UNIVERSITÀ DEGLI STUDI DI MILANO

Machine Learning Paper
Muffins versus Chihuahuas Image Classifier
Statistical Methods for Machine Learning

Michela Mazzaglia
michela.mazzaglia@studenti.unimi.it

2023-2024

Abstract

Neural networks have become a fundamental component of modern machine learning. The aim of this paper is to train a neural network for binary classification of Muffins and Chihuahuas based on images from the dataset proposed by the *Statistical Methods for Machine Learning* module at the University of Milan. The training process proceeds according to some specific tasks, such as the full implementation in code using Keras API. Finally, this paper tries to provide an overview of a specific deep learning concept such as neural networks and focuses on their structure, training process, and evaluation methods.

Contents

1 Introduction

A Neural Network, as its name defines, is a network of neurons composed by units or nodes. An Artificial Neural Network, is a machine learning algorithm with a specific architecture. After being named from our biological systems, the two do not share much else. It receives an input (a single vector), and transforms it through a series of hidden layers. Each hidden layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layer, and where neurons in a single layer function completely independently and do not share any connections. The last fully-connected layer is called the output layer and in classification settings, it represents the class scores. The implementation of a neural network can be really effective especially in case of complex and hard input data. It is part of the field of Deep Learning, where developed techniques achieved outstanding performance on many important problems such as computer vision, speech recognition, and natural language processing.
For this experimental project, the dataset analysed contains images of muffins and chihuahuas. A series of tasks were assigned, starting from the color of the images, which were transformed from JPG to RGB (or gray scale) pixel values and scaled down to fasten computations. It followed the experimentation with different network architectures, at least 3, the training and fine tuning of hyperparameters, finishing with the use of k-fold cross validation, with k set to 5, to compute risk estimates. All the obtained results are thoroughly discussed, documenting the influence of the choice of the network architecture and the tuning of the hyperparameters. The training loss can be chosen freely and the reported cross-validated estimates must be computed according to the zero-one loss. The code is retrievable on my Github profile.

2 Data Preprocessing

In order to make computations faster the code was implemented on the Google Colab platform, where it is possible and free to use some units of the GPU available. After having imported all the libraries and installed the necessary packages, the dataset was loaded. There are exactly 5917 files inside the dataset belonging to 2 classes, Muffins or Chihuahuas. The images of muffins and chihuahuas were uploaded with a image width and height of 100 and a batch size of 32. The **batch size** is a hyperparameter and

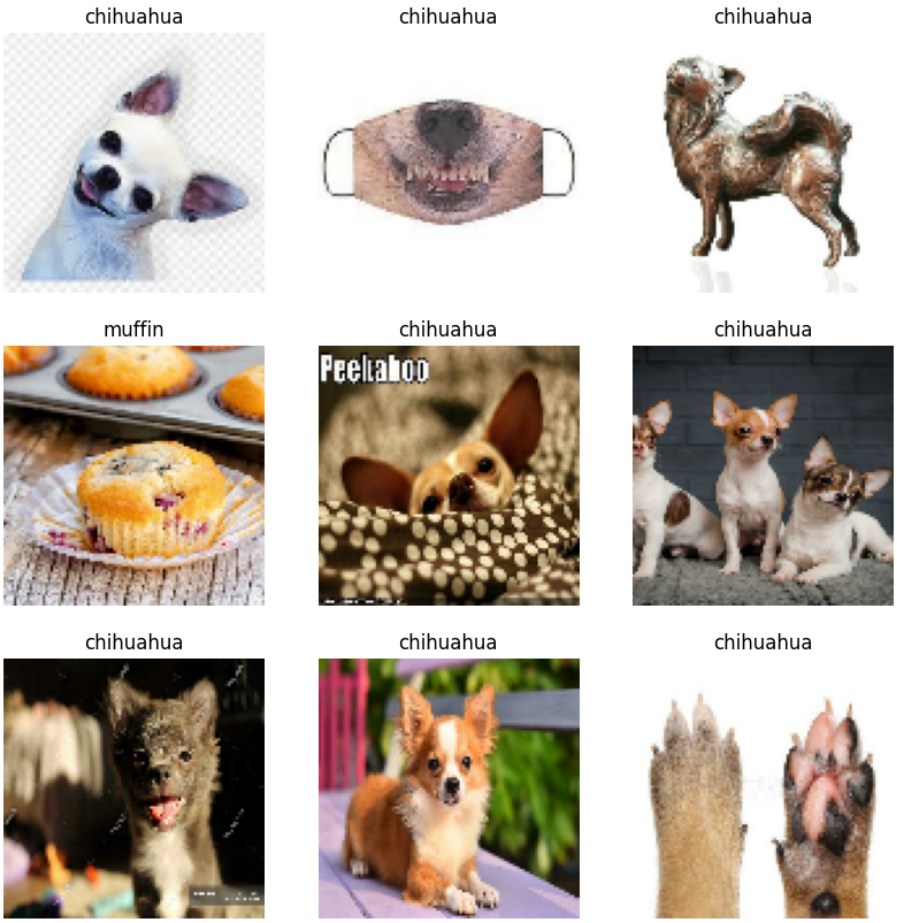

Figure 1: Batch of images

it represents the collection of examples images used and processed during each epoch (loop) of the training of the neural network. This choice is common in deep learning and it is mostly due to its efficiency in the computations and memory. The color mode was set to rgb, which describes how much of red, green, and blue is included in the images. The train and test set were defined, having respectively 4733 and 1184 images, in order to train the model and then evaluate its performance on new unseen data. Using the splitting function provided by *scikit-learn*, the data was divided into train images and labels and validation images and labels setting the split at 0.2, where the validation set is used to monitor the model's performance and tune hyperparameters. The figure hereby 1 shows some images taken from a batch of the train dataset. It is evident that the displayed batch not only is favouring images of chihuahuas other than muffins, but it is also showing what kind of images the dataset includes: not completely clear and not quite specific. Some of the images represent dogs that are not of the defined breed, while others, as the last image in the lower right, represent other characteristics which

