UNIVERSITÀ DEGLI STUDI DI MILANO

Facoltà di Scienze e Tecnologie

Statistical Methods for Machine Learning

Chihuahua and Muffin Classification

Stefano Corizzato - 20208A

1 Introduction

In this project, a convolutional neural network (CNN) was developed and trained to perform binary classification, distinguishing between images of muffins and chihuahuas.

Binary classification refers to a type of classification task that involves predicting one of two possible classes or outcomes, that is, assigning one of two categories to an input. In our case the two categories are "Muffin" and "Chihuahua", and the input is a series of images. CNNs are a particular class of neural networks, used to analyze images, their primary characteristic is the use of convolution operations with the aim of extracting features such as edges, textures, and shapes.

The language used for this project is Python, the various CNN models were built using TensorFlow, a machine learning framework; and Keras, TensorFlow's high-level API

2 Convolutional Neural Network

The architecture of a convolutional neural network is composed of several layers:

- Convolutional layers: The aim is to extract features such as edges, textures and shapes, to do this a convolution operation is applied to the images. This operation involves sliding a filter (kernel) over the input image and calculating the dot product between the filter and portions of the image.

  Different filters are used, with different values in order to enhance certain features of the image. The final result will be a series of images, based on the number of filters used.

- Pooling Layers: The purpose of these layers is to reduce dimensionality and computational load. The pooling operation consists in iterating over the image and for each region of chosen size returning a single value, the most common case is max pooling, which selects the maximum value from each region. E.g. if we use a 2x2 region the result will be an image half the size of the original.

- Dense Layers: they are Layers of neurons in which each neuron receives input from all the other neurons of the previous layer. Before passing through these layers the data, made up of 2-dimensional matrices, are "flattened" to obtain 1-dimensional arrays. The last layer is connected to the output layer which is composed of one or more nodes based on the problem to be solved and the activation function used.

The steps followed for the development of the neural network will be explained below.

2.1 Data pre-processing

The data was taken from Samuel Cortinhas "Muffin vs chihuahua"[1] dataset made available on the Keggle website. The dataset consists of approximately 6000 images of muffins and chihuahuas of different sizes. The data is organized in two folders "test" and "train", within these folders the images are again divided between "chihuahua" and "muffin".

The images were organized into runtime datasets thanks to the function `image_dataset_from_directory`, which generates a `tf.data.Dataset` from image files in a directory.

From the source of the dataset we know that the test set includes 20% of the total files, during the creation of the training dataset we take 20% of the files (16% of the total) and allocate it for the validation set.

The `image_dataset_from_directory` utility allows us to specify various parameters of the dataset, for this project the following configuration was used:

- `directory` = 'archive/train' ('archive/test' for test set)
- `labels`="inferred"
- `label_mode`="int"
- `color_mode`="rgb"
- `batch_size`=16
- `image_size`=(224, 224)

[1]https://www.kaggle.com/datasets/samuelcortinhas/muffin-vs-chihuahua-image-classification

- `shuffle`=True (False for the test set)
- `seed` = 100 (not present for the test set)
- `validation_split`=0.2 (not present for the test set)
- `subset`="both" (not present for the test set)

With this configuration we obtain a dataset of images with labels 0 (for chihuahuas) and 1 (for muffins), encoded using the RGB model, of dimensions 224x224 pixels, enclosed in batches of size 16. The last 4 parameters concern the creation of the validation set.

The images obtained are therefore made up of 3 matrices measuring 224x224. To improve the training process we normalize the images by dividing the pixel values by 255 (maximum possible value) thus obtaining values between 0 and 1.

