Binary classification with Machine Learning

Convolutional Neural Networks (CNNs) for classification of cats and dogs images

Rita Folisi 982523
Giuseppe Lonardoni 07421A

Preface

We declare that this material, which We now submit for assessment, is entirely our 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 our work. We 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 us or any other person for assessment on this or any other course of study.

Contents

1 Introduction

In this report we will address the issue of binary classification, considering a dataset composed of dogs and cats images.

Classification is generally regarded as a typical task in the field of machine learning: it refers to the issue of predicting the correct label given an example data. It has several applications in many real-world scenarios, such as document classification or spam filtering. Indeed, since classification tasks are very common, several algorithms have been proposed in literature, such as K-Nearest Neighbour, Support Vector Machine and so on.

Our case involves binary classification, i.e. it requires to predict only two possible outcomes: each image can be labelled either as "cat" or as "dog". In this scenario, our proposed approach is to employ Convolutional Neural Network frameworks, which are more suitable in case of image classification.

In Section 2 we describe the environment and all the tools used during the project, including libraries and packages. In Section 3 we present the original dataset and describe all the preprocessing operations we have applied in order to build a larger and more fitting dataset. In Section 4 we illustrate a brief introduction to a general Convolutional Neural Network architecture (CNN). In Section 5 we introduce the chosen models and their architectures for our task. In Section 6 we explain the settings of our experiments, the results of which will be discussed in the following Section 7, along with some explanatory visual examples. In the last Section 8 we will draw our conclusion, commenting the results achieved and possible developments and improvements.

All the code is available on GitHub.[1]

2 Environment and used tools

The whole code for this project has been written in Python 3 on Google Colab, an online environment which allows the use of free cloud-based Jupyter Notebooks, a well suited tool for machine learning and data analysis work. We decided to use this platform because it provides access to computing resources, including some powerful GPUs, a rather useful ally when training or fine-tuning our models.

During the development of our project and the implementation of the models, we primarily used the following libraries:

- `OpenCV`, useful for dealing with images in the preprocessing phase;
- `Python Imaging Library - PIL`, to help OpenCV recognizing more image formats;
- `Numpy`, which allows large array manipulations;
- `Tensorflow`, open source library widely used in Machine Learning, since it provides a framework capable to build, train, and finally use deep learning models;
- `Matplotlib`, which allows to create plots;
- `Seaborn`, a data visualization library which refines plots obtained through Matplotlib.

3 Original Dataset and Preprocessing

This project is based on a dataset[2] provided by the professor through UnimiBox. This dataset is composed of 12500 dog images and 12500 cat images, totaling 25000 images. Therefore, the original dataset appears definitely balanced, and lets us avoid further operations to address any kind of class imbalance which could have had a negative impact in performances.

The whole folder has a size of 809 MB: the dogs folder has a size of 439 MB, while the cats one has a size of 370 MB.

The preprocessing procedure involved the following operations:

1. images removal;

[1] https://github.com/ritafolisi/MSA_project
[2] https://unimibox.unimi.it/index.php/s/eNGYGSYmqynNMqF

2. conversion from JPG to RGB or greyscale channels;

3. noise removal through closing;

4. resizing to 128x128 pixels;

5. data augmentation.

Skimming through the images in the dataset, we noticed that some images were not representative for our task, as they did not represent cats or dogs. Some examples can be seen in Figure 1. We also removed images in which both cats and dogs appear together and wrongly labelled images, since they could compromise the classification task.

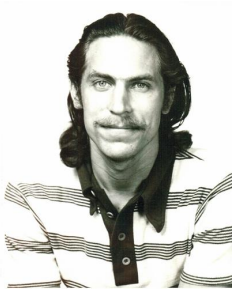

Figure 1: Some images removed from the dataset.

We then cropped the images with white borders, in order to obtain more homogeneous images.

The next step was to convert all the images from JPG to RGB channels: in this case, we also considered creating another folder to convert them in greyscale level, in order to make a comparison and see how the performances of the related models would be affected when working with RGB images and with greyscale images.

In order to reduce possible noise and to better extract features later on in the CNN architecture we initially considered applying a Gaussian filter, but through experiments we noticed that the derived models were not able to work properly, since the blurring applied to the whole dataset compromised the feature extraction. We then applied the closing operator, which is composed by the dilation followed by the erosion of an image. This operation is indeed particularly useful with greyscale level images, but can be advantageous to RGB images too when trying to detect peculiar features such as cat whiskers or dog hair.

Afterwards, we resized all the images to 128x128 pixels, in order to give more homogeneity and uniformity to our dataset, useful when training our models with these examples: designing a model that supports input images of variable size would result in a much more complicated architecture, since standard CNNs wouldn't suffice, and a Fully Convolutional Network would be required.

Since we chose to have squared images but some of them were originally rectangular, we opted to add a black padding. If we had used interpolation, we would not have maintained the proportions within each image; more discussions about this issue can be found in [Has19]. Furthermore, since the next and last step in our preprocessing pipeline will be data augmentation, there was no need to distort or stretch our images already: these kind of transformation will be applied anyway in the following phase.

Finally, we applied data augmentation. Even if the dimension of the dataset is not too low, we decided to increase its size in order to have better training and, as a consequence, better correctness in classifying images. In order to show how much the size of a dataset can influence the error rating, the performances obtained when using an augmented dataset will be compared to the ones obtained with the previous, non augmented dataset. For this purpose we created new images by applying some manipulations and transformations over a randomly extracted set of images (20% of the dataset): we applied rotation, shearing, zooming, flipping and brightness adjustment. These transformations are also useful to show more point of view from which we can approach to an image: for instance, shearing shows how an object can be seen from different angles, which is a useful human feature to replicate.

In Table 1 a brief summary of our generated datasets is provided.

Dataset	Number of dogs	Numbers of cats	Total items
RGB	12451	12445	24896
RGB augmented	24932	24892	49824
Greyscale	12451	12445	24896
Greyscale augmented	24932	24892	49824

Table 1: Datasets generated in this project

4 Introduction to CNN architecture

Artificial Neural Networks (ANN) are traditionally presented in analogy with the animal brain: they're computational structures whose graph representation resembles a biological neural network. An ANN is a collection of connected artificial neurons: these neurons can receive signals and re-transmit them to other neurons, via a connection which is typically weighted. In practice, this signal is a real number multiplied by a coefficient (the weight) that gets adjusted during the training phase, a way of indicating how "important" that signal should be for the net.

Neurons are typically organized in groups referred to as layers, and the function and shape of these layers determine the kind of neural network in use. The most commonly known type of network is the Feed Forward Neural Network (FFNN), where neurons of a layer are connected to neurons of the subsequent layer only: therefore the corresponding graph is acyclic and information flows from the top-most layer, the input layer, to the output layer, eventually traversing all the hidden layers in between. An example of the general structure can be seen in Figure 2.

