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 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 classify 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 **1**

1.1 Exploratory analysis and preprocessing 1

2 Convolutional Neural Network implementation **3**

2.1 First model 3

2.2 Second model 6

2.2.1 Hyperparameter tuning 8

2.3 Final model 13

3 Conclusions **16**

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[1].

In the following sections I will deeply explain the methodology and the architectures used, giving a complete explanation to every decision.

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.

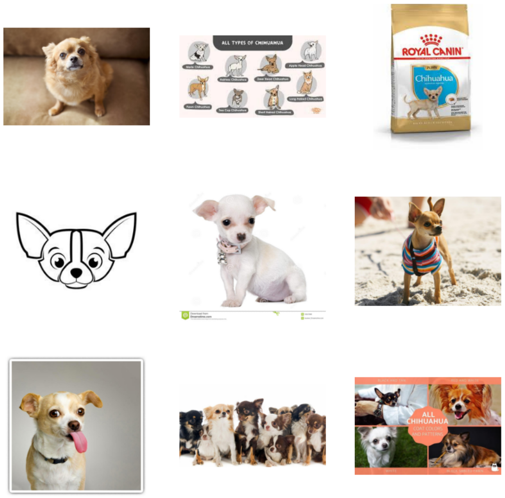

Figure 1: Example pictures of Chihuahuas

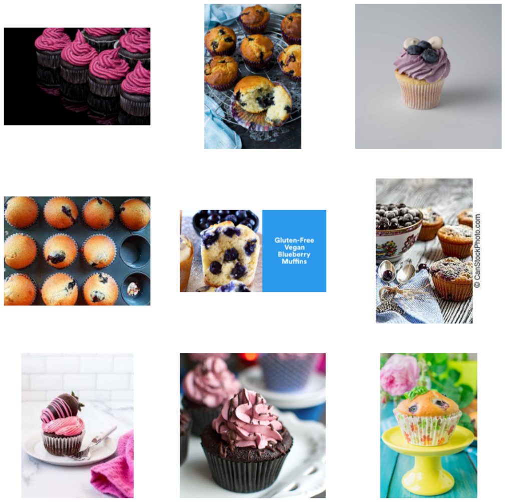

Figure 2: Example pictures of Muffins

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.

[1] The Python code is completely available on my GitHub profile.

- 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*.

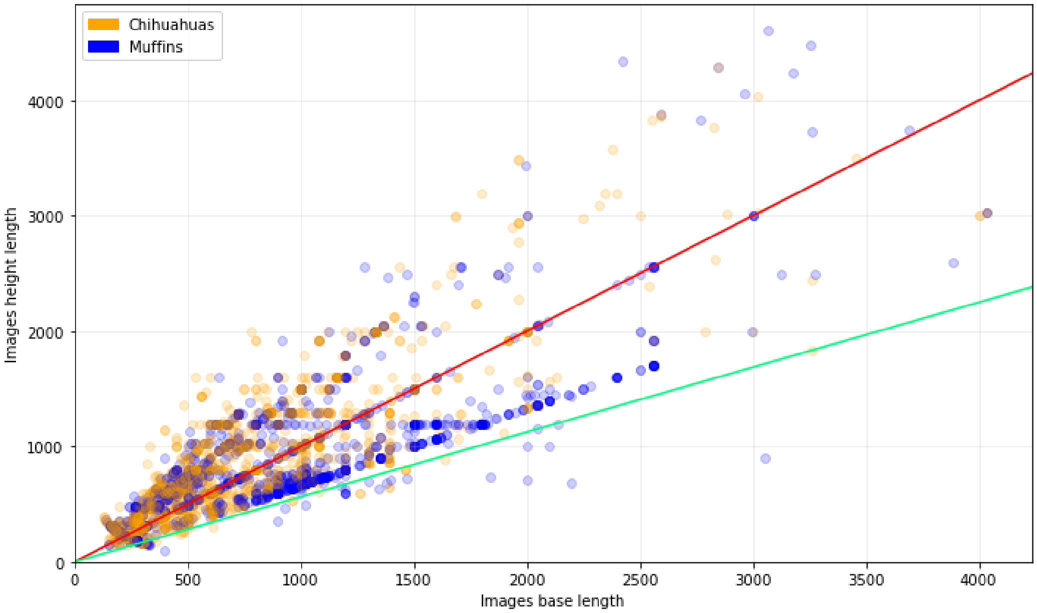

Figure 3: Sizes of the non scaled images in pixels

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 × 150px. Subsequently, I memorized the pictures into a NumPy array and turned them into grayscale. By turning RGB pitcures to grayscale, we lose color information but, since muffins and chihuahuas are more or less of the same yellowish color, I think color 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 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.

