Statitical methods for Machine Learning

Experimental Project

Riccardo Sturla
ID: 13457A

Academic Year 2023-2024

Abstract

The project aims to experiment with different Neural Network architectures to find the best model to correctly classify images of muffins and chihuahuas. The NNs are created on Google Colab using Keras, an high level API for Tensorflow in Python. After a brief explanation of the dataset used and the preprocessing phase, various architectures are tested and evaluated, with emphasis on the progressive improvements. The final model performances are evaluated on a test set and the risk estimates are computed with 5-fold cross validation.

Contents

1 Introduction

The goal of the project is to build a good Convolutional Neural Network for the binary classification of images of muffins and chihuahuas. The CNNs are created on Google Colab, using the TF API Keras for Python.
In the following sections of the report, I am going to briefly analyse the data, explain why Convolutional Neural Network are the best for images classification, and experiment with different architectures to finally find the best model and evaluate its performances.

1.1 Dataset

The dataset used for the experimental project is composed by 5927 images of chihuahuas and muffins, and it has already been split in a training set (4743 images) and a test set (1184 images). The amount of images per class in the training set is quite balanced with a slightly higher number of chihuahuas.

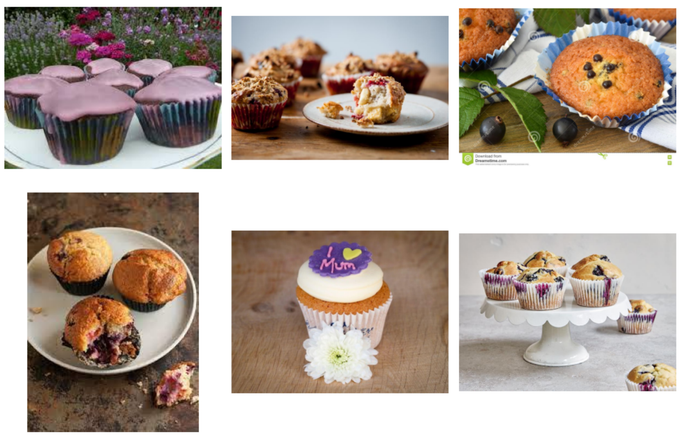


Figure 1: Six random images classified as "Muffin"

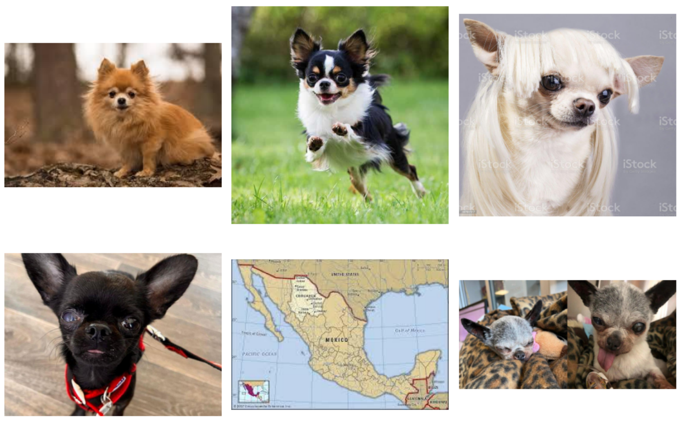


Figure 2: Six random images classified as "Chihuahua"

As shown in Figure 1 and 2, the images are very varied and they not always represent actual muffins or chihuahuas, indeed, analyzing the data, it is possible to find some "noise", with wrong classified images (as the map of Mexico in Figure 2). However, such kind of data represent a small percentage of the total and I decided not to remove them. Another important fact to notice is that the images can also picture drawings or sketches of dogs and muffins, and not the real subjects.

