Experimental Project Report
Statistical Methods for Machine Learning
Neural Networks

Chiara Anni, 28503A, chiara.anni@studenti.unimi.it
Nicolò Pignatelli, 13831A, nicolo.pignatelli@studenti.unimi.it

Academic Year 2022-2023

Contents

1 Abstract

In this report we present our project about neural networks. We are asked to use *Keras*, a *Python* library, to train different binary classifiers in order to distinguish between images of muffins and Chihuahuas from this data set. In particular, after the data preprocessing, i.e. converting images from JPG to grayscale pixel values and scaling them down, we are asked to:

1. experiment with at least 3 different architectures and training hyperparameters;
2. use 5-fold cross validation to compute the risk estimates;
3. discuss the obtained results, especially the influence of the choices of hyperparameters and network architectures on the cross-validated risk estimate.

Also, we have to use the zero-one loss to compute the cross-validated estimates. Overall we follow the steps of the task trying to find the best model for our data. We start with a base multilayer neural network that we improve until we decide to move to a convolutional one that is more appropriate considering the data we have. We manage to achieve a test accuracy of more than 90% thanks to the final model.

2 The data set and data preprocessing

2.1 The data set

The data set is composed by images of Chihuahuas and muffins of different sizes and it is already divided in two sets: training and test, with respectively 4733 and 1184 files.

2.2 Data preprocessing

In order to load separately the training and test set, we use two different custom functions. Thanks to them, we are also able to set the same size for every image, 100 for all the models and 150 for the final one, preserve the label and convert the pictures to grayscale. After this, we shuffle the training set. This is done for the benefit of the learning process, as it helps the model to generalize. We create the feature matrix, which later is converted to a *NumPy* array and scaled by dividing for 255, and the label array for both training and test set. Lastly, it is important to notice that for the multilayer neural network every image has a shape of a square with the side equal to the one decided during the loading, whereas for the convolutional neural network there is also another dimension that is put equal to 1, as we work with grayscale images.

3 Introduction to Neural Networks

A neural network is a machine learning model designed in a way that is inspired by the human brain. In particular, they are intricate networks of interconnected nodes, or neurons, that collaborate to tackle complicated problems. Neural networks are widely used in a variety of applications, including for example image recognition (as in this case), predictive modeling and natural language processing. This happens because they are specifically designed to recognize patterns. The processing units are arranged in layers. Neural networks are typically divided into three parts:

1. an input layer, with units representing the input fields;
2. one or more hidden layers;
3. an output layer, with a unit or units representing the target fields.

3.1 Feed-Forward Neural Networks

One type of neural network is the feed-forward. It conveys information that is processed in a single direction: from input to output layer. Feed-forward neural networks may have hidden layers for functionality, but they are not strictly necessary. As it was stated before, we use *Keras* in order to construct our models. We try a variety of them, as there are different types of feed-forward neural networks. Anyway, they share many things in common, so we are able to present some similarities. For example, once built the architecture, every model must be compiled, meaning that it must be configured for training. For such purpose, the specific function must be provided with some arguments:

- **loss**: a function that measures the discrepancy between the predicted and the actual label. We decide to use the **cross-entropy loss**.
- **optimizer**: the method used for solving the optimization problem and finding the best model. We use **Adam**. The **learning rate**, a crucial hyperparameter that change how much the model responds with respect to the error, is also involved during the choice of the optimizer. During the task we try to tune it.
- **metrics**: a function that evaluates the performance of the model. We choose **accuracy** as we have a classification problem.

All this done, the model must be fit, i.e. trained. Also in this case, besides the feature matrix and the label array, it must be provided with some arguments:

- **batch_size**: the number of samples per gradient updates. It is convenient not to pass the whole data set all at once in terms of memory and speed. During the task we try to tune it.

- **epochs**: the number of times the data are passed through the model. It may be convenient to do this more than once. We set this argument equal to **20** in general and **50** for the best models.

- **validation_split**: the percentage of data used as validation set. We set this argument equal to **0.2**.

Here it follows a brief description of the two types of feed-forward neural networks that we use.

3.1.1 Multilayer Neural Networks

One kind of feed-forward neural network is known as multilayer. It has at least 3 layers, so there is at least one hidden layer, and nonlinear activation functions. In order to build its architecture there are several functions that can be used in *Python*, all taken from the package *Keras*. Here it follows a review of the ones that we operate:

- **Sequential()**: a class that groups a linear stack of layers into a *tf.keras.Model*. It is always followed by the *add()* method, used to increase the network with layers.

- **Flatten()**: a class that flattens the input. In the multilayer case it is always needed as the first layer with the *input_shape* argument put equal to the shape of the images. As we state this here, we will omit to mention this layer in the models.

- **Dense()**: a class that builds the usual layer of a neural network, i.e. the one that connects all the outputs of the previous layer with all the the nodes that are put as its argument and provides an output for each of the latter.

- **activation**: an argument that assigns an activation function to a specific layer. We used **sigmoid** for the output layer and **relu** for all the others.

- **Dropout()**: a class that add a layer that randomly drops out neurons in order to prevent overfitting. The amount of neurons to drop can be specified as a float between 0 and 1 with the argument *rate*.

3.1.2 Convolutional Neural Networks

Another kind of feed-forward neural network is known as convolutional. The peculiarity of this network is the presence of the convolutional layers, i.e. that contain one or more filters that are applied to the input. Thanks to them, this network is particularly effective to deal with image recognition. For building this neural network we use the same functions explained before and other ones, so here we explain only what is new with respect to the multilayer.