2.1 First model

In this subsection I will analyse the first model I implemented in which almost everything was left as default[2]. 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;

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

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**. 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.

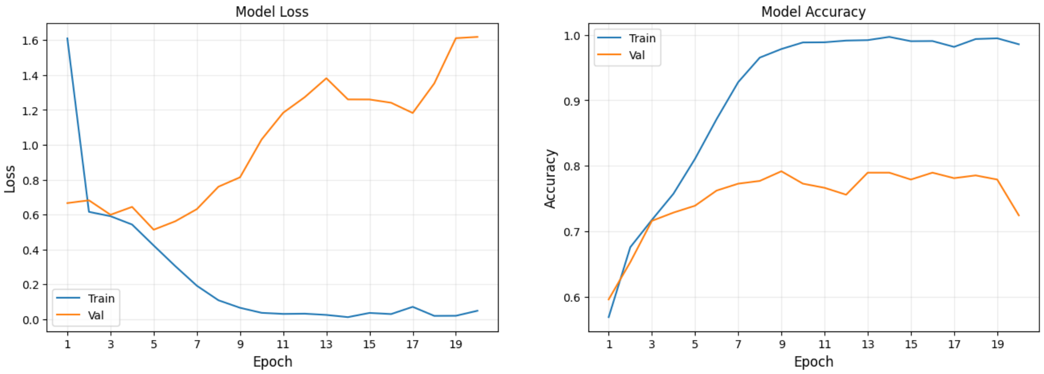

Figure 4: First model, loss and accuracy

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 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 5.

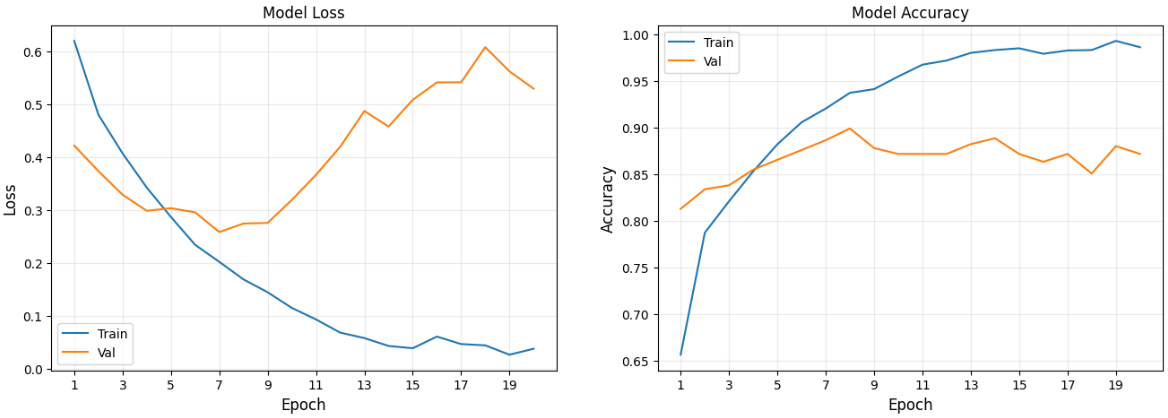

Figure 5: First model + Dropout layer, loss and accuracy

Clearly there is an improvement with respect to Figure 4 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, which should result in a better performance without the need of changing the architecture.

