UNIVERSITÀ DEGLI STUDI DI MILANO

Data Science for Economics

Machine Learning Experimental Project

Chihuahuas vs Muffins Image Classifier

Professor: Nicolò Cesa-Bianchi

Student: Guglielmo Berzano, id: 13532A

E-mail: guglielmo.berzano@studenti.unimi.it

February 2024

Abstract

This work aims to find a good machine learning model for the problem of binary-classifying images of chihuahuas and muffins. The objective is pursued thanks to the implementation of Convolutional Neural Networks (CNNs) through the Keras API of Tensorflow. After a brief description of the dataset and of the preprocessing methods, the report continues by outlining Neural Network theory with a deepening on CNNs, illustrating their main layers and their functioning. In the third section, the report shows the evolution of the model, emphasizing all the steps taken to improve its architecture. After hyperparameter tuning was performed, the final model was able to consistently predict the right class for around 9 pictures out of 10 as demonstrated by the 5-Fold Cross-Validation.[i]

[i] I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

TABLE OF CONTENTS

1 Introduction

The objective of this project is to create a **Convolutional Neural Network** to correctly classify a set of images. The classification is binary and images could represent either *muffins* or *chihuahuas*. The network is created in Python, using the **Keras** API of the **Tensorflow** library.

In the following sections I will deeply explain the methodology and the architectures used and why I decided to use them. The Python code is completely available on my GitHub profile.

1.1 Exploratory analysis and preprocessing

The training dataset is composed by 4,743 total images: 2,559 represent *chihuahuas* and the remaining 2,184 represent *muffins*. First of all, we can we can look at some of the images, to better understand what we are working with.

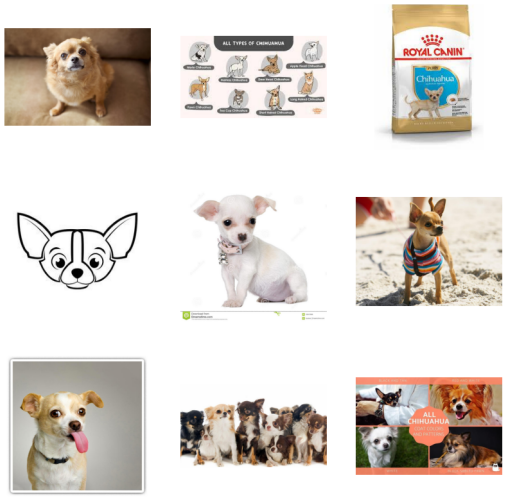

Figure 1: Example pictures of Chihuahuas

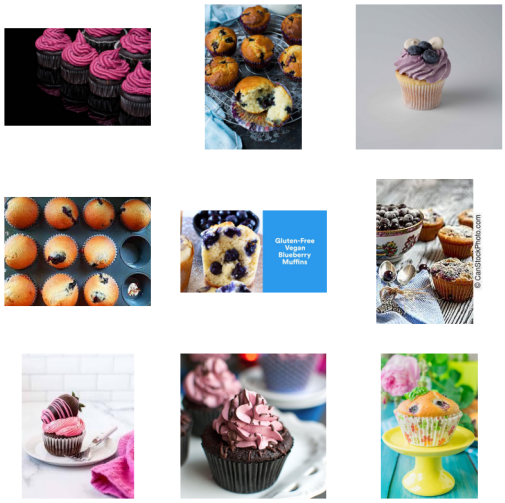

Figure 2: Example pictures of Muffins

By looking at these 18 pictures we already obtain many insights:

- Not all images represent clearly what their labels say, many do not even represent real dogs – or muffins – but drawings, printed objects or video thumbnails instead.

- Images are neither of the same size nor have the same orientation.

Clearly there is a lot of noise in the data which may not be a good thing since noisy data in machine learning models may worsen performance substantially. Nevertheless, there are no corrupted images and thus I decided not to remove the *sort-of-fake* images.

However, one of the compulsory steps to efficiently analyse pictures is to make them of the same pixel-size and thus, to decide how many pixels images should be, we shall analyse deeper image size to recognize patterns or *golden ratios*.

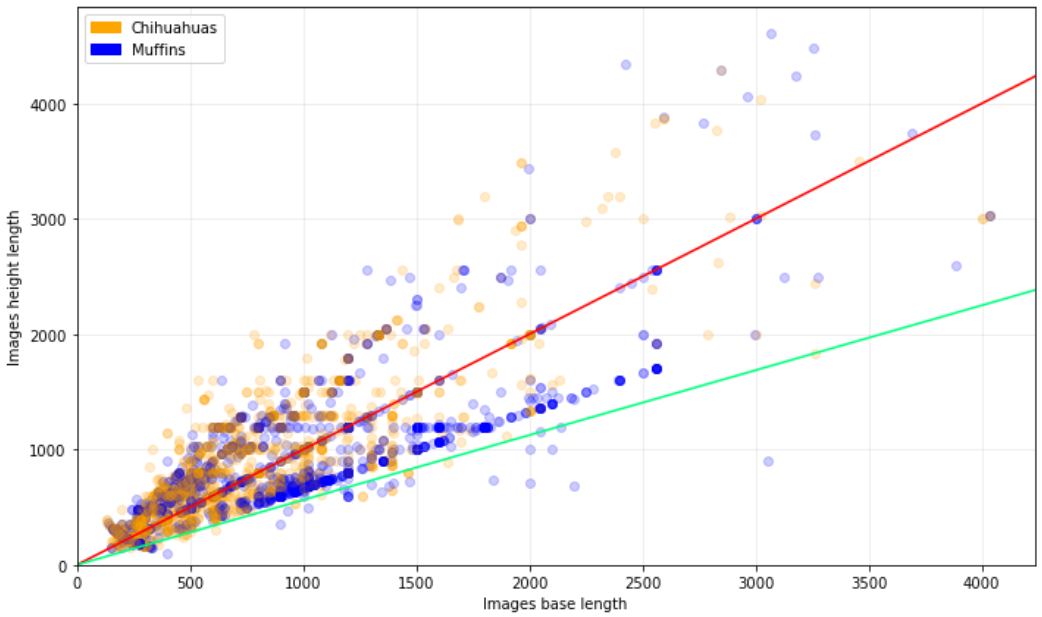

Figure 3: Sizes of the non scaled images in pixels

In the above plot each picture is represented by a point: the darker the point, the more images will have that size. So, by looking at the plot we can conclude that the vast majority of the images are small, few are very large and not many of them lay on the 45 degrees red line which represents 1:1 ratio. For example, muffin images are prevalent below the red line meaning that they are mostly rectangles with the base larger than the height. Given this fact, those images may probably have been taken from videos which are usually rectangles with aspect ratio 16:9. Actually, after having plotted the 16:9 ratio line, depicted in green, we notice that muffin images lay above it

and thus, they do not respect this particular aspect ratio. Therefore, I would say that despite the attempt, there are no evident conclusions we can draw by analysing image size and there is no *golden ratio*.