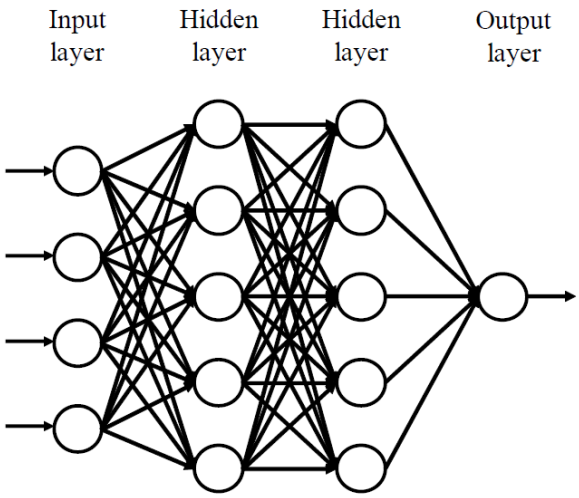

Figure 2: Example of a Feed Forward Neural Network architecture

Convolutional Neural Networks (CNNs) are a particular class of Feed Forward Neural Networks, particularly suitable for visual tasks and widely used in computer vision: they work in a fashion that resembles the functioning of the animal visual cortex, in which a neuron is activated only by a small region of its input. They work by applying a kernel (that can be thought of as a matrix, typically a square matrix) to a region of the input and calculating the convolution between this kernel and the region it covers: the operation is repeated over the full input by "sliding" the kernel to cover the full input matrix. The amount of movement of the kernel is regulated by the *stride*. The key idea in CNNs is applying filters to the input image, through the convolution operation between the kernel and the region covered, in order to extract features. The resulting output of each filter application is called *activation map* or *feature map*. This functionality provides an advantage over classical fully connected topology, as it allows the network to consider spatial correlation in its inputs, while a typical network needs to linearize the input.

Every convolutional layer effectively shrinks the input matrix by compressing a region into a single output for the next layer, allowing for the discovery of bigger and bigger patterns in the image by stacking convolutional layers in a hierarchical way. In order to maintain the size of the original input

and preserve information coming from the borders of the input image, the zero-padding technique has been proposed: the idea is to add a border of pixels all with value zero around the edges of the input images. In this way, the output of every convolutional layer will be:

$$w_{out} = \frac{w_{in} + 2p - k}{s} + 1,$$

where w_{out} is the output dimension, w_{in} is the input dimension, p is the padding size, k is the kernel size and s is the stride size.

The final part of a CNN is usually a fully connected network: the idea is that the convolutional part of the network extracts patterns from the input, even complex and big pattern that would take millions of parameters in a fully connected architecture, and these patterns are then passed as features to a classical network which learns some classification task on them. The overall structure can thus be divided into two sections: the first one is designated for feature learning and extraction, while the second one is employed in the classification task. An example of the general structure can be seen in Figure 3.

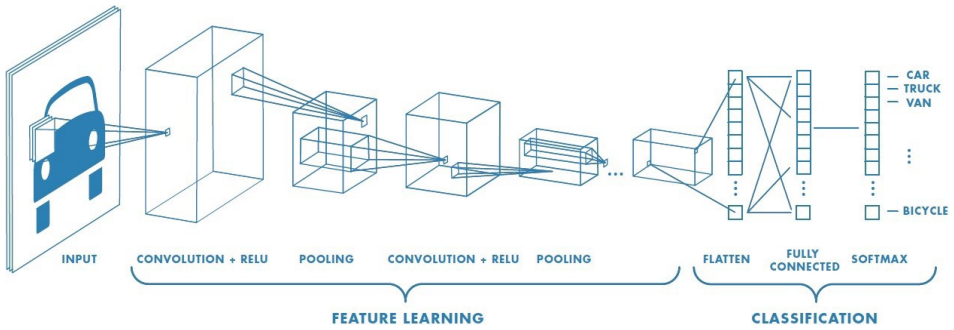

Figure 3: Example of a Convolutional Neural Network architecture

Besides convolutional layers, further operations are defined in order to better extract and learn features. In particular, it is important to reduce the sensitivity to the spatial location. In this way, the model will be spatially invariant, i.e. it will not be sensible to the absolute position of an object in order to classify the overall image. As a matter of fact, when classifying an image of a cat, it is not important where its snout is exactly located, but the relative position to its ears. Therefore, we are more interested in finding the correlation between features, rather than where that exact feature is located. To accomplish this it is possible to downsample the feature maps by using a pooling layer, by summarizing the information held by some pixels in just one.

CNNs are likely to quickly overfit due to the huge number of parameters in the model. To avoid this problem, one of the most used techniques is applying a dropout layer, which randomly drops out nodes (input and hidden layer) in a neural network by setting their activation to 0. Other ways to address the issue of overfitting is adding regularization terms to the convolutional layers, in order to apply weight penalties.

5 Implemented models

In this section we present the models we developed and used in our trials to find the best architecture for our task. After this section our experiments will be presented and the best model will be deployed for the classification task.

We decided to test and compare two kind of CNNs: one with a deeper reach, built using more convolutional layers and filters, and the other one being a shallower model with fewer layers. This will allow us to show the effects of different normalization techniques such as dropout on networks of different sizes, and to compare if and how performances relate to the size and number of parameters in the network.

5.1 Adaptation from VGGNet

The first developed model is freely inspired by the VGGNet-16 CNN. VGGNets are a set of deep convolutional neural networks developed by the Visual Geometry Group, an academic group from the Oxford University, that managed to score particularly well on many classification competitions. The trailing number in the name indicates the number of layers in the network: for example, VGGNet-16 has 16 layers, and VGGNet-19 has 19. While VGGNet is used in multiclass classification, our task is the theoretically simpler binary classification, so while this model will be inspired by the VGGNet architecture, it will also be much smaller.

The basic idea is to have "blocks" of convolutional layers delimited by a pooling layer that halves the size of the outputted matrix; each block has a growing number of convolutional layers of diminishing size but with a growing filter number, stacked on top of each other. The idea behind it is that this highly hierarchical structure should lead to a better pattern recognition. The first block has a convolutional layer with a predefined filter number of 32 and a pooling layer using max-pooling; the second block has two convolutional layers with 64 filters each before the pooling; in the third layer we have three convolutional layers of 128 filters, and so on. The last part of the network is a fully connected component of two layers respectively of 128 and 1 output neuron.

Both the number of blocks and the initial number of filters are parametrized, in order to try out different size of the same architecture.

Given the network size, also compared to the relatively tiny size of the images (only 128x128 pixels) and especially at scale, a number of regularization techniques are applied to the network, apart from the transformations on the dataset itself: each convolutional layer is followed by batch normalization; furthermore, dropout is in place for each layer, with a parametrized dropout rate that can eventually be adjusted at will.

5.1.1 Custom variation to the VGGNet model

Preliminary tests showed that the preceding model did not actually perform very well, and in order to obtain an even slightly better model, another variation was developed with the aim of breaking the strong regularity in the VGGNet-like structure: less layers are inserted in the model but with a bigger filter size; normalization and dropout are kept in the structure.