2.2 Model creation

The first model built has the following architecture:

- First Convolutional Layer: takes as input array of dimensions 224x224x3 and applies 32 filters of dimension 3x3, using the ReLU function as the activation function.
- First MaxPooling Layer: tiling size of 2x2, returns the maximum value.
- Second Convolutional Layer: 64 filters of dimension 3x3 used.
- Second MaxPooling Layer: same as the first maxpooling layer.
- Dropout Layer: this layer randomly sets 20% of the input to zero, the aim is to prevent overfitting.
- Flatten Layer: transform the images in a 1-d arrays.
- Dense Layer: 64 neurons and ReLU as activation function.
- Output Layer: 1 neuron and Sigmoid as activation function.

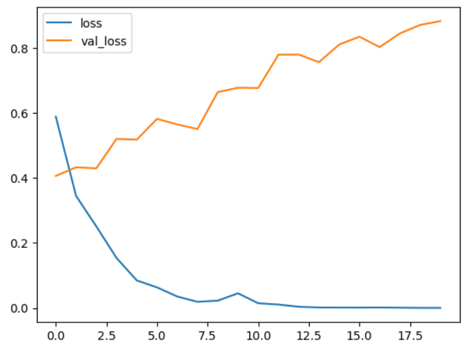

Figure 1: Comparison between the loss of the validation set and loss of the training set

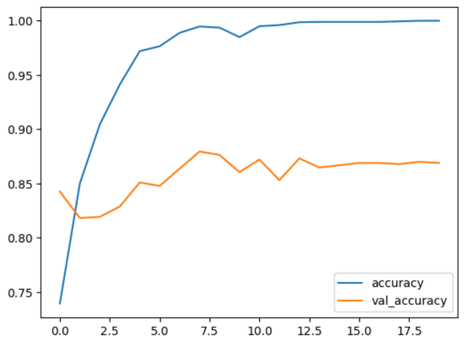

Figure 2: Comparison between the accuracy of the validation set and accuracy of the training set

After training the model for 20 epochs we compare the values obtained from the training set and the validation set.

From the figures 1 and 2 it can be seen that the model is overfitting, i.e. it is unable to generalize and is adapting excessively to the training set. To try to overcome the problem we can use data augmentation.

2.3 Data Augmentation

By data augmentation we mean the process of artificially increasing the amount of data by generating new data points from existing data. It is used in machine learning to increase the diversity of the training dataset without actually collecting new data.

For images this translates into applying small geometric transformations (such as translation, rotation or zoom) and/or color alterations (such as saturation or brightness) to the original images in order to increase the diversity of the training set.

Tensorflow provides several preprocessing layers that allow you to modify images, those used in this project are:

- `RandomBrightness(factor=0.1)` = applies a Brightness filter with random values between -0.1 and 0.1.
- `RandomTranslation(height_factor=0.1, width_factor=0.1)` = applies a transition filter that shifts the height and width of the image by

random values between -10% and 10%.

- `RandomZoom(0.2,fill_mode='nearest')` = a layer that randomly zooms images during training with values between -20% and 20%. Points outside the input boundaries are filled using the nearest values.
- `RandomFlip("horizontal")` = a layer that randomly rotates images along the horizontal axis during training.
- `RandomRotation(0.1)` = this layer will apply random rotations to each image by a random amount in the range [-10% * 2pi, 10% * 2pi].

Only the training set passed through the data augmentation layers, the validation set and the test set did not undergo any alterations.

An example of the effects of data augmentation on images is shown below (figures 3 and 4).

