detect increasingly complex attributes, in our case these would be shapes that correspond to more recognizable facial features such as eyes or nose. These features, despite not representing the entire image, do begin to shed some information about the emotion expressed in the image.

The idea, therefore, would be to not only take into account the filters generated by the last block of the network, but on the contrary, also take advantage of those generated by some of the previous blocks.

This reuse of feature maps can be achieved, as we already introduced, by adding, as was done in DenseNet, with residual layers that connect some of the last blocks of the network directly to the final layer of the body of the network.

Figure 3.26 shows the proposed architecture. As we can see, the left part is the VGG16 model as it was trained during the study, while on the right part we can see the proposed modification: this consists of three residual connections that connect the output of the three blocks before the final block. Each of these connections is made up of the following sequence of layers:

- **Convolutional layers** with 1x1 filters to, as was done in DenseNet, add one more level of depth and control the number of parameters. This is convenient to avoid generating models that are too heavy without it being strictly necessary for their performance, since all the weights generated by the residual layers will be concatenated at the end.

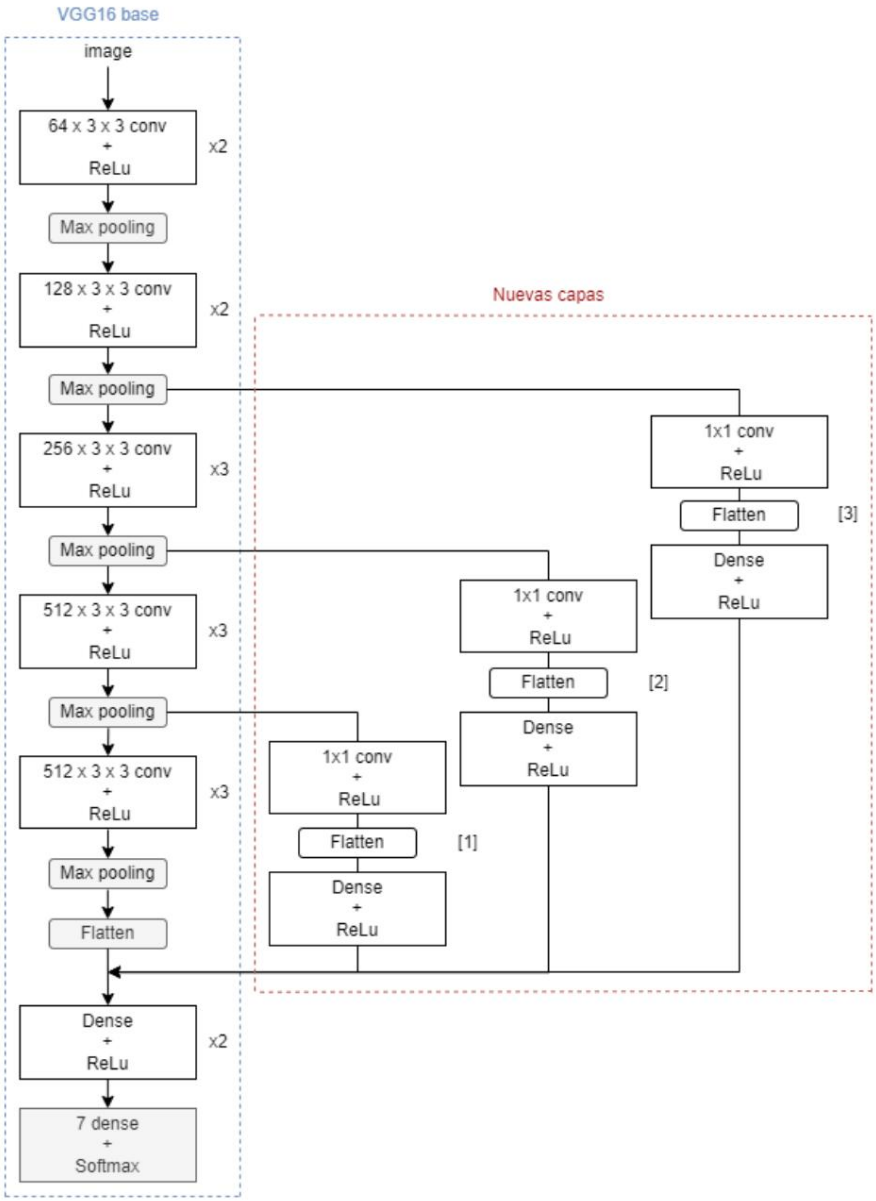


Figure 3.26: VGG16 architecture with residual layers

ÿ **Flatten layers** to reduce dimensionality and subsequently make it possible to concatenate the weights to the last section of the network.

ÿ **Dense layers** to, together with the convolutional layers, provide depth and ca-learning capacity to residual connections.

Search

For the optimization of the proposed architecture, the following hyperparameters will be taken into account:

- **Number of residual blocks (BR)** to include in the architecture. Considering the figure 3.26, the inclusion of residual blocks 1, blocks 1 and 2, blocks 1, 2 and 3 or none of them.
- **Number of filters (NF)** generated by the convolutional layers of size 1x1 prior to the residual blocks. Specifically, performance will be evaluated with 64, 128 and 256 filters.
- **Dense layers at the head of the blocks (CDC).** As we saw in Figure 3.26, each of the aggregated residual blocks culminates in a series of dense layers to provide these blocks with additional learning capacity. for this part In the end we will consider the inclusion of between 1 and 2 layers of 128, 256, 512 and 1024 neurons **(CDCN).**
- The **learning rate (LR)** search space will vary with respect to that used during the study. In this, it was observed that in all cases the Adam optimizer looked favored by the lowest learning rate to be evaluated (0.0001), so in this case will include the learning rates 0.0001, 0.00001, 0.000001 in the search space.
- The inclusion or not of the **data augmentation (DA)** described in 3.3 will be evaluated again .
- **Final dense layers (CDF)** of the network. In this case, since the number of pesos received will be greater than those of the original architecture, the inclusion of 0 to 4 dense layers with 512, 1024, 2048 or 4096 neurons **(CDFN),**
- Regarding the **optimizer,** according to what we deduced from the study of existing architectures, we will only evaluate the performance of the model with the *Adam optimizer.*

Figure 3.38 shows the results obtained after executing the search using te the Bayesian optimization algorithm.

BR	LR	AD	BS	FDL	FDU	BDL	BDU	NF	Acc					Q	R	F1
0	0.0001	False	256	0									0.6390	0.6370	0.6079	0.6098
0	0.0001	True	256	0									0.6330	0.6354	0.6064	0.6077
0	0.0001	False	256	4	4096								0.6314	0.6140	0.6082	0.6022
1	0.0001	True	256	1				512	1	128	64	0.5893	0.5863	0.5340	0.5521	
2	0.0001	False	256	1				512	1	128	64	0.5740	0.5710	0.5221	0.5397	

Table 3.38: Best VGG models with residual blocks

If we look closely at table 3.38, we can see that the best model corresponds to the base VGG architecture without the addition of any residual blocks, with a learning rate of 0.0001, without data augmentation and without any dense layer at the head of the network. This architecture achieves a 0.6098 *F1-score*

As for the next two best configurations, it is again the architecture without residual blocks, but in this case varying the data increase in one case, and the

number of dense layers in another. These three best models obtain very similar metrics to each other. As for the fourth and fifth best models, they are the architecture with one and two residual blocks respectively, where a considerable decrease in the metrics is observed with respect to the models without residual layers. In both cases the model seems to benefit from a final dense layer of 512 neurons, a dense layer at the end of each block of 128 neurons, and a convolutional layer at the beginning of each residual block of 64 1x1 filters. These last two configurations obtain an *F1-score* of 0.55, considerably worse than that obtained by the models without residual blocks.