In Figure 4 a model summary is displayed.

5.2 ReetaNet Model

The second developed model was not inspired by any previous architecture, but it was built up through trial and error technique. The whole structure is composed by five convolutional kernels (size 3x3), followed by a max-pooling (size 2x2) to better extract the most meaningful patterns. In every convolutional layer, we applied a zero-padding, so that the layer's outputs will have the same spatial dimensions as its inputs. After the first two max-pooling layers, we added a Batch Normalization layer in order to speed up training and to improve the accuracy. After the other max-pooling layers, except the last one, we applied a Dropout layer, to prevent overfitting, with a rate of 0.2. After the last max-pooling, we applied a Flatten layer, a Dropout layer with rate 0.3 and two Fully Connected layers, with an activation function set to "ReLU" and "Sigmoid" respectively.

In order to optimize the performances, we tried to integrate kernel regularizers, by adding them to some convolutional layers. Notice that using them in each convolutional layer would lead to an underfitting, so we tried different combinations. For our experiments we employed L2 regularization penalty, which differs from L1 in avoiding sparse weights. However, since this model did not have issues with underfitting, in the end we realized that it would worked better without any kernel regularization.

In Figure 5 a model summary is displayed.

```
Model: "VGGNetLike_iteration_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 Conv2D_0_0 (Conv2D)         (None, 128, 128, 32)      896

 BatchNormalization_0_0 (Bat  (None, 128, 128, 32)     128
 chNormalization)

 Activation_0_0 (Activation)  (None, 128, 128, 32)     0

 Dropout_0_0 (Dropout)       (None, 128, 128, 32)      0

 MaxPooling2D_0 (MaxPooling2  (None, 64, 64, 32)       0
 D)

 Conv2D_1_0 (Conv2D)         (None, 64, 64, 64)        18496

 BatchNormalization_1_0 (Bat  (None, 64, 64, 64)       256
 chNormalization)

 Activation_1_0 (Activation)  (None, 64, 64, 64)       0

 Dropout_1_0 (Dropout)       (None, 64, 64, 64)        0

 Conv2D_1_1 (Conv2D)         (None, 64, 64, 64)        36928

 BatchNormalization_1_1 (Bat  (None, 64, 64, 64)       256
 chNormalization)

 Activation_1_1 (Activation)  (None, 64, 64, 64)       0

 Dropout_1_1 (Dropout)       (None, 64, 64, 64)        0

 MaxPooling2D_1 (MaxPooling2  (None, 32, 32, 64)       0
 D)

 Conv2D_2_0 (Conv2D)         (None, 32, 32, 128)       73856

 BatchNormalization_2_0 (Bat  (None, 32, 32, 128)      512
 chNormalization)

 Activation_2_0 (Activation)  (None, 32, 32, 128)      0

 Dropout_2_0 (Dropout)       (None, 32, 32, 128)       0

 Conv2D_2_1 (Conv2D)         (None, 32, 32, 128)       147584

 BatchNormalization_2_1 (Bat  (None, 32, 32, 128)      512
 chNormalization)

 Activation_2_1 (Activation)  (None, 32, 32, 128)      0

 Dropout_2_1 (Dropout)       (None, 32, 32, 128)       0

 Conv2D_2_2 (Conv2D)         (None, 32, 32, 128)       147584

 BatchNormalization_2_2 (Bat  (None, 32, 32, 128)      512
 chNormalization)

 Activation_2_2 (Activation)  (None, 32, 32, 128)      0

 Dropout_2_2 (Dropout)       (None, 32, 32, 128)       0

 MaxPooling2D_2 (MaxPooling2  (None, 16, 16, 128)      0
 D)

 Flatten (Flatten)           (None, 32768)             0

 Dense (Dense)               (None, 128)               4194432

 DenseResult (Dense)         (None, 1)                 129

=================================================================
Total params: 4,622,081
Trainable params: 4,620,993
Non-trainable params: 1,088
_________________________________________________________________
```

Figure 4: VGGNet-alike Model summary

```
Model: "ReetaNet_iteration_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 Conv2D_0 (Conv2D)           (None, 128, 128, 16)      448

 MaxPooling2D_0 (MaxPooling2  (None, 64, 64, 16)       0
 D)

 BatchNormalization_0 (Batch  (None, 64, 64, 16)       64
 Normalization)

 Activation_0 (Activation)   (None, 64, 64, 16)        0

 Conv2D_1 (Conv2D)           (None, 64, 64, 32)        4640

 MaxPooling2D_1 (MaxPooling2  (None, 32, 32, 32)       0
 D)

 BatchNormalization_1 (Batch  (None, 32, 32, 32)       128
 Normalization)

 Activation_1 (Activation)   (None, 32, 32, 32)        0

 Conv2D_2 (Conv2D)           (None, 32, 32, 64)        18496

 MaxPooling2D_2 (MaxPooling2  (None, 16, 16, 64)       0
 D)

 Dropout_0 (Dropout)         (None, 16, 16, 64)        0

 Conv2D_3 (Conv2D)           (None, 16, 16, 128)       73856

 MaxPooling2D_3 (MaxPooling2  (None, 8, 8, 128)        0
 D)

 Dropout_1 (Dropout)         (None, 8, 8, 128)         0

 Conv2D_4 (Conv2D)           (None, 8, 8, 128)         147584

 MaxPooling2D_4 (MaxPooling2  (None, 4, 4, 128)        0
 D)

 Flatten (Flatten)           (None, 2048)              0

 Dropout_2 (Dropout)         (None, 2048)              0

 Dense (Dense)               (None, 128)               262272

 DenseResult (Dense)         (None, 1)                 129

=================================================================
Total params: 507,617
Trainable params: 507,521
Non-trainable params: 96
_________________________________________________________________
```

Figure 5: ReetaNet Model summary

6 Experiments

Here we describe the process behind our experiments. We decided to perform experiments for the two architectures described in Section 5 and for each dataset we have generated in the preprocessing part. Our experiments are composed by a nested k-fold cross-validation, in order to model hyperparameter optimization and to have more robust results about the performances. In Subsection 6.1 we describe the process of hyperparameters tuning, while in Subsection 6.2 we describe the concept of k-fold cross-validation and how we implemented it. Finally, in Subsection 6.3, we describe the process in our experiments, the datasets employed and the choice of our metrics and loss functions.

6.1 Hyperparameters tuning

In a Machine Learning framework, the peculiar characteristic of the learning process is that a model adjusts, or "learns", by itself based on data it is fed. The training phase for a Neural Network model does just that, feeding data to the model which adjusts its internal weights. Nonetheless a model is not fully qualified by its weights, but there is a set of parameters that relate to the algorithm that are not affected by the learning process but rather affect it: these can be the activation function for each layer, the number of output neurons for a layer, the topology itself, the optimizer to guide the training, and so on.