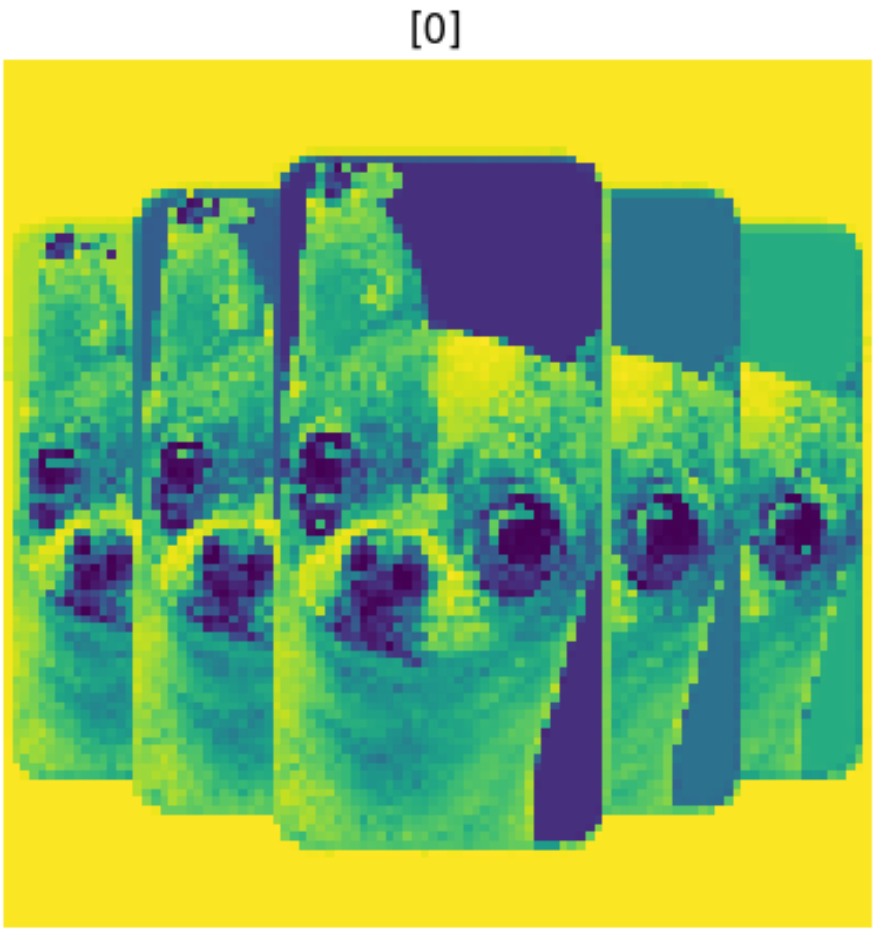

Figure 2: Example image

can lead the neural network to not understand completely the classification process. Since the images' color was set to rgb which is defined in scale over 255, the pixel values were normalized to be between 0 and 1. Moreover, the color mode was changed to gray scale. These changes were done with the aim to fasten the computational and training process of the neural network. The labels were one-hot-encoded with values assigned as 0 or 1, where 0 represent chihuahuas and 1 muffins. An example is shown by the above figure 2.

3 Model definition

In the process of building the model for the Neural Network, the architecture of Convolutional Neural Networks (CNN) was preferred, due to their proven superiority in

handling image data, as discussed earlier. A CNN is quite similar to a simple Neural Network: it is made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other, and it has a loss function on the last (fully-connected) layer. The difference between implementing a simple Neural Network and a CNN relies in its architecture. CNNs make the explicit assumption that the inputs are images, which allows to encode certain properties into it. In addition, they make the forward function more efficient to implement and vastly reduce the amount of parameters in the network. CNNs are the best choice in this case, especially since they can scale very large images thanks to their structure. It is based on layers which have neurons arranged in 3 dimensions: width, height and depth. The depth refers to the third dimension of an activation volume and not to the depth of a full Neural Network, which instead refers to the total number of layers in a network. The output will be a single vector of class scores, given that the CNN architecture reduce the full image (figure 3).