This search gives us a clear conclusion: in no case does the model benefit from the added residual blocks. We can hypothesize that in this case the residual blocks would harm one of the key points of the good performance of these architectures: the pre-training with Imagenet, which provides the first blocks of the network with a great capacity for the recognition of simple shapes at the same time. while the last blocks try to be refined and adapted to the FER2013 database.

After this first search, we reached the following conclusion: since the choice of the VGG16 model as the basis for the modification was largely due to the simplicity of modifying the body of the architecture, and this has proven ineffective, the most sensible thing is to -use this iteration for the search for one's own model to try to find an optimal version of the best architecture obtained in 3.38, this being the **Den-seNet201 architecture.**

3.3.2. DenseNet201 Optimization

To carry out this optimization, we will take into account the following factors, extracted from the conclusions after all the previous study.

- ÿ Firstly, and as already mentioned, during the previous study of the architectures, whose results and conclusions were shown in section 3.2.3, it was observed that the DenseNet201 network was the one that best adapted to the problem, therefore What is logical will be to perform the optimization on the body of this architecture.

- ÿ On the other hand, after carrying out the first iteration of the search for the own model in 3.3.1, it was observed that the modifications made to the body of the network, far from benefiting the model, harmed it, partly due to the benefit that contribute to the model with the pre-trained weights with Imagenet, so in this case we decided to keep the body of the network as intact as possible.

- ÿ Finally, it must be taken into account that all the existing architectures under study were designed and trained on the Imagenet data set, which, as mentioned in section 2.3, has a resolution of 224x224, and since the FER2013 images have 48x48 resolution, the use of techniques to increase said resolution could be beneficial for the model.

Resizing

There are several methods of resizing images based on which method of interpolation is used. Specifically, the Keras library offers us the methods of bilinear, bicubic, nearest neighbor, area, lanzos3, lanzos5, Gaussian and mit-chelcubic interpolation.

To estimate which of these methods is the most appropriate, as in the case of choice of data augmentation, we can make a first visual estimate of the result of the application of the methods.

As we can see in figure 3.27, depending on the type of resizing used, we will obtain different results. At first glance, some techniques such as resizing by nearest neighbors or by area seem to obtain worse quality images.
than the rest, who obtain very similar results between them.

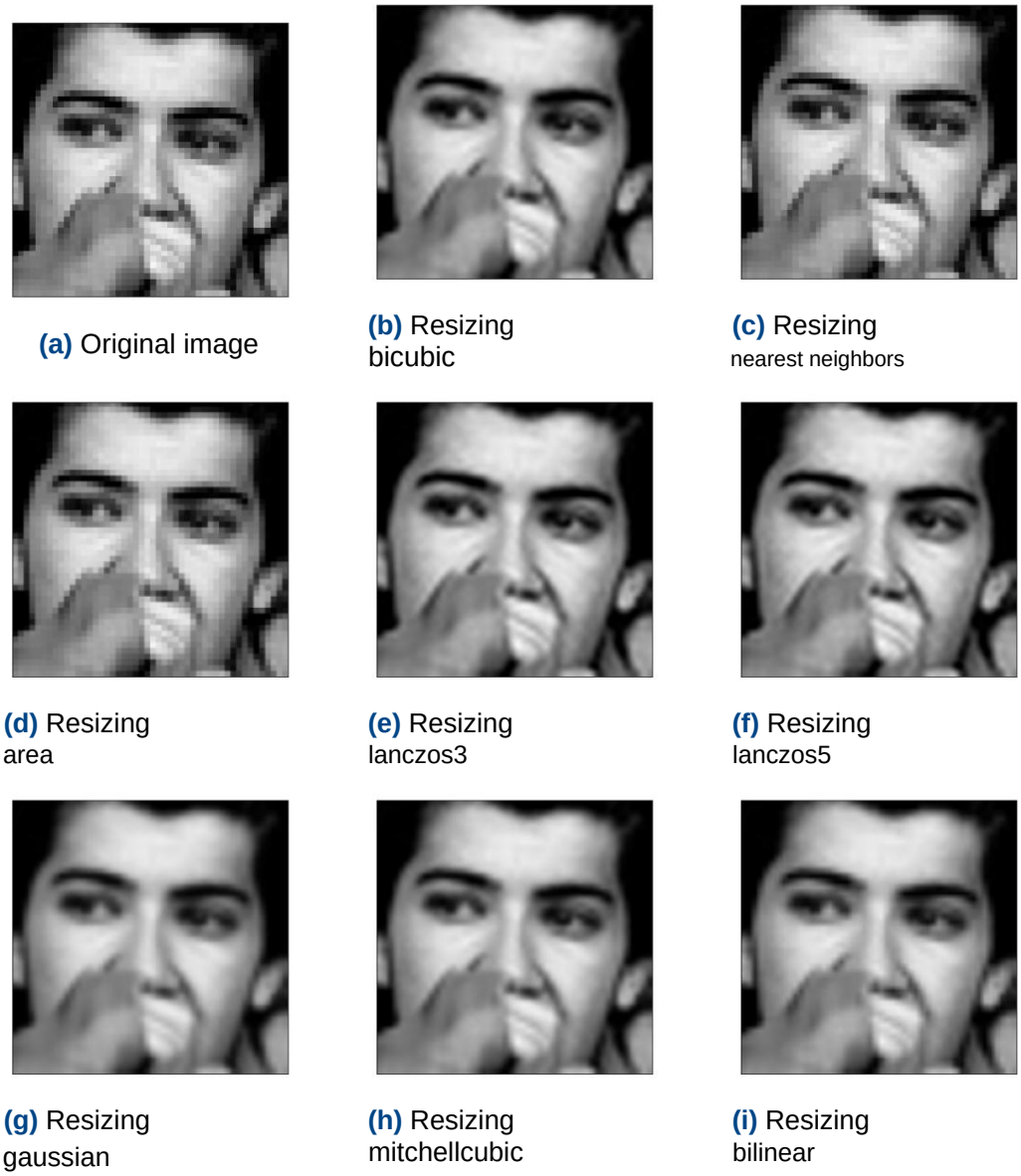
Optimization

In this case, unlike the searches performed for the previous optimizations, A pure search will not be carried out using the Bayesian optimization algorithm. This is because, as we will mention later, the dimensions of the images in the database will be increased. This will considerably increase both
the training time as the computational needs of the network, which makes The idea of carrying out a considerable number of training sessions is unfeasible. On the other hand, a series of manual tests will be carried out to determine to what extent it is possible to increase the resolution of the images so that training is viable.

First of all, we will use the optimal version of the Dense-Net201 architecture as a basis. This, as shown in the table of final results of the comparison (table ??), this is the DenseNet201 version, with *Adam optimizer,* learning rate of 0.0001, without data increase and with a batch size of 256. On this configuration, we will apply 4 different resizings, which is bilinear type resizing in four different resolutions: resizing to resolution 100x100, 125x125, 150x150 and finally 175. In this case, the resulting resolution will not continue to be increased due to computational limitations, but it is considered sufficient to verify the contribution of this type of resizing to the preprocessing of the images. For this first search, again we will resort to the holdout validation technique.

