Binary classification with Machine Learning

Neural Networks for classification of chihuahuas
and muffins images

Elia Togni[1]

[1]Dipartimento di Informatica, Università degli Studi di Milano.

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of our work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any spart of it, has not been previously submitted by us or any other person for assessment on this or any other course of study.

Abstract

This project utilizes Keras to train neural networks for binary classification of muffins and Chihuahuas using image datasets. With a focus on experimenting with different network architectures and training hyperparameters, 5 fold cross validation is employed to compute risk estimates. Results indicate reasonable performance from both multi-layered neural networks and Convolutional Neural Networks (CNNs), with CNNs showing promise, particularly the TogNet CNN. However, concerns of overfitting were noted. The segmentation task revealed better performance by multilayered networks, while data augmentation proved effective in improving model generalization. Nonetheless, the small dataset size impacted task performance. Future work could benefit from increased computational power for more experiments and refined tuning approaches to enhance model performance further.

Contents

List of Figures

List of Tables

I Introduction

1 Image Classification problem

Image Classification is one of the most fundamental and studied topics in the field of machine learning and it refers to the ability to understand and categorize an image as a whole under a specific label.

In Image Classification, given an input image, the goal is to predict the class which it belongs to. While this is not a big deal for humans, when it comes to mimicking perception, computer usually perform fairly poorly compared to them [1]. Indeed, teaching computers to see is a difficult problem that has become a broad area of research interest and both classic computer vision and **Deep Learning** techniques have been developed for decades now with this intention in mind. Classic techniques use **local descriptors** (algorithms or methods that capture information about the local appearance or features of a specific region in an image) to try to find similarities between images, but today, advances in technology allow the use of Deep Learning techniques to automatically extract representative features and patterns from each image [2].

The modern techniques we are referring to are the **Convolutional Neural Networks** (henceforth CNN), a particular class of neural networks commonly used to analyze visual inputs. They are characterized by the extensive application of the convolution operation (and hence the name) to selectively extract features from small portions of the input.

Image Classification is now broadly used in lots of application fields such as medical imaging where they are used to identifying and diagnosing diseases from medical images like X-rays, or such as security and surveillance, where facial recognition, for example, could grant a higher level of security and access control.

2 Chihuahuas vs Muffins classification

This report delves into the challenge of binary classification, specifically, the task of predicting two distinct outcomes. For instance, in our case, each image is categorized as either a *chihuahua* or a *muffin*. In this context, we propose a comparative analysis between the employment of Multi-Layer Neural Network and Convolutional Neural Network frameworks, with a focus on their suitability for image classification.

Our objective is not only to assess the performance disparities between these two structures but also to examine how variations in the network's hyperparameters impact overall performance.

3 Overview

The datasets used include raw RGB images, augmented RGB images, and segmented RGB images. Data augmentation was employed to improve model generalization by exposing the model to a broader range of scenarios during training. Despite these efforts, the relatively small size of the dataset posed a challenge, potentially impacting the effectiveness of the models.

The chosen architectures are:

- **MLNN 512_256_128**: This architecture comprises three dense layers with 512, 256, and 128 units, respectively. It showed moderate performance improvements with data augmentation but struggled with segmented data;
- **CNN 32_64_128**: A more complex model with three convolutional layers followed by max-pooling layers, this architecture performed exceptionally well on the training data with both raw and augmented RGB images. Its performance declined significantly with segmented data, likely due to the loss of critical spatial information. On the testing data, however, the model showed signs of overfitting;
- **TogNet**: A custom model that achieved the highest performance with raw RGB data, demonstrating excellent accuracy, precision, and recall. While augmentation slightly reduced precision, the overall performance remained robust. Segmented data again resulted in decreased performance.

The primary challenge encountered was overfitting, evidenced by the disparity between training and validation losses. To mitigate this, dropout layers were employed, and the performance was monitored using additional metrics such as binary accuracy and precision.

All the code is available on GitHub[1].

II Environment and used tools

Prior to delving into the intricacies of different theoretical frameworks and implementation decisions, this section introduces the primary tools and environment utilized for running the developed code. These choices, in my assessment, are pertinent for executing the models and setting up experiments effectively.

The whole code has been written in a Google Colab environment, which provides Jupyter-notebook, a free notebook for Python3-code execution. I initially chose the Conda environment to work locally but the computational complexity of the task made me rethink my choices. Therefore, I finally chose this environment because of the provided computational power, its simplicity and intuitiveness, which made easier to code with multiple libraries.

Here I present a brief list of the main packages and libraries used:

- Numpy
- Pandas
- PIL
- os
- shutil
- matplotlib
- google.colab
- sklearn
- skimage
- tensorflow
- keras
- gspread

[1] https://github.com/EliaTogni/Statistical-Methods-for-Machine-Learning-Project

III Chihuahua vs Muffin Dataset

1 Data Organization

This project is based on a dataset[2] provided by the professor through Kaggle. This dataset is composed of two different folders for each class of images, a **training** one and a **test** one.

The training folder initially contains 2559 chihuahua images and 2147 muffin images, while the test one initially contains 640 chihuahua images and 544 muffin images.

The whole folder has a size of $510,7$ MB: the chihuahuas folder (comprising both the training and the test folders) has a size of $206,1$ MB, while the muffin one has a size of $304,3$ MB.

After a preliminary analysis, it seemed evident that the dataset was composed of images with different dimensions and weights. These differences will need to be addressed in the preprocessing phase and, for this reason, I chose not to utilize the already prepared folders but to merge the training and the test folders into one for each class, postponing the evaluation and, eventually, the resolution of the possible unbalancing of the data after the image removal step. The current folders contain 3199 chihuahua images and 2718 muffin images.

2 Methodology

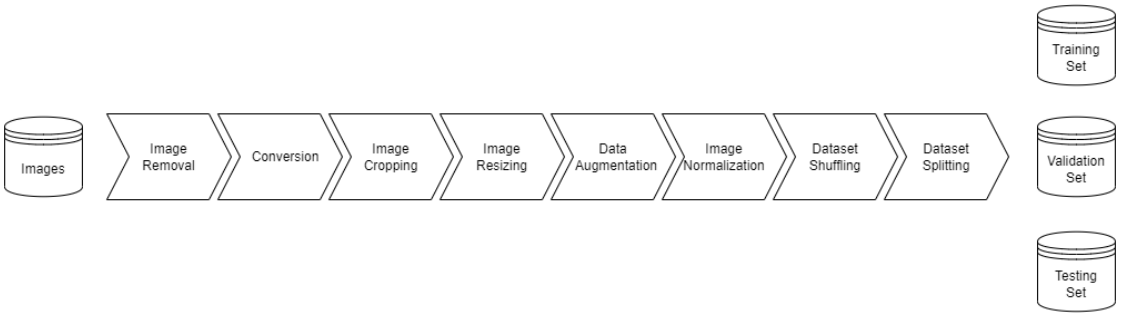

Figure 1: Pipeline of the preprocessing step

The preprocessing steps for the Chihuahua vs Muffin dataset involve several key processes to prepare the data for analysis. First, the dataset undergoes image removal, which includes the visual inspection for images that are not compliant with the given task. Next, unbalance resolution addresses any disparities in the number of images between the two categories. Image conversion ensures uniformity in image format, followed by cropping and resizing to standardize image dimensions. Data augmentation expands the dataset by applying three different type of transformations to existing images. Image normalization adjusts pixel values in the range [0, 1] to simplify the training step. Finally, dataset shuffling randomizes the order of samples for unbiased training and testing. These preprocessing steps collectively enhance the quality and suitability of the dataset for the given machine learning task.

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

3 Dataset Preprocessing

3.1 Images Removal

Given the relatively small size of the dataset, the very first preliminary operation will be a visual inpection made manually, skimming through the images in the dataset and looking for evident errors in classification, like muffin in the chihuahua folder or viceversa, or images that are not representative for our task, as they would not represents one of the two classes at all. The content of the images is various: indeed, often chihuahaus and muffins are not the main content of the image, multiple subjects (like people, present in both classes) appear in the images, a lot of details in background and around the subjects create noise and sometimes also images whose subject belongs to both classes or to neither of them appear in the datasets.

Chihuahua dataset After the first skimming through the chihuahua folder, it is possible to divide the content of the folder into three different subsets:

- chihuahua images (photos or drawings);
- different breeds of dog images;
- not dog images.

The reason for this subclassification will be made clear in the following paragraph.
My first operation will be the removal of the latter subset from the folder, since it would certainly compromise the classification task. The removed images comprise muffins (wrongly labelled images), muffins and chihuahuas (images in which both classes appear together) or neither of them (totally unrelated images). Some examples (e.g., a hospitalized woman, a muffin, both chihuahuas and muffins and a wood plank) can be seen in Figure 2.

Figure 2: A sample of the removed images from the Chihuahua dataset

The folder after the removal contains now 3161 images of different breeds of dogs, so only 38 images were removed.

While what the content of the images that will be fed to the NNs should be clear - chihuahua images, of course - through the initial skimming, I noticed a huge number of different breeds of dog images and, also, of mixed breed dogs. I decided to keep all of them because, by including mixed-breed and different breed images during training, the model learns to recognize common features and patterns shared across different breeds, improving its ability to classify images accurately.

Muffin dataset I followed the same reasoning for the muffin images removal. After the first skimming through the muffin folder, I noticed that it was possible to divide the content of the folder into two different subsets:

- muffin images;
- not muffin images.