1.2 Preprocessing

Images are not of the same size and have different orientations, so the first task to implement is their scaling and resizing. The images are resized to be of size 150 x 150 pixels, then they are memorized into a numpy array and turned into grayscale. Turning RGB images into grayscale ensures to have one layer of depth, instead of three, making the computation much faster, with the drawback of the loss of color information (which however it could be consider not relevant in most cases, given the fact that chihuahuas and muffins are often both brown). Finally, grayscale values are normalized to be in a $[0, 1]$ interval.

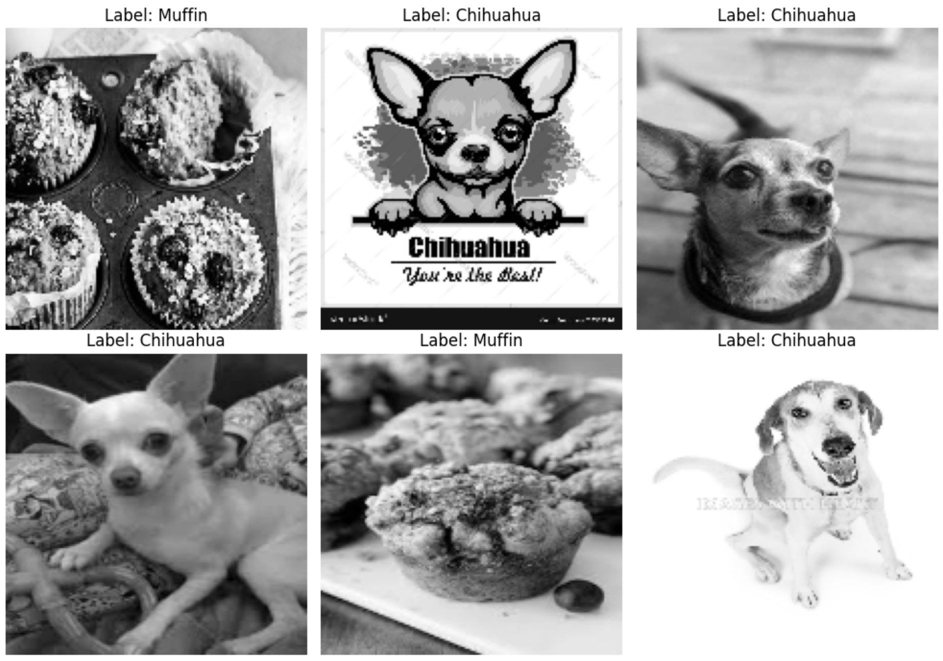


Figure 3: Preprocessed images

2 Theoretical Overview[1]

2.1 Why to use Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are analogous to traditional Artificial Neural Networks as they are comprised of neurons that self-optimise through learning. Each neuron will still receive

[1]Most of the explanations in this section are taken from the paper *"An Introduction to Convolutional Neural Networks"* by Keiron O'Shea and Ryan Nash (see References)

an input and perform a operation (such as a scalar product followed by a non-linear function) as countless ANNs do. From the input raw image vectors to the final output of the class score, the entire of the network will still express a single perceptive score function, i.e. the weight. The last layer will contain loss functions associated with the classes, and all of the regular tools developed for traditional ANNs still apply. The only notable difference between CNNs and traditional ANNs is that CNNs are primarily used in the field of pattern recognition within images. This allows to encode image-specific features into the architecture, making the network more suited for image-focused tasks. Traditional forms of ANN tend to struggle with the computational complexity required to compute image data. Increasing the number of hidden layers in our network is not likely to solve the issue. This is down to two reasons, one being the simple problem of not having unlimited computational power and time to train these huge ANNs. The second reason is stopping or reducing the effects of overfitting. The less parameters required to train, the less likely the network will overfit and improve the predictive performance of the model.

2.2 CNNs architecture

Basic CNNs are mainly comprised of three types of layers. These are convolutional layers, pooling layers and fully-connected layers.