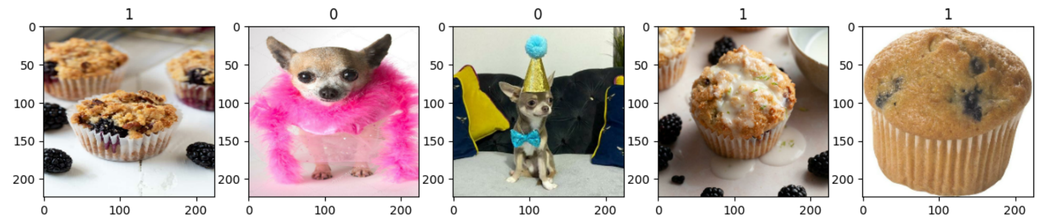

Figure 3: Images before data augmentation

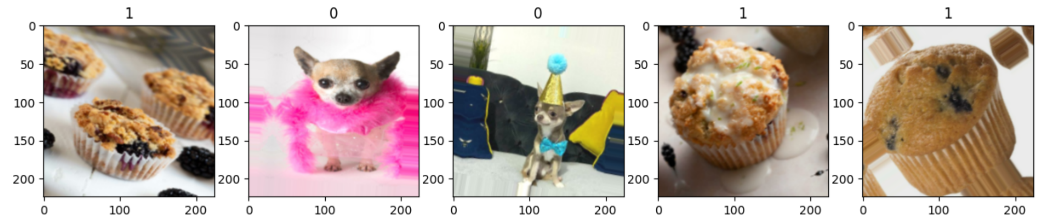

Figure 4: Images after data augmentation

2.4 Models

2.4.1 Model 1

After performing data augmentation on the training set, the first model was trained again using the new images obtained. The model in this case was trained for 30 epochs, leaving the other parameters unchanged.

The results obtained show (figures 5 and 6) an improvement regarding overfitting, thanks to the data augmentation both the validation loss and the accuracy loss are more similar to the values of the training set.

However, due to data augmentation, the accuracy decreased from 1 to 0.90 for the training set and from 0.87 to 0.85 for the validation set. Furthermore, although the validation loss has decreased, it remains a slightly high value at 0.37.

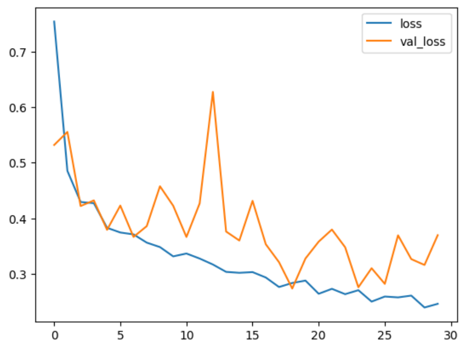

Figure 5: Validation Loss and Training Loss

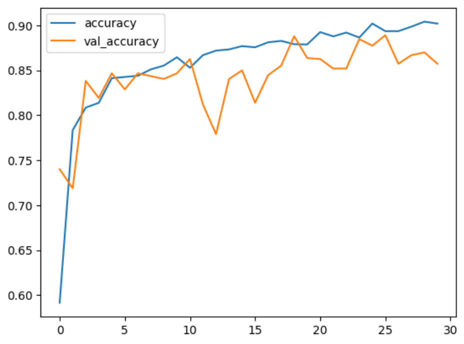

Figure 6: Validation Accuracy and Training Accuracy

2.4.2 Model 2

The second model built tries to solve the problems found in the previous model, i.e. increasing the accuracy for the validation set and decreasing the loss for the validation set.

The second model built has the following architecture:

- First Convolutional Layer: takes as input array of dimensions 224x224x3 and applies 32 filters of dimension 3x3, using the ReLU function as the activation function.

- First MaxPooling Layer: tiling size of 2x2, returns the maximum value.
- First Dropout Layer: this layer randomly sets 20% of the input to zero, the aim is to prevent overfitting.
- Second Convolutional Layer: 64 filters of dimension 3x3 used.
- Second MaxPooling Layer: same as the first maxpooling layer.
- Second Dropout Layer: same as the first dropout layer.
- Third Convolutional Layer: 32 filters of dimension 3x3 used.
- Third MaxPooling Layer: same as the first maxpooling layer.
- Flatten Layer: transform the images in a 1-d arrays.
- Dense Layer: 32 neurons and ReLU as activation function.
- Third Dropout Layer: same as the first dropout layer.
- Output Layer: 1 neuron and Sigmoid as activation function.