- **Conv2D()**: a class that creates the layer in which the convolutional operation occurs. The *input_shape* argument must be specified in the first layer of this kind. Here also the dimensions of the convolution window and the number of output filters must be specified with respectively the arguments *kernel_size* and *filters*.
- **MaxPooling2D()**: a class that creates the pooling layer. It has *pool_size* as an argument, i.e. the dimension of the window in which it takes the maximum value.
- **Flatten()**: in this kind of model this layer must be always inserted without any argument between the first part, in which the convolutional and pooling layers are put, and the second, in which there are the dense layers. As above, we mention this here so we will omit it later.
- **Batch Normalization()**: a class that adds a layer that normalize its input. It is used to help the model to generalize, so to reduce overfitting.

4 Multilayer Neural Networks

4.1 Base model

We start with a really basic model, structured as it follows: a multilayer neural network with two hidden *Dense* layers with respectively 512 and 256 nodes. The final *Dense* layer is always the same for every model in this report: one node with the *sigmoid* as activation function. Other parameters, like the *learning rate* and the *batch_size*, are left with their default values, respectively 0.001 and 32. The results of the accuracy and the loss of the model are shown in *Figure 1*.

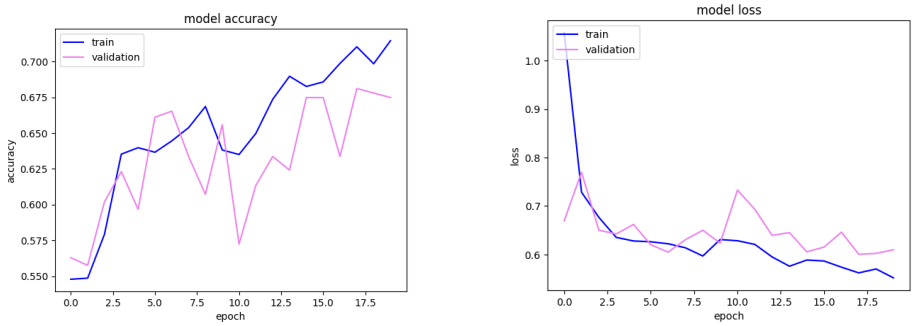


Figure 1: Evaluation of accuracy and loss for training and validation set of the basic multilayer model.

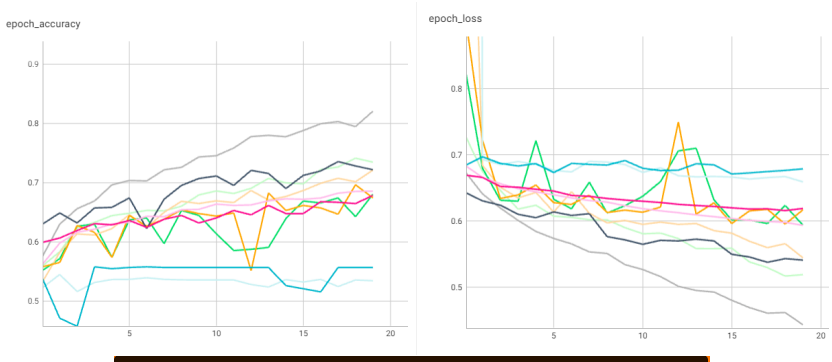
With this really essential model, we obtain an accuracy of **68.11%** on the validation set. The baseline model is the one that predicts always the most frequent class in the data, in this case Chihuahua. It has an accuracy of about **54%** in our data, so this simple model is already a step forward.
There were several parameters that can be modified to obtain better results: the *learning rate*, the *batch size*, the number of *Dense* layers and number of nodes for each layer.

4.2 Optimization of the learning rate

For the base model the *learning rate* is set to 0.001, the default value. Given its importance, we decide to start our analysis by exploring all the combinations for the following values:

- **learning rate**: 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}

In *Figure 2* all the trials are represented. It is immediately clear the improvement given by the right *learning rate*.



Run	Value	Step	Time	Relative
0.0001-learn_rate–32-batch_size\train	0.7351	19	10/10/23, 7:16 PM	13.18 sec
0.0001-learn_rate–32-batch_size\validation	0.6811	19	10/10/23, 7:16 PM	13.18 sec
0.001-learn_rate–32-batch_size\train	0.7216	19	10/10/23, 7:15 PM	12.8 sec
0.001-learn_rate–32-batch_size\validation	0.6748	19	10/10/23, 7:15 PM	12.8 sec
0.01-learn_rate–32-batch_size\train	0.5351	19	10/10/23, 7:15 PM	12.99 sec
0.01-learn_rate–32-batch_size\validation	0.5576	19	10/10/23, 7:15 PM	12.99 sec
1e-05-learn_rate–32-batch_size\train	0.8209	19	10/10/23, 7:16 PM	12.66 sec
1e-05-learn_rate–32-batch_size\validation	0.7223	19	10/10/23, 7:16 PM	12.66 sec
1e-06-learn_rate–32-batch_size\train	0.6862	19	10/10/23, 7:16 PM	12.98 sec
1e-06-learn_rate–32-batch_size\validation	0.679	19	10/10/23, 7:16 PM	12.98 sec

Figure 2: Evaluation of accuracy and loss for the different values of the learning rate.

The model with the *learning rate* equal to 10^{-5} has a peak around **73%**, so we pick this as the value for this hyperparameter from now on. We notice that by just choosing the right number our basic model increases by **5%**. This gives the idea of the importance of the *learning rate* while building a neural network.

4.3 Optimization of the batch size

In the previous models the *batch size* is set to 32, the default value. To optimize the model, we explore all the the following values:

- **batch size**: 32, 64, 128, 256, 512.