After this first refinement, we can observe that indeed the resizing seems to affect the model's metrics very positively, obtaining the best result for the largest resizing evaluated (175x175), which

obtains an *F1-score* of 0.6243 and an accuracy of 0.6829. These results, in the absence of evaluating the consistency of the model, are very promising compared to those obtained in the rest of the models. Once the best resizing has been selected of the 4, we can proceed to evaluate it in the same way that was proceeded during the study of existing architectures: a cross validation with 8 folders followed by 10 iterations incorporating in this case the test set.



(a) Original image

(b) Resizing bicubic

(c) Resizing nearest neighbors

(d) Resizing area

(e) Resizing lanczos3

(f) Resizing lanczos5

(g) Resizing gaussian

(h) Resizing mitchellcubic

(i) Resizing bilinear

Figure 3.27: Types of resizing

Resizing	Accuracy	Precision	Recall		F1
175	0.6829	0.6470	0.6242	0.6243	
150	0.6400	0.6025	0.5752	0.5756	
125	0.6707	0.6423	0.6125	0.6141	
100	0.6322	0.6106	0.5710	0.5754	

Table 3.39: Best final models

Table 3.40 shows the results of the cross validation. We can observe

that mean values have been obtained that are very similar to those obtained in the holdout validation *(F1-score* average of 0.6255 versus 0.6243, and accuracy average of 0.6720 versus 0.6829). This, together with the fact of having obtained a standard deviation within reason, is indicative of good consistency on the part of the model.

Accuracy validation folder	Precision	Recall		F1
0	0.6643	0.6649	0.6236	0.6288
1	0.6712	0.6418	0.6175	0.6199
2	0.6760	0.6301	0.5979	0.6014
3	0.6656	0.6595	0.6330	0.634
4	0.6715	0.6352	0.6276	0.6267
5	0.6870	0.6630	0.6387	0.6421
6	0.6577	0.6463	0.6073	0.6154
7	0.6826	0.6620	0.6324	0.6358
Half	**0.6720**	**0.6223**	**0.6503**	**0.6255**
Half (%)	**67.20%**	**62.23%**	**65.03%**	**62.55%**
Typical deviation	**0.0097**	**0.0139**	**0.0137**	**0.0130**

Table 3.40: Optimized DenseNet cross-validation

Table 3.41 shows the 10 iterations of the model trained on the set of data with the test set included. In this case, an average *F1-score* has been obtained of 0.6645 and an accuracy of 0.6935 with a standard deviation within reason. Are metrics are superior to those obtained in both the holdout and cross validation, so that the incorporation of new images to both the model and the validation set seems to affect the performance of the model very positively. These metrics are also superior to those obtained by the best of the models obtained during the study of existing architectures, which corresponded to this same architecture (DenseNet201) without the application of resizing, which, as we saw, obtained an *F1-score* of 0.6118 and an accuracy of 0.6390.

Finally, we show both the confusion matrix and the ROC curves of the best model obtained in results table 3.41.

Figure 3.28 shows the confusion matrix obtained for the DenseNet model. optimized, in it we can observe an improvement in all the values obtained with respect to the DenseNet version without resizing. We highlight a maximum value of 0.87 for the “happy” class and a minimum value of 0.51 for the “fear” class.

In figure 3.29 we show the ROC curves obtained, the AUC value obtained is

Iteration	Accuracy	Precision	Recall	F1
0	0.6851	0.6699	0.6513	0.6551
1	0.6948	0.6976	0.6472	0.6582
2	0.6989	0.709	0.6629	0.6746
3	0.7003	0.702	0.6668	0.6723
4	0.7005	0.7023	0.6677	0.6750
5	0.6886	0.6891	0.6661	0.6654
6	0.6936	0.6683	0.6669	0.6676
7	0.6803	0.6681	0.644	0.646
8	0.697	0.6896	0.6595	0.6645
9	0.6955	0.6868	0.6627	0.6667
Half	**0.6935**	**0.6883**	**0.6595**	**0.6645**
Half (%)	**69.35%**	**68.83%**	**65.95%**	**66.45%**
Standard deviation	**0.0068**	**0.0160**	**0.0115**	**0.0113**

Table 3.41: Final optimized DenseNet model iterations

0.8156, the highest obtained during the study, which according to range table 2.1 positions to the model as an excellent discriminator.

