UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

Alberto Bertoncini 983833
Massimo Cavagna 983820

Statistical Methods for Machine Learning

Neural networks for binary images classification

July 2022

Plagiarism Declaration

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

Abstract

The main purpose of this paper is to show the methodology applied in image (binary) classification and the results obtained with the use of different neural network structures and hyper parameters. We particularly focused on the binary classification task on a dataset containing cats and dogs images: this task is accomplished with the use of feed-forward and convolutional neural network. Different parameters are tuned over the two type of neural network by using nested cross validation showing the differences in therms of performances: as performances metrics we use the 0-1 loss function and the binary accuracy. After the parameters' tuning, the models' performances are again evaluated with their "best" parameters' values by using both the original dataset and an augmented version of it: in this last evaluation other metrics are taken into account, such as True/False Positive, True/False Negatives and AUROC, in order to have a more general and complete view of the performances.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Image classification is a well known task that attempts to understand the content of the image as a whole and to assing a label to it. In order to fulfill this task machine and deep learning models have been developed for decades now, resulting in the development of the famous convolutional neural network which proved to work very well in image classification and object detection. Both object detection and image classification are now broadly used in lots of application fields such as medicine or security: for example, it may be used to analyze MRI or x-ray images providing the doctors the necessary help in detecting health "problems", or face recognition used to grant the access to sensitive information.
In the following sections, we show all the methods applied to fulfil a binary classification task by using the "simple" multi-layered neural network and the already mentioned convolutional one: the aim is to show not only the comparison between the two structures in therms of performance, but also how the same network's hyper parameters influences the performance itself. As it will be detailed later on, the main metric used to evaluate the models is the **0-1 loss** which give an intuitive idea of the model's performance: it basically returns an error of 1 if a prediction is wrong, 0 otherwise. It is notable to state here that the 0-1 loss has been used only as evaluation metric for the models, and not, like the "loss" name suggests, as loss function during the training phase.

Chapter 2

Environment

Before going into the details of the various theoretical background and the implementation choice, we present in this section the main tools and environment used to run the code developed, which are, in our opinion, a relevant choice for the execution of the models and the experimental setup.
The whole code has been written in Google Colab which provides free notebooks for Python3 code execution with lot of pre installed packages in a Linux like machine. We chose this environment because of the simplicity and intuitiveness which makes easier to code with multiple libraries. It is also very practical in therms of shareability between developers. What's more, Google Colab provides GPU and TPU for free with a basic account, of course with some time limits per day. The latter limitation had an impact in the experimental setup, bringing us to make some choices over others that will be described later.
The main code components are available on the GitHub repository, so in the Colab notebook is contained only the code necessary to run the experiments.
Here we present a brief list of the main packages and libraries used:

- Pillow (PIL) package: it provides all the functionalities to read, write, modify the images of various extensions and formats.
- Numpy: in order to manage the images as arrays
- Json: it is useful to saves dictionaries (like the training results) as json file.
- Matplotlib: well known library used to draw different plots.
- Seaborn: together with Matplotlib, we used it to refine our plots.
- Tensorflow: it provides all the basic functionalities to build, train and use deep learning models with Keras and all the dataset management tools. It provides specific functionalities to manage images, and other data type, making the life easier in the kind of task describe in this paper.

Chapter 3

Image Preprocessing

3.1 Dataset

We started our classification task by observing the data available: in this case all the images used to train and validate the various models presented later on are available in a folder in UnimiBox and downloadable using the link provided by UnimiBox itself.
The images are already divided into two folders (one for each category), with the same number of images (12500 each) for a total of 810 MB (Cats: 370 MB, Dogs: 440 MB). Of course, this pre partitioned and well balanced dataset allowed us to skip a lot of complex operations over the data (like the re balancing), but, although this can be considered an "easy" dataset to be managed, we started observing the data to ensure they are as clean as possible.

3.2 Preliminary operations

Given the small size of the dataset the very first preliminary operation has been made "by hand", observing the images, looking for evident errors in classification (like a cat image in the dog folder and vice versa), or images that do not represents one of the two animals at all. The content of the images is various: indeed, often cats and dogs are not the main content of the image, the presence of multiple subjects (like people or subjects belonging to both the classes), a lot of details in background and around the subjects (structures, domestic backgrounds, or cats/dogs boxes) and sometimes also images whose subject does not belong either to cats or dogs (such as an Einstein photo, or kennels' objects). Certainly, the images where the presence of one of the two animal is not present have been removed.
The next step is the transformation into a standard format and size: these images have different sizes and extensions like JPEG, PNG, BMAP or GIF, so, in order to make them more handy, they have been converted into jpeg in RGB mode. During the conversion two corrupted images were detected one in each folder. The amount of healthy images after the deletion of the corrupted and not "task concerning" ones is 12480 images for the cats folder and 12477 for the dogs folder.
Another important aspect is the size of the images. In each folder the size of the images are not the same, so a resizing step has been necessary, but, here, we encountered the first limitation of

the environment described: the first attempt was to use the average dimensions as standard size for all the images with the purpose of not twist the proportions, but this led to a limitation in the models size, which were not able to fit in the Colab memory. Thus, after few more attempts all the images have been resized to a constant size of 75x75 pixels which seemed to be a good trade off between the quality and the models management.
During the first quick scan, we noticed also another frequent problem that is the white background: sometimes the real picture is confined in an corner or it is centered with a large uniform (white or other color) border, so we cropped them before the resizing excluding useless features.

3.3 Data augmentation

As mentioned in the previous section, the number of images is not huge, so data augmentation helped in generating new data from the existing ones enlarging the amount of samples.
These techniques are used to create new data starting from the already in posses, performing some changes over them but with a high confidence that the modified ones fall in the same class of the source ones. Two kind of changes have been applied over a randomly extracted set of samples (25% of the dataset): transformations and filters.
The transformations take the image and change it keeping the pixel values the same. These kind of changes have been not applied only to increase the number of training examples, but also to avoid that the network could learn useless features like the perspective or particular background details. Although there exists quite a large number of transformations available, the selected ones are:

- Mirroring: the image is mirrored with respect to the vertical axis;
- Rotation: the image is rotated with a random angle (with step of 10 degrees);
- Flipping: the image is mirrored with respect to the horizontal axis.