Again, I made some strong assumptions in the removal process: I considered the raw muffin batter, while it was already in the stamp, to not be a muffin yet and, therefore, I removed it from the dataset.

Consequently, all the images representing chihuahuas (wrongly labelled images), muffins and chihuahuas (images in which both classes appear together) or neither of them (totally unrelated images) have been removed. Some examples (e.g., a plant of marijuana, an ill child, the ingredients necessary to make muffins but not a fully formed muffin yet and empty muffin stamps) can be seen in Figure 3.

Figure 3: A sample of the removed images from the Muffin dataset

The folder after the removal contains now 2590 images of muffins, so 128 images were removed.

3.2 Unbalance Resolution

Classifiers usually assume the training data to be composed by balanced data and an equal cost of misclassification, that is, the consequences or costs associated with making a particular type of mistake in classification are the same for all classes. In other words, the classifier treats all types of errors (false positives and false negatives) equally costly[3].

Naive classifiers attempt to reduce global quantities like the error rate, thus tending to misclassify examples from the minority class. The risks of learning from an unbalanced dataset are mostly identified by the failure of generalizing inductive rules for minority class, that is, the difficulty in learning over more features but less samples and the fact that naive learners are often biased towards the majority class, and by the risk of overfitting[3].

But when a dataset is unbalanced? Assume points to be vectors $x \in \mathbb{R}^n$. It is possible to distinguish between **relative rarity** and **absolute rarity**. With the term relative rarity are identified the cases in which the minority class is outnumbered, but not necessarily rare, e.g., $10^6|10^3$. In this case, the minority class can be accurately learned. With the term absolute rarity are identified the cases in which the minority class is not just outnumbered, but also not well defined, e.g, $10^3|1$.

One common heuristic to determine if a dataset composed of two different classes is unbalanced is to calculate the class distribution or class **Imbalance Ratio**. Here's a simple heuristic:

$$\text{Imbalance Ratio } \rho = \frac{\text{Number of instances in Majority Class}}{\text{Number of instances in Minority Class}}$$

If the Imbalance Ratio is close to 1, the dataset is relatively balanced. If the imbalance Ratio is significantly greater than 1, it indicates an unbalanced dataset, with the larger value representing the degree of imbalance.

Class Imbalance Degree	**Proportion of Minority Class**
Extreme	<1% of the dataset
Moderate	1 - 20% of the dataset
Mild	20 - 40% of the dataset

Table 1: Classification of degree of imbalance in data [4].

Good results can be obtained, regardless of class disproportion, if both groups are well represented and come from non-overlapping distributions[5]. In the considered case, the Imbalance Ratio is equal to $1,22046$ (3161 instances of chihuahuas images vs 2590 instances of muffins images), which is a score sufficiently close to 1 to proceed without having to make any action to fix the unbalance.

3.3 Image conversion to RGB

The next preprocessing step involves converting the dataset into the RGB color scheme.

3.4 Image Cropping and Image Resizing

A recurrent characteristic of the dataset was the presence of a large white border around several images. The white border surrounding an image contains no relevant information for the classification task, therefore I trimmed the borders to help the CNN focus its attention on the essential features within the image, reducing noise and potential distractions that could hinder classification accuracy. Removing this border was made to help the model focus on the discriminative features within the image, making it more robust to variations in background and irrelevant details.

One of the defining characteristics of Neural Networks is that they receive inputs of the same size: each neuron so all images need to be resized to a fixed dimension before inputting them to the CNN. It is possible to resize the images larger than the fixed size (in one dimension or both) down to the desired one using three possible approaches[6]:

1. further cropping their border pixels;
2. scaling the images down using interpolation;
3. zero-padding.

Cropping could cause the missing of features or patterns that appear in the peripheral areas of the image. Scaling could deform the features or patterns across the image. However, since deforming

patterns is still preferable than losing them, scaling results to be the reasonable choice to resize larger images[6]. Therefore, I decided to utilize zero-padding because it has two advantages in comparison with scaling: it does not carry the risk of deforming the patterns in the image, while scaling does, and it speeds up the calculations in comparison with scaling, resulting in better computational efficiency. The reason of this improved efficiency is that neighboring zero input units (that is, pixels) will not activate their corresponding Convolutional unit in the next layer[6].

A choice of great importance in terms of model performance is the fixed size. A size that is too large will result in high accuracy but extensive memory usage and a larger neural network. Thus, increasing both the space and time complexity. On the other hand, a size that is too small will result in a more manageable but less accurate network. It is obvious now that choosing this fixed size for images is a matter of tradeoff between computational efficiency and accuracy. The selected size for the images was 128×128.

3.5 Data Augmentation

One of the characteristics of these networks is that they are heavily reliant on big data to avoid overfitting. Due to the relatively small size of the dataset, I decided to rely on **Data Augmentation**, a data-space solution to this problem which encompasses a set of techniques that enhance the size and quality of training datasets such that better Deep Learning models can be built using them[7].

To build useful models, it's essential for the validation error to continue decreasing alongside the training error, a goal achievable through Data Augmentation. The augmented data will represent a more comprehensive set of possible data points. Creating more diverse data through augmentation ensures the model to be trained on a broader range of scenarios, potentially improving its performance on unseen data.

Since the restricted dimension of the dataset, I chose to perform Data Augmentation. In fact, while many other strategies for increasing generalization performance focus on the model's architecture itself, Data Augmentation approaches overfitting from the root of the problem, the training dataset. This is based on the assumption that it is possible to extract more information from the original dataset through augmentations. These augmentations artificially increase the training dataset size by either data warping or oversampling[7].

Furthermore, considering the safety of data augmentation is fundamental in ensuring the integrity and reliability of machine learning models. In fact, certain transformations might generate unrealistic or misleading data points, leading to erroneous model predictions. All of the chosen augmentations were evaluated not only to be safe but to also improve significantly the robustness of the model.

It is possible to distinguish different techniques for augmenting data:

- **geometric transformations**: randomly flip, crop, rotate, stretch, and scale images. Care should be taken when applying multiple transformations on the same images, as this has the potential to reduce model performance;
- **color space transformations**: randomly change RGB color channels, contrast, and brightness;

- **noise injection**: injecting a matrix of random values, usually drawn from a Gaussian distribution. Adding noise to images can help NNs learn more robust features;
- **kernel filters**: randomly change the sharpness or blurring of the image;
- **random erasing**: this technique was specifically designed to prevent overfitting by altering the input space and, consequently, to combat image recognition challenges due to occlusion. By removing certain input patches, the model is forced to find other descriptive characteristics;
- **mixing images**: blending and mixing multiple images by averaging their pixel values.

For this purpose, new images were created by applying some augmentations over a randomly extracted set of images (25% of the dataset). I randomly extracted 640 images from the first class and approximately 800 images from the second class, in order to make the sizes of the two datasets closer. The total number of added images will be 1440, approximately 25% of the dataset.

The chosen augmentations are described in the following paragraphs.

Geometric transformations This family of transformations takes the image and changes it, keeping the pixel values the same. These changes have been applied not only to increase the number of training examples, but also to prevent the network from overfitting to specific features like perspective or background details that may be present in the original training data but are not relevant to the classification task. The main disadvantages of geometric transformations are the increased need of memory, the transformation costs and the higher training time.

- **flipping**: this geometric transformation consists in flipping the horizonal axis;
- **mirroring**: this geometric transformation consists in flipping the vertical axis;
- **rotation**: Rotation augmentations are done by rotating the image right or left on an axis of a random degree.

Color space transformation This family of transformations changes the pixel values, resulting in the consequential change of the colors of the pixels. Since color variations occur naturally in real-world scenarios due to factors like lighting conditions, camera settings, and environmental factors. by augmenting the dataset with color space transformations, I simulate these real-world variations, enabling the model to learn more representative features and generalize effectively. Furthermore, these changes have been applied in order to differentiate the color scheme, thus avoiding the network to learn biases over some particular palette or scheme of the image's lighting.

Similar to geometric transformations, the main disadvantages of color space transformations are the increased need of memory, the transformation costs and the higher training time. Additionally, color transformations may discard important color information and thus are not always a label-preserving transformation.

In effect, color space transformations will eliminate color biases present in the dataset in favor of spatial characteristics but, for some tasks, color could be a fundamental distinctive feature.

- **brightness**: this color space transformation consists in the decreasing or increasing of the pixel values by a constant value. For example, when decreasing the pixel values of an image to simulate a darker environment, it may become impossible to see the objects in the image;
- **contrast**: this color space transformation involves modifying the distribution of pixel intensities within an image to increase or decrease the difference between light and dark areas;
- **saturation**: this color space transformation involves adjusting the intensity or purity of colors within an image while keeping the brightness and contrast levels constant.

Kernel filters Kernel filters are modifications that change the pixel values, resulting in the change of the arrangement of the pixels. Filters are a very popular technique in image processing to sharpen and blur images. These filters operate by moving a matrix of size $n \times n$ across an image, applying either a box blur filter, causing the image to become blurrier, or a high-contrast vertical or horizontal edge filter, resulting in sharper edges in the image[7].