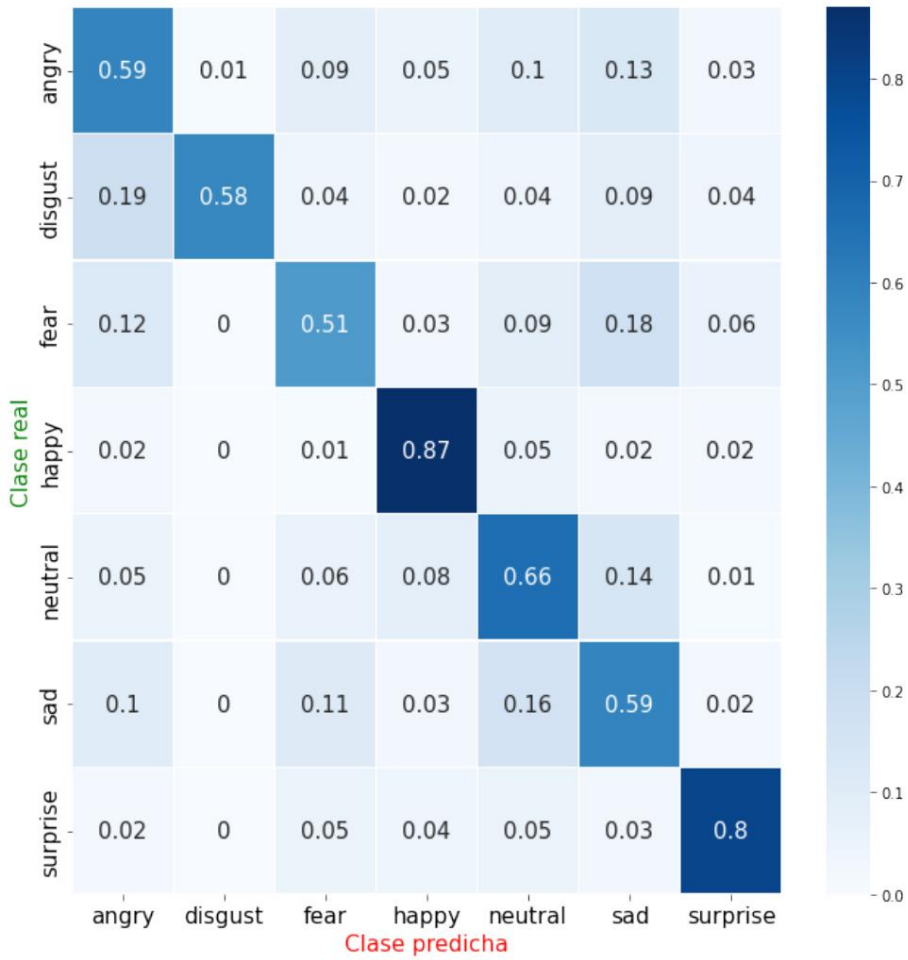


Figure 3.28: DenseNet optimized confusion matrix

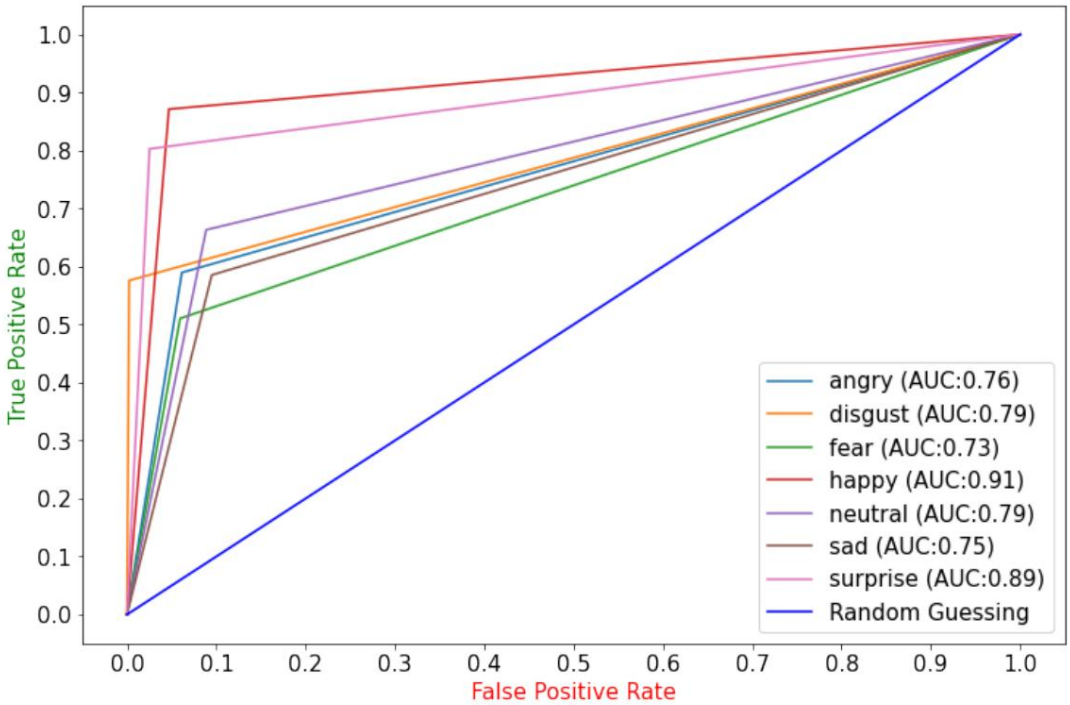


Figure 3.29: DenseNet optimized ROC curves

4. Conclusions and future work

After developing this project, we can highlight the following conclusions:

First of all, we have learned the fundamental concepts behind neural networks and *deep learning.* After the study and design of several architectures, we can conclude that when designing architectures for this type of models, the methodology based In trial and error it is the most successful, since, unlike other models of machine learning, in this case it is impossible to predict its results before to execute it on the specific database. This lack of intuitiveness when designing architectures is what highlights the relevance of algorithms search in the network optimization process. In our case, the use of an informed search algorithm such as the Bayesian optimization algorithm allowed us find the optimal settings for the hyperparameters based solely on the results of previous iterations.

Despite this, it is worth mentioning that knowledge of the fundamentals of these models is useful when limiting the search space for hyperparameters to those that are really relevant and to draw useful conclusions about the best models obtained.

Secondly, we have learned the fundamentals behind the VGG, Res-Net, DenseNet and Inception architectures in their different versions, optimizing them to obtain the best Possible model for FER2013. During the validation of these models, the importance of using validation methods that ensure the consistency of the model was observed, introducing concepts such as holdout and cross validation, and training sets, validation and testing.

Regarding model evaluation, we have learned to interpret the different metrics such as accuracy and *F1-score,* and to select each of them based on the nature both the problem to be solved and the database that will be used for it. In our case, since we had an unbalanced database, we decided to opt for the optimization of the *F1-score metric.* This point is a key difference from other

studies carried out, in which accuracy is used as a base metric for both their evaluation and optimization, obtaining somewhat less honest results.

Finally, after trying to design our own model for this problem, we came to the conclusion that when trying to apply one of these models to solve a real case, it is worth considering the use of transfer learning methods to try to reuse existing architectures before designing an architecture from scratch, not only because these architectures have a prior study and foundation that supports their performance, but also because of the usability of the prior knowledge of these networks in their pre-trained version. These architectures can be a solid base on which to apply improvements, either in the form of small modifications or, as in this case, adapting the preprocessing of the data.

As a result of all the previous study, we have managed to optimize the DenseNet201 architecture until obtaining final metrics of 69.35% *accuracy,* 68.83% precision, 65.95% recall and 66.45% *F1-score.*

As an idea for possible **future work,** the implementation of our final trained architecture in an application for the recognition and analysis of emotions in video could be proposed. Said application could be able to detect the different emotions expressed by the people who appear in the video and use this information to generate reports for subsequent analysis. This could be useful for certain entertainment venues such as restaurants or cinemas, which could implement cameras to collect information about the emotions expressed by customers who try the product and who have given their prior consent, to later try to use the data to generate an analysis of, among other things, acceptance or rejection of the product.

All the code used along with the results obtained in Jupyter note-book format is attached to the digital medium.

A. Annex 1: Search algorithms

A.1. Bayesian optimization

The Bayesian search algorithm for optimizing the hyperparameters of a model [Frazier, 2018] is an alternative to other traditional search algorithms such as grid search or random search.

This algorithm, unlike the others, performs an **informed search** when optimizing the hyperparameters, that is, it prioritizes the search with the hyperparameters that have returned the best results in past searches to reach the optimal configuration more quickly. .

The operation of the algorithm is as follows:

First of all we must define the **objective function,** which is the function that relates the different configurations of the hyperparameters with the error obtained from the network (be it the mean square error, the cross entropy, etc.). In this function, unlike the loss function (introduced in section 2.1.3), we do not know the distributions of the error function, so we cannot resort to the gradient descent method to find the global minimum. .

As an alternative to gradient descent, the Bayesian optimization algorithm creates a probability model of the objective function using several samples of it, which we will call the surrogate model, which will be the probability representation of the objective function, that is, P *(objective function score | hyperparameter).*

The most common surrogate model is a Gaussian process model that models *P(x|y),* using the pairwise history (hyperparameter, objective function score) as *x* and *y* to construct the multivariate Gaussian distributions.