Filters, on the other hand, are modifications that change the pixel values, resulting in the change of the colors or the arrangement of the pixels. This changes have been applied in order to differentiate the color scheme, thus avoiding the network to learn a bias over some particular palette or scheme of the image's lights (light colors $\rightarrow$ dogs). Also in this case, there exists a lot of filters and lot of combination, but we selected a small number of them:

- Blurring: the image is blurred setting each pixel to the average of the pixels in the radius of 5;
- Unsharpening: the images detail evidence is increased, it exploit a Gaussian blur and subtracting it to the original image it is possible obtain and increment the contrast for the smallest details;
- Spreading: the image's pixels are randomly spread with a maximum distance of 15 from the original position;
- Contrast: increment the contrast of the image with a factor of 100;
- Color: adjust the balance of the colors in the image.

3.3.1 Segmentation

The transformations and filters gave us a real new bigger dataset, but we already stated the variety in the images content: even if some filters, like spreading and blurring, 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 (like rug's motif). 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, in order to simplify the images a K-Means clustering algorithm has been applied over each image: every pixel has been set to the value of the centroid of its cluster, with a total of 4 cluster per image. In this way, the images became very simple and only the principal shapes and colors survived.
Note that we will refer to this new dataset as the "segmented" one, although the word "segmentation" may lead to think to a more complex task, the reason is that in most of the resulting images the subjects are highlighted discarding lots of useless details already mentioned. The result of the data segmentation and augmentation has been four datasets that we used separately: the original (used in the hyper parameter tuning), the augmented and the segmented with its augmented version.

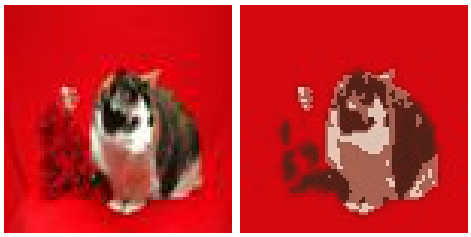


Figure 3.1: Comparison of original and segmented image.
On the left the original picture resized to 75x75 pixels. On the right the "segmented" one.

Chapter 4

Models

4.1 Artificial Neural Networks

Artificial neural networks (ANNs) are nowadays broadly used in several fields, given their powerful capability in the classification and regression tasks. In this section we present briefly some useful notions related to the main structure of the machine learning models we took into account in our work.
A neural network is described as a graph G = $\langle N, E \rangle$ where N in the set of nodes who are the computational bricks of the entire architecture, while E is the set of weighted edges. Each neuron (node $\in N$) first computes the product of the weights of incoming edges with the corresponding incoming inputs, then an activation function is applied over it.
Mathematically, the output of a neuron is:

$$\sigma(w^T x)$$

where σ is a function chosen arbitrary.
In this paper, we refer only to feed forward neural network: these models are called *feed forward* because the flow of information goes only in "one direction" meaning that there are no cycles in the graph representation of the network.
Here we present the two kind of feed forward neural networks used.

4.1.1 Multi layered Neural Network

Multi Layered Neural Network (MLNN) is a kind of feed forward neural network whose structure is organized into *layers* of neurons and connections exists only from a layer to the subsequent one. In particular, there exist three main type of layers:

- **Input**: this layer has no incoming edges, it takes in input the data as they are and it is in charge of spreading it to the next layer.
- **Output**: This layer is in charge of giving an output who will be the prediction made by the network.
- **Hidden**: all the layers in between the input and the output layers.

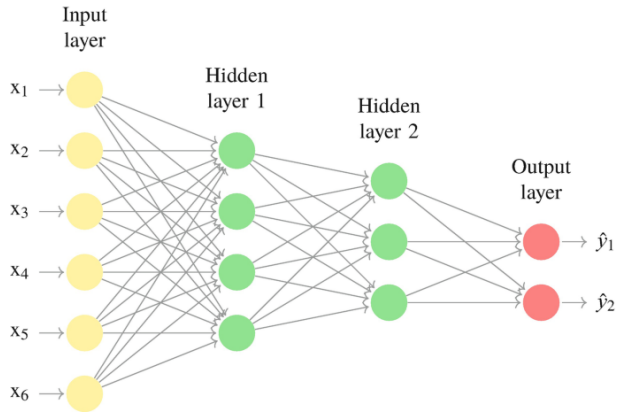


Figure 4.1: Multi Layered Neural Network with complete connectivity (no missing edges)

4.1.2 Convolutional Neural Network

Convolutional neural networks (CNNs) are a kind of ANN that became very used in computer vision tasks: the fact that this kind of network is able to detect patterns inside images make them particularly suitable for the task we are addressing in this paper.
Here is a brief introduction describing the main components:

- Convolutional layer: this layer's purpose is to extract features from the images and uses a *kernel* (which is a vector/matrix) whose dimensions are smaller then the image' ones. Imaging the the picture into consideration as a matrix, it starts from the top-left angle, select the image's submatrix of kernel's dimensions and multiplies it with the kernel itself. This procedure is applied over all the image thus obtaining a feature map matrix.

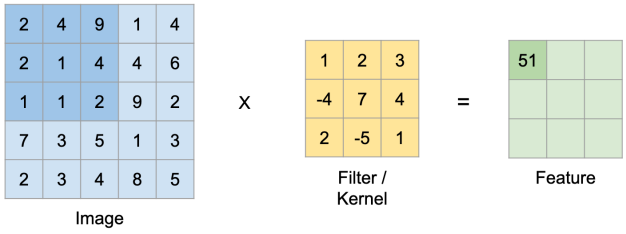


Figure 4.2: Example of a convolutional layer with 3x3 kernel size

- Pooling layer: as the convolutional, the pooling layer is part of the networks which extracts information from the image. It basically 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 as shown in *figure 4.3*.
- Fully connected layers: these layers are the same previously described as multi-layered neural network.

The final structure of this type of network is shown in *figure 4.4*: as it can be seen, the network is built by the repetition of convolutional and pooling layer ending with a MLNN.