However, images must be rescaled so that they all have the same size and, since there is no size better than the other, I arbitrarily decided 150px $\times$ 150px. Subsequently, I memorized the pictures into a NumPy array and turned them into grayscale. By doing so, we moved from tridimensional *tensors*, to bidimensional matrices. Two dimensional images can be seen as "tridimensional matrices", also known as *tensors*. Why images have two dimension is clear but may not be immediate why they have a third one. The reason behind this must be searched in the composition of pixels, which are made up by three color channels, in most cases red, green and blue, from here "RGB". Colors are stored in sort-of vectors, creating depth to each pixel. Each number in those vectors must belong to the set $[0, 255]$ having, then, around 16 million total combinations and, thus, colors. By turning RGB into grayscale, we lose color information but, since muffins and chihuahuas are more or less of the same yellowish color, I think it cannot be used as a discriminant to understand whether an image represents a dog or a muffin. Grayscale ensures to have images with just one layer of depth layer, instead of three layers as it was for RGB, making computations much faster. The last operation I performed was a normalization of the grayscale-values shrinking their interval from $[0, 255]$ to $[0, 1]$.

2 Neural Network theory

In this section I will briefly explain the theory behind Neural Networks with a specific focus on CNNs. In particular, I will deepen the structure and the parameters of the layers contained in Keras, with a in-depth study of those used in the architectures of this project. Lastly I will explain my strategy for hyperparameter tuning.

2.1 Training set, Validation set and Test set

Before describing the structure of a generic Neural Network, let's analyse how data must be arranged for a supervised machine learning model to work. It is fundamental to have three datasets which may also be acquired by subsetting the original data-source, if it is big enough. The sets are known under the names of: *training set*, *validation set* and *test set*. The first one is used to train the model, the second one – which can also be a random subset of the training set – is used to check the performance of the trained model on new data while the last one, the test set, is the one used for evaluating the real performances of the model on completely new data.

2.2 General Neural Network structure

A Neural Network is a complex mathematical object widely used nowadays for solving a variety of problems by making machines reason mirroring the human brain. Indeed, their functioning is based on nodes, also referred to as *neurons* and on links which connect the nodes. Each node of the network belongs to a layer and, if the Neural Network is *feed-forward*[1], then each layer is in some way connected to the following one. This generates the following non-recursive structure:

- **Input layer**: takes normalized training data and passes them to the following hidden layers.
- **Hidden layer(s)**: execute mathematical and algebraic operations to best fit the training data. This is achieved by tuning the weights, thus the importance that each link has. Here with "link", I am referring to the fact that each node will be linked with all the nodes in the previous layer and each of those links will have a weight, that is a number close to 0 but that, realistically, could be anything belonging to $\mathbb{R}$.

[1] In feed-forward neural networks, data is being "processed" from a beginning point to an ending one. There are no recursions nor loops, just a linear process from the beginning to the end.

- **Output layer**: output the result computed by the Neural Network.

Essentially, the network receives something numerical as input which will then be passed to the hidden layers. Here, each node will multiply the inputs it receives with the weights of the links from which it is getting the input. Then, it sums the results and passes it to an **activation function** which returns an output which is then passed to the following nodes with a process known as *forward propagation*. This process goes on up until the network finishes. Among the many activation functions available, in this project I implemented the following ones:

- **ReLU** which stands for **Rectified Linear Unit**. This function has become widely used in the last years, especially in Convolutional Neural Networks, for its simplicity of computation and for the many advantages that it provides in terms of gradient descent and of not being affected by the *vanishing gradient problem*, allowing models to learn quicker and better. Essentially, what this function does, is just to make negative values equal to zero and leaving the positive ones as they are, introducing non-linearity which has a great effect on prediction power. The ReLU activation function is described by the following formula:

$$\sigma(x) = \max\{0, x\} \tag{1}$$

  where σ is the usual way to define activation functions and $x \in \mathbb{R}$.

- **Sigmoid** is the activation function used for the output layer. Its task is to transform the value that it gets as input to either 0 or 1, depending on whether that number is positive or negative. This is useful for classification purposes since, by extremizing values to 0 or 1 we can easily distinguish the two classes. In this case, if the output is 0 then the model predicts the image to be a chihuahua otherwise it is predicted as muffin. This activation function is described by:

$$\sigma(x) = \frac{e^x}{1 + e^x} \tag{2}$$

  as before, σ is activation function itself and $x \in \mathbb{R}$.

The two activation functions depicted above may be represented by the following plots.

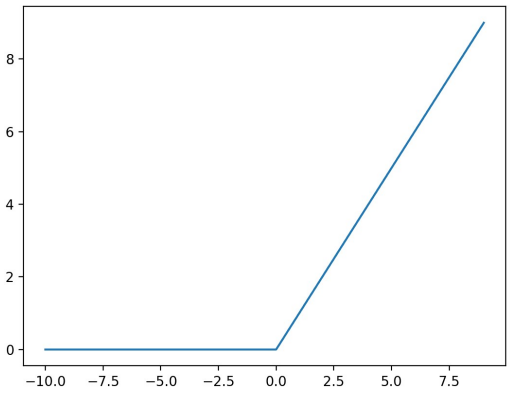

Figure 4: ReLu plot

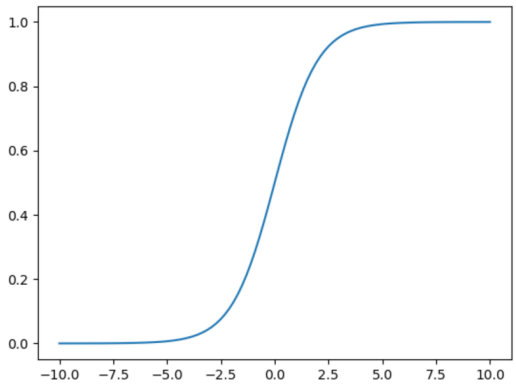

Figure 5: Sigmoid plot

Neural Networks learn by analysing training data in loop: from the input layer to the output layer and then again from the beginning to the end. These cycles are called **epochs**, and in each one the network analyses the entire training set. Specifically, in each epoch, sample data are passed to the input layer in **batches**, that are subsets of the training set, and the greatness of these batches is known as **batch_size**. Thus, each epoch is composed by n batches and ends when the network has analysed all the n batches.

Finally, after the Neural Network completed the learning process and was evaluated on validation data, results must be compared to real values to check how well the model performed. This is analysed by two measures: the **loss** whose formula is known as **loss function** and the **accuracy** which can be of the training set or of the validation set. The first one computes a number belonging to $\mathbb{R}^+$ which describes the error between the true values and the predicted ones while the **accuracy** measures the precision of the model in the interval $[0, 1]$. Among the many types of losses available, Keras provides the **binary cross-entropy** which will be the loss of choice for this project[2].

Moreover, while compiling the model, it is crucial to pass an **optimizer** whose job is to find the weights for the links that minimize the loss function. In this project

[2]Click here to learn more.