This surrogate model will be used for the selection of the following hyperparameters to evaluate. To do this we will need to include it in an acquisition or selection function so that the next hyperparameter to select will be the one that maximizes the function

of acquisition. The most commonly used acquisition function is the expected improvement, whose equation is shown in A.1.

$$MEy\ \ddot{y}\ (x) = \ddot{y} \int_{\ddot{y}\ddot{y}}^{\ddot{y}} \max(y\ \ddot{y}\ \ddot{y}\ y, 0) \cdot PM(y|x)dy \quad (A.1)$$

Being:

ÿ PM(x|y): the surrogate model. :

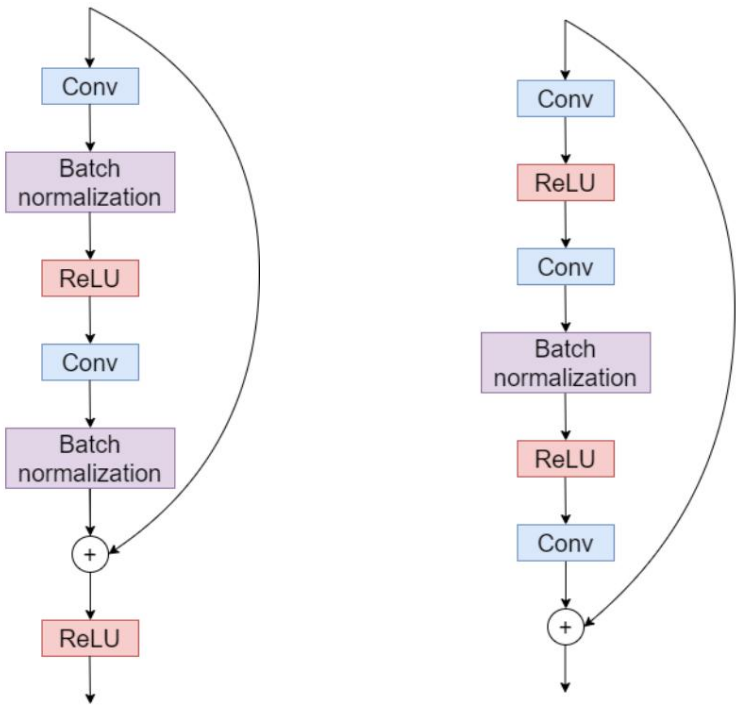
ÿ and ŷ the minimum score in the objective function observed so far.

ÿ y: the new score on the model .

Therefore, once the new model has been trained, the real error value obtained will be used. do as a new sample to update the surrogate model

B. Annex 2: Additional block diagrams

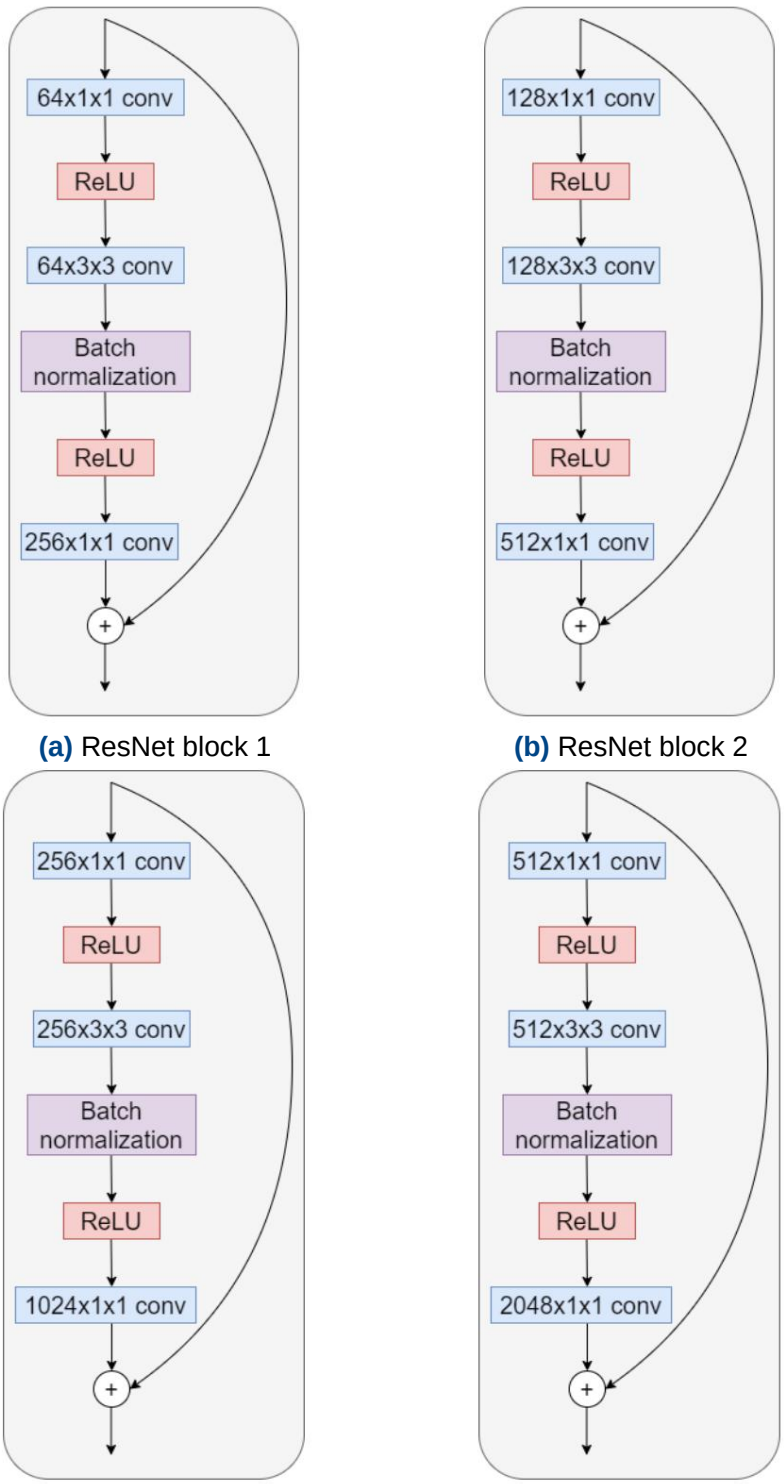
B.1. ResNet block diagrams



(a) ResNetV1 block

(b) ResNetV2l block

Figure B.1: ResNetV1 and ResNetV2 blocks

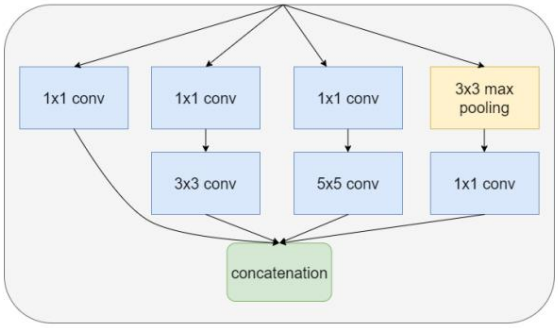


(a) ResNet block 1 **(b)** ResNet block 2

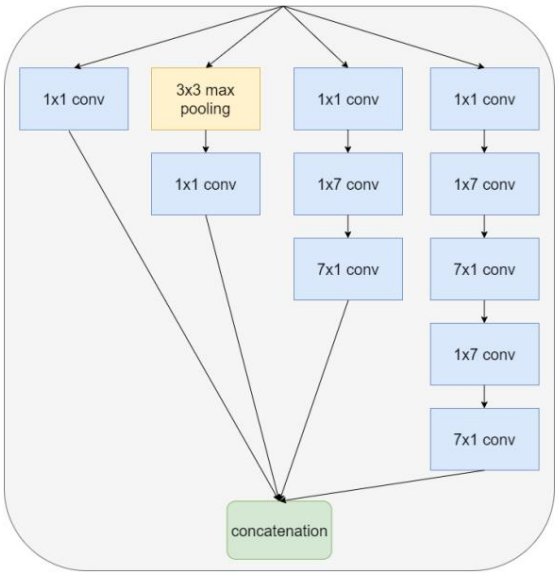
(c) ResNet block 3 **(d)** ResNet block 4