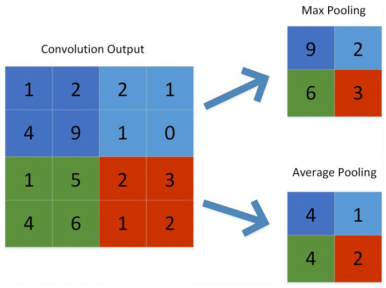


Figure 4.3: Example of a pooling layers with size 2x2

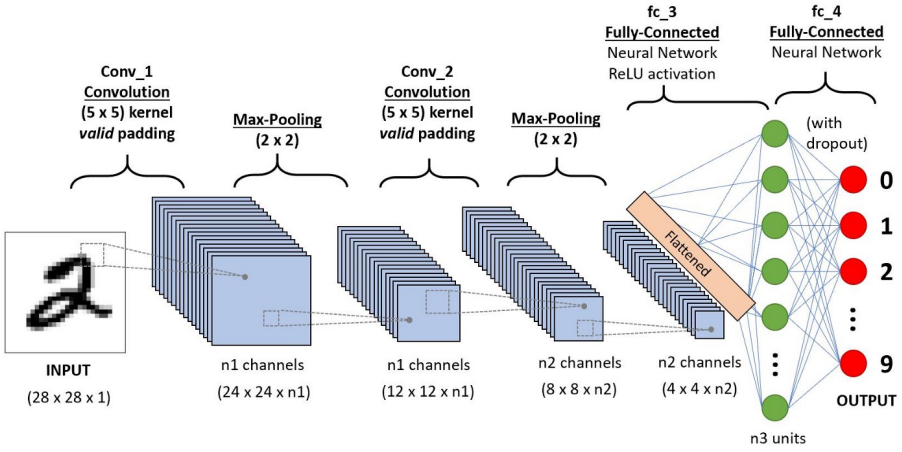


Figure 4.4: Example of CNN for multi-class classification task

4.2 Network structure choice

The choice of the number of layers and neurons per layer is one of the hardest to be made. Here we applied a similar approach for both the MLNN and the CNN that is starting from a very simple small network having one small layer and then increasing both the number of layers and the number of neurons.
All the configurations has been tested using the pre processed data before the augmentation.

4.2.1 MLNN

In the MLNN, we selected the sets of layers listed below: it is notable that we started with a small network with 256 neurons and then we progressively incremented them by adding layers. We also studied the effect of a "reversed" network, by inverting the order of the layers.

- 256
- 128, 256
- 256, 128
- 1024, 512, 256, 128
- 128, 256, 512, 1024

In this phase each network structure has been evaluated by five fold cross validation (whose

functioning is described below) and by the following default values:

MLNN	
Parameters	**Values**
Optimizer	SGD
Learning rate	0.01
Dropout	0.3
Epochs	70

Table 4.1: Default settings

In *Figure 4.5* are shown the results of the runs: the first observation can be made is that the more layers we add the lower loss we get in the training and validation set. In general, it seems the **reversed** networks performs worse then the same with decreasing number of neurons. Regarding this observation, we put in evidence the results of the bigger networks, because, besides the performance, it seems that the reversed networks has a smaller distance between the training and validation results.

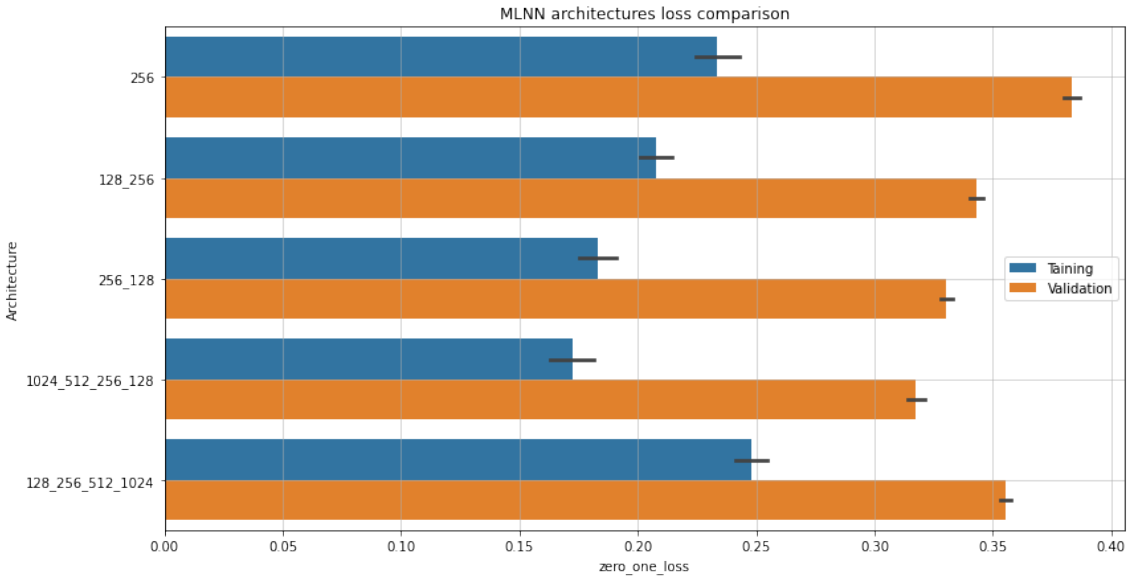
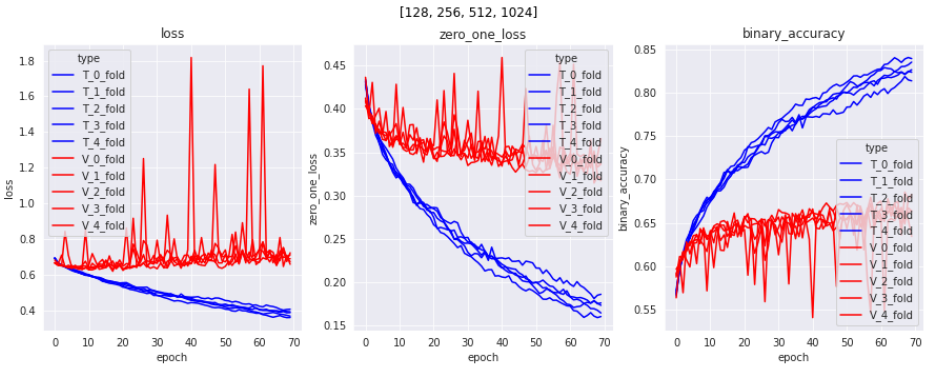
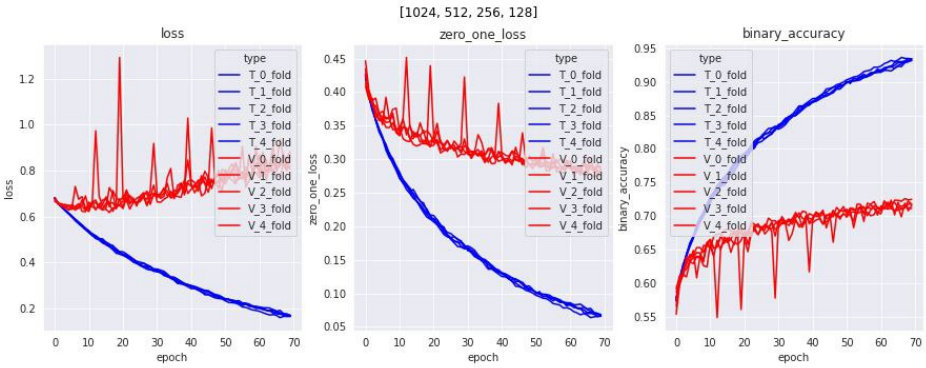