- **unsharp masking sharpening**: this kernel filter involves exploiting a method commonly known as **unsharp masking sharpening** to increase the contrast along edges within the image: a Gaussian blur, that is, a filter that reduces detail and noise in an image by applying a Gaussian function to each pixel and its surrounding pixels, is applied to the original image and it is then subtracted from the original image itself, thereby accentuating the edges. Sharpening images could result in encapsulating more details about objects of interest;
- **blurring**: this kernel filter involves, as the name suggests, blurring the image and reducing noise, resulting in a smoothed version of the image. Intuitively, blurring images for Data Augmentation could lead to higher resistance to motion blur during testing. The blurring is obtained through a **box blur**, that is, by setting the value of each pixel to the average of the pixels in the radius of 3.

3.6 Image Segmentation

Even if some filters simplify a picture giving almost only the main shape, in some cases this is not enough to exclude the features we know are completely irrelevant. The problem in the recognition of the principal content of the image is given by the fact that an image could contain a lot of different details that increase the difficulty of the task of classification, thus, I decided to also apply **segmentation** to the datasets. Segmentation is the process of partitioning an image or video into meaningful regions to identify and differentiate objects or regions of interest. The goal is to classify each pixel or region of the image as belonging to one of two classes: foreground or background.

The **Simple Linear Iterative Clustering** algorithm is employed on each image, assigning every pixel to the centroid of its respective cluster, called **superpixel**. By combining pixels in the image plane based on their mean color and spatial position, the technique clusters pixels in a five-dimensional color and picture plane space to create small, nearly uniform superpixels[8].

The result of the data segmentation and augmentation has been six different datasets:

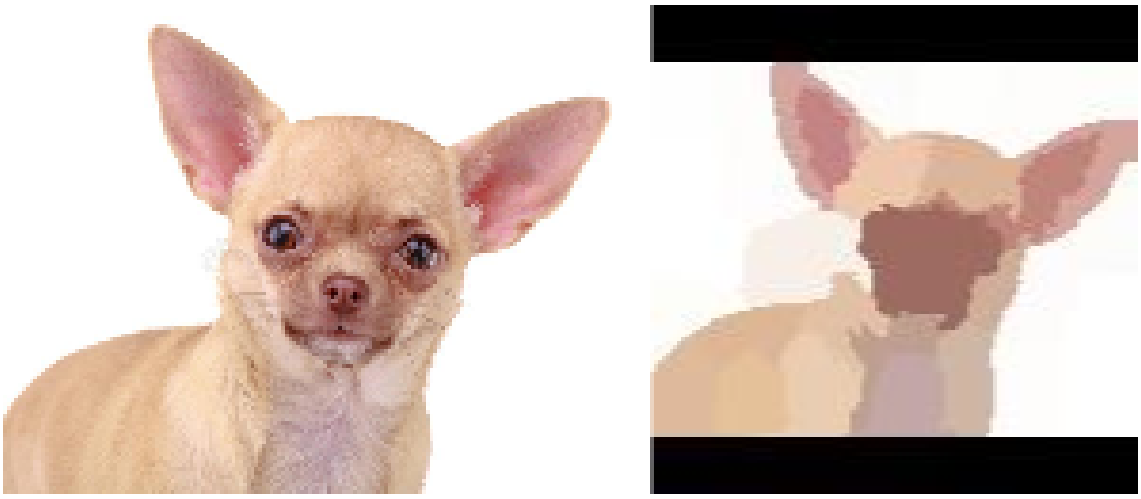

Figure 4: An example of a segmented image

- the original dataset (used in the hyper parameter tuning step);
- the augmented dataset;
- the segmented dataset.

In the table below a brief summary of our generated datasets is provided.

Dataset	Number of Chihuahua	Number of Muffin	Total images
RGB	3161	2590	5751
RGB_Augmented	3801	3390	7191
RGB_Segmented	3161	2590	5751

Table 2: Cardinality of each dataset

3.7 Image Normalization

Each image in the datasets is normalized to pixel values between 0 and 1, with 0 being black and 1 being white. This is done by dividing all pixels by the maximum pixel value. Since the output of the NNs will be between 0 and 1 (because we are dealing with a classification task), normalizing the input will ensure that the input features (pixel values) will have similar scales. This can help accelerate the convergence of optimization algorithms during model training, as it reduces the likelihood of vanishing or exploding gradients and it is also useful to have comparable values in every layer of a neural network.

3.8 Dataset Shuffling

Before splitting, a shuffle operation is executed to eliminate potential patterns or biases in the data collection process. It is important to note that shuffling the dataset will not address issues if the

data is not representative of the entire population. However, if there is a systematic pattern in how the data was collected or ordered (e.g., all images of one class followed by another), shuffling will disrupt this inherent pattern, granting a more randomized distribution of the data. The shuffling will also ensure that each batch of training data represents a diverse mix of images from different categories or classes.

IV Theoretical Background

1 Artificial Neural Network

Artificial Neural Networks (ANNs) are a large and complex class of predictors applicable to a variety of fields, given their powerful capability in tasks like classification, regression, and sequence prediction.

A Neural Network is usually depicted as a graph $G = \langle V, E \rangle$, where V is the finite set of vertices or nodes and E is the finite set of weighted edges. The vertices $v \in V$ are called **neurons** or units and the edges $e \in E$ are called **connections**.

Each neuron first computes the product of the weights of incoming edges with the corresponding incoming inputs and then applies an activation function over it.

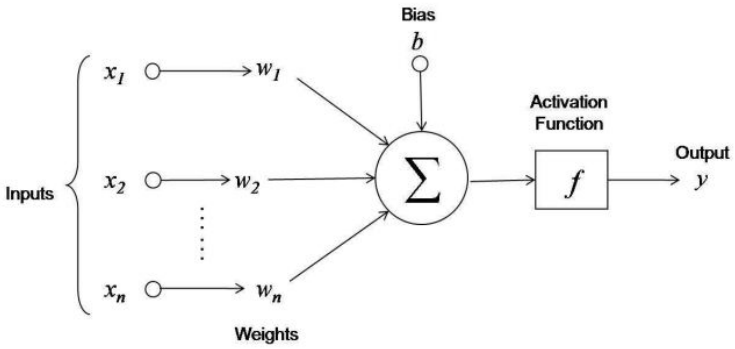

Figure 5: An example of a Neural Network's Neuron

I will only refer to and consider **feed forward neural networks**, that is, models in which there are no loops or cycles in the graph representation of the network. The term feed forward refers to the fact that the computation can be transmitted only along the directed connections and that the output of a layer becomes the input of the following one[1].

1.1 Multi Layer Neural Networks

The simplest form of feedforward NN is the Multi Layer Neural Network, a class of artificial neural networks characterized by their layered architecture, consisting of multiple interconnected nodes organized into three main type of layers:

- **input layer**: this type of layer represent the entry point of a neural network. Its shape should be equal to the shape of the inputs. It is in charge of propagate the input without any change to the successive layer. The nodes in this type of layer have no incoming edges;
- **output layer**: this type of layer is in charge of returning the output of the network. The nodes in this type of layer have no outgoing edge;
- **hidden layer**: this type of layer comprises all the layers between the input one and the output one. All the nodes in this type of layer have both incoming and outgoing edges.

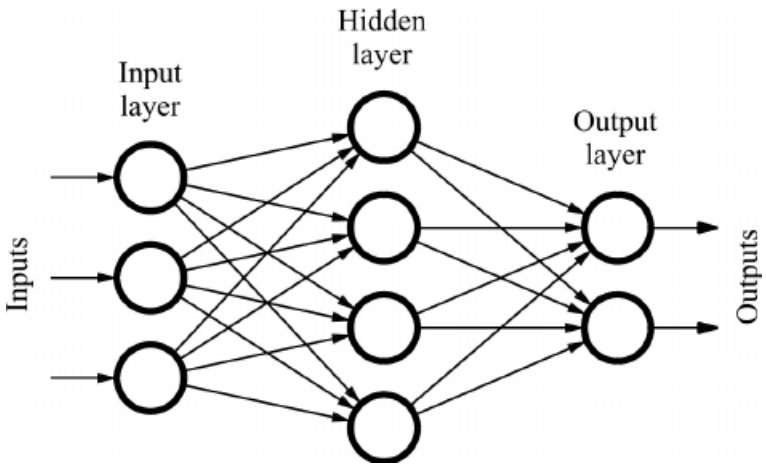

Figure 6: An example of a Feed Forward Neural Network

2 Convolutional Neural Networks

One of the largest limitations of traditional forms of ANN is that they tend to struggle with the computational complexity required to compute image data. To choose a type of network more suited for image-focused tasks, whilst further reducing the parameters required to set up the model, **Convolutional Neural Networks** (CNNs) are the obvious choice. As previously stated, CNNs are a particular subclass of Neural Networks that became popular in computer vision tasks, such as image recognition and classification. They are inspired by the visual processing mechanism of the human brain, specifically the **receptive fields** of neurons in the visual cortex, in which a neuron is activated only by a small region of its input.