Figure B.2: ResNet Blocks

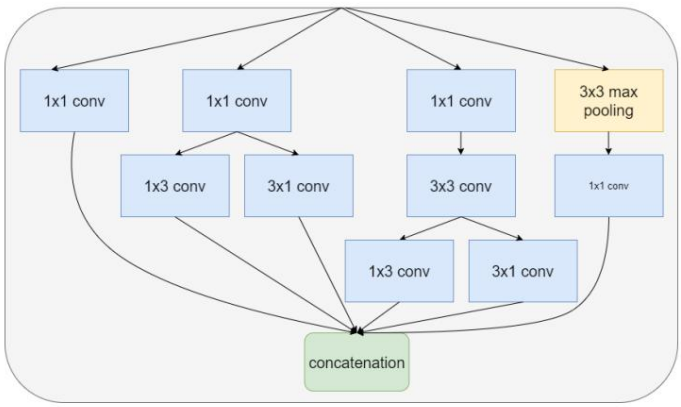
B.2. Inception block diagrams



(a) InceptionV2 block A



(b) InceptionV2 block B



(c) InceptionV2 C block

Figure B.3: InceptionV2 Blocks

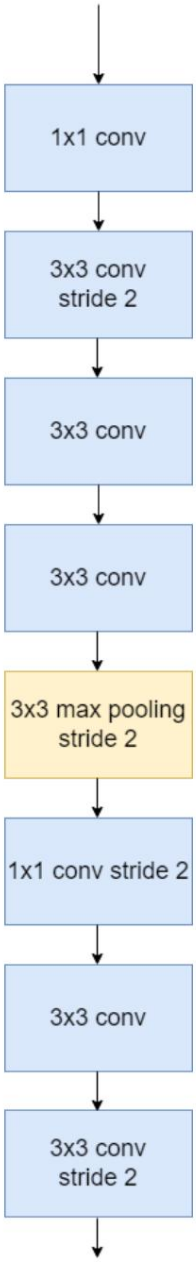
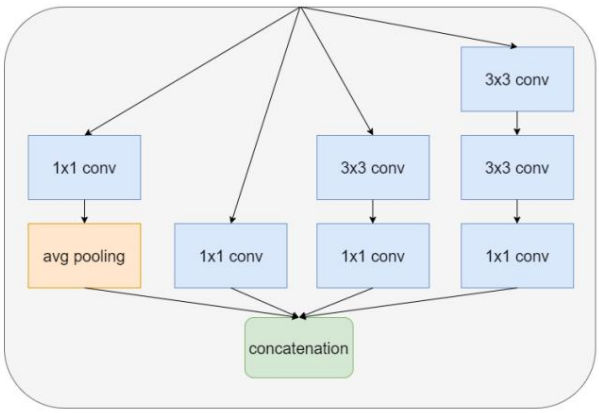
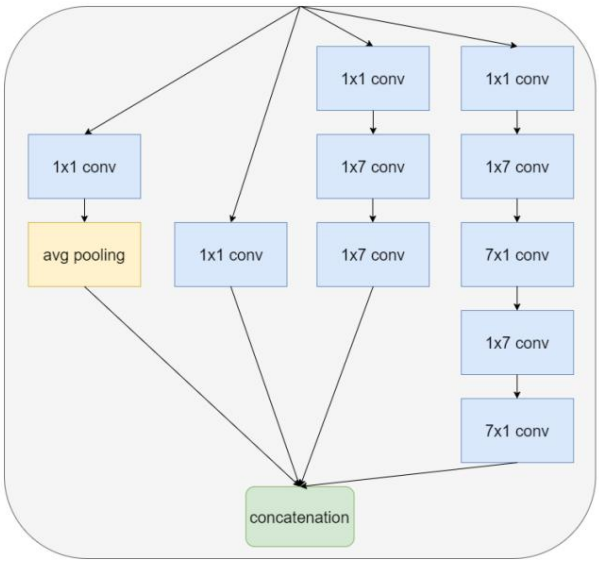