I implemented the **Adam** optimizer which stands for "adaptive moment estimation" which was found to be very efficient and effective. The parameter that will influence the behaviour of the optimizer is the **learning rate** that is realistically a value in the set $[0.0001, 0.1]$ but could actually assume any value as long as it is close to 0. It is important to tune carefully this hyperparameter because the risk is that by setting a very high learning rate – like 0.2 – the optimizer will "jump" along the function without ever stabilizing while if we set it too low – like 0.00001 – the optimizer will think to be already at a point of minimum and will not move. There is also another risk in which it is possible to reach a local minimum without being able to escape dooming the network to a sub-optimal solution. So, the learning rate should be big enough in order not to be stuck into local minima but small enough not to jump around like crazy.

The last concept I want to cover is **back propagation**. Finally, when an epoch is concluded, the network computes the gradient of the loss with respect to the weights using the *gradient descent* algorithm. The gradients will be propagated backward with a process known as **back propagation** with the aim to lower the error by updating the weights according to the learning rate of the optimizer. The main idea here is that the final result of a network is due to computations done in the layer before the output and then in the layer even before and so on. So, the network adjusts the weights of the links in a way such that the final gradient of the loss is the smallest possible.

2.3 CNN layers and hyperparameters

Convolutional Neural Networks are a special type of NN particularly good at classifying spatial data like images or videos. In particular, CNNs have many peculiar layers which can be tweaked to reduce overfitting or to reach higher accuracies or lower losses. The following list contains all the layers I used for the creation of this project with a brief description on what they do and on their hyperparameters in Keras.

- **Conv2D**: the standard convolutional layer for bi-dimensional images. Some of the most important parameters of this layer are:

– **Filters**: a positive integer which denotes the number of matrices used in the computations which usually are 32, 64, 128 or other powers of 2. Values assumed by the filters will be determined by the stochastic gradient descent process.

– **Kernel_size**: a tuple of two positive integers which determines how many squared pixels the convolution layer must analyse at the time. As explained by the *Visual Geometry Group* (VGG) of Oxford in [3], a deep network with a concatenation of 3×3 kernels outperforms a smaller network with bigger filters. Indeed, they found out that a sequence of two 3×3 sized kernels is able to "see" the same field of a 5×5 and a concatenation of three "sees" the field of a 7×7. This decreases much computations and accelerates the learning process.

– **Stride**: a positive integer number which determines of how many pixel the *kernel* must shift to the right – or down – after the computation between a *filter* and the *kernel* is completed. Usually it is a good practice to select a *stride* value smaller than the *kernel_size* since if this is not the case, we would skip pixels in the computations.

– **Padding**: it determines whether the image should be surrounded by – at least – one layer of zero pixels or not. This may be useful not to lose information.

Essentially, given a square image, the convolution layer "takes" the first *kernel_sized* pixels on the top left of the image and performs a product with the first *filter* which is a matrix having the same dimensions of the *kernel*. After the computation is completed, the analysed kernel is moved to the right by the *stride*'s value and, if it is not possible, it goes down by the *stride* value and "teleports" to the left side of the image, following a sort of Z pattern. This process is performed for every single *filter* and the result is stored in a tensor whose dimensions are

equal to:

$$Height_{out} = Width_{out} = \frac{Dim_{input} - Kernel\_size + 2 \cdot Padding}{Stride} + 1 \qquad (3)$$

and $Depth_{out}$ equal to the number of filters. Clearly, since the *stride* is at the denominator, having a value greater than 1 is an effective way to reduce the resolution of the output. Then the tensor is passed to the next layer[3].

- **MaxPooling2D**: performs dimensionality reduction over the result of the convolution layer.
  - **Pool_size**: this parameter is a tuple of two integers which denotes how many squared pixels should the layer check before taking the maximum. The most used value is 2×2.
  - **Stride**: look at *Conv2D*. If no value is manually passed, the default one will be exactly the pool_size.
  - **Padding**: look at *Conv2D*.

  The way the dimensionality reduction works is the same as in equation (3). Note that in this case the *depth* dimension remains untouched.

- **Flatten**: this layer is used to transform the obtained tensor to a numeric vector.

- **Dense**: this is what it is normally referred to as *fully connected layer*. In this layer, every input is connected to every output and serves the purpose to make the classification itself. The only mandatory parameter to this layer is the number of neurons which must be a power of 2. Since this layer works by assigning to each link a weight, among the optional parameters to pass there are the **kernel regularizations l1** and **l2** which try to overcome the problem of having nodes with too large outputs. Indeed, by running the NN, it may happen that some neurons become crucial for the classification problem which may, in turn, lead

[3]Click here to see an animation that shows how the convolution algorithm works.

to an increase in overfitting and overall disappointing performances. These two methods rely on the *lasso* and *ridge* regression respectively and, in order to operate, they both need a parameter $\lambda \in \mathbb{R}^+$ where the closer is λ to 0, the smaller the impact of the regularization. Other regularization tools are *bias regularizers* and *activity regularizers* but will not be covered in this project.

The *flatten* layer is important because it makes possible to do the switch from convolution layers to fully connected layers. Indeed, the *dense* layer only accepts as input a mono dimensional vector of numbers which is exactly the output of the *flatten* layer.

- **Dropout**: one of the most common ways to prevent overfitting. This layer is traditionally used after a fully connected layer select some nodes with probability $p \in [0, 1]$ and to make all the weights related to those nodes equal to zero. By selecting p between 0.5 and 0.8, the network is forced to become more and more robust, reaching good performances without the need of having many nodes.

  Nowadays however, the *dropout* role in CNNs is changing because many recent implementations use $p = 0.1$ or $p = 0.2$ after a *Conv2D* layer, namely [2]. These studies show an improvement in performances but I will not implement those results in this project.

- **BatchNormalization** – from now on BatchNorm or BN: this layer can be added after a *dense* in order to normalize the result. This is another way to deal with overfitting by managing and normalizing large weights that some links may have. This process is done per-batch and contains parameters that can be trained and improved in each epoch.

  BatchNorm and Dropout serve very similar purposes of decreasing the overfitting problem with the goal of having higher general performances and of mitigating the vanishing gradient problem. Why and when to use one or the other is not a simple question to answer. As shown in [1], in CNNs, BatchNorm showed a better

overall performance without perturbing significantly the general architecture of the network, just by tuning the learning rate of the *Adam* optimizer. This was shown to be counterproductive when combined with *dropout* layers, unless p was set very precisely. Furthermore, another important factor is to decide where to put these layers but actually it seems that trial and error is the best method – if combined with the right theoretical assumptions.

Usually, in CNNs, there is a deep sequence of *Conv2D* and *MaxPooling2D* layers which together form a sequence of convolution layers. After this sequence, there is always a *flatten* layer followed by at least one hidden *dense* layer and finally a *dense* layer that serves as output. These sequences of layers identify the nature of the Convolutional Neural Network and is known as **architecture**.