- **Convolutional layers**: computes the convolutional operation of the input images using kernel filters to extract fundamental features.
- **Pooling layers**: aim to gradually reduce the dimensionality of the representation, and thus further reduce the number of parameters and the computational complexity of the model. The pooling function can be max or average. Max pooling is commonly used as, in most of the cases, it works better.
- **Fully-connected layers**: also called **Dense**, because every neuron in a the layer will be fully connected to every neuron in the prior layer. Dense layer is used to classify image based on output from convolutional layers.

3 Convolutional Neural Network

3.1 Base Model

I started the analysis with a basic CNN model as a benchmark to evaluate further results and improvements. Such model is composed of:

- 3 **Conv 2D** layers with respectively 32, 64 and 64 kernel filters of size 3x3. Each layer use the Relu activation function and the first one takes as argument the input shape of the data (images). No padding is used.
- 3 **MaxPooling2D** 2x2 (pool size) layers, one after each Convolutional layer.
- A **Flatten** layer which takes the output of the previous layers, “flattens” them and turns them into a single vector that can be an input for the next stag.
- A **Dense** layer with 64 units and Relu activation function and a final dense layer with 1 unit and **Sigmoid** activation function. The Sigmoid activation function gives as output a value between 0 and 1, that makes it the perfect solution for binary classification problems.

All the models are compiled with the **binary cross-entropy loss** and the **Adam** optimizer. Binary Cross Entropy is a commonly used loss function in machine learning, particularly in binary classification problems. It is designed to measure the dissimilarity between the predicted probability distribution and the true binary labels of a dataset.
Adam optimizer is a method belonging to the stochastic gradient descent family, that is based on adaptive estimation of first-order and second-order moments and it has been proven to perform very well in CNNs.
The batch size, which is the number of samples per gradient update, for the base model is set to 32. The train will always be done using a 10% validation split.

3.1.1 Results

The performances of the model are presented in terms of loss (left side graph in Figure 4) and accuracy (right side graph of Figure 4).

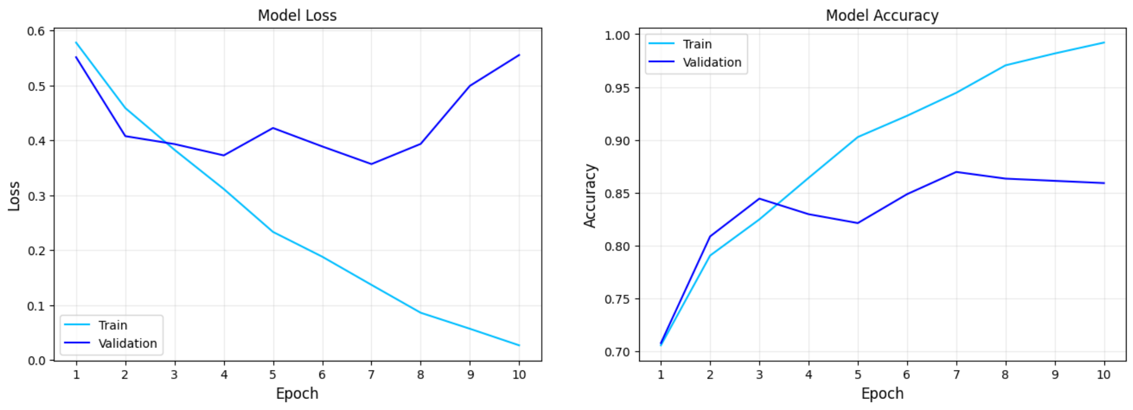
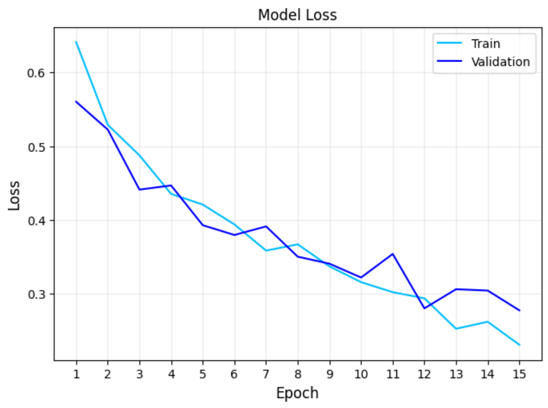


Figure 4: Base model results

The model presents an obvious overfitting problem with train and validation metrics starting to diverge after just 4 epochs. Moreover, the accuracy on the training set reaches 99% very quickly, while the validation accuracy stays under 86%.