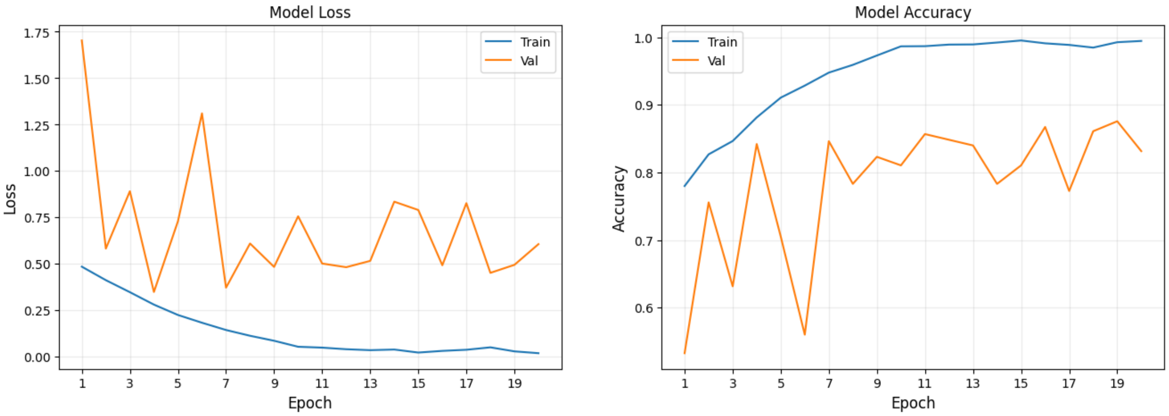

Figure 6: First model + BatchNorm layer, loss and accuracy

Actually, 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 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 5, it simply cannot obtain good performances due to the big overfitting problem that it has, further analyses and tuning are needed.

2.2 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.

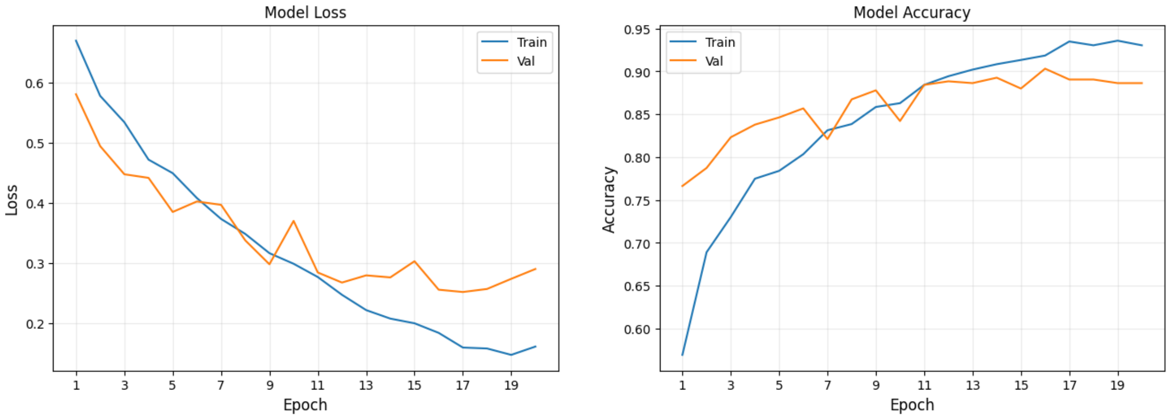

Figure 7: Second model, loss and accuracy

It is immediate to notice that there is a clear improvement with respect to Figure 5, 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 5, 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[3]. 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.

[3] 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*.