Figure 3: Evaluation of accuracy and loss for the different values of the batch size.

We can see a trade-off between validation accuracy and overfitting: as the batch size increases, the first one decreases but also the other one, so there is a positive and a negative effect simultaneously. We decide to keep the *batch_size* equal to 32, the one that maximizes the accuracy, as we think that we can solve the overfitting problem later, with other techniques.

4.4 Optimization of the number of nodes and layers

We start investigating the optimal number of nodes for each layer. In particular, as we could expect, we find that the networks with higher number of nodes in the first layer and progressively fewer in the following ones obtain better results. This is due to how the neural networks work.
We try models with a structure of three, four and five *Dense* layers. We perform a cycle with this possible numbers of nodes:

- **Number of nodes**: 64, 128, 256, 512, 1024

These values are chosen because they are powers of two and also because they are not so high or small, as we do not want to complicate or simplify too much the structure.

4.4.1 Structures with three *Dense* layers

In *Figure 4* is possible to observe the resulting best models for the analysis. Here we started with 3 *Dense* layers of different sizes.

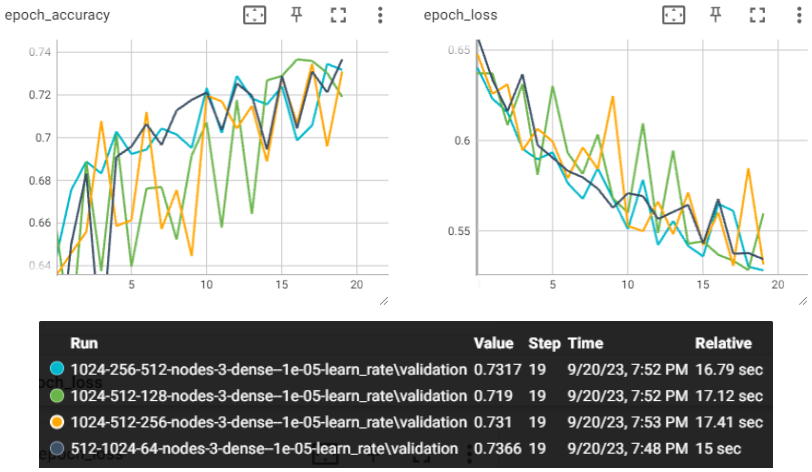


Figure 4: Evaluation of accuracy and loss of the best models with 3 *Dense* layers for different number of nodes.

As it is clear reading the legend, almost all of the best models starts with a *Dense* layer with 1024 nodes and also the second layer generally has a high number of nodes. The best structure shows an accuracy of **73.66%** for the validation set. As it is possible to see, all the models are very unstable, both with respect to the accuracy and the loss. Obviously this is not desirable in a neural network, as in this case it is impossible to be confident about the result that the classifier retrieves with the architecture we built.

4.4.2 Structures with four *Dense* layers

As we did before, we perform the cycle of the possible number of nodes on four *Dense* layers. Also this time the results lead to similar best structures, with many nodes at the beginning.

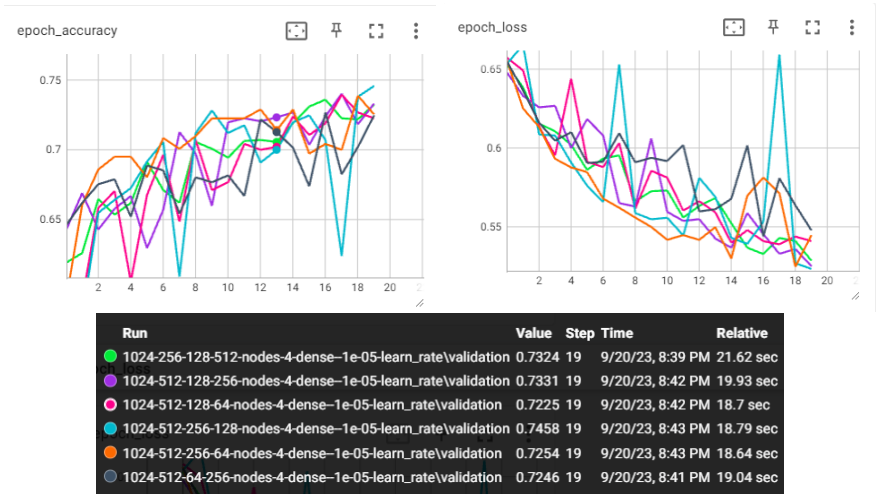
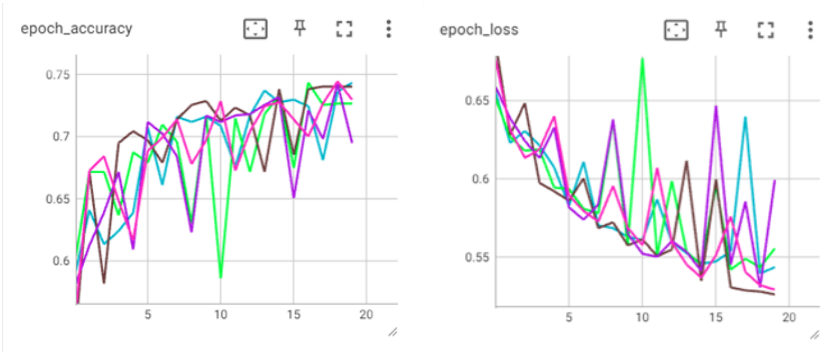


Figure 5: Evaluation of accuracy and loss of the best models with 4 *Dense* layers for different number of nodes.