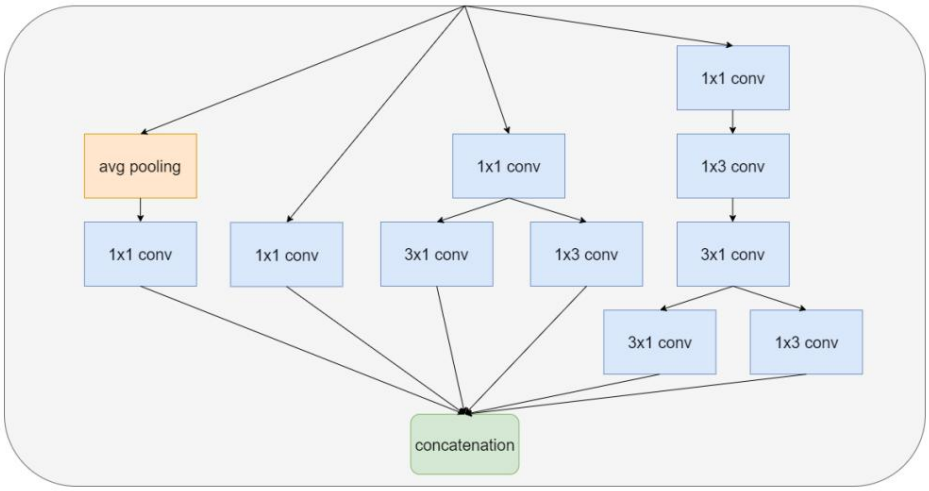
Figure B.4: InceptionV4 initial block



(a) InceptionV4 block A

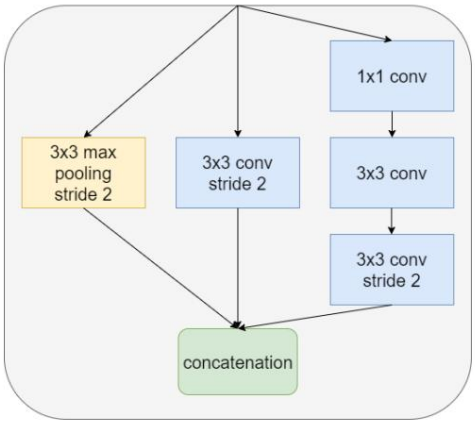


(b) InceptionV4 block B

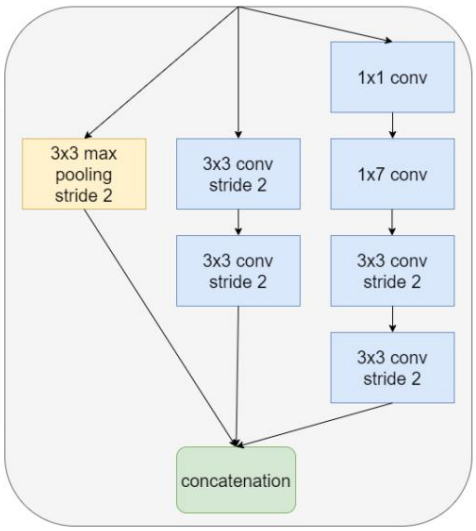


(c) InceptionV4 C block

Figure B.5: InceptionV4 Blocks

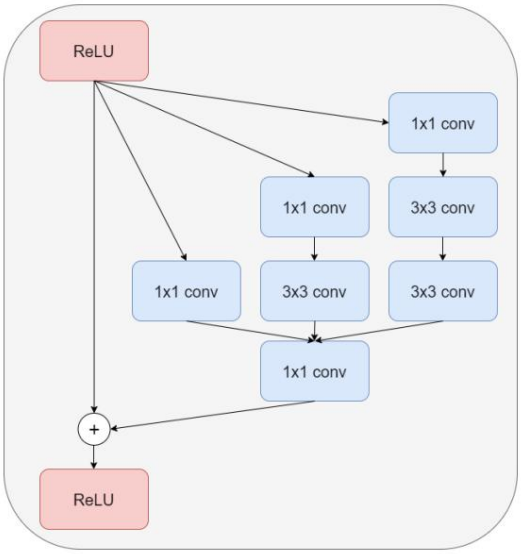


(a) InceptionV4 reduction block A

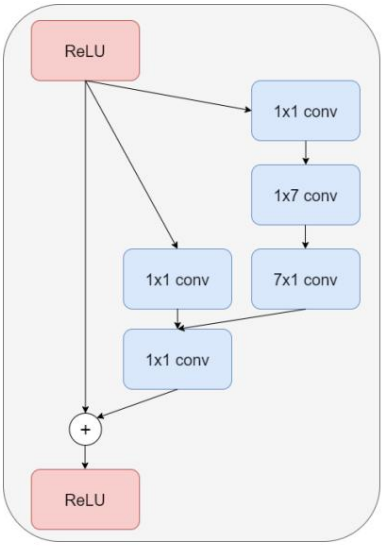


(b) InceptionV4 reduction block B

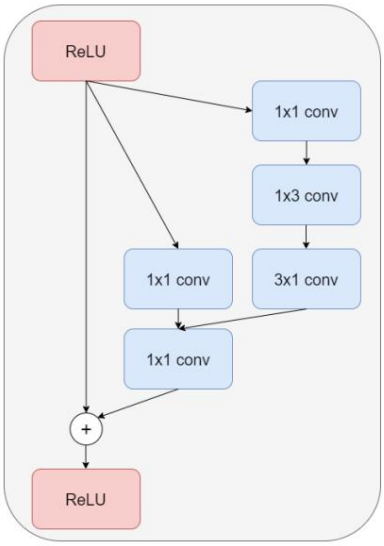
Figure B.6: InceptionV4 reduction blocks



(a) Inception-ResNet block A

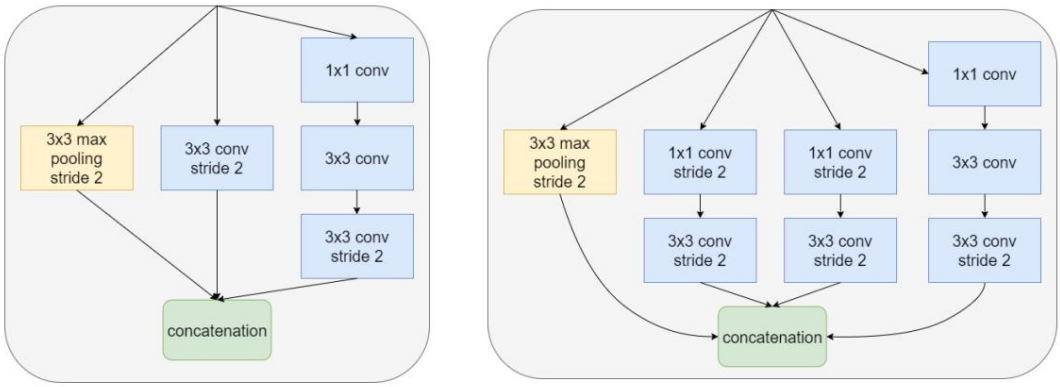


(b) Inception-ResNet block B



(c) Inception-ResNet block C

Figure B.7: Inception-ResNet Blocks



(a) Inception-ResNet reduction block A **(b)** Inception-ResNet reduction block B

Figure B.8: Inception-ResNet reduction blocks

C. Annex 3: Additional layers

C.1. Batch normalization

The batch normalization layers are applied to avoid the imbalance between the outputs of the different neurons during the training of the network, since these do not have to be on the same scale (a neuron can have outputs in the range [0-1] while another in the range [1000-10000]).

This imbalance can cause two relevant problems. The first refers to the slowdown of both the learning and inference of the network, and the second is more related to the performance of the network, since high output values cause corruption of the gradient values (this This is known as the gradient exploitation problem.

Batch normalization layers have the function of normalizing the outputs of the layer above so that the mean corresponds to 0 with a standard deviation of 1.

To do this, the following formula is applied for each of the outputs:

$$z = \frac{x \ddot{y} m}{yes} \cdot g + b \qquad (C.1)$$

Being:

ÿ z the new normalized output.

ÿ x the value of the current output.

ÿ m the average of the outputs.

ÿ s the standard deviation.

ÿ gyb parameters to be adjusted during training.

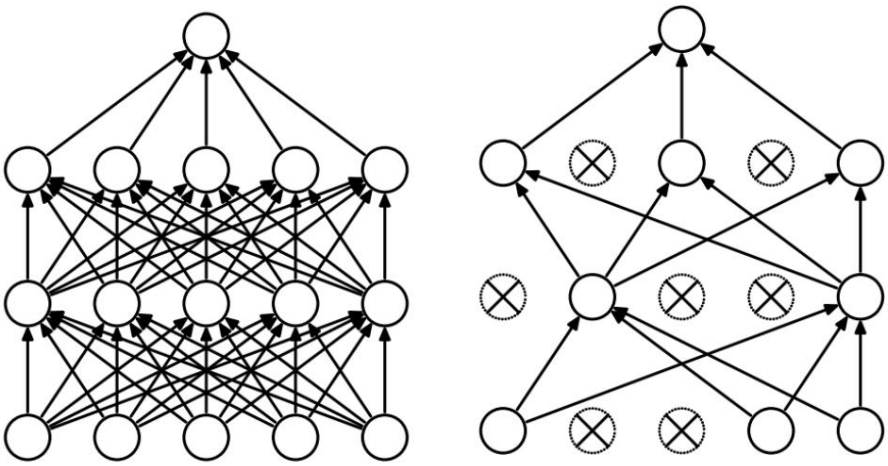
[Sergey Ioffe, 2015]

C.2. Dropout

Dropout is a technique used to avoid overfitting during training of neural networks.

This is based on approximating one of the most promising solutions in terms of avoiding this phenomenon: the combination of models. This technique consists of inferring the result from the average of the result returned by several different neural networks, which would imply either training several different architectures, or training the same architecture several times, so in most cases This method is computationally unfeasible.

The Dropout technique is a way to approach this method in a much more efficient way. The idea, as shown in Figure C.1, consists of randomly removing certain neurons from the network temporarily during training, ignoring both their input and output connections, thus temporarily varying the architecture of the network, in this way, From a neural network with n number of neurons we can potentially obtain 2 n different models.



(a) No dropout **(b)** With dropout

Figure C.1: Dropout

The behavior of the neurons to which Dropout is applied will be different depending on whether it is the training phase or the inference phase.