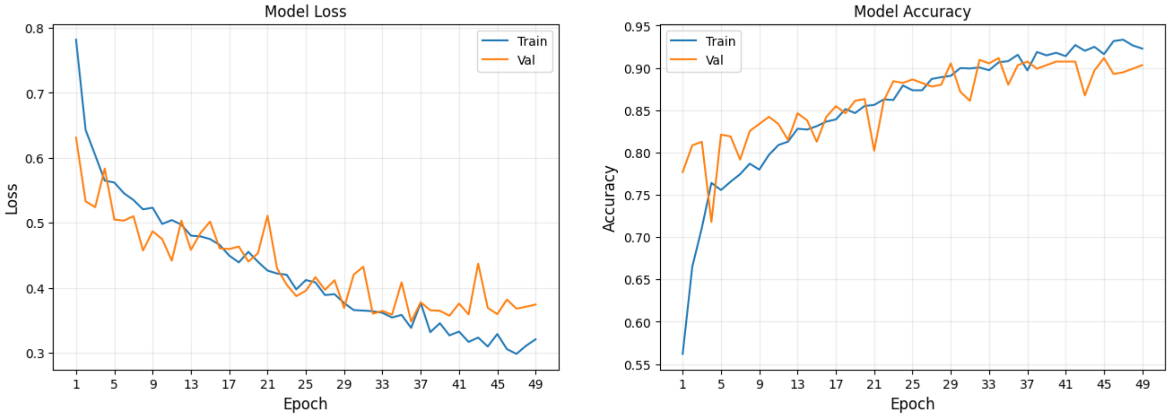

Figure 8: Second model + l2 regularization, loss and accuracy

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.

2.2.1 Hyperparameter tuning

To perform the optimization, I first 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.

In this analysis, everything will be left as it is unless explicitly stated and recall that *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 8.
- **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 8, 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:

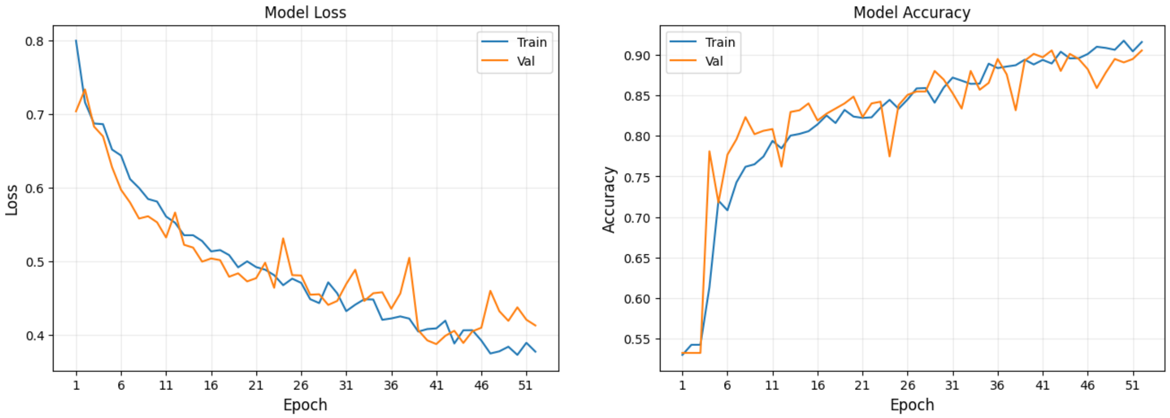

Figure 9: Second model, filters tuned, loss and accuracy

The model performed quite well, reaching a validation accuracy of **0.905** at the 42$^{\text{nd}}$ 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.

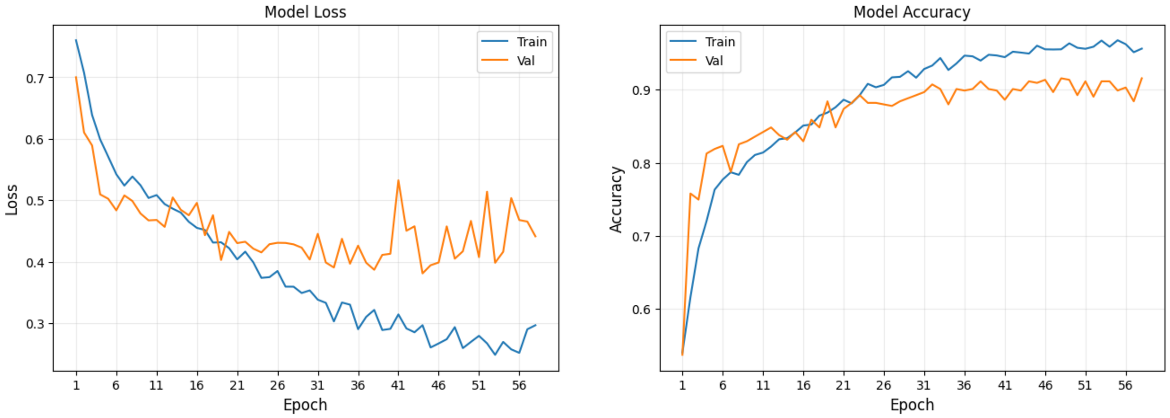

Figure 10: Second model, dropout tuned, loss and accuracy

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.