Figure 4.5: MLNN zero-one loss values for each architecture

In *Figure 4.6* we show the results obtained in these two last architectures: as *Figure 4.5* states, the reversed network performed worse that the other one and it also seems the results of the latter one are more "stable". Of course, the results of both networks shows poor performance in the validation sets, stating the tendency to overfitting. Because this well marked behaviour, the choice we made is the reversed architecture, due to its slower degrowth of loss and a weaker overfitting behaviour.



(a) 128 256 512 1024



(b) 1024 512 256 128

Figure 4.6: Performance of the biggest network and its reversed. From the left: BinaryCrossEntropy, 0-1 loss and BinaryAccuracy.

4.2.2 CNN

In the case of the CNN we took a similar approach by setting some default values

CNN	
Parameters	**Values**
Optimizer	SGD
Learning rate	0.01
Dropout	0.3
Kernel size	(3, 3)
Pool size	(3, 3)
Epochs	50

Table 4.2: Default settings

and a set of hidden layers:

- 128, 256
- 256, 512
- 32, 64, 128
- 64, 128, 256

- 128, 256, 512

Here, we highlight the fact we used only increasing number of neurons through the layers: this is done because at each layer the idea is to capture bigger and bigger patterns.
Again we plotted the 0-1 loss values for all the architectures: while in the multi-layered there was some kind of relationship between the increasing number of layers (and neurons) and the decreasing loss, here we have complete different results. At a very first look it seems the smallest network (128_256) is the best, but from a further analysis we concluded it was not: in *Figure 4.8* are reported the history of the performance along all the epochs and it shows how the smallest network started to overfit after nearly 35 epochs, whilst the bigger network is, besides the "instability" and lower accuracy values, still maintaining similar values for both the training and validation set.

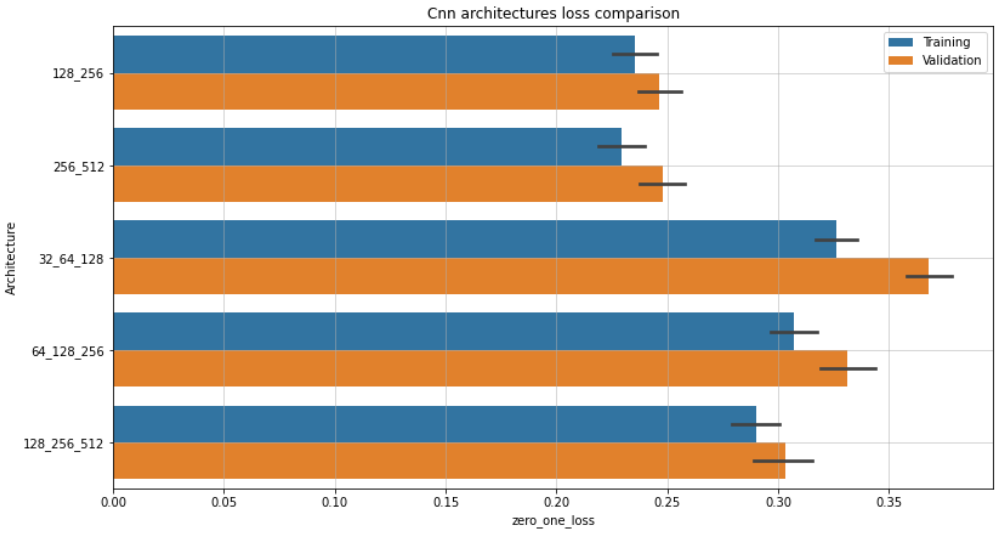
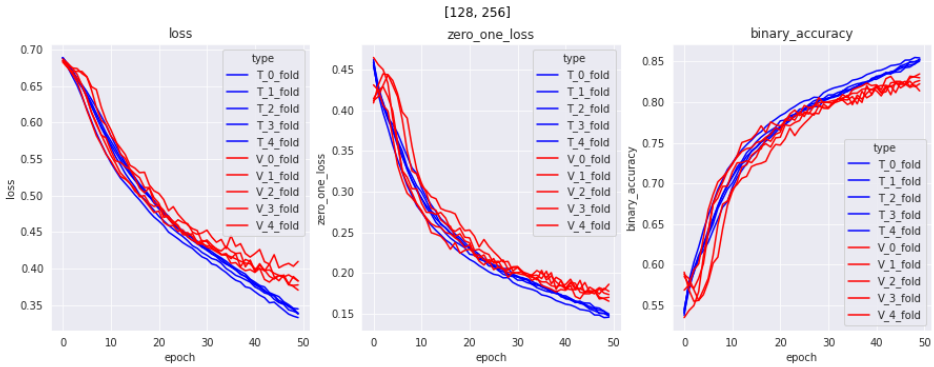
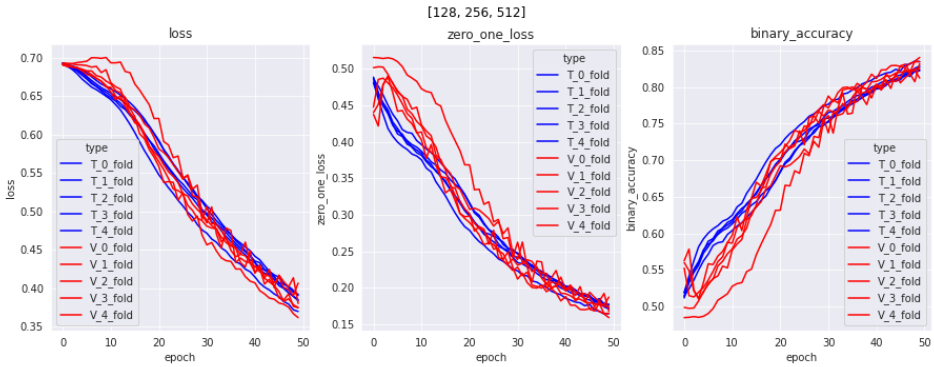


Figure 4.7: CNN zero-one loss values for each architecture



(a) 128 256



(b) 128 256 512

Figure 4.8: Performance of the smallest network and the biggest. From the left: BinaryCrossEntropy, 0-1 loss and BinaryAccuracy.