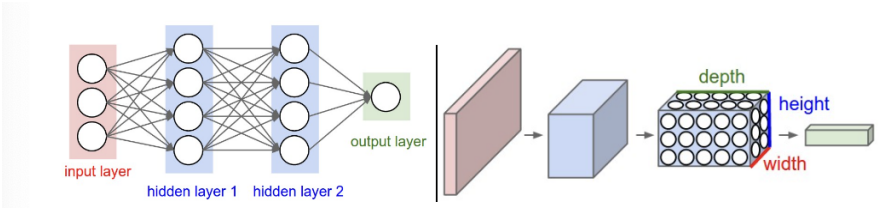

Figure 3: Left: A regular 3-layer Neural Network. Right: A CNN neurons and layers

In the course of this project, the sequential model was implemented as it describes the flow of information through the networks' layers. The first architecture was created as follows:

- **Input Layer** (InputLayer): it defines the input shape of the data, which in this case is (100, 100, 1), indicating an input image size of 100x100 pixels with a single channel (grayscale).
- **Convolutional Layer**(Conv2D): each convolutional layer has 64, 64, and 32 filters respectively, with a kernel size of (3,3). The activation function used is ReLU.
- **Max Pooling Layers** (MaxPooling2D): each max-pooling layer reduces the spatial

dimensions of the input data by taking the maximum value in each window. The size of the pooling window is (2,2).

- **Flatten Layer** (Flatten): it converts the output of the convolutional layers into a one-dimensional vector, which prepares the data for input into the fully connected layers.

- **Dense Layer** (Dense): it is a fully connencted layer with 128 neurons and ReLU activation function.

In the final layer of the neural network architecture, a Dense layer was employed to serve as the output layer. This layer was configured with a single unit, as the task at hand entails binary classification. To facilitate this classification, the **Sigmoid** activation function was chosen for the output layer. The utilization of the sigmoid activation function ensures that the output of the model is constrained to a single value, thus effectively distinguishing between the two classes, namely Muffins and Chihuahuas.

3.1 Understanding the CNNs architecture

It is imperative to provide a brief elucidation for the understanding of the architecture.

The initial component of the neural network architecture, the "Input Layer", plays a pivotal role in orchestrating the flow of input data to subsequent layers. It is pertinent to emphasize that the Input Layer does not process the entire image as a singular entity; rather, it partitions the image into smaller, manageable blocks. This segmentation facilitates the processing of localized features and enables the network to extract meaningful information from the input data as shown in the figure below (Figure 4).

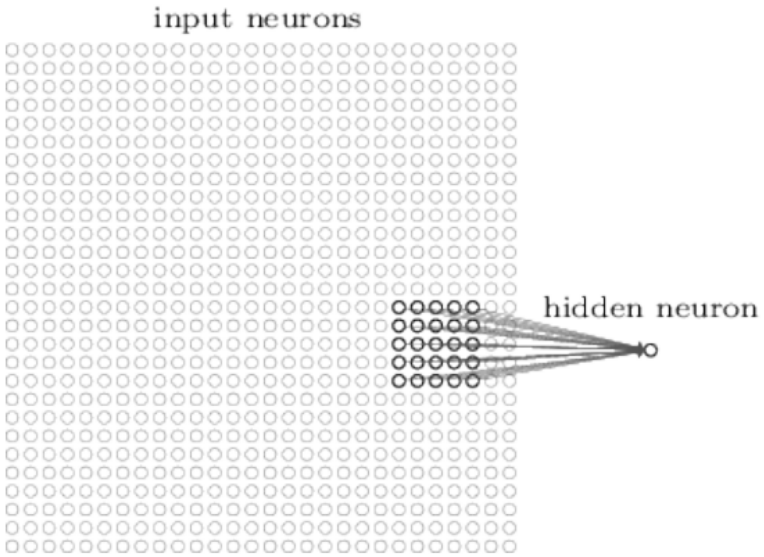

Figure 4: The input data scaled into blocks

Consequently, there are the so-called "Hidden Layers". These layers operates by performing some mathematical transformations on the data. The idea used is the one of the *Stochastic Gradient Descent* which estimates the gradient of a small sample chosen randomly over the training inputs. Through these layers an adjustments of the weights and biases is performed, in order for the output to get closer to the real value. For example, the **pooling** layer serves the purpose of reducing inputs of varying sizes to a standardized size. Within this context, the maximum pooling operation is employed, where each block within the input is scrutinized, and the block presenting the maximal value is retained. This operation effectively extracts the spatial information while preserving salient features, thus contributing to the model's capacity to generalize across diverse inputs. The intuition is that once a feature has been found, its exact location is not as important as its rough location relative to other features. This is valuable also for the **convolutional** layer, which uses the **kernel** representing a matrix that captures the patterns of the input data. The kernel, also known as filter, slides through the image performing a dot product between its values and the pixel values. This activity is called "convolution" and the sliding is determined by the **stride**, which sets how many pixels are processed during the convolution. To ensure that the dimensions between the input data and the output data are equal, another parameter of the convolutional layer **padding** (*same*) is used. Moreover, depending on the kernel size it is possible to gather more information on the features of the images.

The final layer of connections in the network is the Output Layer, often a fully-connected layer, which connects the information from all the previous layers to the two different outputs in order to classify the images. The strength of the connection between these neurons is determined by the weights parameter. During the training of the neural network, these weights adjusts in order to minimize the loss function and improve the model's performance. Each convolutional and fully-connected layer has an **activation function** that is used to introduce non-linearity in the neural network, essential to finally make the model understand complex patterns that structure real-world data. The chosen ones are the **ReLu** and **Sigmoid**. While the properties of the sigmoid function have been previously discussed, it is also important to discuss the characteristics of the other one. ReLu stands for Rectified Linear Unit and it provides an efficient training, performing better than other functions like tahn, while avoiding the problem of the *Vanishing Gradient Descent*. It states that for negative inputs the output of the ReLu activation function is zero, leading to sparse activations of neurons during the training process, while for positive inputs it behaves as a linear function.