3.2 Improvements and new architectures

A commonly used method to reduce overfitting in machine learning is a regularization technique called **"Dropout"**. Dropout is a technique where randomly selected neurons are ignored during training: they are “dropped out” randomly. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass, and any weight updates are not applied to the neuron on the backward pass. The effect is that the network becomes less sensitive to the specific weights of neurons. This, in turn, results in a network capable of better generalization and less likely to overfit the training data.
The new model incorporating dropout is built as the base model with two dropout of 60%, one after the last *Conv2D* layer and one after the *Flatten* layer.



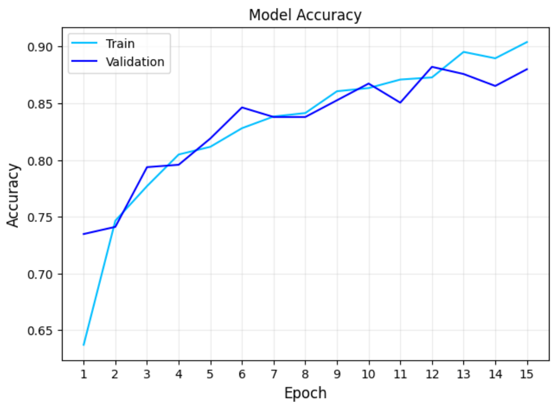


Figure 5: Results of the model with dropout

Adding dropout significantly reduce the overfitting of the algorithm, which reaches an accuracy of 88% on the validation test in the last epoch. Running the algorithm for more epochs (30), however, leads it again to some overfit, while the general accuracy remains pretty much the same.
Regarding the number of epochs used to train the model, from now on an **early stopping** criteria is introduced. Early stopping is a regularization technique theoretically used to stop the training right before an algorithm starts to overfit. For the purpose of the project, I set an early stopping with patience 5, monitoring the validation loss, which means that the training will stop if there is no improvement in the validation loss for 5 consecutive epochs.

Another important parameter to consider which could improve the performances of the model is the **learning rate**. The learning rate is a hyperparameter that determines the size of the steps taken during the optimization process. It controls how much you want to update your weights with respect to the loss gradient. A high learning rate means taking large steps, which might cause to overshoot the minimum. On the other hand, a low learning rate means smaller steps, which can lead to slow convergence or getting stuck in a local minimum.
The next experiment introduces a learning rate of 0.0005 and also a **L2 kernel regularizer** with lambda of 0.001. The L2 regularizer is one of the most common type of regularization. It consists in updating the general cost function by adding another term known as the regularization term λ. Due to the addition of this regularization term, the values of the weight matrices decrease, so the weights decay towards 0. This will theoretically lead to simpler models and it will also reduce overfitting to quite an extent.