2.3.1 Hyperparameter tuning

After having found a model that performs decently, a crucial step is to optimize the hyperparameters to make it perform even better. To do so, I tried to use the **Keras Tuner** library but faced many errors and found many a sub-optimal solutions. This made me search for alternative methods and what I decided to do was to modify one parameter at the time looking for the best ones. Of course I am aware of the limitations of this method: if I optimize a parameter and then optimize another one, I cannot be sure that the first parameter is still the best one given the new second parameter and so on, the more parameters there are, the harder the problem becomes. Nevertheless, I think that this method, given the computational power at my disposal, will generate a good – or at least acceptable – proxy of the best solution.

Now that theoretical concepts and parameters have been explained, I will implement some CNNs to try to predict whether an image represents a chihuahua or a muffin.

3 Convolutional Neural Network implementation

In this section, I will first illustrate a base model and, from that, I will describe the steps taken to try to fix problems and improving the architecture.

3.1 First model

In this subsection I will analyse the first model I implemented in which almost everything was left as default[4]. The model consists of:

1. **Conv2D** layer with 32 filters and kernel_size of 3×3;
2. **MaxPooling2D** layer with a pool_size of 2×2;
3. **Conv2D** layer with 64 filters and kernel_size of 3×3;
4. **MaxPooling2D** layer with a pool_size of 2×2;
5. **Conv2D** layer with 64 filters and kernel_size of 3×3;
6. **MaxPooling2D** layer with a pool_size of 2×2;
7. **Flatten** layer;
8. **Dense** layer with 64 nodes;
9. **Dense** layer with 1 node that serves as output.

Each layer uses the **ReLU** activation function except the output layer which uses a **sigmoid**, as discussed in subsection 2.2. The model was compiled with the **Adam** optimizer and the **binary cross-entropy** loss. **Epochs** are 20, **batch_size** is equal to 32 and the validation set is obtained by taking a casual 10% of the training set. Results are summarised below.

As we could have expected, the model suffers heavily from overfitting: while the training loss tends to zero, the validation loss increases steadily. Similarly, the training

[4]With this I am referring to parameters like the *stride* or the *padding* in the convolution layers.

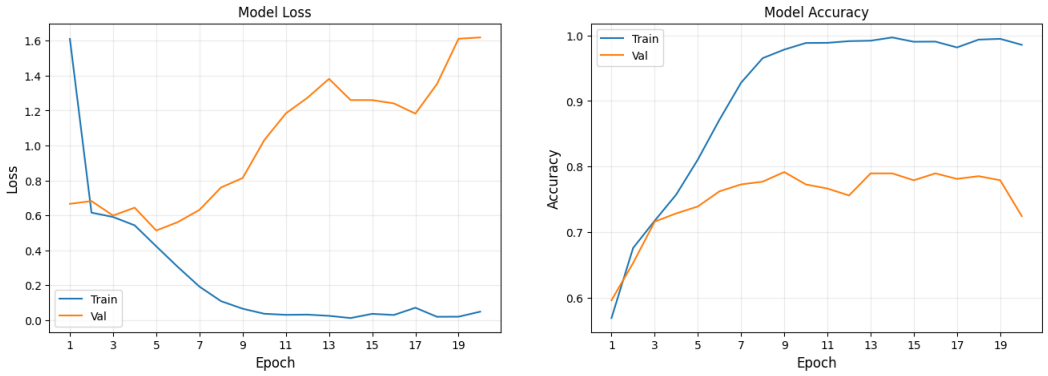

Figure 6: First model, loss and accuracy

accuracy increases almost up to 1 while validation accuracy never goes beyond 0.8. Having regard to the results, it is crucial to apply some methods to reduce overfitting like a dropout, a BatchNorm layer or some regularization. For example, I tried to add a dropout after the dense layer setting $p = 0.5$ and got the plots in Figure 7.

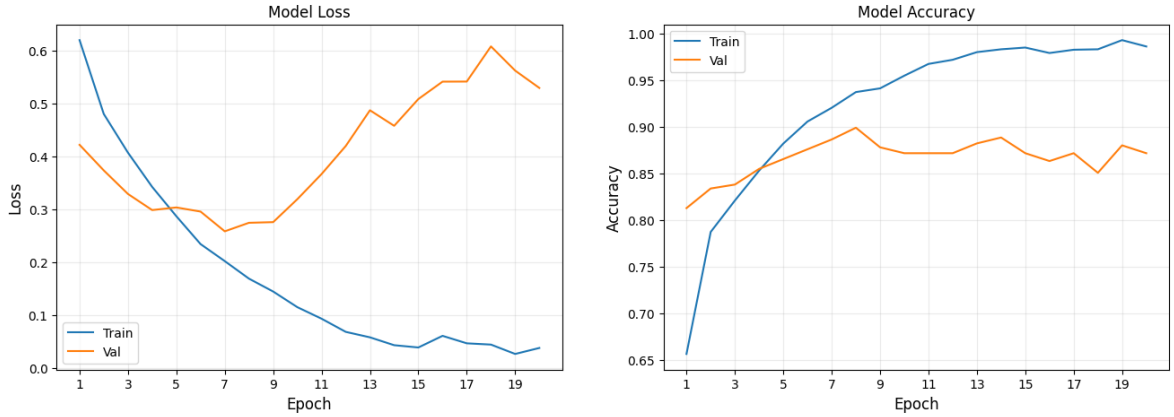

Figure 7: First model + Dropout layer, loss and accuracy

Clearly there is an improvement with respect to Figure 6 for both validation loss and accuracy: the latter grew up to 0.90. Moreover, I tried to implement the suggestions showed in [1], by inserting a BatchNorm layer instead of a dropout and depicted the results in the following plots.

The model is extremely unstable and did not even perform better as hoped: validation loss stabilised around 0.6 and accuracy fluctuated around 0.85. Finally, by

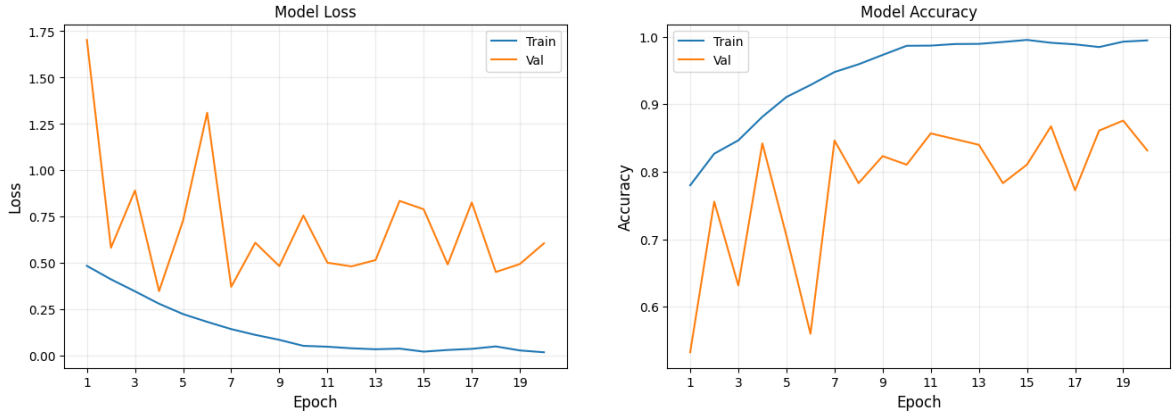

Figure 8: First model + BatchNorm layer, loss and accuracy

comparing the BN model and the Dropout model, we may conclude that BN performed worse by every point of view.