Compared to the first model the following changes have been implemented In order to increase accuracy, an additional convolutional layer with 32 filters and an additional MaxPooling layer were added to reduce dimensionality. Furthermore, 2 more dropout layers have been added after the Maxpooling layers with the aim of preventing any overfitting. Finally, the size of the dropout layer was reduced from 64 to 32 nodes.

After the model was trained for 30 epochs, the results obtained showed (figures 7 and 8) an improvement compared to the first model both in validation accuracy, going from 0.85 to 0.91, and in loss accuracy, going from 0.37 to 0.24. Also regarding the training set, the results show improvements compared to the first model.

2.4.3 Model 3

For the third model, a further convolutional with 64 filters was added before the first maxpooling layer, thus bringing the total to four, the last convolutional layer was modified, bringing the number of filters from 32 to 128. Finally, the number of nodes in the dense layer was increased to 64 and a

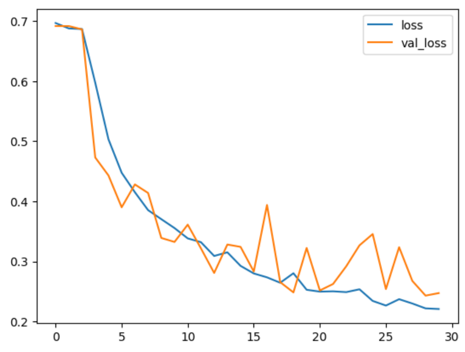

Figure 7: Validation Loss and Training Loss

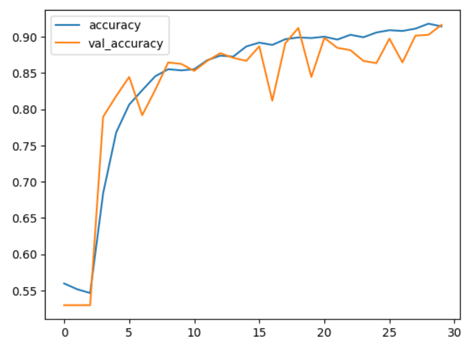

Figure 8: Validation Accuracy and Training Accuracy

dropout layer was added between the dense layer and the output node with a value of 0.4.

The third and final model built has the following architecture:

- First Convolutional Layer: takes as input array of dimensions 224x224x3 and applies 32 filters of dimension 3x3, using the ReLU function as the activation function.
- Second Convolutional Layer: 64 filters of dimension 3x3 used.
- First MaxPooling Layer: tiling size of 2x2, returns the maximum value.
- First Dropout Layer: this layer randomly sets 25% of the input to zero, the aim is to prevent overfitting.
- Third Convolutional Layer: 64 filters of dimension 3x3 used.
- Second MaxPooling Layer: same as the first maxpooling layer.
- Second Dropout Layer: same as the first dropout layer.
- Fourth Convolutional Layer: 128 filters of dimension 3x3 used.
- Third MaxPooling Layer: same as the first maxpooling layer.
- Third Dropout Layer: same as the first dropout layer.
- Flatten Layer: transform the images in a 1-d arrays.

- Dense Layer: 32 neurons and ReLU as activation function.
- Fourth Dropout Layer: this layer randomly sets 40% of the input to zero, the aim is to prevent overfitting.
- Output Layer: 1 neuron and Sigmoid as activation function.