3.2 Configuring the model

Once the model is defined, some other parameters need to be configured for training. This is done by the compile method, which takes as arguments: the optimizer, the loss and the metrics. The first one, as its name defines, implements the optimization on the input weights in order to make better predictions by minimizing the loss. Between the many *optimizers* the one chosen was **Adam**, provided by the Keras module, due to its being computationally effective especially regarding the learning rate. Adam stands for Adaptive Moment Estimation, stating the adaptability of the learning rate that can handle complex and high-dimensional data patterns like images, leading to a faster convergence and better generalization performance in problems such as image classification. The *loss function* instead measures the difference between the predicted labels and the true labels, meanwhile showing how well the model is performing. The **binary crossentropy** was preferred since an image can belong only to one of the two classes (Muffin or Chihuahua). Finally, the **accuracy** is the *metric* that evaluates the percentage of corrected predictions.
The training of the model is performed over a fixed number of epochs. It takes as input

the images and their corresponding labels, the batch size, and the validation images and their corresponding labels.

4 Training process

4.1 The Basic Model

During the training process different architectures were defined. As previously mentioned, the first architecture is known as basic model. The figure below 5 represents the behaviour of the training and validation, loss and accuracy, over 25 epochs.

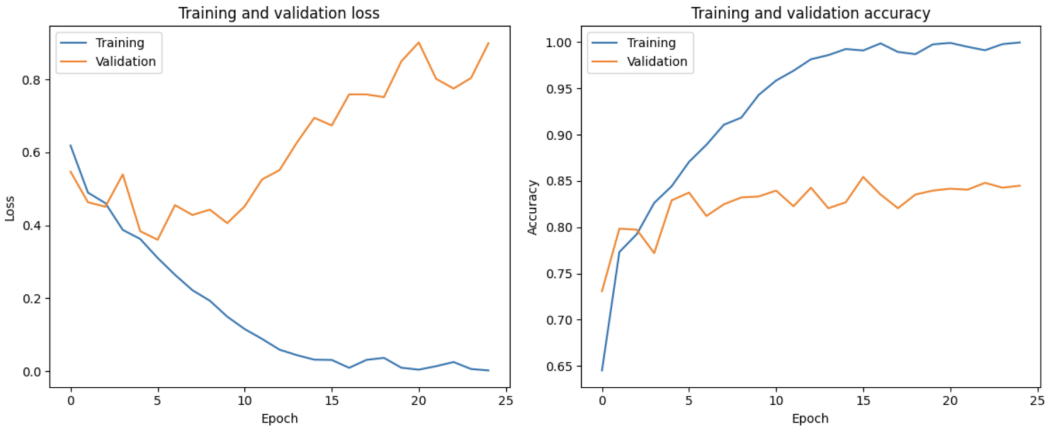

Figure 5: Basic model performance

From the graph it is possible to spot the presence of overfitting. The training loss had very low values around *0.0017*, while the validation loss had higher values around *0.89*. Overfitting shows how, on one hand, the model performs very well on the training data, while on the other hand it performs poorly on unseen data. In order to reduce this difference, several methods were used. The first measure taken was removing a layer. The number of epochs were increased up to 30 with the aim to let the model improve its learning. The results show a reduction of the validation loss (*0.62*) and a slight increase in the validation accuracy, meaning that it helped to reduce the problem (figure 6).

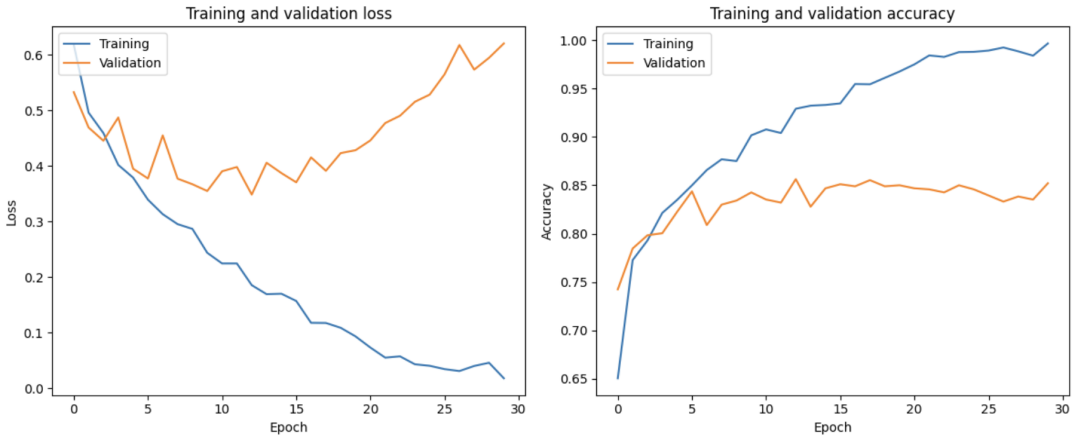

Figure 6: Model performance with removed layer

The second method was adding a **Dropout layer**. It is one of the regularization techniques commonly implemented to reduce overfitting. As its name defines, it is a layer which generally lies between fully connected layers, and "drops out", drops to zero some of the input units in order to prevent the issue. The exact fraction is determined by the dropout rate (a float between 0 and 1 representing the probability). Setting the dropout rate to $p = 0.5$, the validation loss decreased reaching 0.52, as well as the validation accuracy, meaning that there still be room for improvement.