With this cycle, we find the best model with the following structure: 1024, 512, 256 and 128 number of nodes per layer. Here we obtain an accuracy of **74.58%** for the validation set. With respect to the previous models, there is still instability, even if it seems to be less pronounced. However, the best model is the one that suffers the most from this problem, whereas for the others there is a slight improvement.

4.4.3 Structures with five *Dense* layers

The same procedure is reproduced with five hidden layers. Also here, in *Figure 6*, quite all the models start with the highest number of nodes and slightly decrease.



Run	Value	Step	Time	Relative
1024-512-256-128-64-nodes-5-dense–1e-05-learn_rate\validation	0.7434	19	9/20/23, 10:22 PM	21.37 sec
1024-512-64-128-256-nodes-5-dense–1e-05-learn_rate\validation	0.7265	19	9/20/23, 10:19 PM	20.73 sec
1024-512-64-256-128-nodes-5-dense–1e-05-learn_rate\validation	0.7402	19	9/20/23, 10:20 PM	20.95 sec
512-1024-256-128-64-nodes-5-dense–1e-05-learn_rate\validation	0.6948	19	9/20/23, 10:09 PM	20.02 sec
512-1024-256-64-128-nodes-5-dense–1e-05-learn_rate\validation	0.7297	19	9/20/23, 10:08 PM	19.23 sec

Figure 6: Evaluation of accuracy and loss for the different values of the learning rate and batch size.

Here, with the violet model (that has the following structure: 512, 1024, 256, 64), we obtain an accuracy of **74.34%** for the validation set. As the performance is more or less the same with 4 or 5 *Dense* layers, it seems unnecessary to try with 6.

4.5 Optimization of the best model

All these trials were done with the purpose of finding out the best model that fits our data. We started from the optimization of *learning rate* and *batch size* and then we moved to the best number of nodes for three, four or five hidden layers.
At the end of these evaluations we discover that there are two models that work better than the others: the blue one with four layers (with nodes: 1024, 512, 256, 128) and the pink one with the five layers (with nodes: 512, 1024, 256, 64, 128). Considering only the maximum value of their accuracy, the first one is a little bit better. However, it is the only one among all the networks with four layers that reach this value, while for the ones with five layers also the other models work pretty good, so we consider this violet one as our best. Since, generally, the accuracy of the model could increase as the number of the epochs increases, we reproduce this model and change the *epoch* parameter from 20 to 50. This allows the model to learn more from the training images. Increasing the number

of epochs, however, could also have bad responses. In fact, in particular with a data set not very large as ours (4733 images), it may overfit. The model could only learn the training images and struggle in predicting labels of new images.

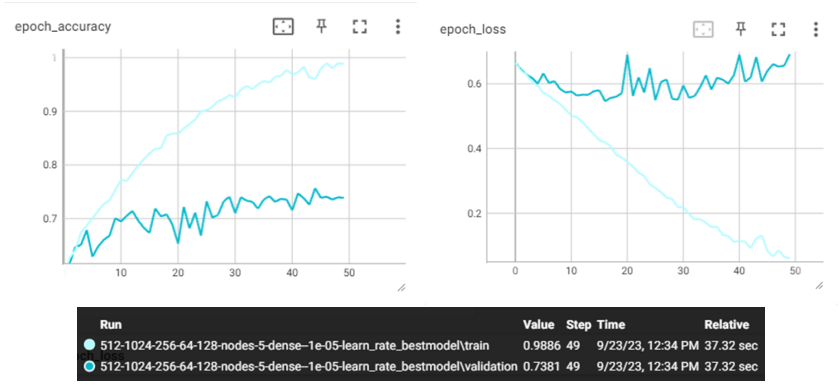


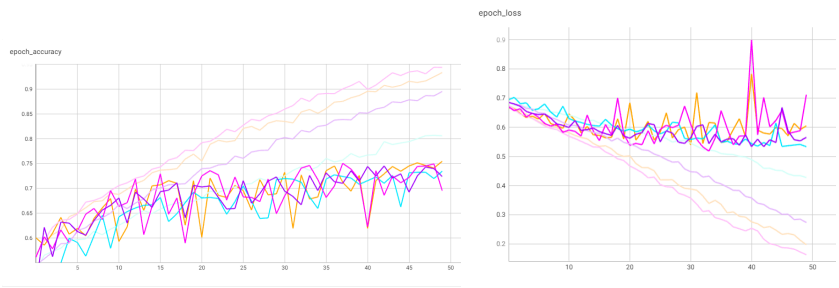
Figure 7: Evaluation of accuracy and loss of the best model with 5 *Dense* layer, using 50 epochs.

The problem of the overfitting stands out very clearly observing the *Figure 7*. The darker line related to the validation set reach a slightly better result over the previous version iterated only 20 times, with a maximum of **75.61%** of accuracy, but the light blue line for the training model goes really close to the **99%** of accuracy. This gap is the clear representation of the overfitting.

4.5.1 Solve the overfitting of the best model and conclusions

One possible solution to solve the overfitting problem is to add one or more *Dropout* layers. We try different approaches. When we set a high rate of dropping, we obtain a huge reduction of overfitting but the model also suffers from the elimination of many values in the training, reducing the accuracy. Vice versa, the risk is to not solve the problem.
Considering only the validation accuracy, the best model is the pink one with two *Dropout* layers with rate 0.1 (*Figure 8*), followed by the orange one with only one *Dropout* of 0.2, with a peak of **75%** of accuracy. Both this two, however, still present a strong overfitting problem, with a gap around 20%. The only one that reduces the training accuracy is the blue one, with one *Dropout* of 0.5. But this reduction in the data, due to the deletion of some nodes, worsen also the model.