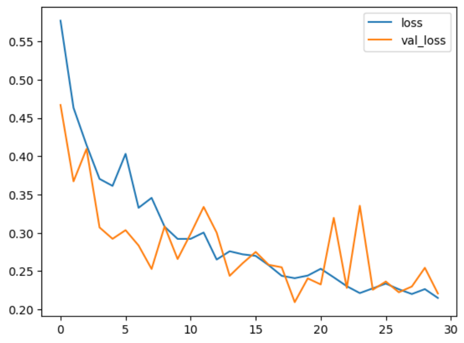

Figure 9: Validation Loss and Training Loss

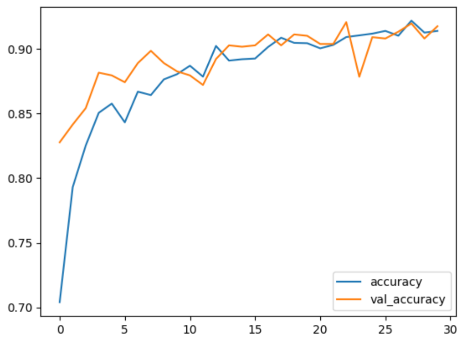

Figure 10: Validation Accuracy and Training Accuracy

After training the model for 30 epochs the results obtained are very similar to those obtained from the second model, that said, thanks to a lower validation loss (0.22) this model was chosen for hyperparameter optimization.

3 Hyperparameter Optimization

A hyperparameter is a parameter whose value is used to control the learning process. Some examples can be the learning rate, batch size or kernel size in convolutional layers.

Hyperparameter optimization or tuning is the problem of choosing a set of optimal hyperparameters for a learning algorithm.The result of hyperparameter optimization is a list of hyperparameters that produces an optimal model that minimizes a predefined loss function.

There are several ways to perform hyperparameter tuning, the most famous are Grid search: given subsets of values for each hyperparameter, the algorithm performs an exhaustive search using every possible combination; Random search: randomly selects the values to use in each iteration; and

Bayesian optimization: starts randomly and tries to learn from the results obtained by creating a probabilistic model of the function.

Thanks to the `keras.tuner` framework it was possible to specify for which hyperparameters and for which values to perform the tuning, and the objective to minimize, in our case the validation accuracy. Below are the parameters chosen for tuning and the range of values in which to carry out the search:

- Dropout hidden layer: from 0.05 to 0.25 with with step of size 0.1
- First Convolution layer: from 16 to 64 with step of size 16
- Second Convolution layer: from 32 to 128 with step of size 16
- Third Convolution layer: from 32 to 128 with step of size 16
- Fourth Convolution layer: from 32 to 128 with step of size 16
- Last dropout layer: from 0.2 to 0.5 with with step of size 0.1

5 trials were performed and the results obtained were: **0.05** for hidden dropouts; **64** for the first convolutional, **80** for the second, **64** for the third, **128** for the fourth and **0.2** for the final dropout.

This combination of hyperparameters returned a validation accuracy of 0.92, therefore not improving the result obtained from the non-optimized model. A possible explanation is the high number of parameters provided compared to the number of trials performed.

4 Evaluation

After training the network with the developed hyper parameters, we evaluate the performance of the network using the `predict` function. on the test set. The function returns a value between 0 and 1, the value is then rounded to the nearest integer, in other words if the value is less than 0.5 the predicted label will be chihuahua (0), if the value is greater than 0.5 the label predicted will be muffin (1).

From the results obtained it is possible to build a confusion matrix to have a quick understanding of the quality of our model. We can see that out of 640 images of chihuahuas, 593 were categorized correctly and 47 incorrectly,