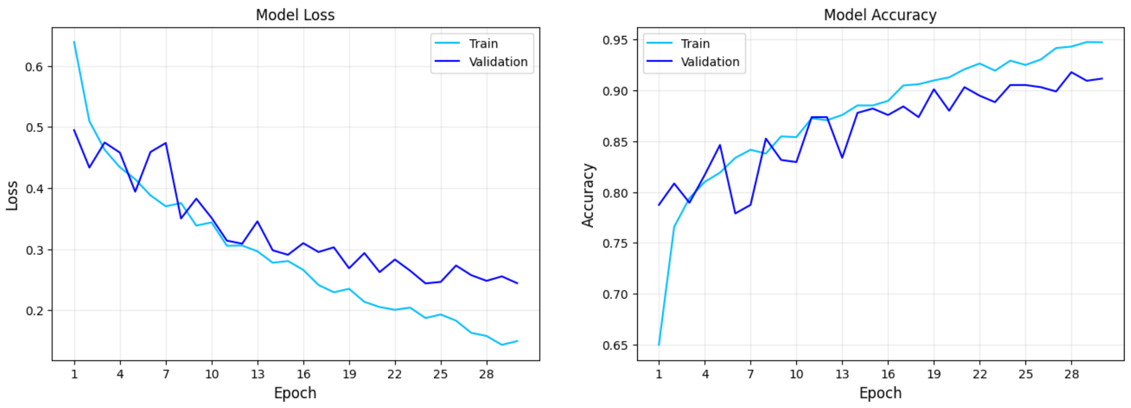


Figure 6: Results of the model with learning rate and L2 regularization

The model is able to reach almost 92% of validation accuracy with still some overfitting (around 3% of accuracy) in the last epochs: the best result so far.

Finally, I decided to add a **batch normalization** layer and to slightly change the general architecture of the model. Batch normalization works by normalizing the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation. It should speed up training and regularize the model.
The new model is structured as follow:

- 3 **Conv 2D** layers with respectively 32, 64 and 128 kernel filters of size 3x3. Each layer utilizes the Relu activation and no padding is used.
- 3 **MaxPooling2D** 2x2 (pool size) layers, one after each Conv layer.
- A 0.6 **Dropout** layer.
- A **Flatten** layer.
- Another 0.6 **Dropout** layer.
- A **Dense** layer with 128 units and Relu activation function and a final dense layer with 1 unit and **Sigmoid** activation function.
- In between the dense layers, a **batch normalization** layer is added.

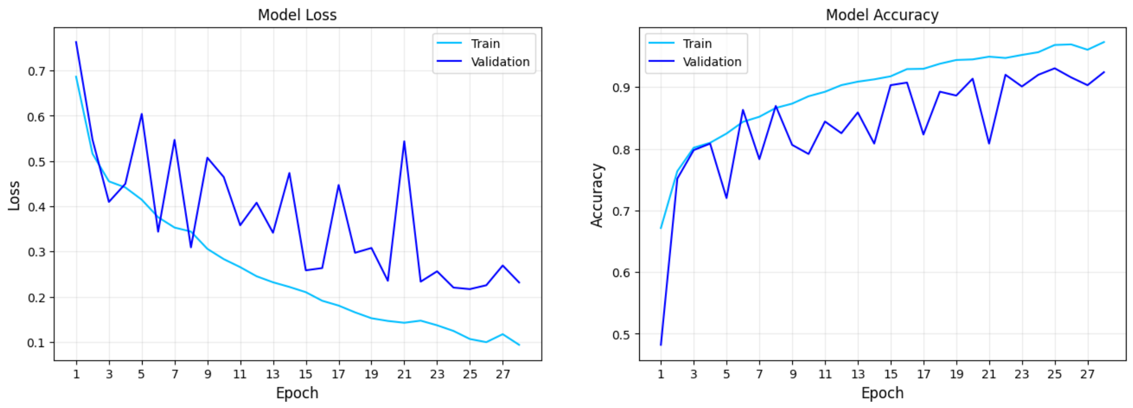


Figure 7: Results of the model with batch normalization layer

The new model seems to have general better performances with peaks of more than 93% of validation accuracy, but it does not solve the overfitting problem and it looks unstable. It could be the case to perform some better hyperparameter tuning and see if there could be some improvements.