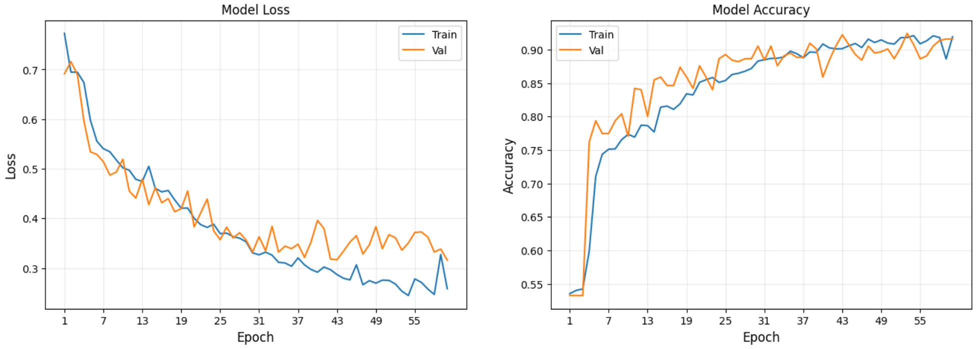

Figure 11: Second model, λ parameter tuned, loss and accuracy

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.

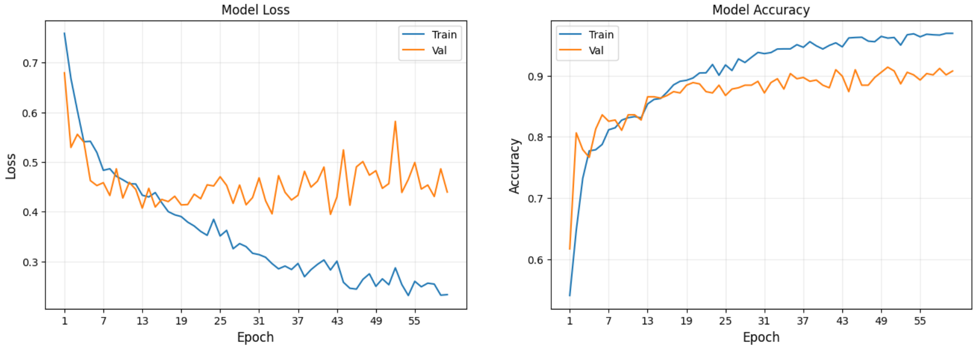

Figure 12: Second model, dense nodes tuned, loss and accuracy

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 $\mathbf{3 \times 3}$ 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 [2], 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 35th epoch. Moreover, the loss performed better than the previous models stabilising around 0.45. Results are shown below.