4.2.3 Final architecture choice

These are the layers choice for each neural network:

- **MLNN**: 128, 256, 512, 1024
- **CNN**: 128, 256, 512

In practice, the neural networks we chose contains also some additional feature, indeed the multi-layered NN at first gave us very poor results with a training and test error about 50%, we then add a normalization step between each layer and that improved our results.
Regarding the CNN, we added a fully connected layer of 128 neurons, after the features extraction part: the choice of this size has been made based more on the Colab limitations than on a real tuning.
As activation function we chose ReLU in the hidden layers and a sigmoid (logistic) in the output neruon for both the MLNN and the CNN. This choice was taken basing on the fact that the deepest networks could not be well trained, or not trained at all. Regarding this fact, it will be showed later the results in the test set with a sigmoid as activation function in the hidden layers.

4.3 Hyper-parameter tuning

The tuning of the hyper-parameter is a tricky part of the learning process. We needed to define

1. the main hyper-parameters we wanted to make the tuning of.
2. the tuning methods.

Here are the list of hyper parameters for both the MLNN and the CNN:

Hyper-parameters	
MLNN	**CNN**
Optimizer	Optimizer
Learning rate	Learning rate
Dropout	Dropout
-	Kernel size
-	Pooling size
-	Activation func.

Table 4.3: Hyper parameters list

It is evident the unbalance in the hyper parameters selected to be tuned in the multi layered NN and the convolutional one: the choice is made due to the first results in which we noticed how a "simple" fully connected network is not able to recognize the patterns inside the images, conversely it seemed to be able to learn by heart the single images, which is, of course, not the task's objective.

4.3.1 Estimation algorithms

Once the hyper-parameters have been defined, we proceeded with the choice of the tuning technique: there are several algorithms to fulfill this task but, again, here we got the limitation of our environment which made us nearly impossible to run an algorithm like Grid Search. We preferred to rank our hyper parameters and then apply a **Nested Cross Validation** algorithm sequentially.

Nested Cross Validation

The algorithms described above is the one actually implemented in our work: it takes the dataset and split it in *train* and *test* set, then, for each provided value of the hyper-parameter, performs a **five fold cross validation** thus to estimate the average performance and saves the hyper parameter value that led to the lowest 0-1 loss. The selected value is then employed in the retraining of the model, and the result is evaluated on the test partition. This process is repeated three times over different splits of the dataset in order to avoid bias of a "lucky" partition. The result is an estimation of the error for each theta (if selected by the five fold cross validation as a good values for a particular split, otherwise the error estimate is ∞).

Five Fold Cross Validation

Algorithm 1 Nested Cross Validation

```
Require: hyperparameters, dataset, model
  n-folds = splitDataset(dataset)
  finalError = inf
  for train, test in n-folds do
    error = inf
    for theta in hyperparameters do
      tmpError = FiveFoldCrossValidation(theta, train, model)
      if tmpError < error then
        error = tmpError
        bestTheta = theta
      end if
    end for
    trainedModel = trainF(model, bestTheta, train)
    bestThetaError = evaluate(trainedModel, test)
    add(errorEstimation[bestTheta], bestThetaError)
  end for
  for theta in hyperparameters do
    errorEstimation[theta] = mean(errorEstimation[theta])
  end for
  return errorEstimation
```

Algorithm 2 Five Fold Cross Validation

```
Require: theta, dataset, model
  n-folds = splitDataset(dataset)
  error = 0
  for train, validation in n-folds do
    trainedModel = trainF(model, theta, train)
    error += evaluate(trainedModel, validation)
  end for
  return error/5
```

The internal procedure of five fold cross validation performs the training of the model five times over different splits of the dataset provided as argument. These trained models are evaluated on the current validation partition and the errors are summed together. The result is the mean of the error as an estimation of the risk of the model with that particular hyper-parameter value.

4.3.2 Multi Layer neural network hyper-parameter tuning

The multi layer neural network presents three principal hyper-parameters in its structure, as already mentioned in table *Table 4.1.* We recall the strategy for the hyper parameter tuning is **conditioned by the Colab limitations**, thus we started from the basic setting found in *section 4.2* and we applied the nested cross validation on the parameters **sequentially**, using 10 epochs for each training phase. We also highlight the fact that the hyper parameter tuning has been done only looking at the **0-1 loss**.
Here are shown the results:

Multi layered optimizer	
Optimizer	**Zero-one loss**
SGD	∞
Adam	0.3446
Nadam	0.3434

Table 4.4: ML optimizer risk

Multi layered learning rate	
Learning rate	**Zero-one loss**
0.001	0.3518
0.005	0.3374
0.01	0.3434
0.1	∞

Table 4.5: ML learning rate risk

ML dropout	
Dropout rate	**Zero-one loss**
0.2	0.3301
0.4	∞
0.6	∞

Table 4.6: ML dropout rate risk

4.3.3 Convolutional neural network hyper-parameter tuning

The very same strategy has been applied over the convolutional neural network:

Convolutional optimizer	
Optimizer	**Zero-one loss**
SGD	0.3490
Adam	0.3068
Nadam	∞

Table 4.7: Convolutional optimizer risk

Convolutional learning rate	
Learning rate	**Zero-one loss**
0.001	0.3068
0.005	0.5027
0.01	0.4980
0.1	0.4954

Table 4.8: Convolutional learning rate risk

Convolutional dropout	
Dropout rate	**Zero-one loss**
0.1	0.4035
0.3	0.2721
0.5	∞

Table 4.9: Convolutional dropout risk

Convolutional kernel size	
Kernel size	**Zero-one loss**
(2, 2)	0.5043
(3, 3)	0.2524
(4, 4)	∞

Table 4.10: Convolutional kernel size risk

Convolutional pooling size	
Pooling size	**Zero-one loss**
(2, 2)	0.2685
(3, 3)	∞
(4, 4)	0.3793