CNNs are comprised of three types of layers[9], each of which has a different role determined by its structure:

- **convolutional layer**: this type of layer is characterized by the convolution operation and its purpose is to extract features from the images. To do so, it uses a kernel (which is a squared vector/matrix typically of odd dimension), called **convolutional kernel**, which are smaller than the dimension of the initial image. To extract features, the CNN calculates the convolution between this kernel and the region covered by it. This procedure is repetitively

applied and shifted along all the input dimensions and the amount of movement of the kernel is regulated by the **stride**. A CNN layer can contain multiple filters, each of which produces a different **feature map** matrix, capturing different aspects of the input. Furthermore, since these kernels are shifted along the entire input, the presence of a certain feature will be captured no matter where it occurs in the input. Each convolutional layer efficiently reduces the size of the input matrix by consolidating regions into single outputs for subsequent layers. This hierarchical stacking of convolutional layers enables the recognition of increasingly complex patterns within the image;

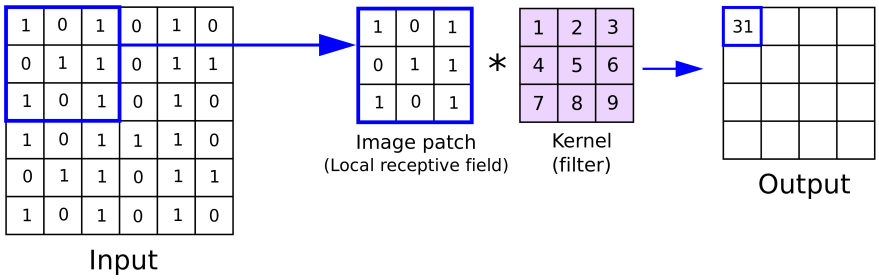

Figure 7: An example of a Convolutional Kernel

- **pooling layer**: this layer is the components of the networks in charge of aggregating features in a feature map using a certain window size juxtaposed across the entire feature map. In particular, this layer is built to reduce the sensitivity of the model to the spatial location. In this way, the model will be spatially invariant, that is, it will not be sensible to the absolute position of an object in order to classify the overall image. To accomplish this, it is fundamental to focus on finding the correlation between features rather than where that exact feature is located. The pooling layer basically downsaples the feature map: it extracts a value from a submatrix/vector of the matrix/vector it takes as input: this value can be, for example, the maximum or the average one. By doing so, the layer summarize the information held by some pixels in just one. Reducing the dimension of feature maps allows CNNs to increase in depth without significantly increasing the computational load on the network;

- **dropout layer**: to avoid the problem of quickly overfitting due to the huge number of parameters in the model, 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;

- **fully connected layer**: these layers are the same previously described as Multi Layer Neural Network. Usually, the concluding segment of a CNN comprises a fully connected network. The concept is that the convolutional section of the network discerns patterns from the input and forwards them as features to a traditional network, which then undertakes a classification task

based on these features.

The overall structure of a CNN can thus be divided into two major sections: the first one is designated for feature learning and extraction while the second one is employed in the classification task.

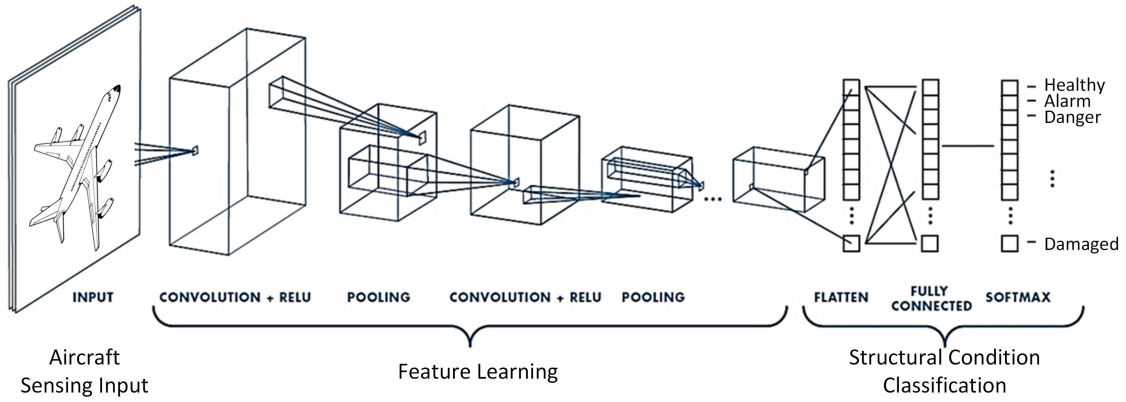

Figure 8: An example of the structure of a Convolutional Neural Network

V Models

In this section I present the models I developed and used in my trials to find the best architecture for the proposed task. I opted for testing and comparing a Feed Forward Neural Network built from scratch with two kind of Convolutional Neural Networks: one built starting from an existing CNN and adapting it to the assigned task, and the other one built from scratch with fewer layers and parameters.

By comparing different architectures, I aim to demonstrate the impacts of various normalization techniques like dropout on networks of different sizes. Additionally, I seek to assess whether and how performance correlates with the scale and the number of parameters of the network.

One of the fundamental modeling choices regarded the number of layers and neurons in each one of them. Here I applied a similar approach for both the MLNN and the second CNN, that is, I started from a small network composed by one layer and then I increased both the number of layers and the number of neurons. All the configurations have been tested using the preprocessed data before both the augmentation and the segmentation.

All the configurations have been tested using the preprocessed data before both the augmentation and the segmentation.

1 MLNN

In regard to MLNN, I've opted to parameterize certain values to discover the optimal configuration of neuron count and layer number. This approach facilitates the exploration of the neural network's

design space, aiming to identify the most fitting model for this specific classification task. I've experimented with the following sets of neurons per layer:

- $[128]$
- $[256]$
- $[256, 128]$
- $[128, 128]$
- $[256, 128, 64]$
- $[128, 128, 128]$
- $[512, 256, 128]$
- $[128, 256, 512]$

Each one of these architectures iterates over a list of hidden layer units, applying batch normalization, activation function, dropout regularization, and dense (fully connected) layers sequentially. The batch normalization layer improves the training speed and stability of neural networks by normalizing the input to each layer. The activation layer applies an activation function element-wise to the output of the previous layer to introduce non-linearity to the network, allowing it to learn complex patterns and relationships in the data. The dropout layer randomly sets a fraction (20%) of input units to zero during training, which helps prevent overfitting by introducing noise and reducing the interdependence of neurons.

Each network architecture has been evaluated by 5 fold cross validation (described in details in section VI), choosing some hyperparameter values as default. The first choice of parameterized values was the following:

2 CNN

In regard to CNN, I've opted for a quite simple configuration. The Convolutional Neural Network presented follows a typical architecture for image classification tasks. It consists of three main types of layers: convolutional layers, max-pooling layers, and fully connected layers. The convolutional layers apply filters to the input image to extract feature. The three convolutional layers have increasing numbers of filters and they are followed by a maxpooling layer, which reduces the spatial dimensions of the feature maps retaining the most important informations. The final convolutional layer is flattened to transform the 2D feature maps into a 1D vector, which is then fed into fully connected layers. These two layers learn to classify the extracted features into different classes, with the last layer producing a single output using a sigmoid activation function for binary classification. The purpose of this specific CNN is to serve as a comparative benchmark against the MLNN previously presented.

```
Model: "model_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_8 (InputLayer)        [(None, 128, 128, 3)]     0

 flatten_7 (Flatten)         (None, 49152)             0

 rescaling_7 (Rescaling)     (None, 49152)             0

 batch_normalization_21 (Ba  (None, 49152)             196608
 tchNormalization)

 activation_21 (Activation)  (None, 49152)             0

 dropout_21 (Dropout)        (None, 49152)             0

 dense_28 (Dense)            (None, 512)               25166336

 batch_normalization_22 (Ba  (None, 512)               2048
 tchNormalization)

 activation_22 (Activation)  (None, 512)               0

 dropout_22 (Dropout)        (None, 512)               0

 dense_29 (Dense)            (None, 256)               131328

 batch_normalization_23 (Ba  (None, 256)               1024
 tchNormalization)

 activation_23 (Activation)  (None, 256)               0

 dropout_23 (Dropout)        (None, 256)               0

 dense_30 (Dense)            (None, 128)               32896

 dense_31 (Dense)            (None, 1)                 129

=================================================================
Total params: 25530369 (97.39 MB)
Trainable params: 25430529 (97.01 MB)
Non-trainable params: 99840 (390.00 KB)
_________________________________________________________________
```

Figure 9: Summary of the MLNN model

3 TogNet

TogNet is a Convolutional Neural Network builded and tuned for this specific task to avoid the overfitting that characterized the previous network. To do so, the model implements dropout layers after

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 rescaling_5 (Rescaling)     (None, 128, 128, 3)       0

 conv2d_6 (Conv2D)           (None, 126, 126, 32)      896

 max_pooling2d_6 (MaxPoolin  (None, 63, 63, 32)        0
 g2D)

 conv2d_7 (Conv2D)           (None, 61, 61, 64)        18496

 max_pooling2d_7 (MaxPoolin  (None, 30, 30, 64)        0
 g2D)

 conv2d_8 (Conv2D)           (None, 28, 28, 128)       73856

 max_pooling2d_8 (MaxPoolin  (None, 14, 14, 128)       0
 g2D)

 flatten_5 (Flatten)         (None, 25088)             0

 dense_16 (Dense)            (None, 128)               3211392

 dense_17 (Dense)            (None, 1)                 129

=================================================================
Total params: 3304769 (12.61 MB)
Trainable params: 3304769 (12.61 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 10: Summary of the CNN model

each pooling layer and before the first dense layer. This approach forces the network to learn more robust features, preventing the model from relying too heavily on any single neuron. Consequently, dropout layers improve the generalization ability of the CNN, leading to better performance on new, unseen data.