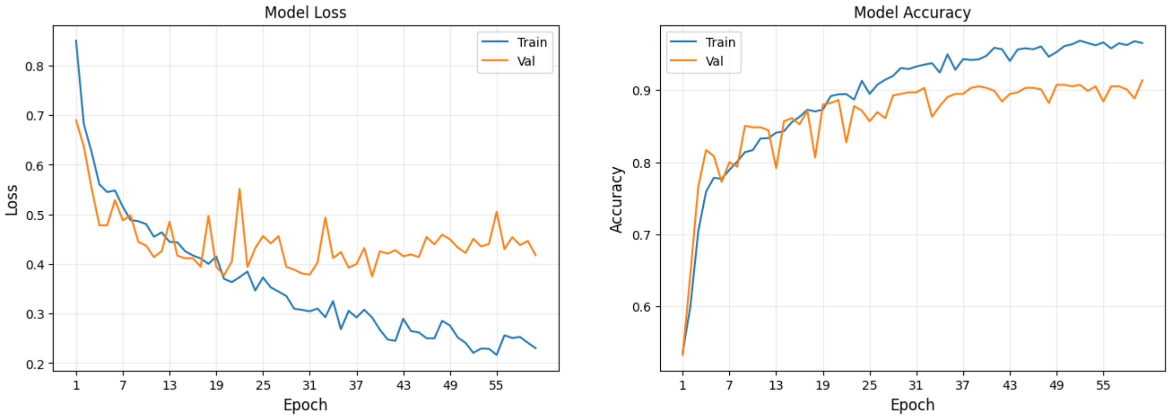

Figure 13: Second model, batch_size tuned, loss and accuracy

2.3 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;
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: $\boldsymbol{val\_accuracy} = \mathbf{0.9263}$ and $\boldsymbol{val\_loss} = \mathbf{0.4769}$, 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.

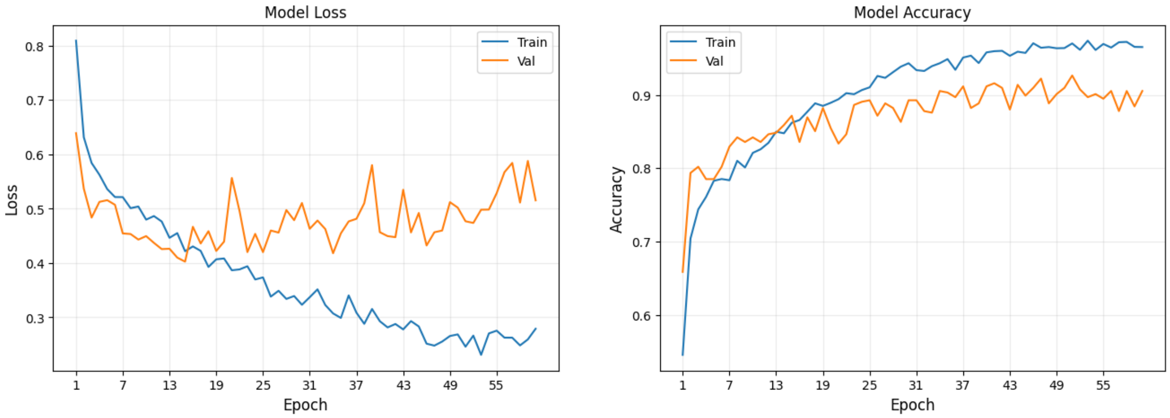

Figure 14: Final model, loss and accuracy

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 15.

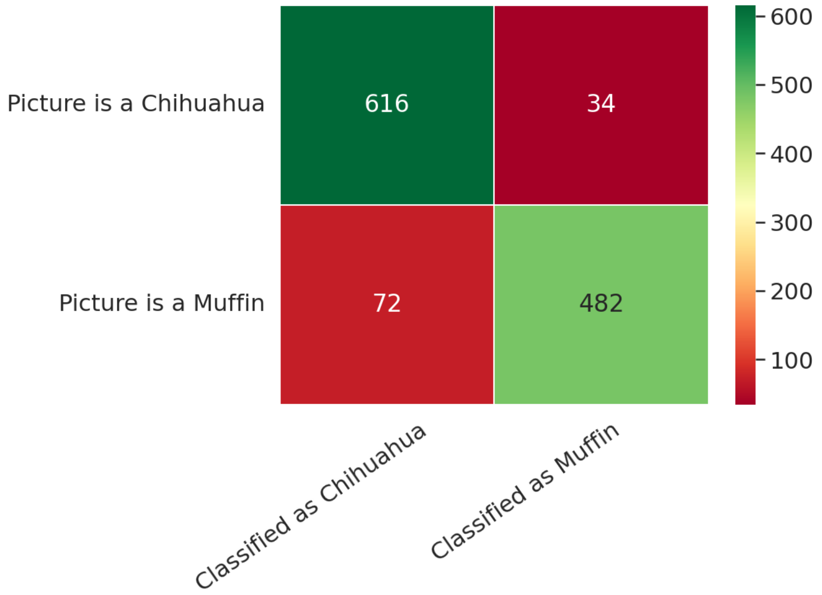

Figure 15: Confusion matrix for the final model

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.

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