EXPERIMENTAL PROJECT ON NEURAL NETWORKS

Course of Machine Learning and Statistical Learning

MSc in Data Science for Economics

Author: Adriano Meligrana

Index

1. Introduction

The aim of this project is to experiment with different neural network architectures to try to solve a binary classification task on images of Chihuahuas and muffins. The dataset[1] used to train and evaluate the networks' performance is composed by approximately 6000 unique and already labeled images, scraped from Google Images. Three types of architectures of increasing complexity have been chosen to be tested to tackle the task:

- Regularized Multilayered Perceptrons;
- Regularized Convolutional Neural Networks;
- Residual Neural Networks with transfer learning.

The work is divided in two main sections, the first describes the general methodology used to perform the experiments, along with a brief description of the different components of the learning process. The other presents the results achieved by the tested models.

[1] The dataset comes from Kaggle, and it is available at the following link: `https://www.kaggle.com/datasets/samuelcortinhas/muffin-vs-chihuahua-image-classification`

2. Methodology

This section aims to explain the steps that were performed during the realization of the project, with a particular emphasis to the methodological issues encountered.

2.1 Data transformations and data loading

Some preliminary transformations on the images were used to scale them at a suitable size. In particular it was chosen a size of 128x128 because at smaller scales some images were nearly impossible to associate with a specific label.

The color channel of the images was preserved because empirically it seemed associated with a higher accuracy but no extensive testing was performed, at the same time the pretrained ResNet models were trained with colored images, so using a gray-scale transformation (and so changing also the first layer to make it work with it) would have worsened the performance of those models because the weights weren't adjusted to such an input. Apart from that, the transferred residual neural networks used also some slightly more involved transformations, namely, cropping and normalization of the image pixels; applying them was necessary for optimal transfer because those networks were pretrained with images transformed in this way. It is important to notice that these last transformations are independent to the training, validation and test set, so that no information is leaked from one set to the other. Besides, some data augmentation techniques have been examined during the first exploratory phase of the analysis, however, since the models didn't seem to perform substantially better with an augmented dataset, these techniques weren't eventually used during the hyper-parameter tuning phase, given the increased running time this would have implied.

The data loading process involved the decision on the batch size. This should be carefully chosen because it involves a trade-off between the accuracy of the gradient estimate and the required memory as well as the convergence speed during training. The more the batch size increases, the more the gradient computation should be precise, because it takes into consideration a bigger number of examples. However, at the same time, you need more memory because you need to store a bigger number of examples; in addition, since the number of updates of the weights gets higher with a smaller batch, the training should converge faster. A mini-batch of 32 images was chosen to accommodate the memory limits of the GPU in use, a Nvidia GTX 1650 with 4 GB of memory. At higher batch sizes it wouldn't have been possible to train all the models tested. Another motivation for this choice was to speed up training by exploiting vectorization, so to be able to fit the models in a shorter amount of time.