However, even considering the model of Figure 7, it simply cannot obtain good performances due to the big overfitting problem that it has, further analyses and tuning are needed.

3.2 Second model

A way to fix the problems of the previous model may be to decrease the number of filters and raise the dropout probability. Given these considerations, the second model will be characterized by the following architecture:

1. **Conv2D** layer with 32 filters and kernel_size of 3×3;
2. **MaxPooling2D** layer with a pool_size of 2×2;
3. **Conv2D** layer with 32 filters and kernel_size of 3×3;
4. **MaxPooling2D** layer with a pool_size of 2×2;
5. **Conv2D** layer with 64 filters and kernel_size of 3×3;
6. **MaxPooling2D** layer with a pool_size of 2×2;

7. **Flatten** layer;
8. **Dense** layer with 64 nodes;
9. **Dropout** layer with $p = 0.7$;
10. **Dense** layer with 64 nodes;
11. **Dense** layer with 1 node that serves as output.

As before, each layer is equipped with a **ReLU** activation function except the output, which has a **sigmoid**. The optimizer is again **Adam**, the loss is **binary cross-entropy**, the **epochs** are 20, the **batch_size** remains 32 and the validation set is obtained by a casual 10% of the original training set. Results are summarised in the following plots.

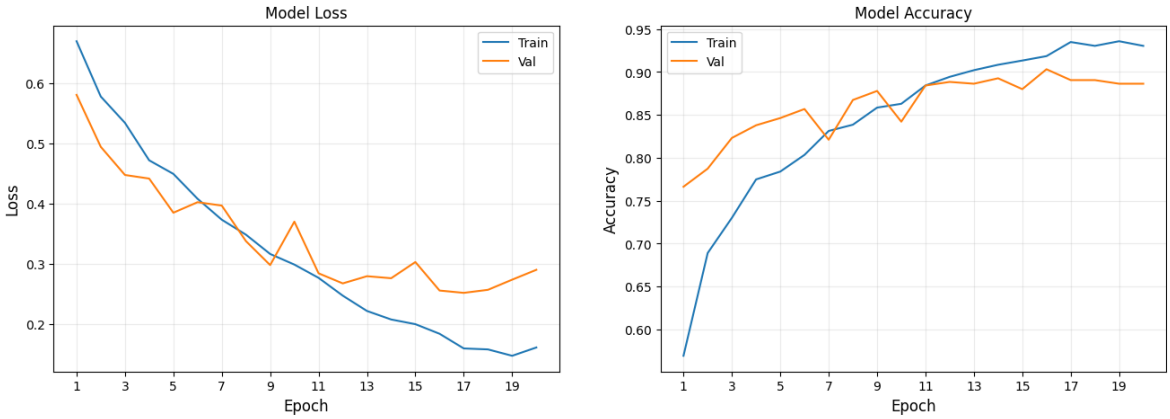

Figure 9: Second model, loss and accuracy

It is immediate to notice that there is a clear improvement with respect to Figure 7, especially considering the loss. Validation accuracy remained quite high throughout the whole process, stabilizing just below 0.90. Similarly, validation loss performed well decreasing significantly with respect to the previous model. In the end, we got: ***val_accuracy* = 0.8863** and ***val_loss* = 0.2901**.

This model seems promising, it would be interesting to increase the number of epochs and to initialize an **early_stopping** callback to see how much it is able to improve. Essentially, the early stopping is a class that keeps track of model performances and

stops the learning process when there is no significant improvement in a parameter of reference i.e. validation accuracy or loss. Before doing so however, I would first address the problem of overfitting since, despite it is better than Figure 7, it is still present. To achieve this goal, instead of increasing the dropout probability, I implemented a **kernel l2 regularization**, increased the number of *epochs* to 60 and introduced an *early_stopping* with a patience of 15 epochs[5]. To be precise, I decided to apply the l2 regularization on the first fully connected layer after the flatten one, layer number 8 of the last numerated list. Results are summarised in the following plots.

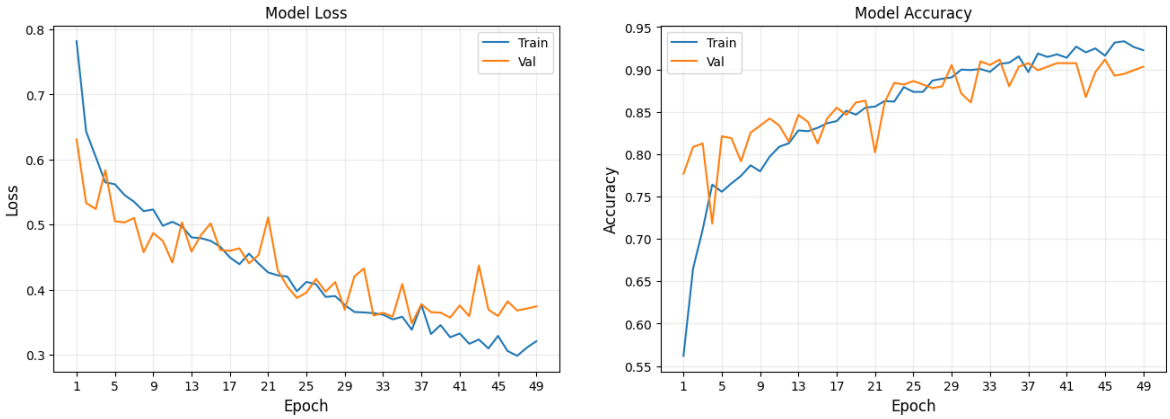

Figure 10: Second model + l2 regularization, loss and accuracy

In this model, overfitting and underfitting are almost non-existent and, according to the validation accuracy, this is the best model obtained so far. The learning process was stopped at epoch 49 and the model stored the weights it had at epoch 37, in which it achieved a very good level of validation accuracy of **0.9116**. Moreover, the losses jointly decreased throughout all the epochs except in the last ones. Actually, considering that they have a maximum difference of around 0.1, it is not a real issue.

Now, since the model performed well already by itself, it would be interesting to see which results it can reach if hyperparameters are tuned accordingly.

[5]This means that if, say, at epoch 32 *val_accuracy* is 0.91 and after 15 epochs, at epoch 47, *val_accuracy* never increased, then the process is stopped at epoch 47 and the model is stored with the weights it had at epoch 32, when it reached the maximum level of *val_accuracy*.

3.2.1 Hyperparameter tuning

To perform the optimization, I will change one parameter at the time, leaving everything else as it is unless explicitly stated, recall that the *epochs* are 60 and the *batch_size* is 32. Moreover, the models will be fitted with the *early_stopping* callback monitoring the validation accuracy with a patience of 15.