3.2.1 Hyperparameters tuning

The tuning of the hyperparameters was firstly done through numerous attempts and then using the KerasTuner framework: the values which seemed to perform better in the previous experiments were used as possible values for the Tuner. KerasTuner uses a search algorithm to test and evaluate different hyperparameters values and select the best ones. I choose to use the Bayesian Optimization search, which keeps track of past evaluation results and use them to form a probabilistic model, mapping hyperparameters to a probability of a score on the objective function. The functioning of the algorithm is shortly described in five steps:

1. build a surrogate probability model of the objective function;
2. find the hyperparameters that perform best on the surrogate;
3. apply these hyperparameters to the true objective function;
4. update the surrogate model incorporating the new results;
5. repeat steps 2–4 until max iterations or time is reached.

The tuning with KerasTuner, given the computational capabilities that I have on the free version of Google Colab, has some limitations. Specifically, it is possible to run a limited amount of different trials and for a small amount of epochs. That causes the results to often be not efficient and to return just a possible set of values which *could* perform the best among the ones tested. With those assumptions, I tried to tune the model of Figure 7, hoping in some improvements in stability and overfitting. Unfortunately, the hyperparameters values obtained by the Tuner did not perform better.

3.3 Deeper model

Deep Neural Networks could be very powerful, but they could also lead to overfitting problem due to their complexity. In this section, I decided to experiment with a deeper architecture with respect to the ones used before. After numerous tests, the model chooses is built as follow:

- 6 **Conv 2D** layers with respectively 32, 32, 64, 64, 128 and 128 kernel filters of size 3x3. Each layer use the Relu activation. In this case, to preserve the spatial size of the input through all the convolutional layers and prevent an excessive dimensionality reduction, **padding** is used.
- 6 **MaxPooling2D** 2x2 (pool size) layers, one after each Conv layer.
- A **Flatten** layer.
- 3 **Dense** layers with 128, 64 and 32 units and Relu activation function, and a final dense layer with 1 unit and **Sigmoid** activation function.
- 2 **Dropout** of 0.6 after the two Convolutional layers with 128 filters and 1 **Dropout** of 0.7 after the first Dense layer.

L2 regularization was added on all the convolutional and dense layers with a lambda of 0.0001; the learning rate choosen was 0.0005.

The model is clearly more complex than the ones tested before. The greater number of convolutional layers and the extra dense layers could increase the capability of classify the images but, as said before, could also lead to bigger overfitting problems.

3.3.1 Results

The model was trained for 60 epochs with early stopping and batch size of 64.

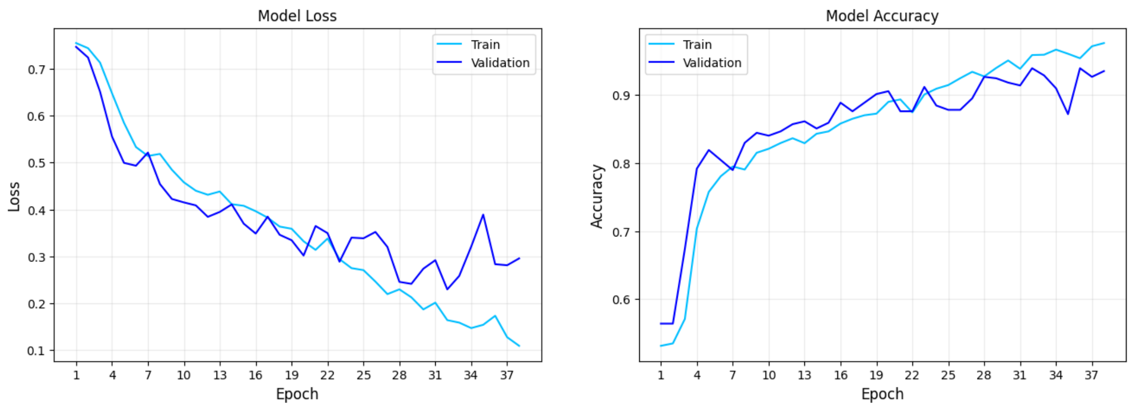


Figure 8: Results of the model

The results are pretty stable with a validation accuracy of 91/92%. However, there is still a slight overfitting in the last epochs, before the early stopping of the training. All in all, it is a good model and the increased complexity does not affect in negative terms the performances.