Secondly, the dropout rate was increased up to $p = 0.7$ to discover if it could improve overfitting. Comparing the results with the previous one, it is possible to spot an advancement in both training loss and accuracy, where the first one decreased and the latter increased. However, the validation loss had higher values as well as the validation accuracy describing a persistent overfitting although a small improvement in generalizing unseen data (figure 7).

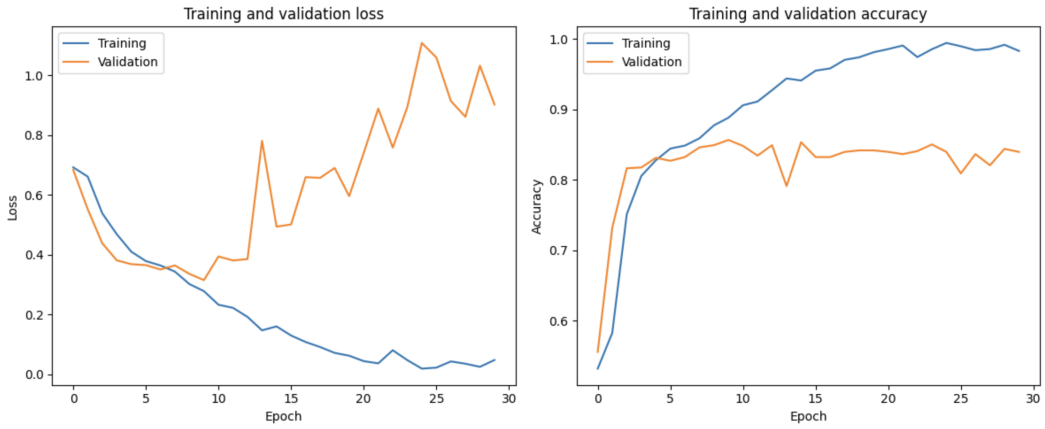

Figure 7: Model performance with Dropout

The third method was using **Batch Normalization**. This layer introduces a normalization step at each layer's inputs. As an alternative to the dropout, the model performance was quite unstable showed by figure 8, although it reached the highest validation accuracy so far of around *0.92* at the 21st epoch.

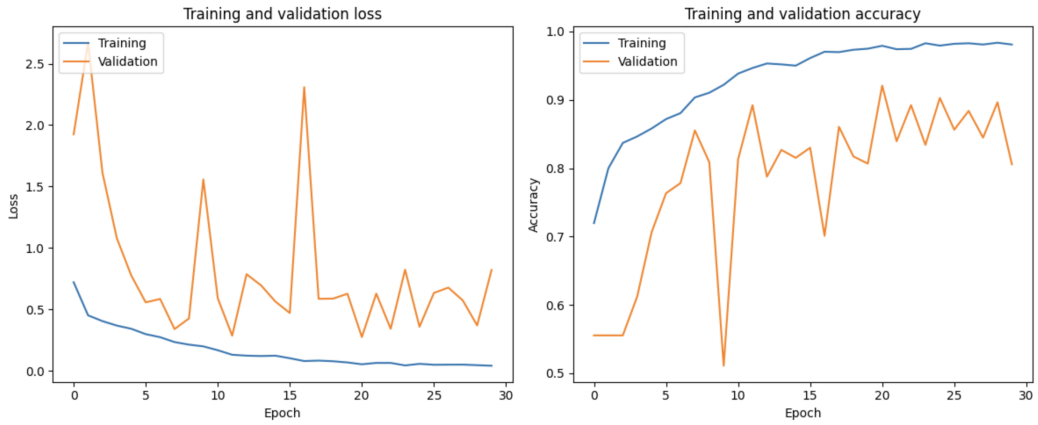

Figure 8: Model performance with Batch Normalization

The Dropout validation accuracy compared to this was overall stable around *0.85*, but with Batch Normalization the model generalizes better as stated by the lower validation loss. The training needs further improvements.

4.2 The second architecture

Going through the training process a second architecture was built. Its structure is the following:

- **Conv2D**: with 32 filters of size (3,3) and ReLU activation function.
- **MaxPooling2D**: with a pool size of (2,2) to downsample the spatial dimensions.
- *The previous two are repeated two other times.*
- **Flatten**: the input from the other layers are transformed into a 1D vector, preparing it for the fully connected layers.
- **Dense**: with 64 units and ReLU activation.
- **Dropout**: with a rate of 0.7 to prevent overfitting by randomly setting a fraction of the input units to 0 at each update during training.

- **Dense**: with 16 units and ReLU activation.
- **Dense**: as output layer with 1 unit and sigmoid activation for binary classification.

The model starts with low accuracy and high loss, indicating it's not performing very well. As training progresses, both training and validation metrics improve, suggesting the model is learning and generalizing better. Around the 15th epoch, the validation accuracy peaks, and the loss decreases, suggesting this is where the model performs best on unseen data. After this point, the model starts to overfit slightly, as the training accuracy continues to increase, while the validation accuracy flattens and the validation loss starts to increase (figure 9).