The first parameter I will optimize is the **learning_rate** of the **Adam** optimizer, checking the values $\{0.01, 0.001, 0.0001\}$. I will briefly discuss in the following bullet points the results, without inserting superfluous plots.

- **learning_rate = 0.01**: in this case, the model seems unable to learn since neither the training accuracy nor the validation accuracy were decreasing, they remained almost constant at around 0.53.
- **learning_rate = 0.001**: see Figure 10.
- **learning_rate = 0.0001**: overfitting is close to zero since the validation accuracy follows closely the training accuracy, same goes for the loss. The best value for the validation accuracy, achieved at the 60^{th} epoch, was **0.88** and for the validation loss **0.3592**. While the loss is slightly better than the one obtained in Figure 10, the accuracy is simply too low and the learning process is quite slow. Probably this model could be improved by increasing the number of convolution layers, allowing it to learn faster but this is not the point of the analysis.

After these comments we can quite confidently state that the default value equal to **0.001** is indeed the best one.

Now my goal is to optimize the number of **convolution filters**. In this case, I assumed that the number of filters could be in the set $\{32, 64, 128\}$. Thus, I created a *for-loop* which, at each cycle, trains a model using as filters a combination of those three numbers i.e. $\{32, 32, 32\}, \{32, 32, 64\}, \ldots, \{128, 128, 128\}$, resulting in 3^3 models trained. Since the number of models to check is relatively high, I decided to set the value of the *patience* parameter of the early_stopping callback to 10 to save some time.

Results suggest that the best numbers of **Conv2D filters** are, respectively **64, 64, 128**. So, with these filters, we have the following plots:

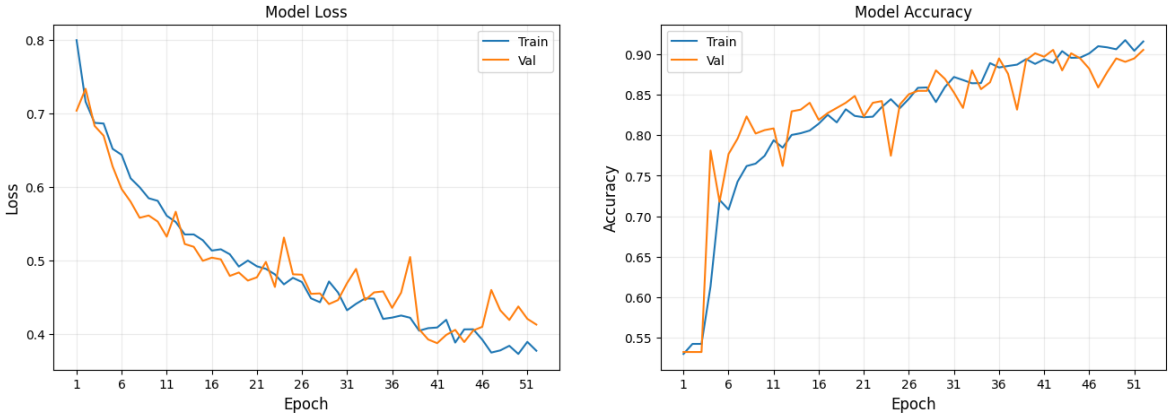

Figure 11: Second model, filters tuned, loss and accuracy

The model performed quite well, reaching a validation accuracy of **0.905** at the 42^{nd} epoch. Not a bad result but we could probably obtain something better if we also tune the **dropout probability**. To do so, I checked the values $\{0.4, 0.5, 0.6, 0.7\}$ and found that $\boldsymbol{p = 0.6}$ generates the best validation accuracy, equal to **0.9158**. Losses and accuracies are shown below.

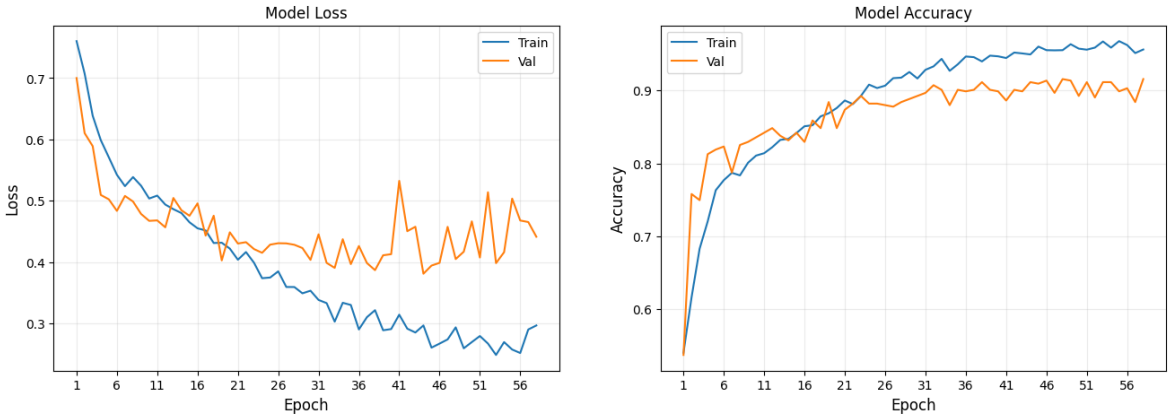

Figure 12: Second model, dropout tuned, loss and accuracy

The training loss decreased steadily going even lower than 0.3 but unfortunately validation loss, after a good decrease, stabilized around 0.45. On the contrary, accuracy

is quite promising even though, similarly to the loss, the validation one stabilized around a fixed value.

The next step is the tuning of the hyperparameter $\boldsymbol{\lambda}$ of the **l2 kernel regularization**. I grid-searched the values $\{0.1, 0.01, 0.001, 0.0001, 0.00001, 0\}$ where, the closer λ is to 0, the lower the impact of regularization. After having run the models, I found that the model that yields the highest validation accuracy is $\boldsymbol{\lambda = 0.01}$, in which $\boldsymbol{val\_accuracy = 0.9242}$ and $\boldsymbol{val\_loss = 0.3361}$. Results are depicted in the following plots.