3.3.2 Hyperparameters tuning

For the ”deep architecture”, I've experimented with many parameters and layers, adding batch normalization, changing the l2 regularizer and everything that was possible to modify. The model in Subsection 3.3.1 is the result of such experiments, which are in fact considered as tuning.
On the other hand, hyperparameters tuning with KerasTuner suggests some changes, but, after a few tests, the model obtained seems to perform slightly worse than the original one. Once again, the automatic tuning shows to have some limitations for this kind of project and manually adjust the hyperparamters still looks like the best option.

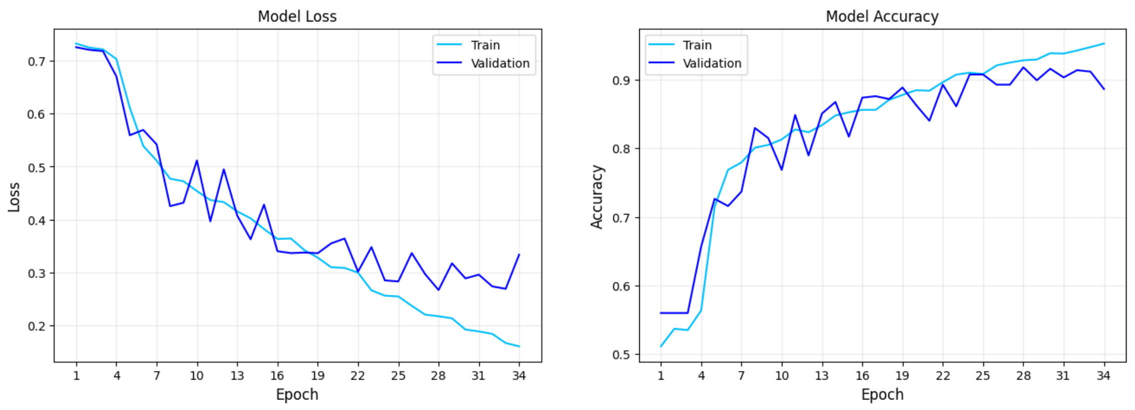


Figure 9: Results of the tuned model

4 Final Model

The final model which I choose to keep and evaluate on the test set is the one presented in Subsection 3.3.1, built as follow:

- **Conv 2D** layers with respectively 32, 32, 64, 64, 128 and 128 kernel filters of size 3x3. Each layer use the Relu activation. **Padding** is used.
- 6 **MaxPooling2D** 2x2 (pool size) layers, one after each Conv layer.
- A **Flatten** layer.
- 3 **Dense** layers with 128, 64 and 32 units and Relu activation function, and a final dense layer with 1 unit and **Sigmoid** activation function.
- 2 **Dropout** of 0.6 after the two Convolutional layers with 128 filters and 1 **Dropout** of 0.7 after the first Dense layer.
- **L2 regularizer** on all the Convolutional and Dense layers with a **Lambda** of 0.0001
- Adam optimizer with **learning rate** of 0.0005
- **Batch size** of 64 and 60 **epochs** of training

I choose to evaluate this model instead of a simpler one like the one in Figure 6, because the performances were similar, but the overfitting was less persistent and the loss was stabler.

4.1 Performance on the Test Set

The evaluation of the final model on the test set reports a loss of 0.267 and an accuracy of the 91.6%. I subsequently built the confusion matrix to show the exact classification of the images.