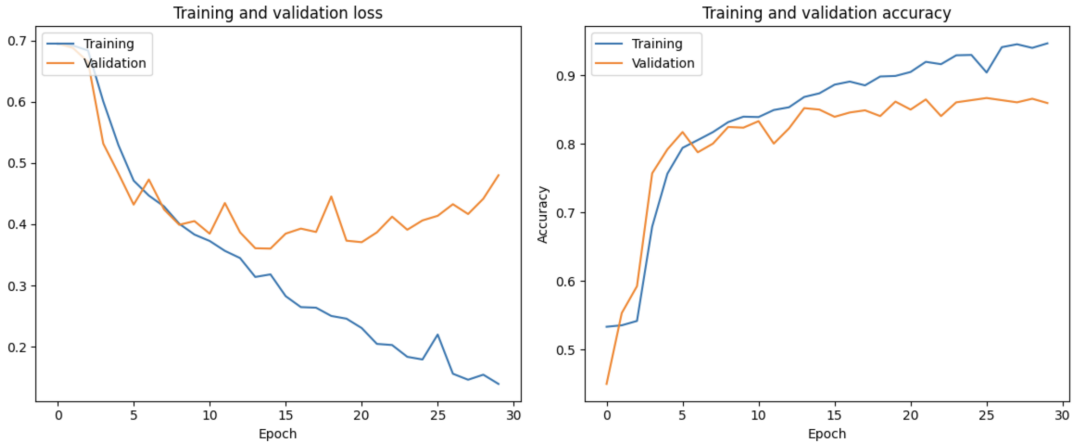

Figure 9: Second architecture performance

This architecture appears to outperform the previous one in terms of stability. Another method to additionally reduce overfitting can be applied: **L2 regularization**. This parameter is present in the Dense layer and interacts with its weights. It relies on the Ridge regression function, which adds a penalty to the loss function proportional to the square of the weights. Thanks to its regularization strength parameter λ it is possible to control overfitting. Another method is the L1 regularization, which relies on the Lasso function pushing some values towards zero, reason why L2 regularization was preferred. Besides, it produces smoother weights value, which can be beneficial for image data where each pixel or feature can contribute valuable information to the final prediction. To terminate the training process when the model's performance ceased to improve, **Early Stopping** was introduced. This technique permits to monitor the validation performance based on a *patience* parameter, which specifies the number of

epochs tolerated with no improvements before stopping. Its last parameter *(restore best weights)* allows to return to the model the weights with the best performance during training (figure 10).

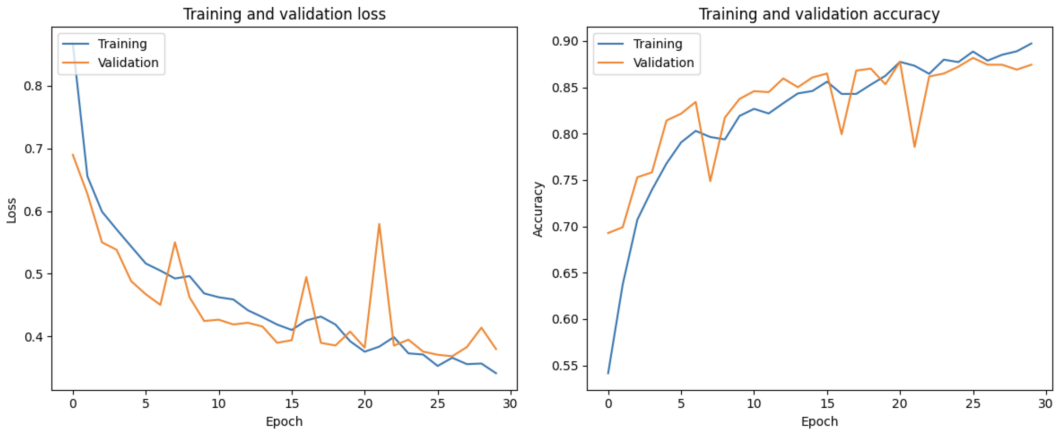

Figure 10: Model performance with L2 regularization

Subsequently, the number of epochs on the same model was increased up to 50, which made possible to register an improvement in the validation accuracy reaching *0.89*. It is clear that the model learnt to generalize and going through the training process it reached a good level of generalization on unseen data. In the last experimentation on this architecture, the technique of **padding** was implemented to preserve spatial information of the images, as well as an increase in the number of neurons in the first Dense layer (128). Comparing these two different sequential models, the last one presents a lower validation accuracy which is traduced in higher overfitting (figure 11).

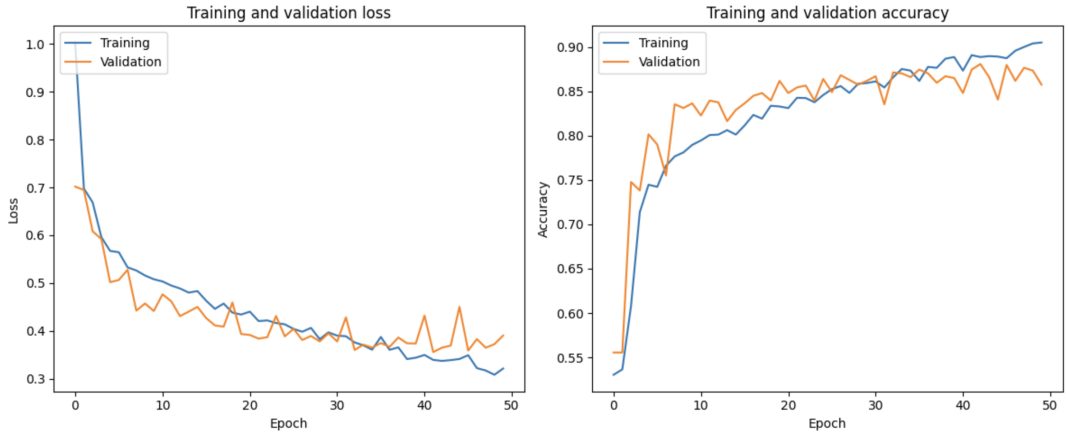

Figure 11: Second architecture final performance

4.3 The Third architecture

A third architecture was created with the aim to find another high performance model. Its structure was the following:

- **Conv2D**: with 64 filters, a kernel size of (3, 3), and 'same' padding.
- **Conv2D**: with 128 filters, again with a (3, 3) kernel size and 'same' padding.
- **Conv2D**: with 64 filters and always 'same' padding.
- **MaxPooling2D** Layers: Each of these layers with (4, 4).
- **BatchNormalization** Layers: right after each maxpooling layer.
- **Flatten**
- **Dense** Layers: with 128 units and ReLU activation, along with L2 regularization.
- **Dropout**: with a rate of 0.7.
- **BatchNormalization**
- **Dense**: with 16 units and ReLU activation.
- **Dense**: with 1 unit and sigmoid activation.

The training was performed over 30 epochs, as done with its predecessors. Since the initial epochs, both training and validation loss decrease, almost by reaching the same level, while the validation and training accuracy increase, showing a considerable generalization. The last epochs show a return of a slight overfitting, which was tried to be additionally reduced by increasing the level of regularization and by extending the number of epochs. During this next training the validation accuracy reached *0.91* at the 22th epoch. However, it maintained a validation loss of *0.5*, while the training loss got smaller indicating the permanent presence of the difference (figure 12).

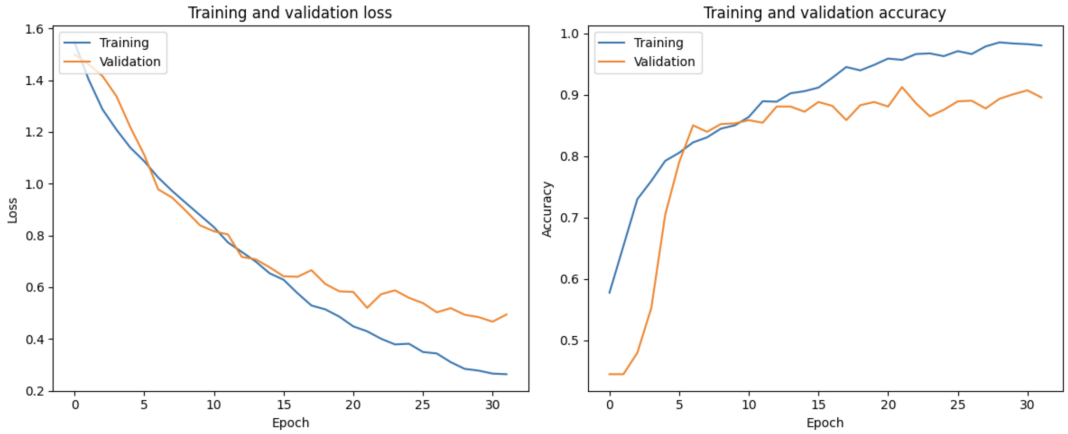

Figure 12: Third architecture final performance

This architecture is going to be used for a further hyperparameter tuning in the next section.

5 Hyperparameter tuning

Optimizing the performance of the models is extremely important. It was done with the aim to find the best hyperparameters that led to the highest accuracy on the validation set. Hyperparameters are just variables manually set before the training and that can manage the performance of the model. Since there is no possibility to know which hyperparameters give the best results in advance, the activity of hyperparameter tuning is needed in order to experiment.

Firstly, the number of nodes inside each convolutional layer was optimized. After having set the values of 128, 64 and 32, the best validation accuracy of *0.89* was found with this combination of nodes' layers: *64, 128, 128*. Secondly, the learning rate was optimized. The best learning rate found was *0.001* with a validation accuracy of *0.91*. Lastly, the size of the pools and kernels. It was found that having a pool size of 4 and a kernel size of (3,3) had the best validation accuracy of around *0.90*. The final model was trained setting the results just discussed (figure 13).