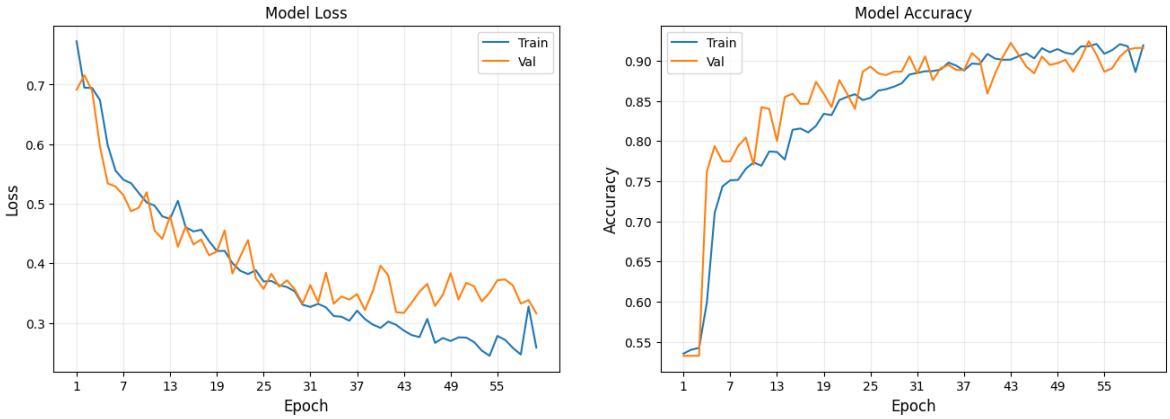

Figure 13: Second model, λ parameter tuned, loss and accuracy

Among the left parameters to tune, there is the number of nodes in the fully connected layers. The checked values are in the set $\{32, 64, 128\}$ and, similarly to what I did for convolutional filters, I checked combinations like $\{32, 32\}, \{32, 64\}, \ldots, \{128, 128\}$ for a total of 3^2 models. The highest validation accuracy was reached with **64** nodes on the first dense and **64** on the second one, in particular $\boldsymbol{val\_accuracy = 0.9136}$ and $\boldsymbol{val\_loss = 0.4470}$. Results are showed below.

One of the last hyperparameters to tune is the **kernel_size** of the first convolution layer. I am not going to check the best kernel size for the other convolution layers because I assume that 3×3 is indeed the best one for every layer different from the first. Notice that before the following computations, I reset the *patience* value of the early_stopping callback to 15. Results showed that, indeed, the best kernel_size is $\mathbf{3 \times 3}$

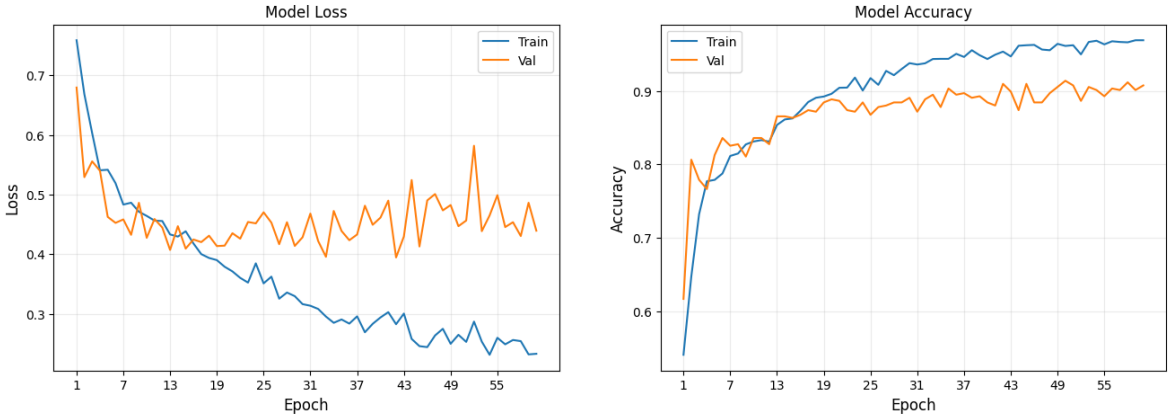

Figure 14: Second model, dense nodes tuned, loss and accuracy

which showed a great stability around $\boldsymbol{val\_accuracy \approx 0.9}$ in the last epochs and also acceptable levels of overfitting. Moreover, it seems that there is no need to increase the *stride* and thus to introduce *padding* because by leaving the *stride* equal to 1, each time the kernel moves exactly 1 pixel to the right – or down and this ensures that each pixel is seen by the convolution layers at least once for each "kernel-filter multiplication". Furthermore, as explained in [3], if many convolution layers are concatenated one after the other, each with a 3×3 kernel_size, the field of view of the layers enlarges without increasing the number of trained parameters resulting in better overall performances.

The next step is to tune the **batch_size**, choosing among $\{32, 64, 128\}$. According to the results, $\boldsymbol{batch\_size = 64}$ yielded the best validation accuracy, remaining stable around 0.9 never decreasing below 0.88 after the 35^{th} epoch. Moreover, the loss performed better than the previous models stabilising around 0.45. Results are shown below.

3.3 Final model

After having performed hyperparameter tuning, the best model I found has the following architecture:

1. **Conv2D** layer with 64 filters, kernel_size of 3×3, stride of 1 and no padding;

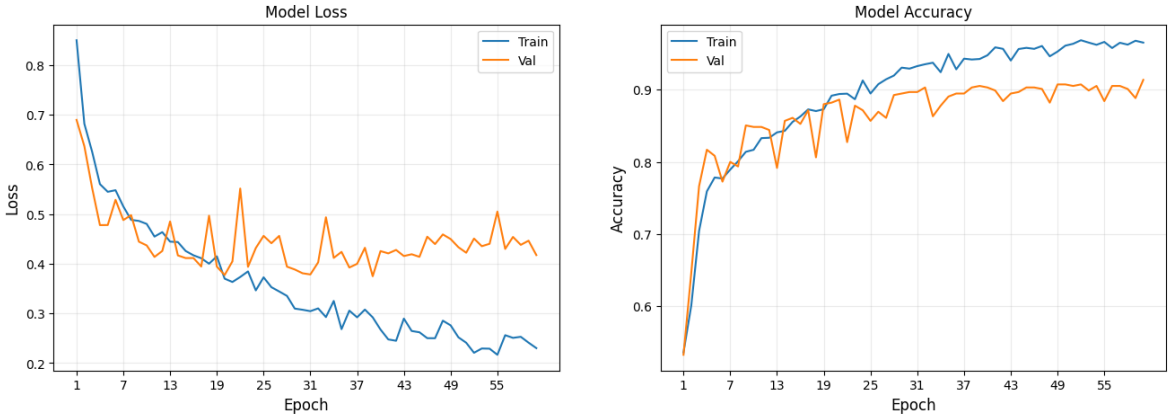

Figure 15: Second model, batch_size tuned, loss and accuracy

2. **MaxPooling2D** layer with pool_size of 2×2, stride of 2 and no padding;
3. **Conv2D** layer with 64 filters, kernel_size of 3×3, stride of 1 and no padding;
4. **MaxPooling2D** layer with pool_size of 2×2, stride of 2 and no padding;
5. **Conv2D** layer with 128 filters, kernel_size of 3×3, stride of 1 and no padding;
6. **MaxPooling2D** layer with pool_size of 2×2, stride of 2 and no padding;
7. **Flatten** layer;
8. **Dense** layer with 64 nodes and kernel_regularizer l2 with $\lambda = 0.01$;
9. **Dropout** layer with $p = 0.6$;
10. **Dense** layer with 64 nodes;
11. **Dense** layer with 1 node that serves as output.