It learns less and then predicts worse on the validation set: in fact it obtains only **73.5%**.



Run	Value	Step	Time	Relative
bestmodel–1-drop(0.2)\train	0.9337	49	9/27/23, 5:19 PM	38.67 sec
bestmodel–1-drop(0.2)\validation	0.755	49	9/27/23, 5:19 PM	38.67 sec
bestmodel–1-drop(0.5)\train	0.8064	49	9/27/23, 5:18 PM	38.14 sec
bestmodel–1-drop(0.5)\validation	0.735	49	9/27/23, 5:18 PM	38.14 sec
bestmodel–2-drop(0.1)\train	0.944	49	9/27/23, 5:21 PM	38.14 sec
bestmodel–2-drop(0.1)\validation	0.6959	49	9/27/23, 5:21 PM	38.14 sec
bestmodel–2-drop(0.2)\train	0.8954	49	9/27/23, 5:20 PM	39.73 sec
bestmodel–2-drop(0.2)\validation	0.7244	49	9/27/23, 5:20 PM	39.73 sec

Figure 8: Evaluation of accuracy and loss of the best model with 5 *Dense* layer, using 50 epochs and different *Dropout* layers.

To resume all the works done on multilayer neural network, the best structures result to have 5 hidden layers. The overfitting is still present and the maximum accuracy obtained on the validation set is around **75%**. As we said at the beginning, this is only the preliminary analysis because additional *Convolutional* layers will help the to fit our data set composed by images.

5 Convolutional Neural Networks

We now consider new architectures, with also *Convolutional* layers. From these models we expect a great improvement in results, because this type of layer is specially designed to fit the data sets composed by images.

5.1 Base model

As we did before, we start with a simple model. The first Convolutional Neural Network is structured in this way:

- Two *Convolutional* layers with respectively 64 and 32 *filters* but same *kernel_size* of 3x3
- MaxPooling2D layer in between with *pool_size* equal to 2x2
- *Dropout* layer with rate equal to 0.5
- *Dense* layer with 64 nodes.

Everything else is left with the default values. The results are plotted in *Figure 9*.

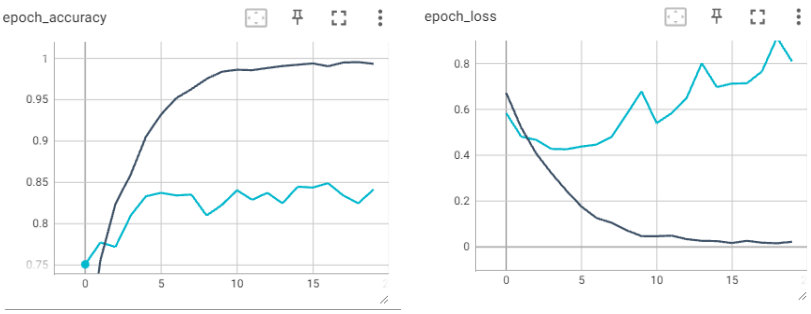


Figure 9: Evaluation of accuracy and loss for training (black) and validation (light blue) set for the basic Convolutional Neural Network.

The base model has a validation accuracy of almost **85%**. We can notice that there is a huge overfitting and that the validation accuracy after a certain epoch remain more or less stable. Still, as expected, the base convolutional model is way better than the best multilayer one. Anyway, there is clearly room for improvement.

5.2 Optimization of the learning rate and batch size

As in the multilayer case, we optimize the *batch_size* and the *learning rate*. This time we explore all the combinations for the following values:

- **learning rate**: 10^{-3}, 10^{-4}, 10^{-5}
- **batch size**: 32, 64

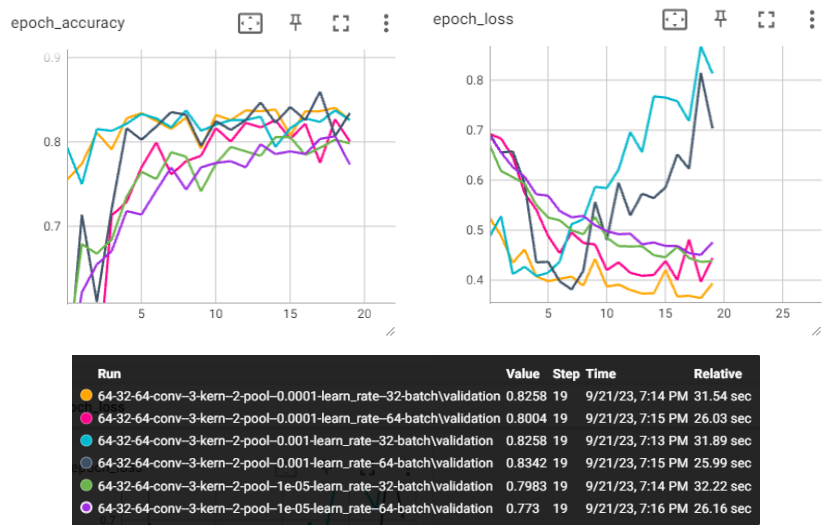


Figure 10: Evaluation of accuracy and loss for the different values of the learning rate and batch size.

The *Figure 10* resumes the results of the various models. The yellow one seems to be the best, in particular looking at the validation loss. This model has a *learning rate* of 10^{-4}, a *batch_size* of 32 and reaches around **82%** of validation accuracy, so strangely the accuracy is worse than before.