Table 4.11: Convolutional pooling size risk

Convolutional activation function	
Function	**Zero-one loss**
ReLU	0.2685
Sigmoid	0.5013
Softplus	0.5024

Table 4.12: Convolutional kernel size risk

4.4 Risk estimate

From the previous section we extracted the best configuration of hyper parameters, considering the ones which gradually decrease the loss, in table are reported the corresponding values for each NN:

MLNN	
Parameters	**Values**
Optimizer	Nadam
Learning rate	0.005
Dropout	0.2

Table 4.13: MLNN Final settings

CNN	
Parameters	**Values**
Optimizer	Adam
Learning rate	0.001
Dropout	0.3
Kernel size	(3, 3)
Pool size	(2, 2)

Table 4.14: CNN Final settings

Once the final structure (layers and neurons) and hyper parameters is defined, the networks have been evaluated using the **five fold cross validation** previously described. In particular, we applied the estimate over the four different configuration of the dataset:

- Original: the pre processed dataset.
- Augmented: the original dataset plus the randomly modified images.
- Segmented: each image is managed with K-Means.
- Segmented & augmented: each image is managed with K-Means and then augmented.

4.4.1 MLNN risk estimate

MLNN risk estimate	
Dataset	**0-1 loss**
Original	0.3393
Augmented	0.2965
Segmented	0.3675
Segmented augmented	0.3175

Table 4.15: Five Fold Risk Estimate for each dataset on the tuned multi layered neural network

The *Table 4.15* and *Figure 4.9* show how the MLNN is performing better with the original augmented data. The reduction of the details by using the K-means algorithms seems to have the opposite effect than the desired one: indeed, apparently the MLNN is able to memorize the images as they are, and, by simplifying them, we unconsciously helped the network in this task instead of prevents the overfitting.
While the segmentation is not giving good results, the augmentation apparently has a mitigating effect over the overfitting, although not so significant.

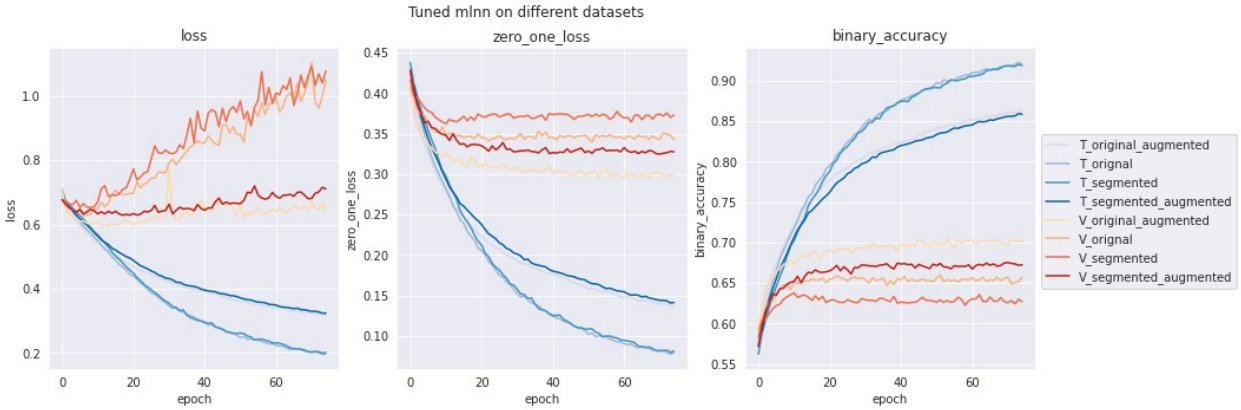


Figure 4.9: The average trend of the tuned MLNN with the five fold cross validation over different datasets

4.4.2 CNN risk estimate

CNN risk estimate	
Dataset	**0-1 loss**
Original	0.1292
Augmented	0.1516
Segmented	0.1744
Segmented augmented	0.2018

Table 4.16: Five Fold Risk Estimate for each dataset on the tuned convolutional neural network

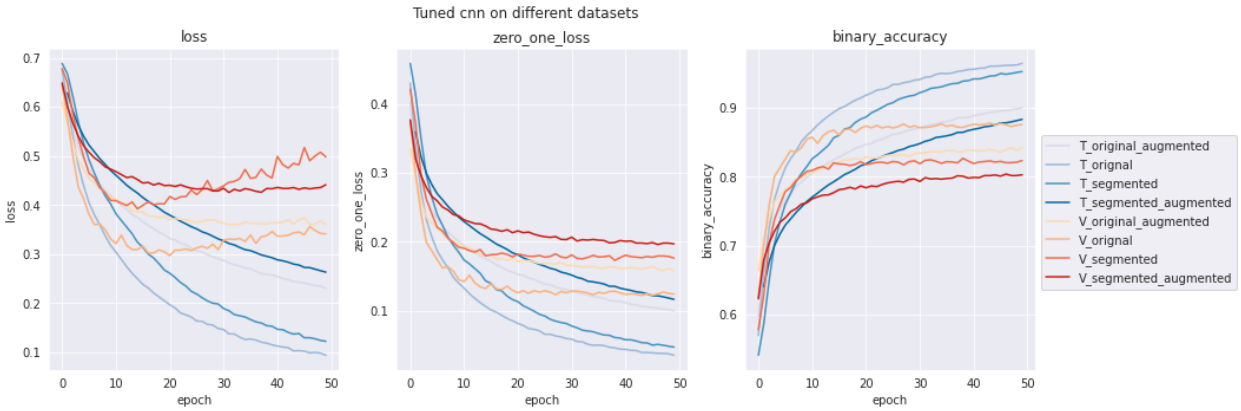


Figure 4.10: The average trend of the tuned CNN with the five fold cross validation over different datasets

The convolutional neural network is, in general, performing better than the MLNN, as expected: even though the results over all the datasets shows the tendency to overfit, in *Table 4.16* it is notable the capability of the network to maintain the loss under the 20%.
The effect of the segmentation is not the one expected: considering the functioning of the convolutional layers, the networks is not able to recognise the patterns. In other words, the segmentation process deleted too many details the network must recognize in order to fulfill the task.

Although the augmented dataset is apparently giving worse results that the not augmented ones, it is notable in *Figure 4.10* how, on average, the network over the not augmented data overfits faster: the poor performance over the augmented data can be imputable to the higher number of data available, giving the network more epochs the same performance may be reachable.