Each layer is equipped with a **ReLU** activation function except for the output layer which has a **sigmoid**. Other paramters are:

- **Loss**: binary-crossentropy;
- **Optimizer**: Adam with $learning\_rate = 0.001$;

- **Batch_size**: 64;
- **Epochs**: 60;
- **Early_stopping**: patience of 15 epochs.

Results are depicted in the plots below. In particular, we have that the model reached: $\boldsymbol{val\_accuracy = 0.9263}$ and $\boldsymbol{val\_loss = 0.4769}$, according to the accuracy this is the best model obtained so far. Moreover, on the accuracy-side, overfitting does not seem a a big problem but it becomes worse on the loss-side where the difference between validation loss and training loss reaches, at its maximum, 0.2.

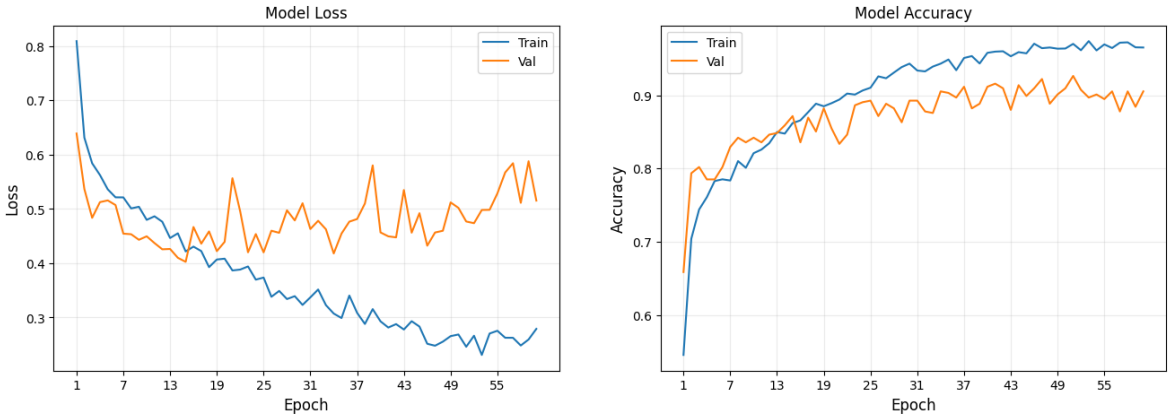

Figure 16: Final model, loss and accuracy

Nevertheless, the model seems quite stable and performs well on data that has never seen before since the evaluation over test-data is totally in line with what was measured for the validation set. Indeed:

- **test loss**: 0.4995
- **test accuracy**: 0.9119

In particular, the confusion matrix is shown in Figure 17.

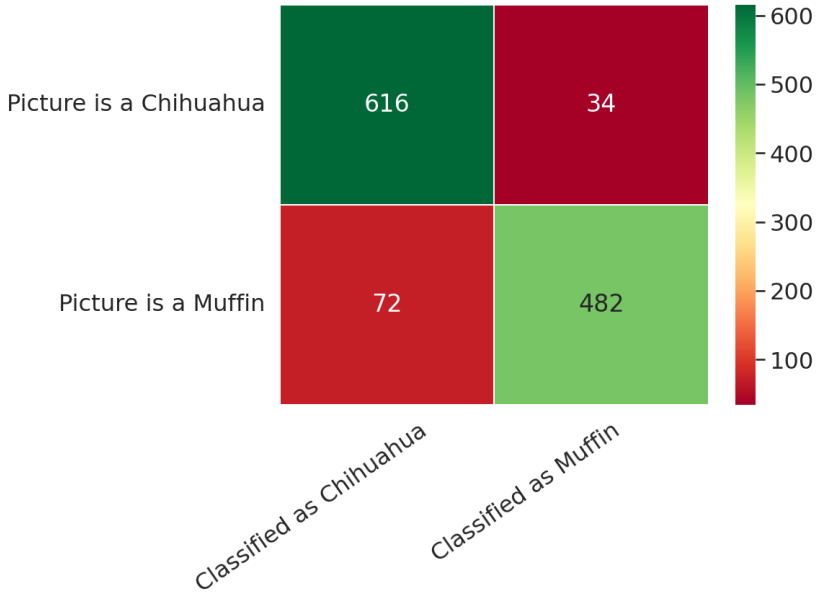

Figure 17: Confusion matrix for the final model

The model better classifies images of chihuahua rather than images of muffins. Indeed, only **5.231%** of chihuahua images are misclassified while, for muffins, this percentage is **12.996%**. One explanation behind this may be that, as said in subsection 1.1, in the training set there are about 400 more images of chihuahuas with respect to muffins, making the set unbalanced. Overall, the total percentage of misclassified images is **8.039%**.

Lastly, I performed a **5-fold Cross-Validation** upon the final model to compute the risk estimates using the **0-1 loss**. Results are summarised in the following table.

Table 1: Cross-Validation results

K-Fold	**0-1 Loss**
Fold 1	0.1043
Fold 2	0.1085
Fold 3	0.1032
Fold 4	0.1002
Fold 5	0.1075
Average	**0.1047**

The model is extremely stable with a value for the loss in line with what we have found previously: it is able to correctly classify about 90% of all the test pictures.

4 Conclusions

In this project I had to implement, through Tensorflow's API Keras, some Convolutional Neural Network (CNN) to correctly classify a dataset composed by 4,743 images representing either chihuahuas or muffins.

After a brief presentation of the dataset and of the preprocessing methods, I analysed the basic theoretical concepts behind Neural Networks, with a particular focus on CNNs.

In the third section I focused on the real CNN implementation, trying to improve model's performances at each modification through the hyperparameter tuning. Unfortunately this was not always the case, especially for the validation loss. I have been wondering a lot about why this was the case. Still, given the *sigmoid* activation function depicted in Figure 5, it may be that the model predicts an X-value very close to zero, hypothetically in the interval $[-2, +2]$ in which the **certainty** of the model is the lowest, allowing the model to *guess* which is actually represented in the image. This may generate situations in which the model predicts an image to be, say, a chihuahua with just 0.51 probability. If the image is, indeed, a chihuahua then everything is fine,

otherwise the model will be severely "punished" with a significant increase in the loss function. Actually, since the model has both a high loss and a high accuracy we can say that it is an able guesser.

However, since the final model was good enough to correctly classify around 9 images out of 10, I would consider this a good result given the computational power at my disposal and my level of experience. Yet, I am aware that there is still room for improvement, especially regarding the loss which should be reduced in some ways.

References

[1] Christian Garbin, Xingquan Zhu, and Oge Marques. Dropout vs. batch normalization: An empirical study of their impact to deep learning. *Multimedia Tools Appl.*, 79(19–20):12777–12815, May 2020.

[2] Sungheon Park and Nojun Kwak. Analysis on the dropout effect in convolutional neural networks. In Shang-Hong Lai, Vincent Lepetit, Ko Nishino, and Yoichi Sato, editors, *Computer Vision – ACCV 2016*, pages 189–204. Springer International Publishing, March 2017.

[3] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.