VI Experiments

1 Hyperparameters Tuning

A key feature of the machine learning process is the model's ability to adjust its internal parameters autonomously, based on the data it receives. During the training phase of a Neural Network, the model modifies its internal weights in response to the provided data. However, a model's effectiveness is not solely defined by these weights; it is also influenced by a set of parameters known as

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
rescaling_8 (Rescaling)      (None, 128, 128, 3)       0

conv2d_15 (Conv2D)           (None, 126, 126, 32)      896

max_pooling2d_15 (MaxPooli   (None, 63, 63, 32)        0
ng2D)

dropout_17 (Dropout)         (None, 63, 63, 32)        0

conv2d_16 (Conv2D)           (None, 61, 61, 64)        18496

max_pooling2d_16 (MaxPooli   (None, 30, 30, 64)        0
ng2D)

dropout_18 (Dropout)         (None, 30, 30, 64)        0

conv2d_17 (Conv2D)           (None, 28, 28, 64)        36928

max_pooling2d_17 (MaxPooli   (None, 14, 14, 64)        0
ng2D)

dropout_19 (Dropout)         (None, 14, 14, 64)        0

flatten_8 (Flatten)          (None, 12544)             0

dense_22 (Dense)             (None, 128)               1605760

dropout_20 (Dropout)         (None, 128)               0

dense_23 (Dense)             (None, 1)                 129

=================================================================
Total params: 1662209 (6.34 MB)
Trainable params: 1662209 (6.34 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 11: Summary of the TogNet model

hyperparameters. These hyperparameters, which include the network's topology, activation functions for each layer, the number of output neurons per layer, and the optimizer guiding the training, shape the learning process without being directly modified by it.

Hyperparameters are fundamental for achieving high model accuracy. The process of selecting the optimal hyperparameters, called **hyperparameter tuning**, involves repeatedly training the model while altering these hyperparameters. The objective is to explore various combinations to identify the configuration that yields the best performance.

Different techniques have emerged to explore the hyperparameters' combinations space. GridSearch involves exhaustively searching through a predefined set of hyperparameters, evaluating all possible combinations. While thorough, it can be computationally expensive. Random Search, on the other hand, samples hyperparameters randomly within specified ranges, often finding good combinations more efficiently than GridSearch.

However, due to the limitations of the chosen environment, which made running an algorithm like GridSearch nearly impossible, I decided to run sequentially the 5-fold cross validation, using 10 epochs for each training phase. For each provided value of the hyperparameters (which can be observed in the table below3), I performed a 5-fold cross validation to estimate the average performance and I saved the hyperparameter value that led to the lowest Zero-One loss. The selected values are then employed in the retraining of the model, and the result is evaluated on the test partition. This process is repeated three times using different seeds to split the dataset in order to avoid bias of a peculiar partition.

Learning Rate	0.001	0.01	0.1
Dropout	0.1	0.2	0.4

Table 3: Chosen Hyperparameters

2 Nested Cross Validation

Estimating the performance of a machine learning model accurately is crucial for ensuring its effectiveness in real-world applications. One common technique used for this purpose is **k-fold cross validation**. In k-fold cross validation, the dataset is divided into k subsets or folds of approximately equal size. The model is then trained and evaluated k times, each time using a different fold as the test set and the remaining folds as the training set. This allows the model to be tested on different portions of the data, providing a more reliable estimate of its performance.

However, when hyperparameter tuning is involved, simply using k-fold cross validation may lead to overfitting to the validation set, as the hyperparameters may be selected based on their performance on this specific subset of data. This is where nested cross validation comes into play. Nested cross validation adds an outer loop to the process in which the dataset is still divided into k folds, but each fold is used as a separate validation set exactly once. Within each iteration of the outer loop, an inner loop of k-fold cross validation ($k = 5$ in our case) is performed to tune the hyperparameters of the model. The hyperparameters are selected based on their performance on the inner cross validation loop, ensuring that they are optimized without leaking information from the test set.

By using nested cross validation, a more unbiased estimate of the model's performance is obtained because the hyperparameters are tuned based on their performance on unseen data. Additionally, by averaging the performance metrics obtained from the outer loop across all iterations, we obtain a more reliable indication of the model's overall effectiveness.

3 Performance Metrics

I selected the **Binary Cross Entropy** (or Log Loss) as the training loss function. This loss function is particularly well-suited for binary classification because it increases when the prediction diverges from the target label, penalizing incorrect predictions more heavily and therefore encouraging the model to produce probabilities closer to the true labels. Furthermore, this loss is differentiable, which facilitates efficient gradient descent optimization, leading to faster and more reliable convergence during training.

The other metrics used to evaluate the networks are:

- **Binary Accuracy**: this metric calculates the accuracy of the model, which is the proportion of correctly classified samples out of the total number of samples. It's calculated as (true positives + true negatives) / (true positives + true negatives + false positives + false negatives);
- **Precision**: Precision measures the proportion of true positive predictions out of all positive predictions made by the model. It's calculated as true positives / (true positives + false positives). Precision is a measure of the model's ability to avoid false positives;
- **Recall**: Recall measures the proportion of true positive predictions out of all actual positive samples in the dataset. It's calculated as true positives / (true positives + false negatives). Recall is a measure of the model's ability to identify all positive samples;
- **Area Under the ROC Curve** (AUC): AUC measures the area under the Receiver Operating Characteristic (ROC) curve, which plots the true positive rate (recall) against the false positive rate (FPR). AUC provides an aggregate measure of the model's performance across all possible classification thresholds. 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.

For the risk estimate, instead, I employed the Zero-One loss function, which is defined as follows:

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

where $\widehat{y}$ is the label predicted by the model and y is the target label.

4 General Settings

In building our datasets, I started applying a mapping to the labels "chihuahua" and "muffin", in order to better compute the error between the prediction and the actual label. Therefore, for this project "chihuahua" was assigned the label 0 (False), while "muffin" the label 1 (True).

5 MLNN

As previously specified, in my experiments, I first used 5-fold cross validation to evaluate which MLNN architecture was the most effective in the context of this classification task.

Initially, I considered only the following architectures:

- [128]
- [256]
- [128, 128]
- [256, 128]
- [128, 128, 128]
- [256, 128, 64]

as I wanted to observe how different network depths would affect the outcome. The first choice of parameterized values used was the following:

Parameters	**Optimizer**	**Learning rate**	**Dropout**	**Epochs**
Values	ADAM	0.001	0.2	10

Table 4: Hyperparameters initially chosen to evaluate the MLNN models

However, the differences in the zero one loss and in the binary cross entropy values from one architecture to another weren't significant enough to take a decision. Therefore, I increased the number of epochs to 15:

Parameters	**Optimizer**	**Learning rate**	**Dropout**	**Epochs**
Values	ADAM	0.001	0.2	10

Table 5: Hyperparameters finally chosen to evaluate the MLNN models

Even though the increase in the number of epochs, as shown in the histogram below, no architecture substantially outperformed the others in terms of Zero-One loss (observable in Figure 12) on the validation set. For this reason, I would have liked to try training the different architectures with a greater number of epochs, but I was limited by the chosen environment.

Therefore, I tested two additional architecture, as deep as the previously tested architectures but with a greater number of neurons in each layer. Furthermore, in the second new architecture, the number of neuron in the hidden layer is increasing instead of decreasing. The reasoning behind this structure is that the early layers should learn simpler and more general features and this should require less neurons. On the other hand, the deeper layers should learn more complex features, therefore requiring more neurons. The proposed new architecture are:

- [512, 256, 128]
- [128, 256, 512]

Overfitting occurs when our learning algorithm fails and returns a predictor with low training error. One way to discover overfitting is to plot the training and validation accuracy at each epoch during training. During training, the model continuously learns from the training data, which may lead to increasing training accuracy. However, if the model starts to overfit, the validation accuracy

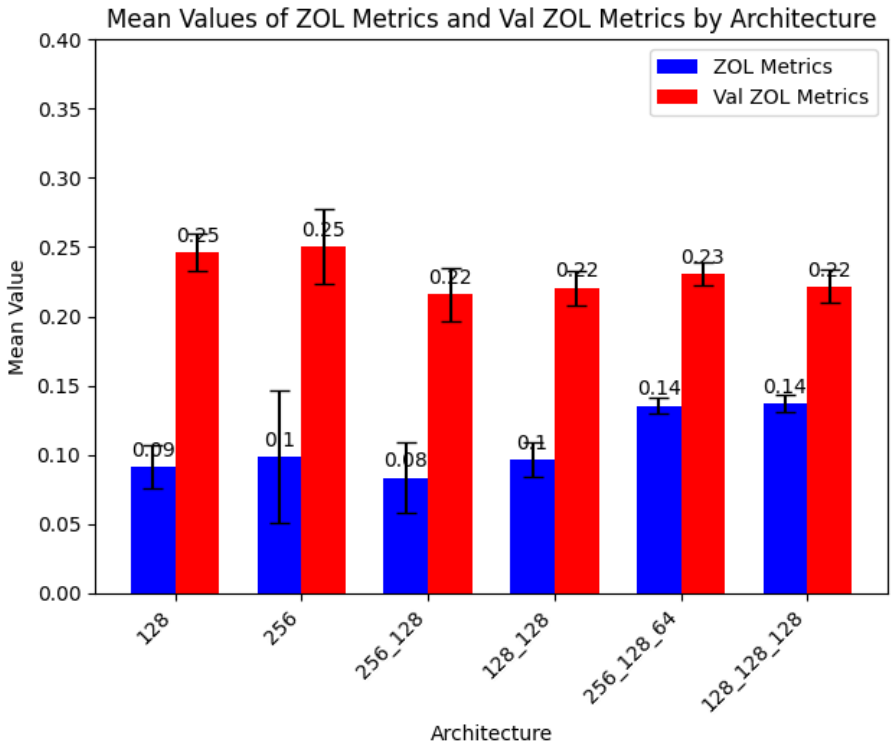

Figure 12: Histogram of the Zero-One loss for the MLNN architectures considered initially

may plateau or even decrease, indicating that the model is not generalizing well to new data. In contrast, if both training and validation accuracy increase together, it indicates that the model is learning the underlying patterns of the data effectively without overfitting. By comparing the trends of training and validation accuracy over epochs, it is possible to identify when overfitting begins to occur.