These are called "hyperparameters", and play an important role in the accuracy of the resulting model. The process of choosing the best hyperparameters is referred to as hyperparameters tuning, or hypertuning: the base idea is to simply train the model many times, each time changing one of these hyperparameters, in order to try out all the possible combinations and choose the best performing one.

Different techniques have emerged to explore the hyperparameters' combinations space, the most naive being GridSearch: the whole search space is divided in discretized sets of values which the hyperparametrs are allowed to take, and the model is tested on these values only, providing a sort of sampling of the search space. A variant for GridSearch is RandomSearch, where instead of a regular grid over the hyperparameters, a random extraction is done each time. Both of these techniques often prove unhelpful due to the sheer size of the hyperparameters space: a model with only two hyperparameters, each of which may assume 4 different values, will need $4 \times 4 = 16$ runs to cover the whole search space, and adding another hyperparameter again with 4 values brings the amount of required runs to 64; this without considering parameters that may assume continuous values, as opposed to discrete ones (e.g., dropout rate is in the range $[0, 1] \in \mathbb{R}$). While highly inefficient, these brute-force methods remain common because the relations between each hyperparameter, or how one choice influences another, is not evident and often is not known at all (for example, while the effect of batch normalization is evident, its cause is still debated), and the only option left is to try them all, with great use of resources such as processing power and time.

For this reason, and because our development platform, Colab, has some limits that are easily reached, we opted for a novel approach called Hyperband [LJD+16] already implemented in the keras-tuner library[3]. This algorithm, as stated by the researchers behind it, has its primary focus on resource allocation, and instead of making heavy use of resources to optimize the selected configuration, it tries to find an optimal solution by improving solution evaluation, offering a fast way to rank randomly sampled combination of hyperparameters in order to focus only on the most promising ones.

For our experiments, we considered the following hyperparameters:

- `Optimizer`, used to change attributes in our models such as weights and learning rate, in order to reduce the losses and to get better results faster. The choice was among "adam", "nadam", "rmsprop".
- `Activation function` for each convolutional layer. The choice was among "sigmoid", "ReLU" and "softplus".

In Table 2 and in Table 3, results of hypertuning are shown for each architecture and each dataset. Notice that we only consider the best model (according to the zero-one loss) among all those created during the k-fold cross-validation for brevity reasons, but all the results are visible in our Github repository.

[3] https://keras.io/keras_tuner/

As you can see, while there are manifold choices for the optimizer, the activation function chosen is always the ReLU. This may happen because ReLU suits training of CNNs, since it helps to reduce the vanishing gradient problem. Another reason may be that ReLU is a sparse activation function and therefore it allows the network to focus on the informative features of each image and ignore the less important ones, making the network more robust to noise and less prone to overfitting. Moreover, ReLU is less computationally expensive, making it easier and faster for larger and deeper networks to be trained.

Hyperparameters	Final choice	Reference dataset
Optimizer	nadam	RGB
	rmsprop	RGB augmented
	adam	Greyscale
	nadam	Greyscale augmented
Activation Function	ReLU	RGB
	ReLU	RGB augmented
	ReLU	Greyscale
	ReLU	Greyscale augmented

Table 2: Hyperparameters choice for VGGNet-alike

Hyperparameters	Final choice	Reference dataset
Optimizer	adam	RGB
	nadam	RGB augmented
	nadam	Greyscale
	nadam	Greyscale augmented
Activation Function	ReLU	RGB
	ReLU	RGB augmented
	ReLU	Greyscale
	ReLU	Greyscale augmented

Table 3: Hyperparameters choice for ReetaNet

6.2 K-fold Cross Validation

Evaluating a model means assessing its performance in order to understand if it will perform well: this "performance" of a model corresponds intuitively to the expected risk, and what we want to calculate is an estimate of this risk. This estimation can be carried out through a procedure known as K-fold Cross Validation.

The estimation is a simple iteration of the usual method of fitting a machine learning algorithm on a training set and evaluating it on a separate test set. Given an initial dataset, we partition it in k parts, or folds, then for k times we choose a different fold to test the model on after having trained it on the remaining $k-1$ folds. At the end of the process we will have tested the model on k different folds: the errors obtained evaluating the model k times will be averaged and give us the cross-validation estimate for the expected risk of the model.

A variation exists to the standard cross-validation procedure, called the Nested K-Fold cross-validation, which allows for hyperparameter tuning during the estimation: during the training phase, the training part of the dataset is again split into a training part and a surrogate testing part, also called the validation set; the hypertuning performs a risk minimization among the combination of hyperparameters, using the result on the validation set to evaluate them. Therefore, at each iteration, the Nested cross-validation evaluates the model with the hyperparameters that minimized the risk for that iteration.

The parameter k in the (Nested) K-fold cross-validation can be chosen to increase the number of folds or their size: the extreme case of k being equal to the cardinality of the dataset is known as *leave-one-out*. Common values for k are 5, 10 or 12: in our experiments we decided to operate a 5-fold Nested cross-validation.

6.3 General settings of our experiments

As specified before, in our experiments we employed a nested 5-fold cross-validation for our four datasets, in order to compute the risk estimation. Therefore, we generated 20 models in total for each architecture. In this way we can have more robust and precise insights about the performances, with much less bias.

In building our datasets, Keras applied an automatic Label Encoding to the labels "cat" and "dog", in order to better compute the error between the prediction and the actual label. Therefore, for this project "cat" was assigned the label "0" (False), while "dog" the label "1"(True).

Before getting started with our experiments, we needed to split each dataset in training set, validation set and test set. Firstly, we split the whole dataset in training set ($TrainSet_1$) and test set ($TestSet_1$)in order to perform the external 5-fold cross-validation. In this case, the split was set to 0.8 for training and 0.2 for testing. In each fold, we perform the hypertuning through Hyperband. Therefore we split $TrainSet_1$ into a secondary training set ($TrainSet_2$), a validation test ($ValSet$) and a secondary test set ($TestSet_2$). As a matter of fact, Hyperband needs both a training set and a validation set in order to choose the best configuration of hyperparameters. We decided to also create another test set, in order to evaluate the performances after each iteration of the Hyperband. For this second part, the split was set to 0.2 for $ValSet$ and 0.1 for $TestSet_2$. In Table 4, all the sets with their size are presented.

Original dataset	$TrainSet_1$	$TestSet_1$	$TrainSet_2$	$ValSet$	$TestSet_2$
RGB/Greyscale augmented	39872	9952	27911	7974	3987
RGB/Greyscale non augmented	19936	4960	13955	3987	1994

Table 4: All the sets generated from each dataset

Furthermore $TrainSet_1$ and $TestSet_1$ are rather class-balanced. Regarding the non augmented datasets, we have 9973 cats versus 9963 dogs in $TrainSet_1$ and 2472 cats versus 2488 dogs in $TestSet_1$. In the augmented datasets, we have 19934 cats versus 19938 dogs in $TrainSet_1$ and 4958 cats versus 4994 dogs in $TestSet_1$.