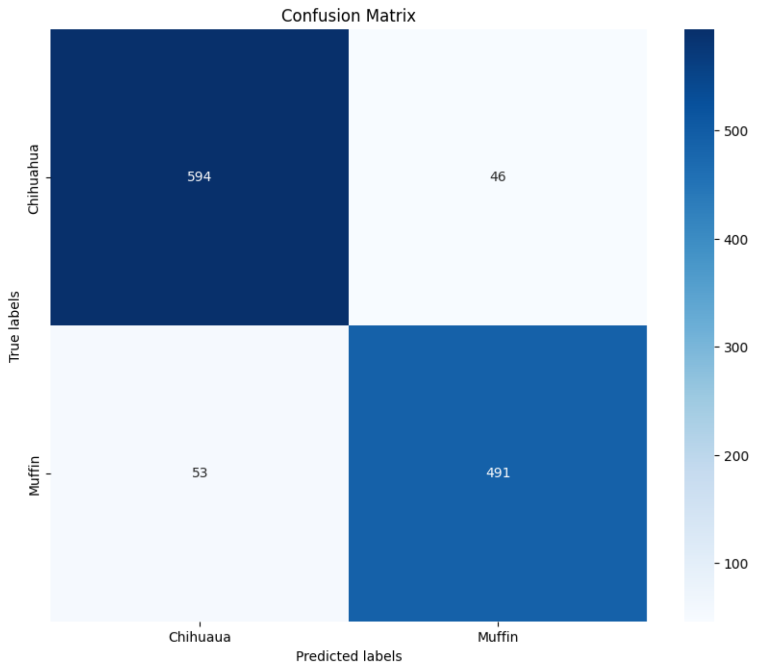


Figure 10: Confusion Matrix of the Final Model

The matrix shows a lightly higher missclassification on the images of muffins, probably due to a bias following the composition of the training dataset (more chihuahuas images). In fact **7.7%** of Chihuahuas images are wrongly classified against a **10.8%** of the Muffins. Overall, with 91.6% of accuracy, the model missclassifies **8.4%** of the images.

4.2 5-fold Cross Validation

K-fold cross validation is used to estimate the skill of a machine learning model on unseen data. A limited sample of data is used in order to estimate how the model is expected to generalize. The basic steps of K-fold CV are:

1. Shuffle the dataset randomly.
2. Split the dataset into k groups

   For each unique group:

3. Take the group as a hold out or test data set
4. Take the remaining groups as a training data set
5. Fit a model on the training set and evaluate it on the test set

6. Retain the evaluation score and discard the model
7. Summarize the skill of the model using the sample of model evaluation scores.

For the project, I used a 5-fold CV and evaluate the risk estimates using the Zero-One Loss. The results are showed in the following table:

K-Fold	**0-1 Loss**
Fold 1	0.106
Fold 2	0.075
Fold 3	0.109
Fold 4	0.089
Fold 5	0.112
Average	0.098

The cross-validation shows some light "instability" in the model, however the average value for the Zero-One Loss is **0.098**.

5 Conclusion

In this project, I implemented different Convolutional Neural Networks using the Tensorflow API, Keras, on Google Colab, with the goal of correctly classify images of muffins and chihuahuas.
After the dataset analysis and preprocessing, I started by training a basic model, then I gradually tested various architectures, introducing and tuning new hyperparameters, with emphasis on the progressive improvements.
The final model achieves 91,6% of accuracy on the test set and 0.098 as cross-validated risk estimate. Given the computational power at my disposal, I would consider it a satisfactory result. Yet, I'm also aware the results also suggest that, with more time and computational power, it would be possible to improve both accuracy and loss.

References

- Keiron O'Shea and Ryan Nash, *"An Introduction to Convolutional Neural Networks"*, 2015
- Sinam Ajitkumar Singh, Swanirbhar Majumder, *"Deep Learning Techniques for Biomedical and Health Informatics"*, 2020
- Srivastava et al., *"Dropout: A Simple Way to Prevent Neural Networks from Overfitting"*, 2014
- Cortes, Corinna and Mohri, Mehryar and Rostamizadeh, Afshin, *"L2 Regularization for Learning Kernels"*, 2009
- Web Resources: machinelearningmastery.com, analyticsvidhya.com, towardsdatascience.com