After considering all of this, I have decided to choose architecture 512_256_128 for the hyperparameter tuning phase because it has returned a better Zero-One loss and it also appears to be more stable than the others. The binary accuracy of the other classifiers exhibits a pattern marked by significant fluctuations over time, displaying peaks and valleys. These fluctuations suggest variations in the classifiers' performance across different epochs, indicating potential challenges in maintaining consistent predictive accuracy.

The next step is the hyperparameter tuning. It is important to emphasize that this step has been done only looking at the Zero-One loss.

These are the results:

Therefore, the final settings for the MLNN are the following:

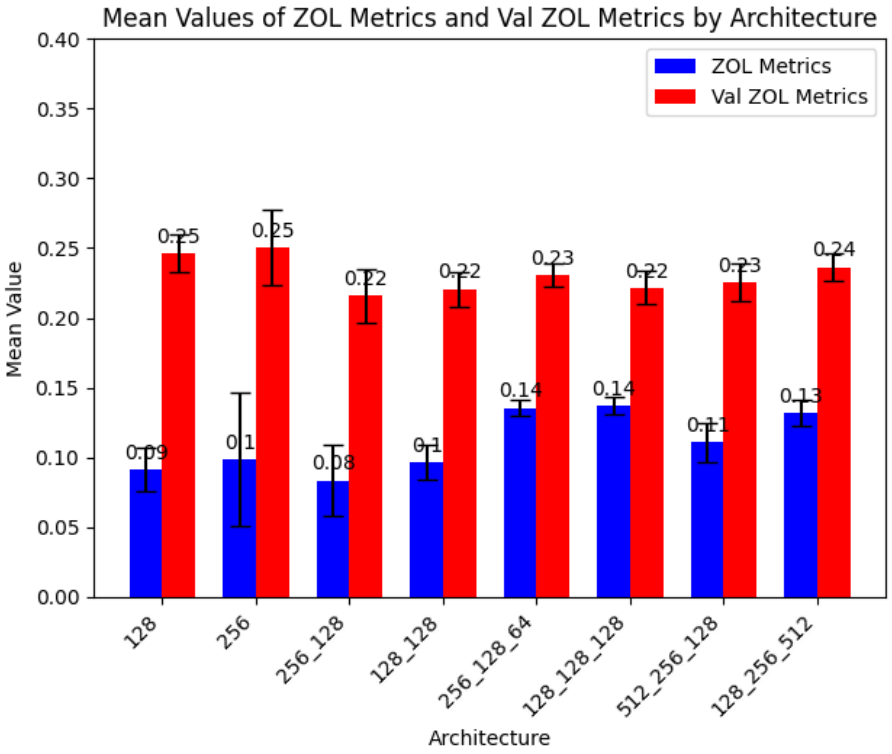

Figure 13: Histogram of the Zero-One loss for all the MLNN architectures considered

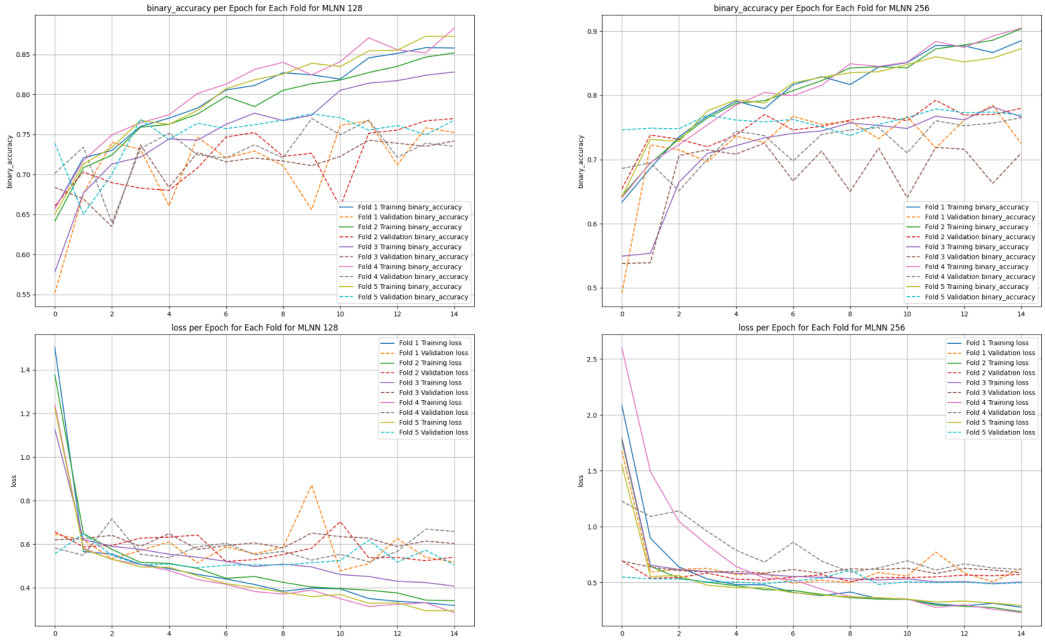

Figure 14: Loss and Binary Accuracy of the 128 and 256 MLNN

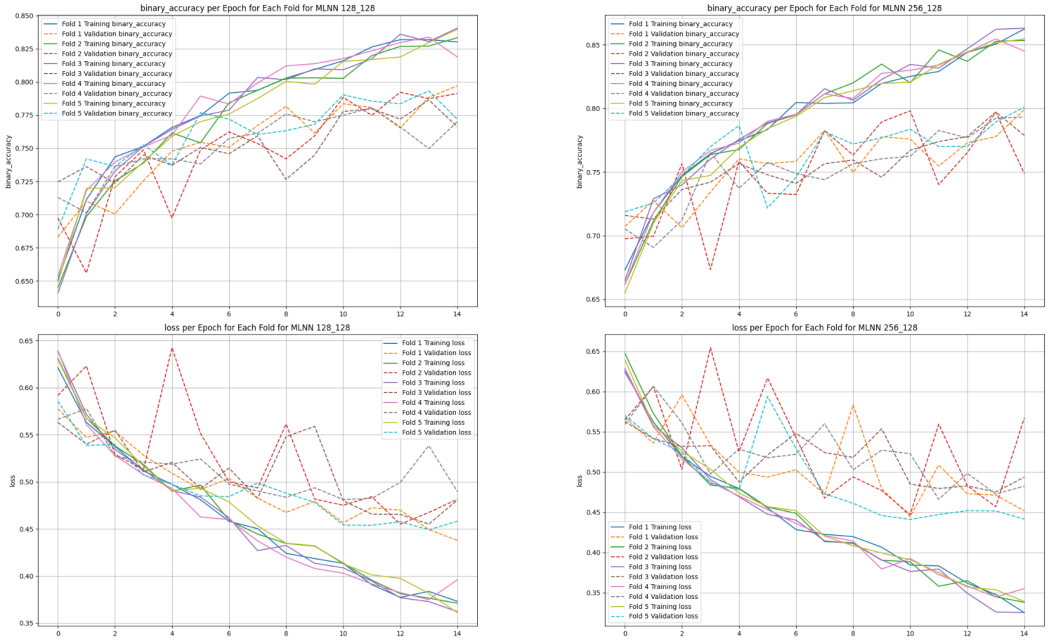

Figure 15: Loss and Binary Accuracy of the 128_128 and 256_128 MLNN

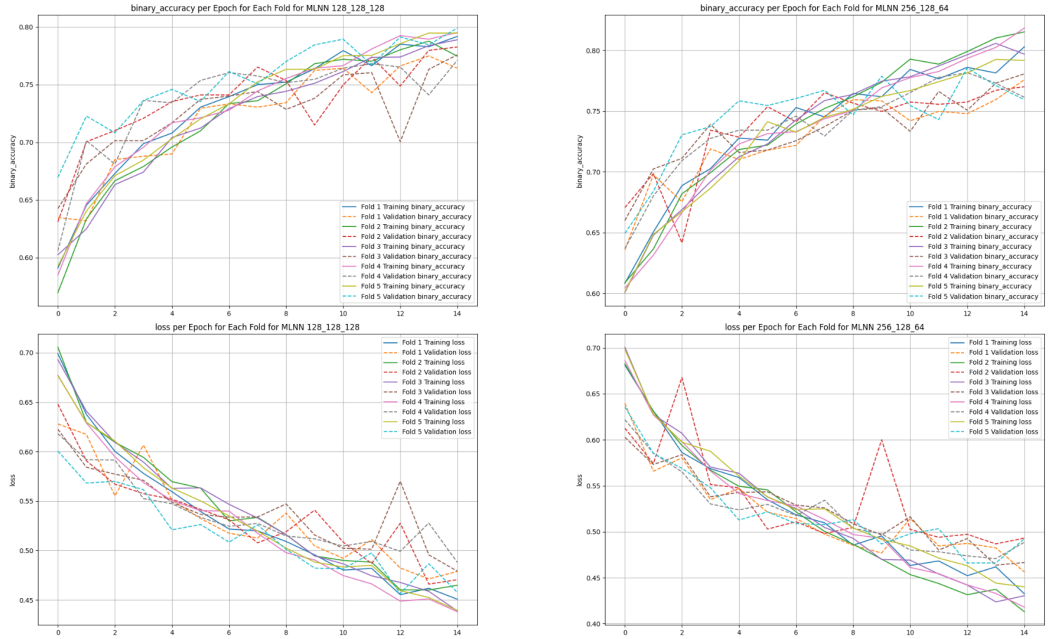

Figure 16: Loss and Binary Accuracy of the 128_128_128 and 256_128_64 MLNN

6 CNN

The idea behind the choice of this baseline CNN is literally to perform a comparison between one of the simplest CNN possible with a more elaborated MLNN. Tipically, the performances of CNN