During training, we decided to use the Binary Cross Entropy (or Log Loss) as training loss function. This loss function increases when the prediction diverges from the target label. In Figure 6 a graph of the loss function is shown, when the actual label is equal to 1. As the prediction approaches 1, the loss function slowly decreases. On the other side, when the prediction decreases, the log loss increases quickly. Therefore, there is higher penalization for those errors with high confidence.

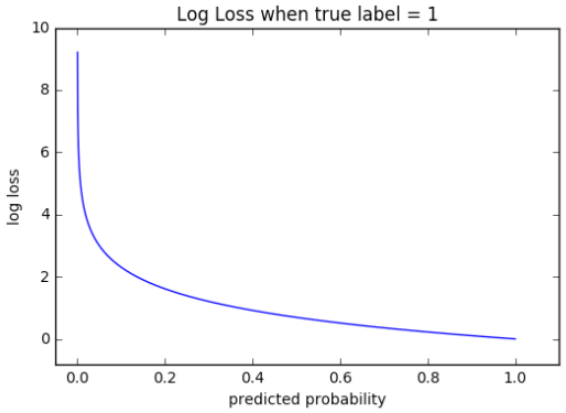

Figure 6: Binary Cross Entropy / Log Loss function

For the risk estimate, instead, we employed the zero-one loss function, which is defined as follows:

$$l(y, \hat{y}) = \begin{cases} 0 & \text{if } y = \hat{y} \\ 1 & \text{otherwise,} \end{cases}$$

where y is the target label and $\hat{y}$ the prediction made by the model.

In our experiments, we did not choose a fixed number of epochs, but we relied on Early Stopping. This is a useful tool to stop training when the chosen metrics do not significantly improve over a certain number of epochs defined by the paramater `patience` (here set to 6).

Regarding the metrics employed to evaluate our models, our choice fell on the following ones. For all the metrics listed here, the range is from 0 to 1.

- `Accuracy`, defined as the number of correct prediction made by the model divided by the total number of predictions made.
- `Recall`, defined as the number of positive instances correctly predicted divided by the total number of positive instances. Through this metric, we can measure the ability to detect positive samples.
- `Precision`, defined as the number of positive instances correctly predicted divided by the total number of instances classified as positive. Through this metric, we can measure the reliability in classifying an instance as positive, without misclassifying a negative instance as positive.
- `AUC`, defined as the area under the ROC Curve. The ROC Curve is a graph created by plotting the Recall against the false positive rate (FPR). Given a randomly selected positive instance and one randomly selected negative instance, AUC explains how much the model can distinguish between the two classes. The higher the AUC, the better the model is at predicting the corresponding class.
- `Specificity`, defined as the number of negative instances correctly predicted divided by the total number of negative instances. Through this metric, we can measure the ability to detect negative samples.

Notice that those metrics are usually employed when dealing with unbalanced data. Even though this is not the case, we decided to consider these metrics in order to have a better analysis on how our model works. For example, without considering the specificity, the model could have great precision and recall just simply predicting everything as true.

For the sake of completeness, we also considered the values of true positive, true negative, false positive and false negative in our statistics.

7 Results and Visualization

After performing our experiments, we collected all the results regarding the performances within tables and graphs. In Table 5 and Table 6 the average values for each metric within the 5-fold cross-validation are presented for ReetaNet and VGGNet-alike.

Dataset	Accuracy	Precision	Recall	AUC	Cross Entropy	Zero-One Loss
RGB augmented	0.804	0.826	0.877	0.858	0.380	0.196
RGB non augmented	0.818	0.824	0.913	0.872	0.357	0.181
Greyscale augmented	0.787	0.799	0.870	0.848	0.416	0.213
Greyscale non augmented	0.796	0.814	0.877	0.861	0.390	0.204

Table 5: Average results from 5-fold cross-validation for ReetaNet

Dataset	Accuracy	Precision	Recall	AUC	Cross Entropy	Zero-One Loss
RGB augmented	0.821	0.723	0.724	0.866	0.341	0.179
RGB non augmented	0.865	0.911	0.825	0.961	0.355	0.135
Greyscale augmented	0.755	0.754	0.554	0.848	0.507	0.245
Greyscale non augmented	0.845	0.895	0.813	0.952	0.418	0.155

Table 6: Average results from 5-fold cross-validation for VGGNet-alike

On average, we noticed that the augmented datasets had worse performance than the non augmented ones. This may be explained because sometimes data augmentation can lead to a slight

overfitting and therefore it results in worst performances. Moreover, we noticed that not all the models generated in each fold have the same training behaviour. Some models train for more epochs, while others train for fewer ones, resulting in worst performances. This can be seen clearly in Figure 7 and Figure 8, where some models train for 20 epochs, other for 10 epochs or even less. This is due to the Early Stopping technique we described before: unfortunately, because of the limited resources provided by Colab, we could not run those models on more epochs. We also noticed that there is always a fold in which the model generated goes much worse than the average. An example can be seen for the RGB augmented dataset employing ReetaNet model: while in the other folds the accuracy is about 85% (in the third fold we have 95%), there is one fold (in this case, the last one) in which the accuracy is 50%. This has a great impact on the statistics and we noticed that it happened in most of the cases. Our hypothesis is that there could be a bad initialization of the weights and the model does not learn properly (in correlation also with the previous problem of few epochs highlighted before). Another hypothesis could be an unfortunate split of the data, with some bias within the training dataset that does not help the model to learn how to identify patterns.

In general, we noticed better performances with RGB datasets over the greyscale ones. Indeed, the color is an important feature to consider when distinguishing cats from dogs, therefore this behaviour is completely reasonable and under our expectations.

In general the performances of both architectures are quite similar, notwithstanding that VGGNet-alike is a much more complex and deeper architecture. Besides all the hypothesis made before, we also thought that it could have issues with vanishing gradient, since it is very deep. This would explain why in some folds there is no improvement of the metrics and therefore the number of epochs is lower, due to the Early Stopping. A solution could be to consider an upgrade of the architecture, by integrating an auto-encoder.

In Figure 9 and 10, all the plots, except for specificity, are normalized. Indeed, in training we have much more data than in testing and therefore comparing pure numbers would result in a confusing visualization.