5.3 Optimization of the number of nodes and layers

Once we have the best values for the *batch_size* and the *learning rate*, we have to find the ones for the number of layers and the number of nodes. So these are the features to tune and the possible values:

- **number of *Dense* layers**: 1, 2
- **number of nodes per *Dense* layer**: 64, 128, 256, 512
- **number of *Convolutional* layers**: 2, 3
- **value of the filters**: 32, 64, 128

5.3.1 Structures with one *Dense* layer and two *Convolutional* layers

We start with one *Dense* layer, i.e. the base model, optimizing its number of nodes. The two *Convolutional* layers remain with respectively 64 and 32 *filters*.

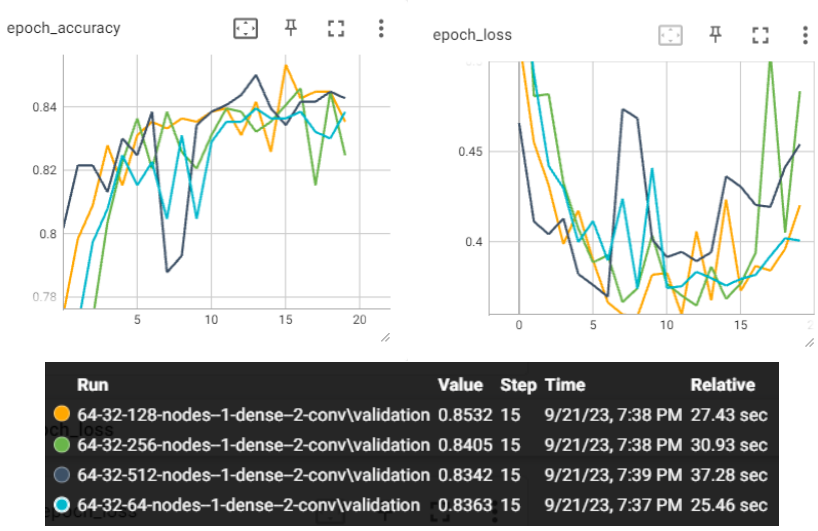


Figure 11: Evaluation of accuracy and loss for the different values of the node for the *Dense* layer.

The best model is the yellow one, with 128 nodes in the *Dense* layer and a validation accuracy of **85%**. It is interesting to notice that all the validation losses from a certain point start to grow. This probably happens because the training accuracy is almost **99%**, so there is clearly overfitting, and so going on with the train worsen the result on the validation metrics.

5.3.2 Structures with two *Dense* layer and two *Convolutional* layers

Let's see now if the model improves with another *Dense* layer.

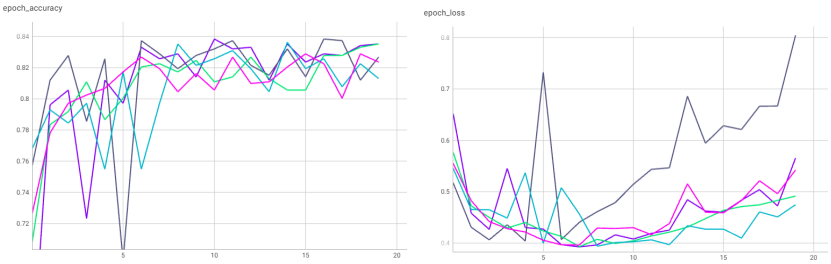


Figure 12: Evaluation of accuracy and loss for the different values of nodes for the two *Dense* layers.

Thanks to *Figure 12*, it can be said that it seems there is no improvement with respect to the previous case. This may happen because of the overfitting. In fact, both validation accuracy and loss from a certain epoch start to behave in a peculiar way: the first does not grow anymore, so the model stops learning, whereas the latter, as it was said before, even rises, so the overall performance is worse. As we can see from *Figure 13*, the training performance does not suffer from this problem.

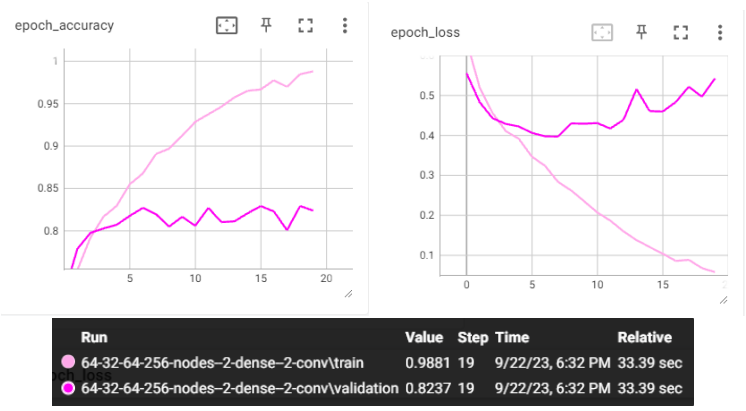
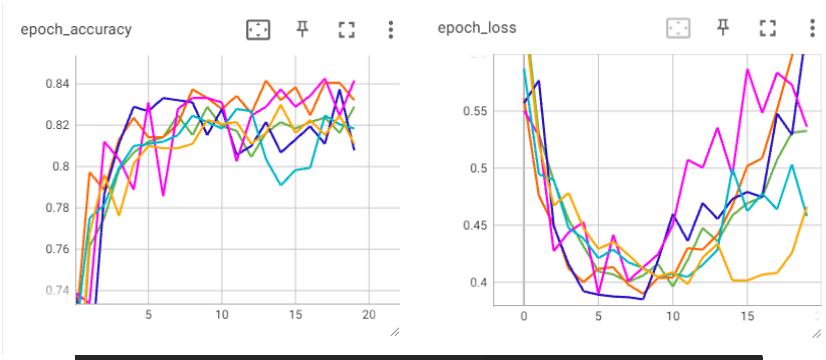


Figure 13: Comparison between training and validation performance of one of the last models.