ÿ During the **training phase,** neurons with Dropout will disappear randomly based on the parameter p, with p being the probability that that particular neuron is maintained in training, as shown in figure C.2a.

ÿ During the **inference phase,** a way to average the result is required and, since it is not feasible to explicitly average all of these models, a different approach is used. This is based on using a single network in validation with all active neurons. The weights of this network will be a reduced version of the trained weights. This, as shown in Figure C.2b, is achieved by multiplying each of the trained weights by the probability of having maintained that neuron during training (p), which ensures that the expected output (under the distribution of Dropout) be the same as the actual output during validation. This scaling allows the 2 n possible networks to be combined into a single one at the time of validation.

(a) Dropout in training **(b)** Dropout on validation

Figure C.2: Dropout phases

Applying this method greatly reduces overfitting in a large number of classification problems [Nitish Srivastava, 2014].

D. Annex 4: Optimizers

D.1. SGD

The “Stochastic Gradient Descendent” optimizer [You et al., 2019], by its acronym SGD, is an algorithm based on gradient descent. The difference lies in the way of calculating the gradient, since while in gradient descent the gradient is calculated based on each of the values of the hyperparameters, in SGD only the gradient of a series of random samples is calculated. (stochastically, hence the name), which reduces computational needs.

D.2. Adam

The Adam optimizer [Bock et al., 2018], also known as “Adaptive Moment Estimation” is an optimization algorithm widely used for its performance and efficiency when training models based on relatively big. It is a relatively complex algorithm, since it tries to combine two other optimization algorithms: **AdaGrad** and **RMSProp.**

- ÿ **Adagrad** or adaptive gradient algorithm is an adaptation of SGD (D.1) that uses different learning rates for each variable. To do this, it takes into account the gradient accumulated in them.
- ÿ **RMSProp** or root mean square propagation algorithm is a variation of AdaGrad in which, for each hyperparameter, only the most recent gradients will be taken into account, rather than the accumulation of gradients.

E. Annex 5: Additional concepts

E.1. Gradient fade

This phenomenon occurs in deep neural networks during network learning through **backpropagation,** and causes the gradient factor to be so small that the first layers of the network cannot update their weights [Hochreiter, 1998].

To understand this phenomenon we must analyze the backpropagation equation 2.9, introduced in section 2.1.3. In it we observe that the output of the neuron (h) multiplies the rest of the equation.

This value is the one obtained directly from the **activation function** of each neuron, and it will be the nature of said function that determines the range in which said value will be found.

If we look at these activation functions, we can see (especially in the sigmoid and tanh types) that a large part of the output values will be between 0 and 1 (or 0 and -1), and this is where the problem lies: according to As the gradient propagates to the first layers of the network, there will be a tendency for it to reduce until it almost disappears in the first layers, which will limit the ability to learn from relatively deep networks.

Bibliographic reference

[April, 2013] April, RR (2013). Fer2013. https://paperswithcode.com/dataset/fer2013. Accessed November 2022.

[April, 2021] April, RR (2021). Gradient descent. https://lamaquinaoraculo. com/computacion/el-descent-of-the-gradient/. Accessed July 2022.

[Baheti, 2022] Baheti, P. (2022). Why do neural networks need an activation function?]. https://www.v7labs.com/blog/neural-networks-activation-functions. Accessed June 2022.

[Baro, 2009] Baro, L.M.S. (2009). Bias-variance compromise.

[Bock et al., 2018] Bock, S., Goppold, J., and Weiß, M. (2018). An improvement of the convergence proof of the adam-optimizer. *arXiv preprint arXiv:1804.10587.*

[Chen, 2019] Chen, C. (2019). Emotion and feeling. https://blogthinkbig.com/neuronal-reds-deep-learning. Accessed November 2022.

[Chollet, 2014] Chollet, F. (2014). Keras documentation. https://keras.io/api/. Access done in July 2022.

[Christian Szegedy, 2015] Christian Szegedy, Vincent Vanhoucke, SIJSZW (2015). Rethinking the inception architecture for computer vision. https://arxiv.org/abs/ 1512.00567. Accessed July 2022.

[Christian Szegedy, 2016] Christian Szegedy, Sergey Ioffe, VVAA (2016). Inception-v4, inception-resnet and the impact of residual connections on learning. https://arxiv. org/abs/1602.07261. Accessed July 2022.

[David W. Hosmer Jr., 2013] David W. Hosmer Jr., Stanley Lemeshow, RXS (2013). *Applied Logistic Regression.* Wiley Series in Probability and Statistics. John Wiley Sons.

[Education, 2020] Education, IC (2020). What are neural networks? https://www.ibm. com/cloud/learn/neural-networks. Accessed November 2022.

[Ekman, 1992] Ekman, P. (1992). An argument for basic emotions. *Cognition and Emotion,* 6(3-4):169–200.

[Fawcett, 2005] Fawcett, T. (2005). An introduction to roc analysis. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4533825/. Accessed July 2022.

[Frazier, 2018] Frazier, P.I. (2018). A tutorial on bayesian optimization. https://arxiv.org/pdf/1807.02811.pdf. Accessed November 2022.

[Gao Huang, 2016] Gao Huang, Zhuang Liu, L. vd MKQW (2016). Densely connected convolutional networks. https://arxiv.org/abs/1608.06993. Accessed July 2022.

[Hochreiter, 1998] Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural networks and problem solutions. *International Journal of Uncertainty, Fuzzi-ness and Knowledge-Based Systems,* 6(02):107–116.

[ichi.pro, 2021] ichi.pro (2021). Cross entropy loss function. https://ichi. pro/es/cross-entropy-loss-function-267783942726718. Accessed July 2022.

[Kaiming He, 2015] Kaiming He, Xiangyu Zhang, SRJS (2015). Deep residual learning for image recognition. https://arxiv.org/abs/1512.03385. Accessed July 2022.

[Karen Simonyan, 2015] Karen Simonyan, AZ (2015). Very deep convolutional networks for large-scale image recognition. https://arxiv.org/abs/1409.1556. Accessed July 2022.

[Nitish Srivastava, 2014] Nitish Srivastava, Geoffrey Hinton, AKISRS (2014). Dropout: A simple way to prevent neural networks from overfitting. https://jmlr.org/papers/v15/srivastava14a.html. Accessed July 2022.

[Paperswithcode, 2022] Paperswithcode (2022). Facial expression recognition on fer2013. https://paperswithcode.com/sota/facial-expression-recognition-on-fer2013. Accessed July 2022.

[Recuero, 2018] Recuero, P. (2018). Do you know how neural networks differ from deep learning? https://blogthinkbig.com/redes-neuronales-deep-learning. Accessed July 2022.

[Sammut and Webb, 2010] Sammut, C. and Webb, G.I., editors (2010). *Mean Squared Error,* pages 653–653. Springer US, Boston, MA.

[Sergey Ioffe, 2015] Sergey Ioffe, C.S. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. https://arxiv.org/abs/1502.03167. Accessed July 2022.

[Taha AA, 2015] Taha AA, HA (2015). Metrics for evaluating 3d medical image segmentation. https://people.inf.elte.hu/kiss/11dwhdm/roc.pdf. Accessed July 2022.

[You et al., 2019] You, K., Long, M., Wang, J., and Jordan, M.I. (2019). How does learning rate decay help modern neural networks? *arXiv preprint arXiv:1908.01878.*