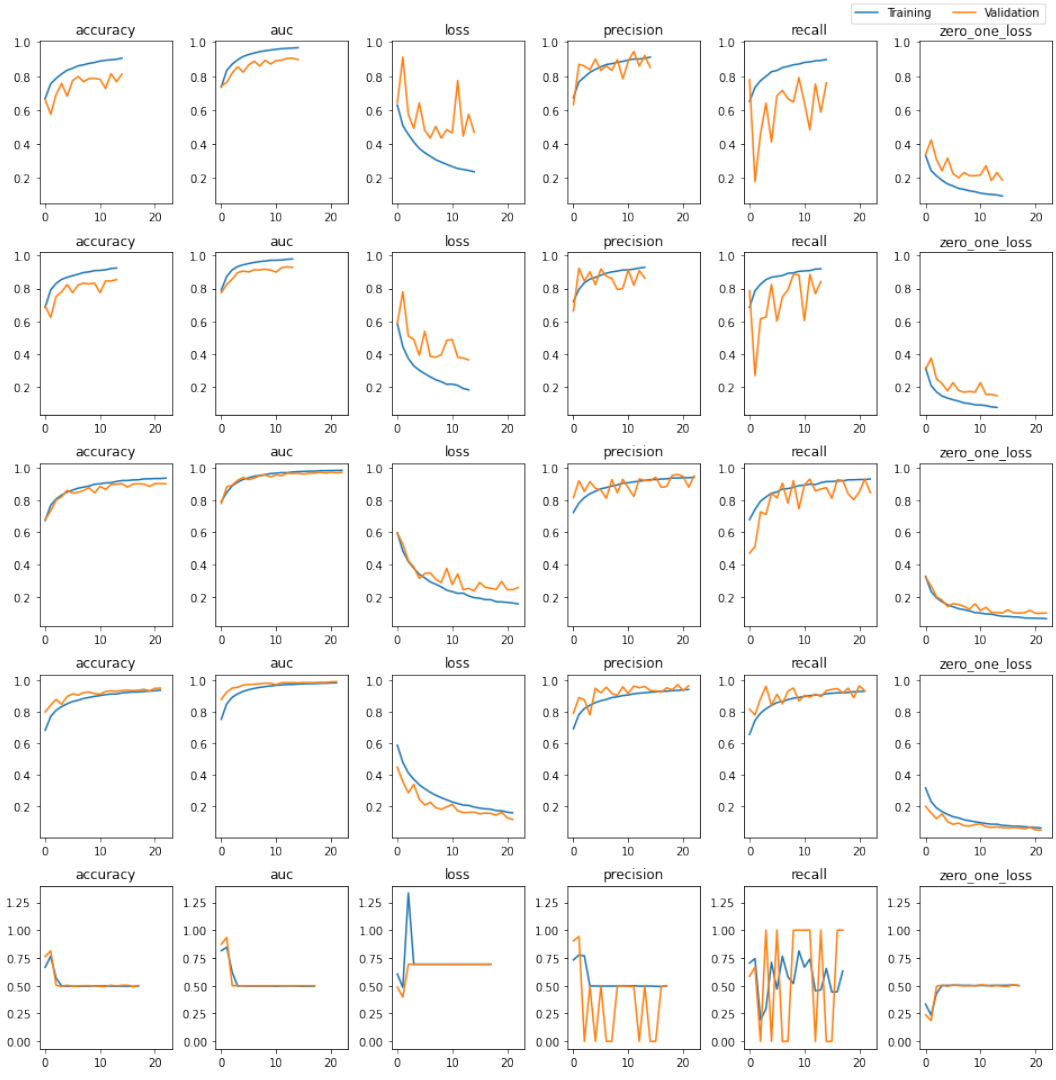

Figure 7: Plots for VGGNet-alike with RGB augmented dataset

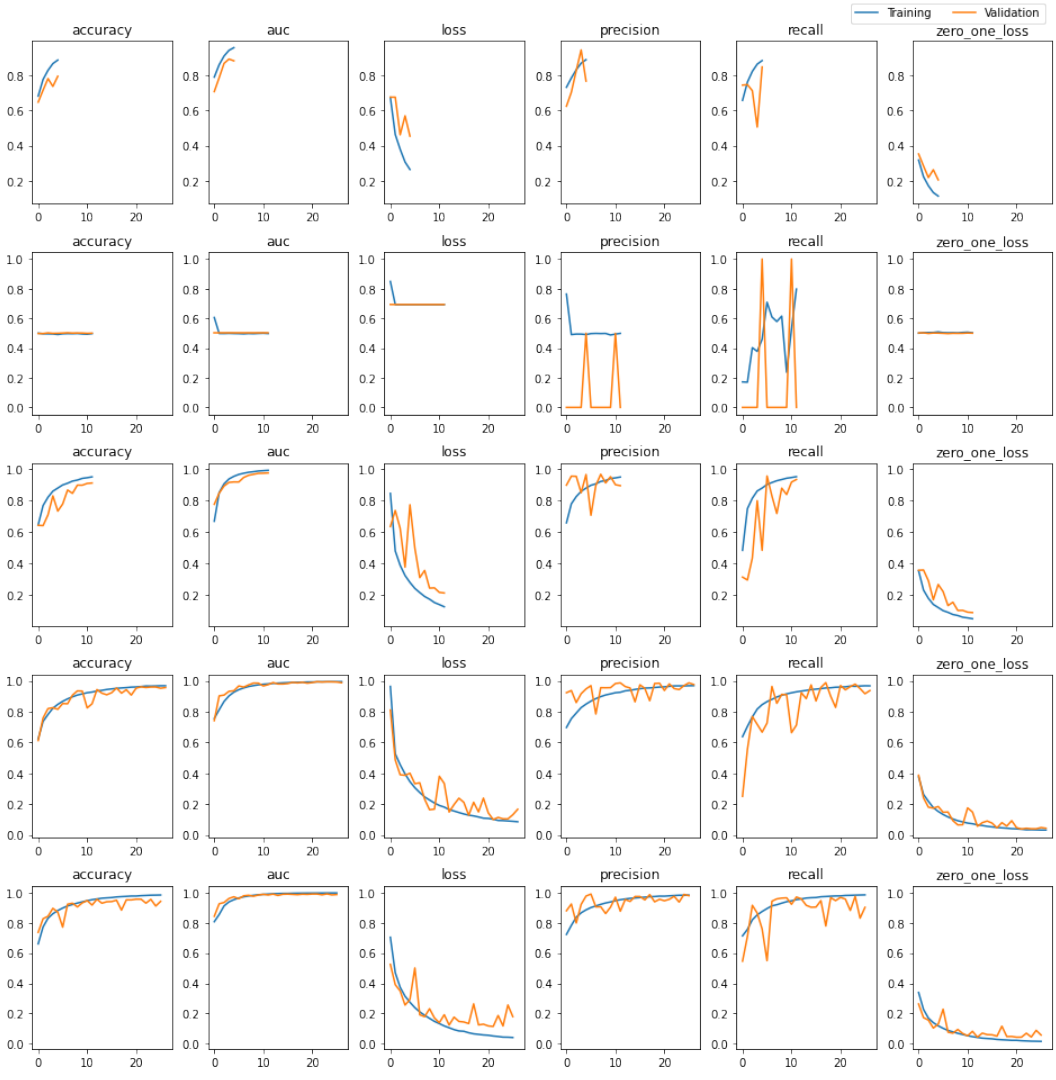

Figure 8: Plots for VGGNet-alike with RGB augmented dataset

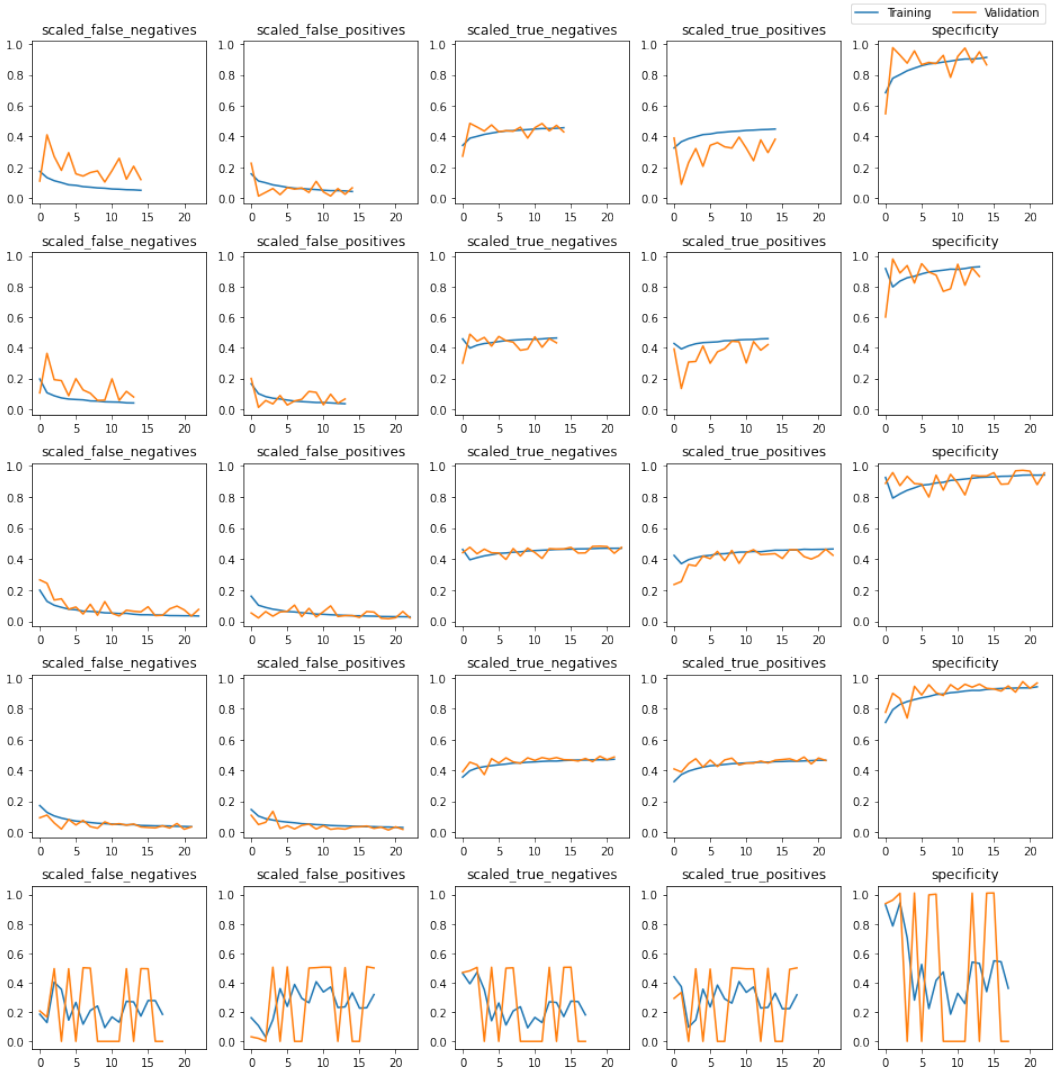

Figure 9: Further plots for VGGNet-alike with RGB augmented dataset

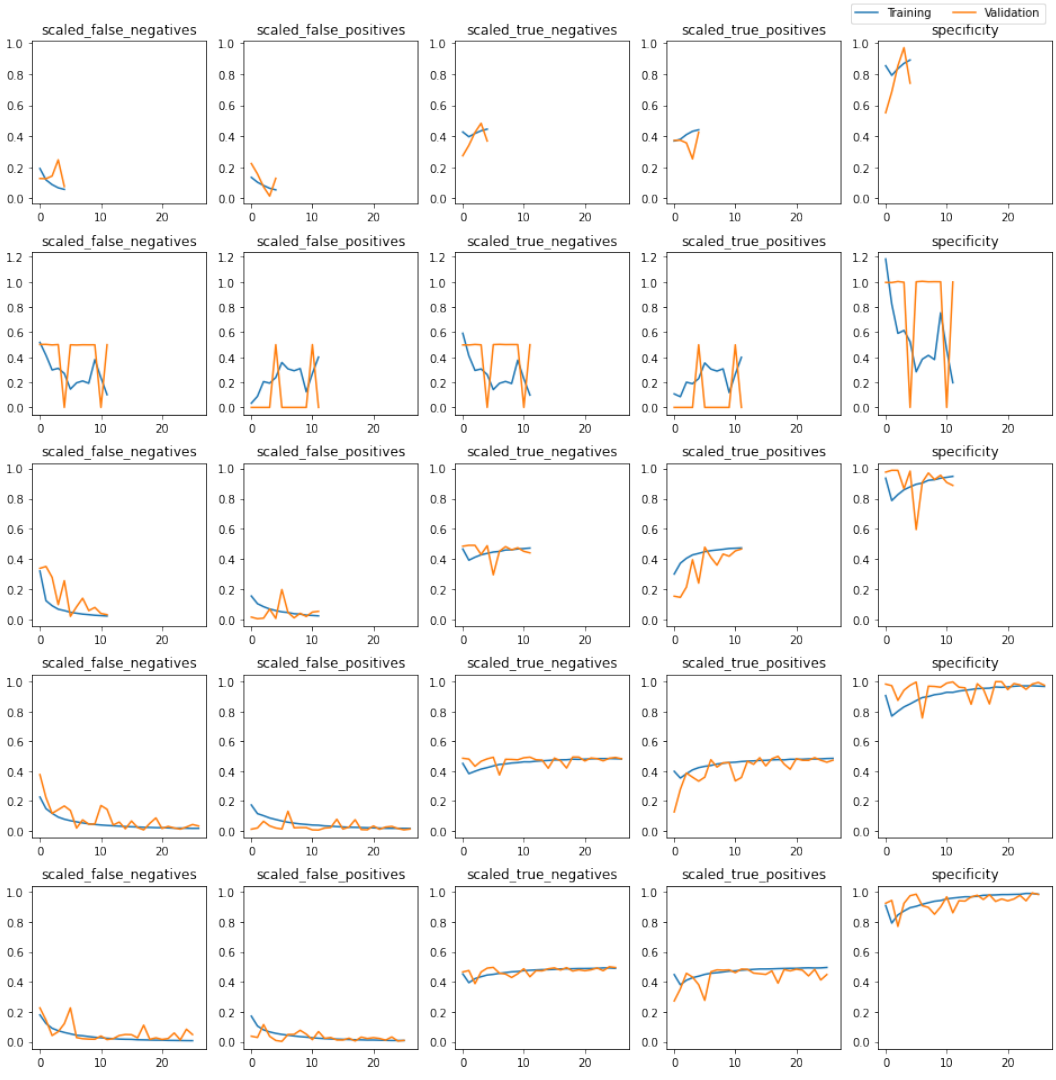

Figure 10: Further plots for VGGNet-alike with RGB augmented dataset

7.1 Visualization

To complete our analysis regarding these two architectures, we decided to employ other visualization tools. For this part, we chose the best model from each architecture obtained during the cross-validation. The reference dataset is the RGB augmented one. A recap of the two best models can be seen in Table 7.