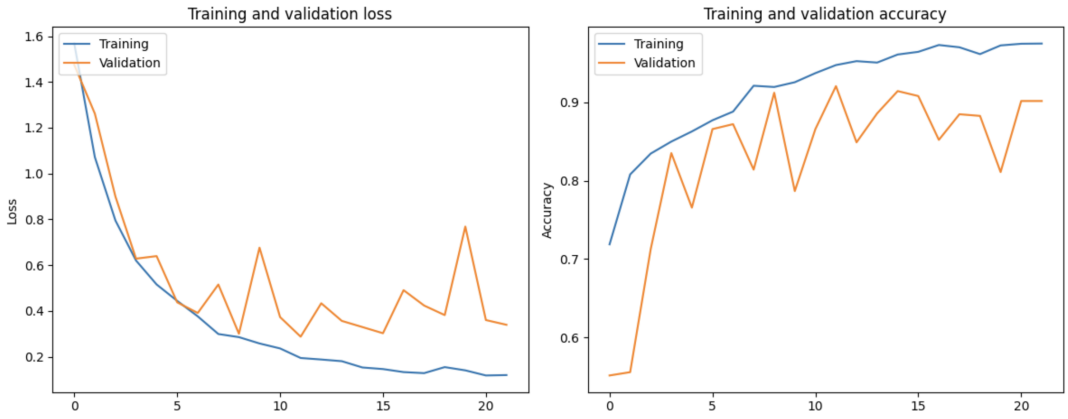

Figure 13: Final model performance with hyperparameter tuning

Since there exist also hyperparameter learning algorithms, the Keras Tuner framework was used to inspect if it would reach better results of the manual hyperparameter tuning of above. The algorithm used was Bayesian Optimization, based on Bayes' Theorem. It did not perform better than the previous optimization, since the highest validation accuracy reached only *0.86*.

6 Evaluation

After discovering the best model through rigorous training and optimization, its performance was evaluated on the test set. It showed an accuracy of *0.88* and a loss of *0.36*. The confusion matrix hereby (figure 14) shows exactly how the model classifies both chihuahuas and muffins, registering a higher misclassification of chihuahuas over muffins, probably because of the presence of not so clear images inside the dataset. Finally, a test was performed on a picture from a friend. The reason was to study the behaviour of the model with data unseen before, outside the dataset given. The final result was a value really close to 0, which represent chihuahuas, proving the worth of the model.

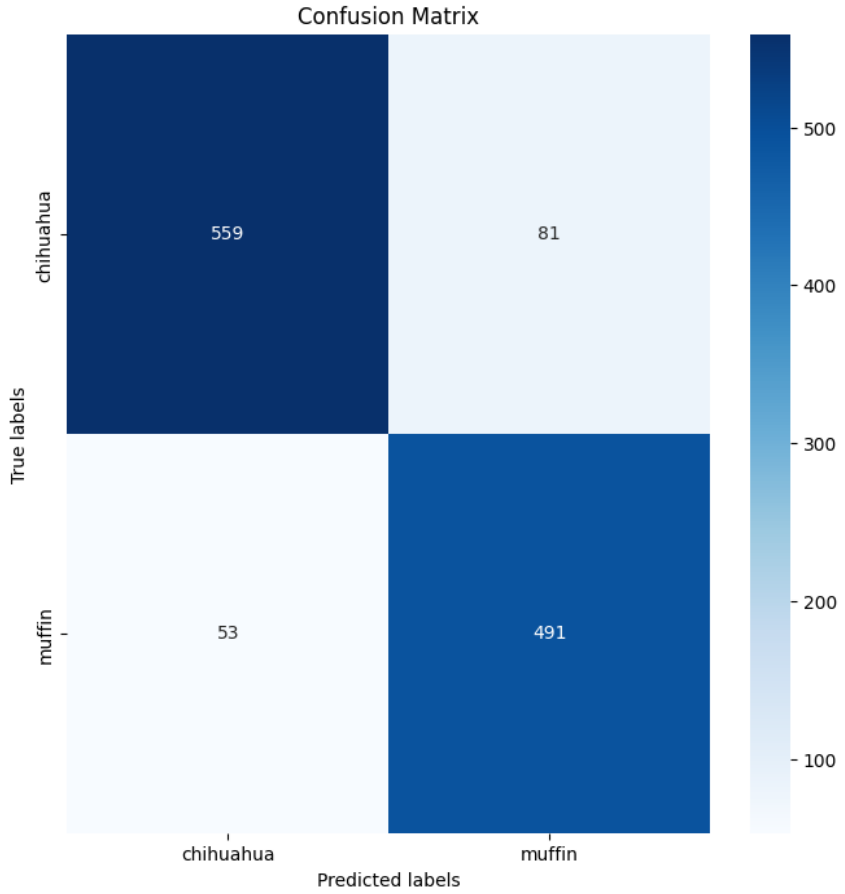

Figure 14: Confusion Matrix

7 K-fold Cross Validation

As last task for this project, a K-fold cross validation was performed. The number of folds determined was 5 and the reported cross validated estimates were computed according to the zero-one loss. The folds are subsets of the data, which are also the number of times the model is trained. At each time, a different fold is used as validation set and the remaining folds as training set. This technique is extremely helpful to understand how the model would perform generally, instead of using the data generated by the train-test split. That is the reason why it is also more reliable. The average accuracy was around *0.88* and the average zero-one loss was *0.11*, as it performed during the evaluation on the test set. Overall, the metrics were consistent showing that the model is performing and generalizing well on unseen data.

8 Conclusion

In conclusion, this paper provides a brief understanding of neural networks, covering their architecture, training process, and evaluation methods according to the tasks assigned. After having discussed the best model found, it was evaluated on data it had never seen before and performed unexpectedly well. Acknowledging the extensive process required to train a neural network and the effort invested in understanding their behaviour, it is clear that there are opportunities for further improvements and a deeper comprehension of the subject in the future.

References

[1] Ian Goodfellow, Yoshua Bengio, Aaron Courville (2017). Deep Learning.

[2] Diederik P. Kingma, Jimmy Ba (2014). Adam: A Method for Stochastic Optimization.

[3] Charles A. Micchelli, Massimiliano Pontil. (2005). Learning the Kernel Function via Regularization.