5.3.3 Structures with two *Dense* layers, two *Convolutional* layers and an additional *Dropout* layer

Given that in the last try we have an additional *Dense* layer, we may try to add a *Dropout* layer in order to solve overfitting. This, in fact, could may be the cause for a worse result with respect to the models with only one *Dense* layer.



Run	Value	Step	Time	Relative
64-32-128-256-nodes–2-dense–2-conv_drop\validation	0.8289	19	9/22/23, 7:51 PM	35.41 sec
64-32-128-512-nodes–2-dense–2-conv_drop\validation	0.8321	19	9/22/23, 7:52 PM	35.24 sec
64-32-256-64-nodes–2-dense–2-conv_drop\validation	0.8078	19	9/22/23, 7:52 PM	40.42 sec
64-32-512-64-nodes–2-dense–2-conv_drop\validation	0.8416	19	9/22/23, 7:56 PM	48.24 sec
64-32-64-128-nodes–2-dense–2-conv_drop\validation	0.8184	19	9/22/23, 7:48 PM	32.92 sec
64-32-64-256-nodes–2-dense–2-conv_drop\validation	0.811	19	9/22/23, 7:49 PM	33.07 sec

Figure 14: Evaluation of accuracy and loss for the models with two *Dense* layers, two *Convolutional* layers and a *Dropout* layer.

As we could see in *Figure 14*, the additional *Dropout* layer do not solve the problem, as the accuracy still does not exceed **84%** and the loss grows. At this point we keep only one *Dense* layer and we try with more *Conv2D* layers.

5.3.4 Structures with one *Dense* layer and three *Convolutional* layers

Here we try with three *Convolutional*, specifically with the same parameters as the other ones.

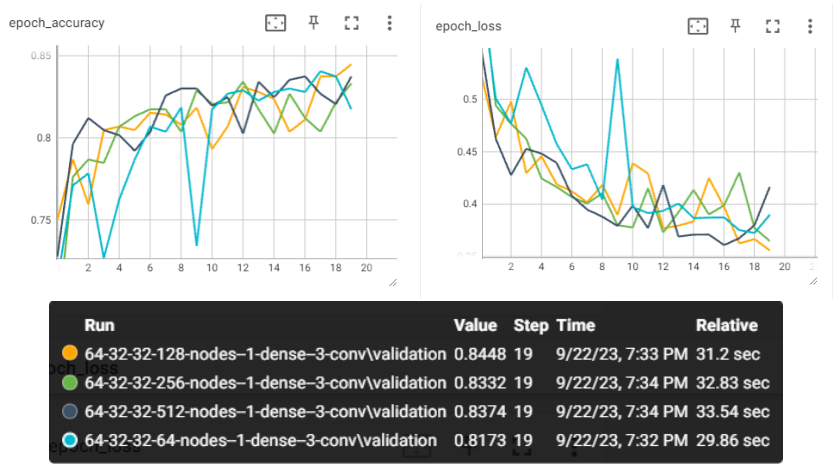
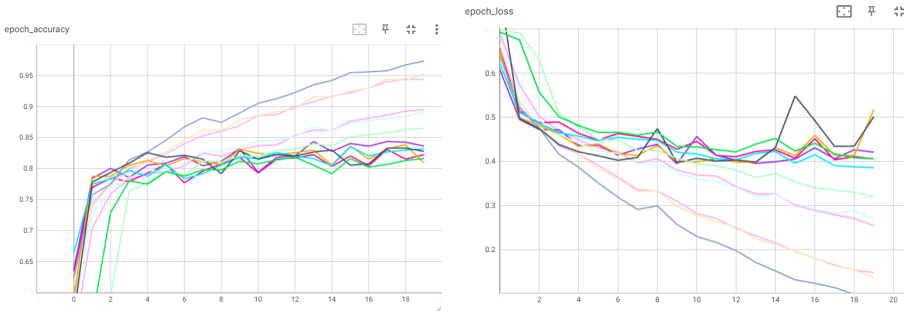


Figure 15: Evaluation of accuracy and loss for the models with one *Dense* and three *Convolutional* layers.

As shown in *Figure 15*, the accuracy is not improved but the loss is promising, as for the first time it does not grow from a certain point. This is a good signal, as it means that there is perhaps room for improving the validation accuracy. We can try to deal with overfitting to see if the model could become better. We start with changing the values for the filters.

5.3.5 Structures with one *Dense* layer, three *Convolutional* layers and different numbers of filters

We try to understand if the values of the filters impact the performance of the model. Here we see the results.



Run	Value	Step	Time	Relative
128-32-64-128-nodes–1-dense–3-conv\train	0.8949	19	10/21/23, 6:18 PM	1.583 min
128-32-64-128-nodes–1-dense–3-conv\validation	0.8226	19	10/21/23, 6:18 PM	1.583 min
128-64-32-128-nodes–1-dense–3-conv\train	0.9432	19	10/21/23, 6:20 PM	1.796 min
128-64-32-128-nodes–1-dense–3-conv\validation	0.8363	19	10/21/23, 6:20 PM	1.796 min
32-128-64-128-nodes–1-dense–3-conv\train	0.9525	19	10/21/23, 6:12 PM	1.175 min
32-128-64-128-nodes–1-dense–3-conv\validation	0.8089	19	10/21/23, 6:12 PM	1.175 min
32-64-128-128-nodes–1-dense–3-conv\train	0.8909	19	10/21/23, 6:11 PM	1.038 min
32-64-128-128-nodes–1-dense–3-conv\validation	0.8321	19	10/21/23, 6:11 PM	1.038 min
64-128-32-128-nodes–1-dense–3-conv\train	0.9733	19	10/21/23, 6:15 PM	1.456 min
64-128-32-128-nodes–1-dense–3-conv\validation	0.8279	19	10/21/23, 6:15 PM	1.456 min
64-32-128-128-nodes–1-dense–3-conv\train	0.865	19	10/21/23, 6:13 PM	1.165 min
64-32-128-128-nodes–1-dense–3-conv\validation	0.8163	19	10/21/23, 6:13 PM	1.165 min

Figure 16: Evaluation of accuracy and loss for the different values of the filters for the three *Convolutional* layers.