Architecture	Accuracy	Precision	Recall	AUC	Cross Entropy	Zero-One Loss
VGGNet-Alike	0.956	0.976	0.937	0.988	0.167	0.044
ReetaNet	0.95	0.97	0.934	0.992	0.116	0.049

Table 7: Best models extracted from cross-validation

First of all, we generated the confusion matrix, in the form of heatmaps, visible in Figure 11. As it can be seen, the models have a good response in detecting true positive and true negative, as it can be also seen in the high values of other metrics such as recall or specificity.

Figure 11: Confusion Matrix of VGGNet-alike and ReetaNet respectively

It is possible to spot a slight overfitting in both models. In order to better visualize this behaviour and understand our models, we tried to exploit a new visualization tool, the GradCAM [SCD+19]. A GradCAM (Gradient-weighted Class Activation Mapping) is a useful tool to have an insight on how a black-box model such as a CNN is working. It provides a visual representation of the attention focus the CNN holds when evaluating an image. Without entering in the technical details, through this tool it is possible to produce a coarse localization map highlighting the important regions in the image for predicting the label. In other words, it shows "where the CNN is looking" to predict the label of an image.

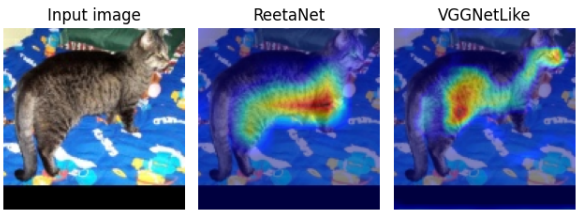

Figure 12: Comparison GradCAM in ReetaNet and VGGNet-alike (target label equal to cat)

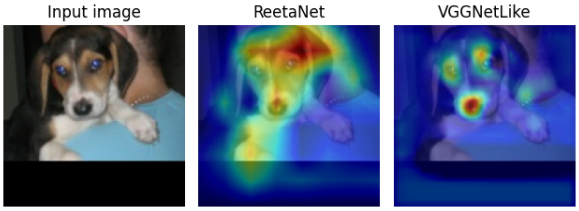

Figure 13: Comparison GradCAM in ReetaNet and VGGNet-alike (target label equal to dog)

In Figure 12 and 13 it is possible to see that in general both models seem to be able to recognize in which area the animal is located. In some cases, for example in 13, it is possible to notice that the VGGNet-alike is focusing on the eyes and nose of the dog in order to make its prediction. On the other hand, ReetaNet is activated on a larger area of the images. This difference is quite present also in other images. Our conclusion is that it is possible that VGGNet-alike overfits slightly more than ReetaNet. As a matter of fact, VGGNet-alike is focusing too much on really small details.

Lastly, we displayed some activation maps in order to take a look on how an image would be processed inside the network. Some example can be seen in Figure 14 and 15, but further visualizations are available on the repository.

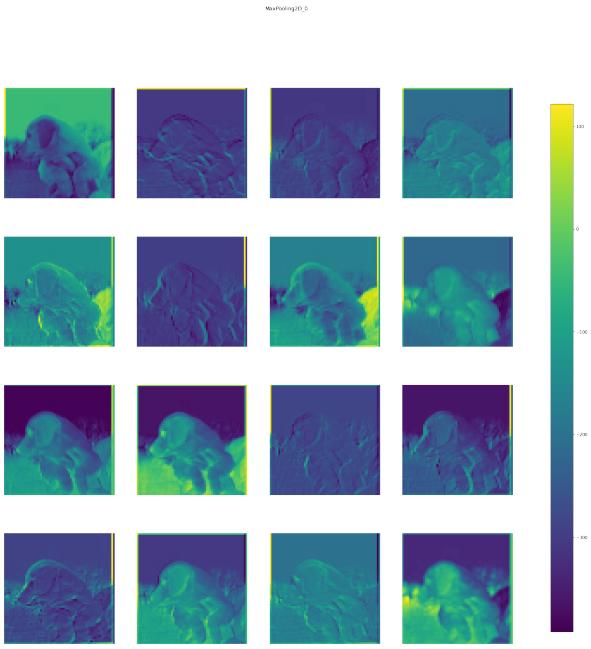

Figure 14: Activation Maps in ReetaNet

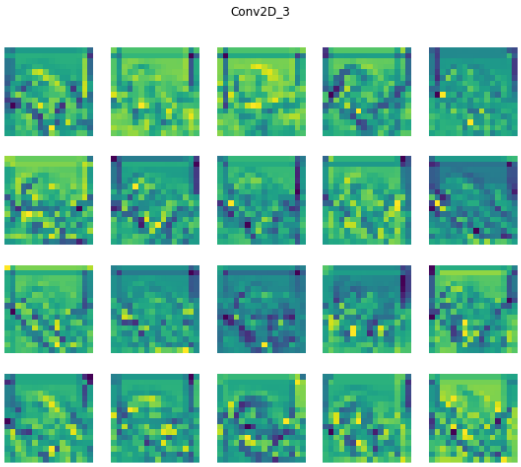

Figure 15: Activation Maps in VGGNet

8 Conclusions

In this report we addressed the issue of binary classification, considering a dataset composed of dogs and cats images. Our approach was to develop two independent CNNs, VGGNet-alike and ReetaNet. After a first preprocessing phase, we employed a nested 5-fold cross-validation. In general, we obtained good results, especially with the non-augmented RGB version of the dataset. Some improvements concern having a better setting without limitation on resources (as it happens in Colab): in this case, it would be useful to tune more hyperparameters and augment the number of epochs in order to build better models. Moreover, the quality of the images within the datasets could have an impact also on the performances. Lastly, future developments of this project involve using transfer learning with pre-trained models, integrating with different datasets and building new architecures.

References

[Has19] Mahdi Hashemi. Enlarging smaller images before inputting into convolutional neural network: zero-padding vs. interpolation. *Journal of Big Data*, 6, 11 2019.

[LJD+16] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. 2016.

[SCD+19] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-CAM: Visual explanations from deep networks via gradient-based localization. *International Journal of Computer Vision*, 128(2):336–359, oct 2019.