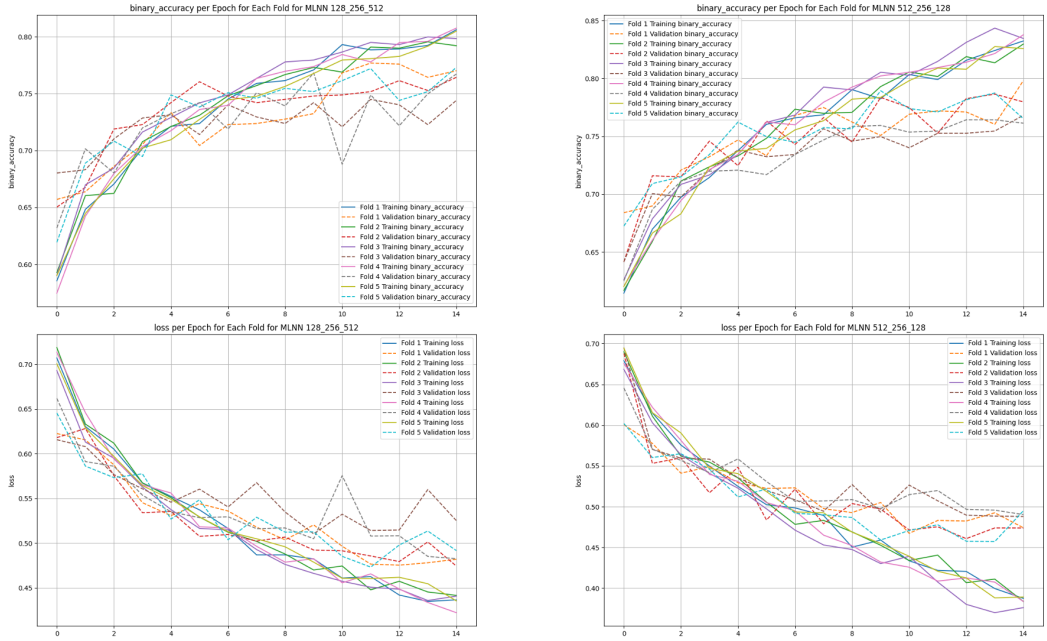

Figure 17: Loss and Binary Accuracy of the 128_256_512 and 512_256_128 MLNN

Learning Rate Dropout Rate	0.1	0.2	0.4
0.001	0.2324	0.2402	0.2301
0.01	0.2532	0.2274	0.2261
0.1	0.3474	0.3314	0.4435

Table 6: Zero-One loss for the MLNN learning rate and dropout rate tuning

Parameters	Values
Learning rate	0.001
Dropout rate	0.4

Table 7: Final MLNN settings

outclass the performances of fully connected networks. Therefore, I started from one really simple architecture, with 3 different convolutional layers (one with 32 filters, one with 64 and one with 128) and pooling size $(2, 2)$.

I chose the Learning Rate and the Kernel Size as hyperparameters to tune. It is important to emphasize again that this step has been done only looking at the Zero-One loss. These are the obtained results:

Learning Rate Kernel Size	3×3	5×5
0.001	0.0892	0.1142
0.01	0.1651	0.1482
0.1	0.2491	0.2490

Table 8: Zero-One loss for the CNN learning rate and kernel size tuning

Therefore, the final settings for the CNN are the following:

Parameters	Values
Learning rate	0.001
Kernel Size	3×3

Table 9: Final CNN settings

However, I noticed a significant disparity between the training loss and the validation loss, as it is possible to observe in the final result table. This disparity in my model, where the training loss was very low while the validation loss was significantly higher, was a signal that the model was likely overfitting. To address overfitting, I considered using techniques such as adding dropout layers, because I already chose a substantial split percentage of the dataset for the training data and I already had applied data augmentation.

7 TogNet

This network is an evolution of the previous one, aiming to reduce overfitting as much as possible.

I decided to set the kernel size and the learning rate, remembering the results of the previous tuning and to choose the dropout rate as parameters to tune instead. These are the results.

Dropout Rate	Zero-One loss
0.1	0.0902
0.2	0.0873
0.3	0.0864
0.4	0.0993

Table 10: Zero-One loss for the Tognet dropout rate tuning

To fully evaluate the reduction of overfitting, during the hyperparameter training of this network, I chose to consider not only the zero-one loss but also the Binary Accuracy and the Binary Cross Entropy loss.

Therefore, the final settings for TogNet are the following:

Parameters	Values
Dropout rate	0.4

Table 11: Final TogNet settings

By observing the loss and binary accuracy curves of the training and validation folds, it can be concluded that overfitting has been significantly contained. This is also evidenced by the final testing of the model.

VII Results and Discussion

After performing our experiments, we collected all the results regarding the performances within tables and graphs.

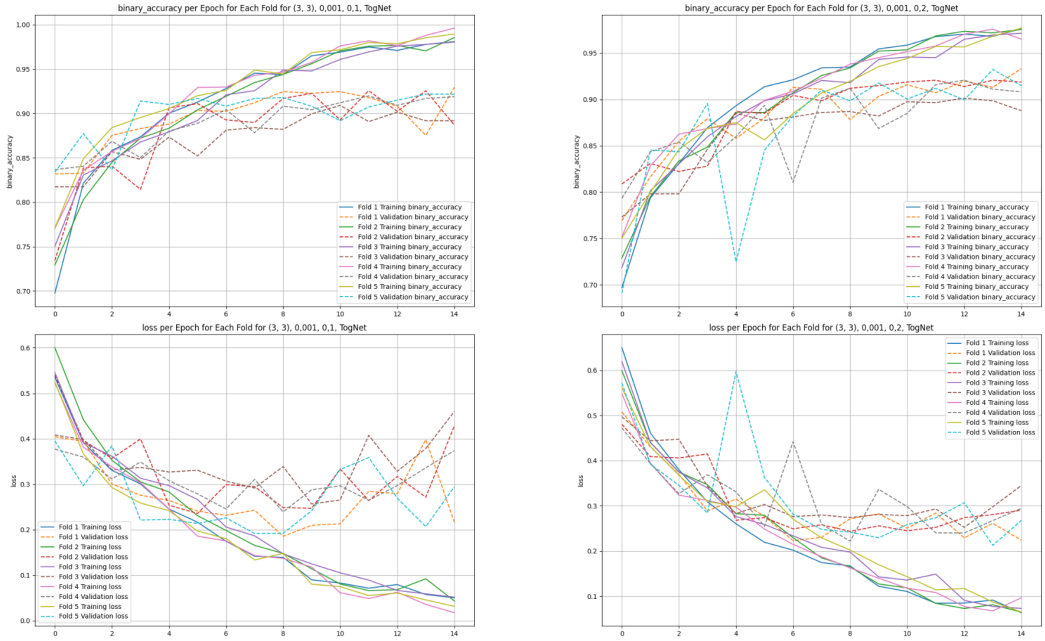

Figure 18: Loss and Binary Accuracy per epoch for each fold of the TogNet with Dropout Rate 0.1 and 0.2