The structure 128-64-32 seems the best, as it reaches the best accuracy and the lowest overfitting.

5.3.6 Structures with one *Dense* layer, three *Convolutional* layers, increased *Dropout* rate, Batch Normalization and different values for *pool* and *kernel size*

The last model has two problems: the accuracy is not very high considering the base model, that is only the starting point, and the overfitting is still large. For the first issue, we can try to tune the values for the kernel and the pool size; for the second we can try Batch Normalization and an increased rate in the *Dropout* layer from 0.5 to 0.75. If this does not work, we may try with more convolutional layers.

To resume, here are the possible values:

- **kernel_size**: 2, 3, 4
- **pool_size**: 2, 3, 4

Figure 17: Evaluation of accuracy and loss for the different values of pool and kernel size.

Among all the possible models, the two in *Figure 17* are clearly the best. As it is possible to appreciate, they are very similar. We choose the blue one, that has *kernel_size* equal to 3, whereas the other is 4, and *pool_size* equal to 4, because it seems that the overfitting problem is slightly better.

5.3.7 Structures with four and five *Convolutional* layers

Anyway, as we stated before, we want to check if adding one or two *Conv2D* layers benefits the network. We do not report the plots as there are already many models in this analysis and because the addition is not profitable: one additional layer worsens the performance, whereas two additional layers retrieve more or less the same result, so we choose to keep the one with 3 *Convolutional* layers as it is simpler. We can say that we have the final model.

5.4 Final model

At this point, we can say that the following is the final model:

- **number of Conv2D layers**: 3, with 128-64-32 *filters*' architecture and *kernel_size* equal to 3
- **number of MaxPooling2D**: 2, with *pool_size* equal to 4
- **number of Dense layers**: 1, with 128 nodes

- **techniques to prevent overfitting**: one Dropout layer between the two parts with rate 0.75 and Batch Normalization
- **learning rate**: 0.0001
- **batch_size**: 32

We train the final model with 50 epoch and the Early Stopping, a technique that stops the training if the metric of interest (in our case, the validation loss) does not improve anymore. It requires a number of epochs after which it stops the model, that we set equal to 15. The utility of this tool is that it saves and retrieves the weights of the model of the last best epoch that it finds. Also, as stated at the beginning and already done for the Multilayer Neural Network, we decide to run the final model with an image size of 150, in order to exploit all the clarity of the images.

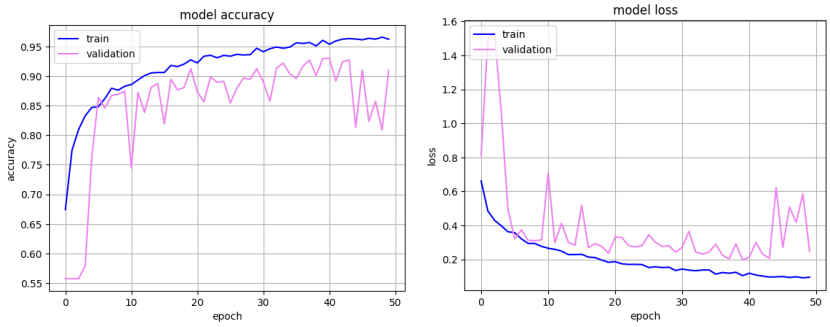


Figure 18: Evaluation of accuracy and loss for the final model.

The best performance is achieved in the 41st epoch with the following results: loss: **0.1186**, accuracy: **0.9538**, validation loss: **0.2131** and validation accuracy: **0.9303**. We evaluate the model with the test set and the results are:

- **test accuracy: 91.05%**
- **test loss: 0.2684**

Still, there is some overfitting that we consider as acceptable.

5.5 Confusion Matrix

In *Figure 19* we show the confusion matrix.
Given that *0* is the code for the Chihuahuas, *1* for the muffins, on the left we read the actual labels and on the bottom the predicted ones, we can say that our model predicts more often a Chihuahua with respect to a muffin, so it is more common for the network to predict the first when in reality the image shows the latter. Perhaps this happens because there are more images of Chihuahuas in the data set to analyse.

Figure 19: Confusion Matrix

5.6 5-fold Cross-Validation

As the last thing, we perform the cross-validation with 5 folds to compute the risk estimates of the final model. As required, we use the **zero-one loss**. Here we report the results:

Fold	Loss
Fold 1	0.082
Fold 2	0,076
Fold 3	0.082
Fold 4	0.068
Fold 5	0.069
Mean	**0.075**

Table 1: The result of the 5-fold Cross-Validation

We notice with delight that the losses are all included in a small interval, which signals that the model is pretty stable.

5.7 Conclusion

In this report we tried to thoroughly discuss all the steps that brought us from a very basic model, that could predict correctly more or less two third of the images, to one that classifies the data in the right way more than nine times out of ten. Anyway, we believe that with more computational resources, time, experience and data there is still room for improving in classifying correctly a muffin and a Chihuahua.

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