4.4.3 CNN comparison: default dataset v. augmented

Before giving the actual comparison between the performance of the CNN over the original (smaller) dataset and the augmented one, we show here some more metrics for both. *Figure 4.11* represents the metrics for the original dataset: besides the three already mentioned, we analyzed also the true/false positive/negative and the AUROC curve.
Recall that the original dataset is composed by about 12.500 examples and the five fold cross validation divides into 80% training and 20% validation set.
From the charts, it is possible to state that the CNN is behaving in the same way for both the True/False Positive and Negative examples: this is an expected result considering the balance between categories. The very same result can be verified in the AUROC curve which is reaching good values (over 0.9 in the validation set).

Observing *Figure 4.12* to affirm that the CNN over the augmented dataset is behaving in the same way over the original one. We recall here the dataset is much bigger ($\sim$37.000 per category).

Focusing more on the comparison between the two runs of the models, *Figure 4.13* shows how in general the augmented dataset makes the models overfit less than the original, even though it converges at higher loss values. The figure also shows the comparison between the performances of the models at the 50th and 75th epoch: regarding the absolute loss values we may state the 75th epoch brought to better results, but with the help of the charts above we may disagree observing how the model start to overfit already in the early stages.
Furthermore, considering the tuning approaches which has been conducted only with the original dataset, we can also say that the results with the augmented one brought to more promising results, considering that the network has been tailored over a smaller and less various set of images.

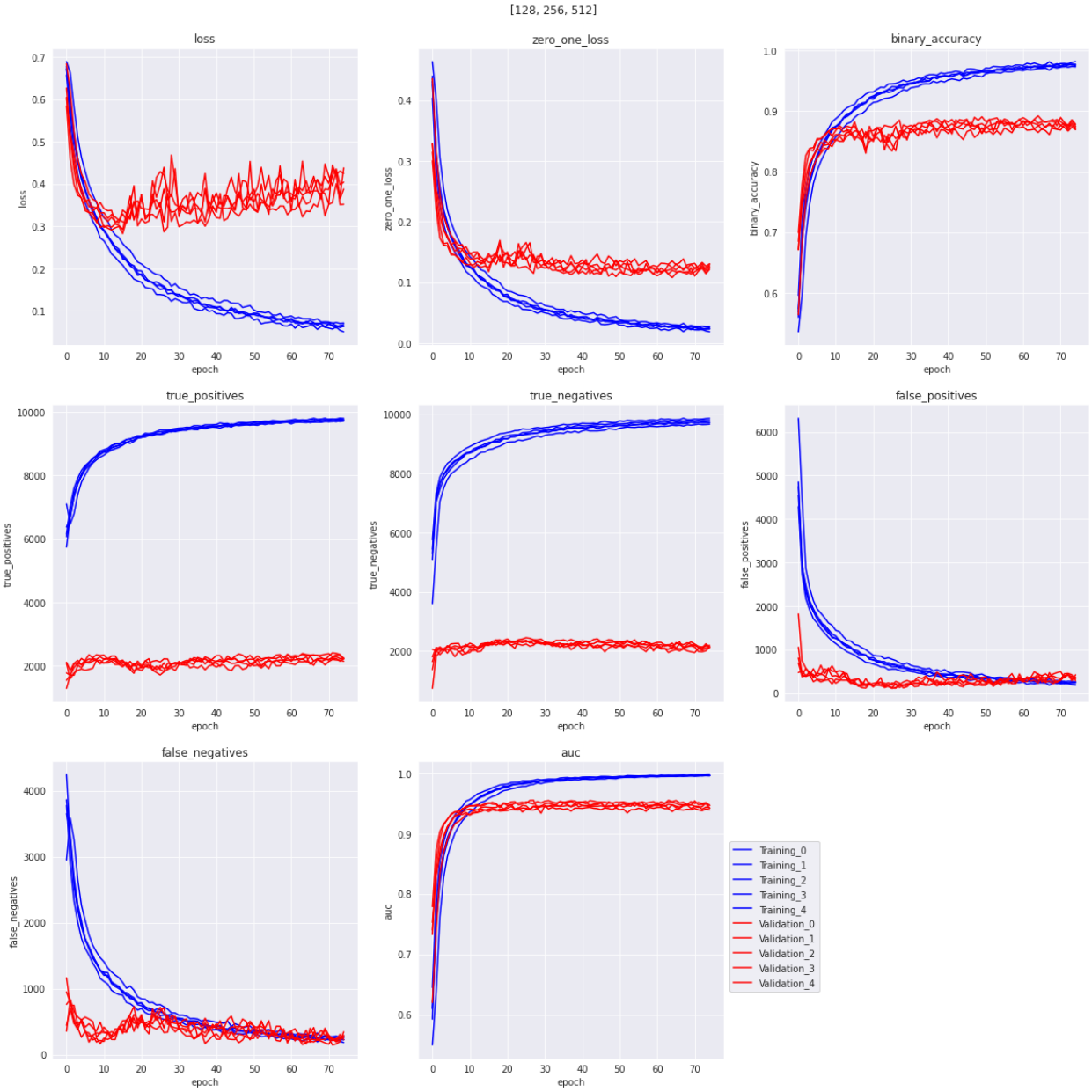


Figure 4.11: The metrics for the CNN performance over the original dataset with 75 epochs