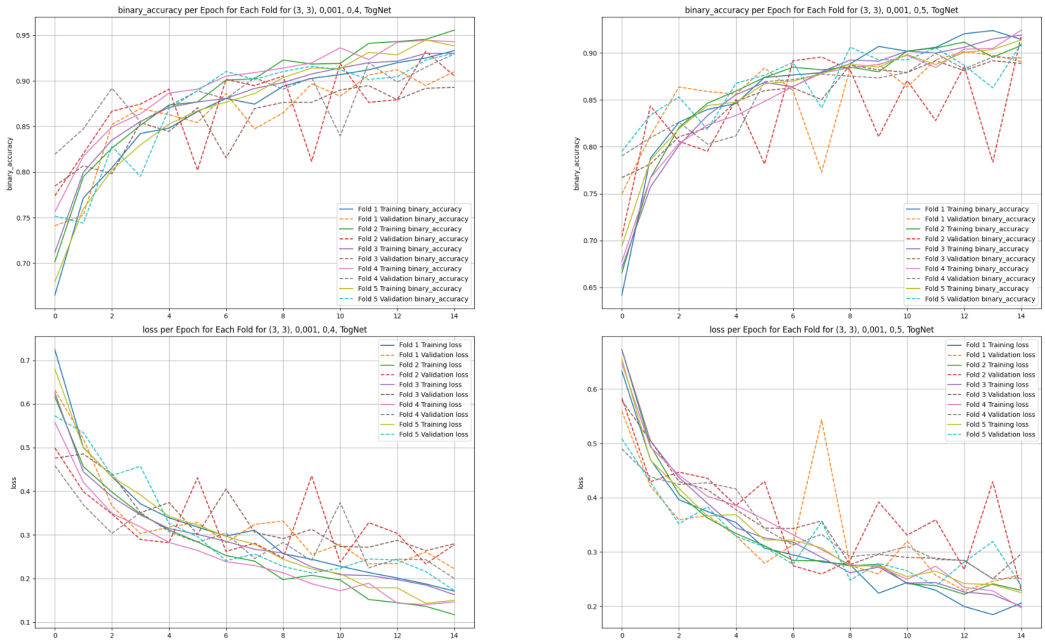

Figure 19: Loss and Binary Accuracy per epoch for each fold of the TogNet with Dropout Rate 0.4 and 0.5

1 Comparative Analysis of Models

After performing the experiments, I collected all the results regarding the performances. In the tables below, the average values for each metric within the 5-fold cross validation are presented for MLNN_512_256_128, for CNN_32_64_128 and for TogNet.

MLNN 512_256_128	Binary Cross Entropy	Binary Accuracy	Precision	Recall	AUC
RGB	0.4363	0.8038	0.7802	0.8008	0.8801
RGB_Augmented	0.4027	0.8292	0.8906	0.8095	0.9154
RGB_Segmented	0.3848	0.7847	0.7767	0.8007	0.8639

Table 12: Performance metrics of MLNN model with different input configurations

MLNN_512_256_128 Binary accuracy is highest for the RGB_Augmented dataset, suggesting that this model is the most accurate among the three in correctly classifying images. This indicates that the RGB_Augmented model has the best overall performance in terms of correctly predicting both positive and negative classes. The higher accuracy demonstrates the model's ability to generalize well to unseen data, making fewer errors in its predictions. This improvement can be attributed to the effectiveness of data augmentation techniques, which likely provide the model with a more diverse, representative and numerous training set, leading to better generalization and accuracy.

The highest precision is, again, achieved with the RGB_Augmented dataset, indicating that the model has a high rate of correctly identified positive images among those it has classified as positive. In this case, the model trained with the RGB_Augmented dataset demonstrates superior performance in accurately identifying true positives, suggesting that data augmentation techniques have significantly improved its classification reliability.

The RGB_Segmented results for the MLNN 512_256_128 model suggest that segmentation, in this case, may have removed or altered essential features required for accurate classification. The lower binary accuracy, precision, and AUC indicate that the model struggles more with the segmented data than with the raw or augmented RGB data. While the binary cross-entropy loss is lower, this does not compensate for the decreased performance in accuracy and precision.

CNN_32_64_128	Binary Cross Entropy	Binary Accuracy	Precision	Recall	AUC
RGB	0.3296	0.9288	0.9173	0.9548	0.9701
RGB_Augmented	0.3626	0.9361	0.9418	0.9227	0.9660
RGB_Segmented	0.6753	0.8316	0.7944	0.8572	0.9026

Table 13: Performance metrics of CNN model with different input configurations

CNN_32_64_128 CNNs have demonstrated clear superiority over MLNNs in terms of accuracy, precision, recall, and AUC. This highlights the importance of using convolutional architectures in image processing, where the ability to capture and learn local spatial features is crucial. However, segmented images did not provide the same rich information as the other versions of the dataset, leading to a significant decrease in the performance of the CNN. This suffering with segmented data

indicates a reliance on full image context. This suggests that while segmentation could have been useful for highlighting specific areas of interest, in this case probably removed important contextual information necessary for accurate classification.

In the case of the CNN_32_64_128 model, overfitting is indicated by the disparity between the training loss and validation loss, with the training loss being significantly lower. This suggests that the model has learned the training data's specific details and noise, which do not generalize to new data.

TogNet	Binary Cross Entropy	Binary Accuracy	Precision	Recall	AUC
RGB	0.2277	0.9288	0.9245	0.9436	0.9737
RGB_Augmented	0.2224	0.9153	0.8940	0.9286	0.9703
RGB_Segmented	0.4125	0.8195	0.8092	0.8970	0.9185

Table 14: Performance metrics of TogNet model with different input configurations

TogNet The TogNet model performs exceptionally well on the raw RGB data. The low binary cross-entropy loss indicates a good fit to the data. The high binary accuracy, precision, recall, and AUC values suggest that the model is both accurate and reliable in distinguishing between the two classes. With augmented RGB data, the model maintains high performance, though there is a slight decrease in binary accuracy and precision compared to the raw RGB data. However, the binary cross-entropy loss is slightly lower, indicating that augmentation helps the model generalize better. Also in this instance, the RGB_Segmented dataset doesn't provide enough information for the CNN to be able to clearly distinguish between the two classes. Overall, the addition of dropout layers in the TogNet model has successfully reduced overfitting, allowing the model to perform well across different datasets, especially in terms of generalization and robustness.

VIII Conclusion

Both the multi-layered neural network and the Convolutional Neural Network demonstrated reasonable performance in terms of Zero-One loss and accuracy for both the original and augmented datasets. However, to obtain more accurate and conclusive results, it is essential to run the models using greater computational power. This would enable to conduct more experiments and implement a more refined tuning approach. Specifically, it would allow to increase the number of hyperparameters to tune, which is not feasible within the constraints of the Google Colab environment. Improved computational resources would enable more extensive exploration and optimization of the models, potentially resulting in substantial performance enhancements.

Concerning my results, they showed that it is possible to obtain good results (besides the overfitting) in the binary classification task with the convolutional neural network, while the multi-layered network showed results far from convincing. On the other hand, the TogNet Convolutional Neural Network performed in a more promising way.

The segmentation task, whose objective was to delete non relevant details, showed better results in the MLNN, making the images too simple for the Convolutional ones. The augmented dataset

proved to be the most effective, as it enhanced the network's ability to understand the data by introducing 'beneficial noise.' This noise helps the network learn truly relevant features by providing a more varied and representative set of training examples. Data augmentation has enhanced the model's ability to generalize to unseen data by exposing it to a wider variety of scenarios during training, thereby improving its overall robustness and performance. However, the small size of the dataset likely impacted the task, despite the augmentation.

1 Future Work and Improvements

A future experiment could consider applying both augmentation and segmentation to the dataset to analyze the changes in the metrics of interest. By combining these two techniques, it might be possible to enhance the model's performance further. Augmentation can increase the diversity of the training data, helping the model to generalize better, while segmentation can focus on the most relevant parts of the images, potentially improving the model's ability to identify key features. This combined approach could lead to improvements in various performance metrics such as the considered ones, providing a more comprehensive understanding of the model's capabilities and limitations. Additionally, experimenting with a less strong augmentation could provide useful results.

References

[1] Rudolf Kruse et al. *Computational Intelligence: A Methodological Introduction*. Springer London, 2013, p. 492. DOI: 10.1007/978-1-4471-5013-8. URL: https://doi.org/10.1007/978-1-4471-5013-8.

[2] Òscar Corominas Lorente, Ian Riera, and Aditya Rana. *Image Classification with Classic and Deep Learning Techniques*. Journal Article. 2011. DOI: 10.48550/arXiv.2105.04895. URL: https://doi.org/10.48550/arXiv.2105.04895.

[3] Marco Frasca. *Learning from Unbalanced data*. University Lecture. 2021.

[4] Gotam Lalotra et al. "Addressing Binary Classification over Class Imbalanced Clinical Datasets Using Computationally Intelligent Techniques". In: *Healthcare* 10 (Aug. 2022), p. 28.

[5] Justin Johnson and Taghi Khoshgoftaar. "Survey on deep learning with class imbalance". In: *Journal of Big Data* 6 (2019), p. 27. DOI: 10.1186/s40537-019-0192-5. URL: https://doi.org/10.1186/s40537-019-0192-5.

[6] Mahdi Hashemi. "Enlarging smaller images before inputting into convolutional neural network: zero-padding vs. interpolation". In: *Journal of Big Data* 6 (2019), pp. 1–13. DOI: 10.1186/s40537-019-0263-7. URL: https://doi.org/10.1186/s40537-019-0263-7.

[7] Connor Shorten and Taghi M. Khoshgoftaar. "A survey on Image Data Augmentation for Deep Learning". In: *Journal of Big Data* 6 (2019), pp. 1–48. DOI: 10.1186/s40537-019-0197-0. URL: https://doi.org/10.1186/s40537-019-0197-0.

[8] Radhakrishna Achanta et al. "SLIC Superpixels". In: *École polytechnique fédérale de Lausanne* (2010), p. 15. URL: https://www.iro.umontreal.ca/~mignotte/IFT6150/Articles/SLIC_Superpixels.pdf.

[9] Keiron O'Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. Journal Article. 2015. DOI: 10.48550/arXiv.1511.08458. URL: https://doi.org/10.48550/arXiv.1511.08458.