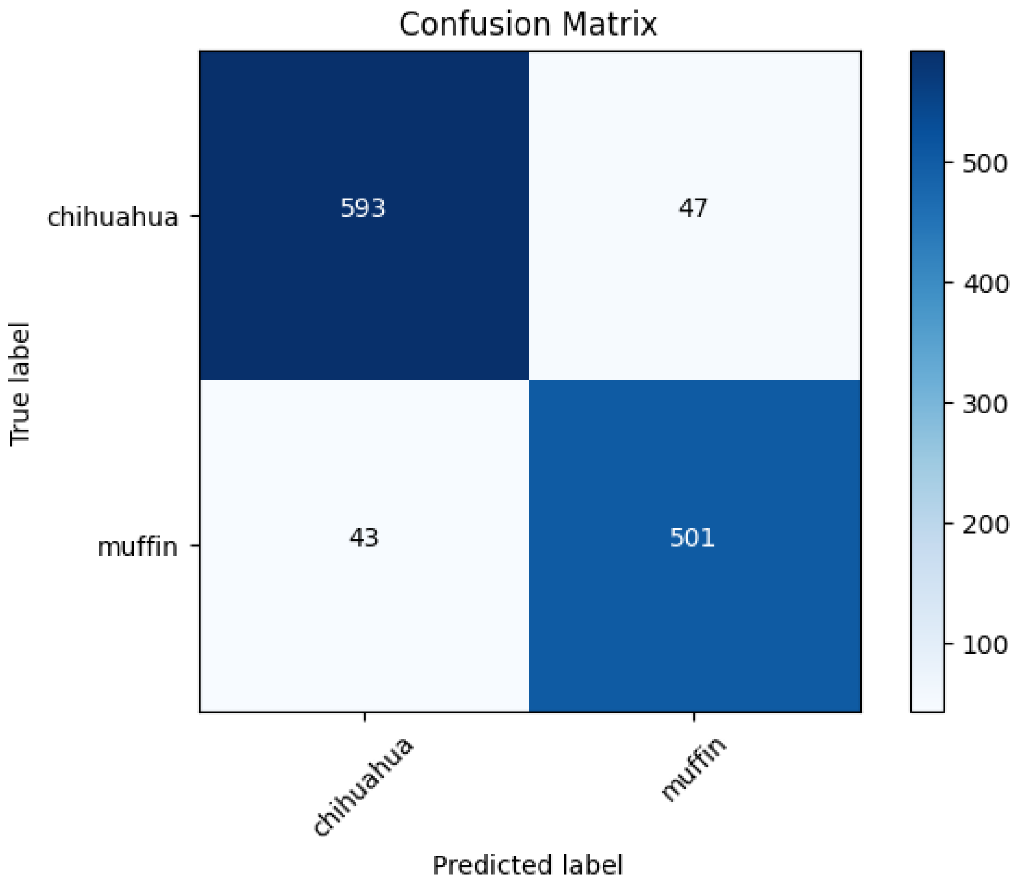

Figure 11: Confusion Matrix

as regards muffins, out of 544 images, 501 were categorized correctly and 43 incorrectly.

From these values it is possible to obtain measures to evaluate the model such as precision, recall and accuracy. The precision for a class is the number of true positives (TP) divided by the total number of elements labeled as belonging to the class; recall is defined as TP divided by the total number of elements that actually belong to the class; finally the accuracy is calculated as the total number of categorized data divided by the total data tested.

Below are the measurements obtained:

	Precision	Recall	Accuracy	Support
Chihuahua	0.93	0.93		640
Muffin	0.91	0.92		544
Total			0.92	1184

From the values obtained we note that the network performs slightly better on images of chihuahuas.

5 Cross Validation

Cross validation is a technique that consists of dividing the data set into different portions and iteratively testing the model on one portion while using the others to train it. The most used technique is called K-Fold cross validation, where the data set is divided into k folds and the model is trained on k-1 portions. The remaining portion is used to evaluate the model. For this project, given the smaller number of data available for the "muffin" class, the Stratified K-Fold Cross-Validation technique was used which ensures each fold maintains the same proportion of observations for each target class as the complete dataset.

Using `StratifiedKFold` of the sklearn library it was possible to obtain indices corresponding to the ends of the folds used, for this project 5 folds were created. For each iteration the model was trained and evaluated on a different fold, the results obtained were then compared with the true values of the labels using the 0-1 loss: this loss function returns 0 when the output is correct and 1 otherwise .

Below are the results obtained from each fold and the total average:

Fold	Loss
First fold	0.08
Second fold	0.10
Third fold	0.12
Fourth fold	0.11
Fifth fold	0.11
Total Average	0.10

The results obtained contain small fluctuations and are in line with the accuracy obtained in the test set, we can therefore consider the model robust.