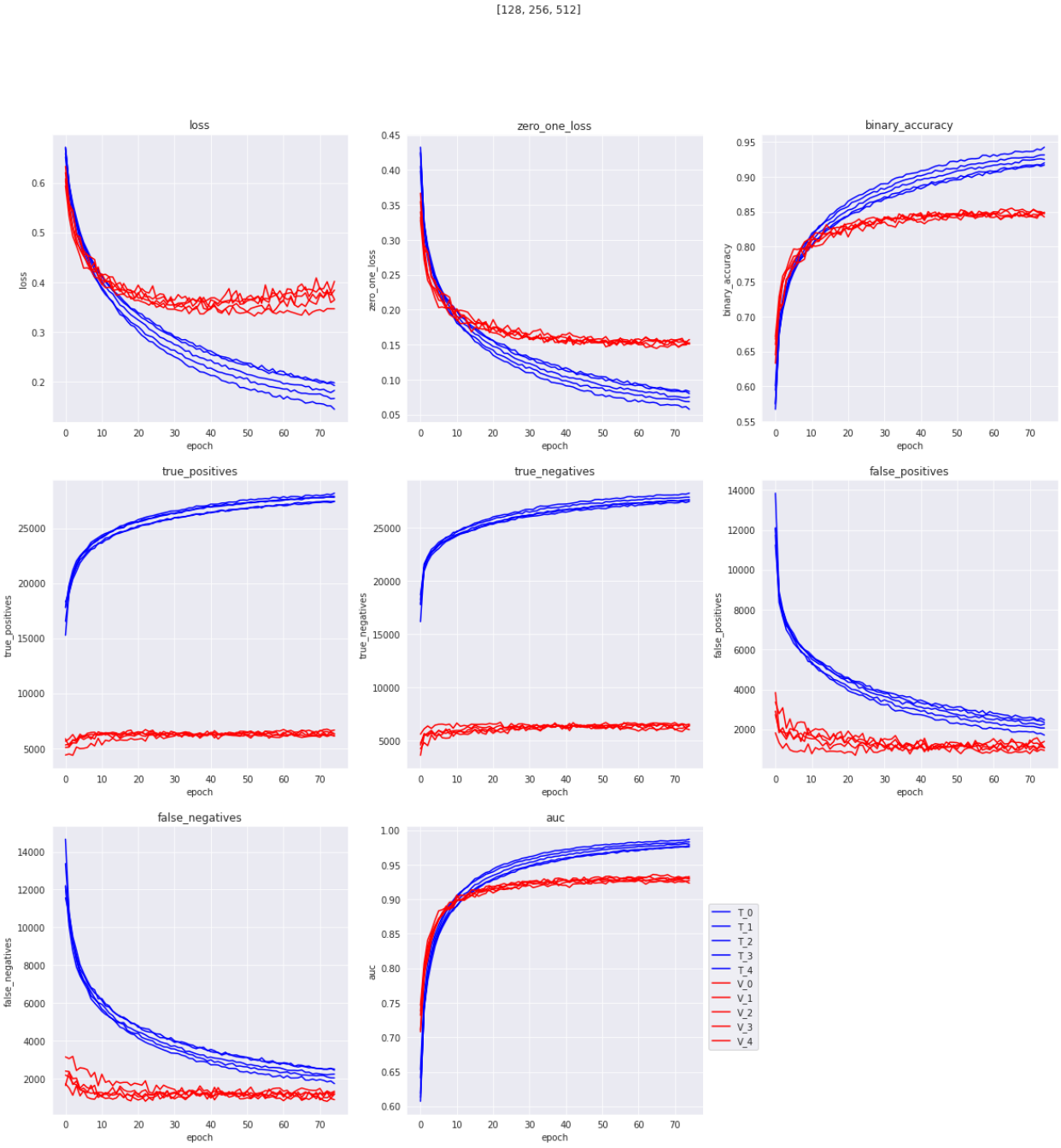


Figure 4.12: The metrics for the CNN performance over the augmented dataset with 75 epochs

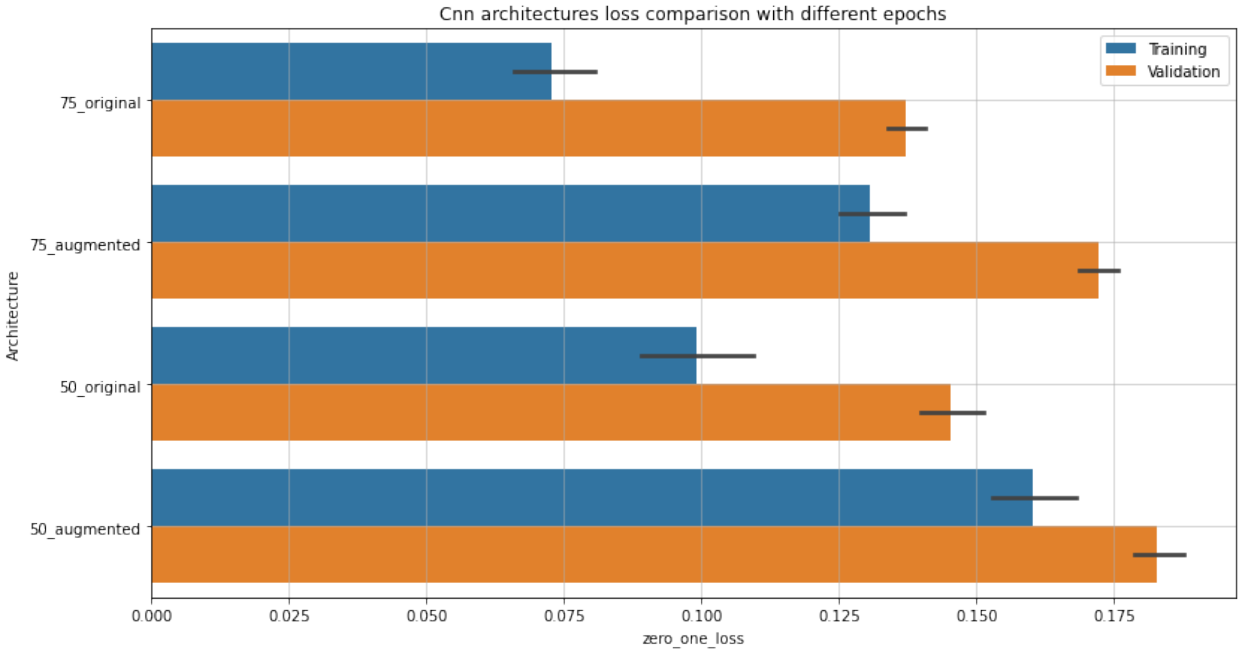


Figure 4.13: Comparison of the 0-1 loss with different epochs

Chapter 5

Conclusion

The results obtained by our analysis can be considered as preliminary: indeed, the multi-layered neural network and the convolutional neural network performances looks promising with respect to the 0-1 loss and the accuracy for both the original and augmented dataset, but, in order to have more accurate results, it is necessary to execute the models with more computational power, which would allow to have more experiments and a better tuning approach, increasing the number of epochs and the number/size of the layers, which is not possible in the Google Colab environment. Concerning our 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 poor results.
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.
The best dataset seemed to be the augmented one that in both network allowed a better understanding of the data, introducing "good noise" making the network learning true relevant features. This dataset has been also useful in order to reduce the overfitting in each tested architecture.