2.2 Training

All three types of neural networks tested are feed-forward, meaning that the direction of the connection between nodes is unidirectional - from the nodes of one layer to the next without any loop or cycles. This has the important implication that the gradient estimation procedure used during training to optimize the network's weights and biases is the same, independently from the type of the neural network in question: backpropagation. Therefore, during training, the standard optimization algorithm was applied: a forward pass to compute the activation values of the output layer for each mini batch in order to obtain the value of the loss function at that time, and then a backward pass to estimate the optimal changes in the weights and biases of the network so to reduce the loss function. This is done informally by computing the optimal changes of the weights and biases between the last layer to the previous, and then propagating back recursively those values to the previous pair of layers, this is why the gradient estimation procedure is called backpropagation. Finally, after calculating those changes (or, more formally, the partial derivatives) of the loss function in respect to the weights and biases of the networks, a variant of the stochastic gradient descent algorithm is used to update the parameters in order to induce a reduction in the loss function. In its simplest form this is done by subtracting the gradient vector multiplied by a factor (the learning rate) from the parameters vector; this last step is due to the classic approximation of differentiable functions in a neighborhood of a point by a first order taylor polynomial. The pseudocode in this image could clarify the procedure a bit more:

```
Algorithm Backpropagation learning algorithm
Input:
    A set of training examples $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$
    A multilayer network with $L$ layers, weights $w_{ij}^l$, and activation function $f$
    Loss function $J(y, o)$
    Learning rate $0 < \alpha < 1$
    Number of epochs $epochs$

 1: for each weight $w_{ij}^l$ in the network do
 2:     $w_{ij}^l \leftarrow$ a small random number
 3: for $i = 1$ to $epochs$ do
 4:     for each training example $(\mathbf{x}, y) \in D$ do
 5:         /* Propagate the inputs forward to compute the outputs */
 6:         for each neuron $i$ in the input layer do
 7:             $a_i^0 \leftarrow x_i$
 8:         for $l = 2$ to $L$ do
 9:             for each neuron $i$ in layer $l$ do
10:                 $z_i^l \leftarrow \sum_j w_{ij}^l a_j^{l-1}$
11:                 $a_i^l \leftarrow f(z_i^l)$
12:         /* Propagate deltas backward from the output layer to the input layer */
13:         for each neuron $i$ in the output layer do
14:             $\delta_i^L \leftarrow \frac{\partial J(y_i, o_i)}{\partial o_i} f'(z_i^L)$
15:         for $l = L - 1$ to 1 do
16:             for each neuron $i$ in layer $l$ do
17:                 $\delta_i^l \leftarrow f'(z_i^l) \sum_j \left(w_{ji}^{l+1} \delta_j^{l+1}\right)$
18:         /* Update the weights using the deltas */
19:         for each weight $w_{ij}^l$ in the network do
20:             $w_{ij}^l \leftarrow w_{ij}^l - \alpha \delta_i^l a_j^{l-1}$
```

Figure 1: Pseudocode for the backpropagation algorithm. The last step is the stochastic gradient descent update. Source: Towards Data Science, backpropagation: Step-By-Step Derivation.

Some choices made during the training phase of the networks tested still need to be explained: the loss function used for the binary classification task at hand was a 2-class cross entropy loss, this is a common choice when training neural networks on binary classification problems in PyTorch. Also, apart from the standard SGD optimizer, we tested also the

Adam optimizer during hyper-parameters tuning because in the literature it is commonly found to reach higher accuracy. One of the reasons is that it adjusts at each gradient computation the learning rates for each parameter, so that some parameter receive smaller/larger updates than with standard SGD. These adjustments can lead to a higher and/or a faster decrease in the loss function.

2.3 Architectures

As stated in the introduction, three main types of neural networks were tested: regularized multilayered perceptrons, regularized convolutional neural networks and residual neural networks with transfer learning.

In this section only the most complex version tested for each architecture will be (briefly) described; the combination of parameters and "sub-architectures" (networks with only some components of the most complex version) tested during hyper-parameter tuning will be reported in the next section.

Multilayered perceptrons are one of the simplest kind of deep neural networks, where nodes in a layer have incoming edges only from the previous one, without any convolutional layer. The most complex version of this architecture that was tested in the performed experiments was a network with two hidden layers using a ReLu activation function and dropout regularization to prevent overfitting. Some remarks on these decisions follow: the ReLu activation function has the advantage to be faster to compute than a Sigmoid activation function because it only involves applying a $\max(0, a)$ where a is the activation value of a node (before the activation function has been applied), rather than exponential transformations. Regarding dropout regularization instead, it prevents overfitting because it randomly drop nodes and their edges, so, intuitively, during training each node has to adapt to different "environments", therefore helping generalization.

Convolutional Neural Networks, instead, involve some convolutional layers. This kind of layers exploit a peculiar property of many problems: nodes representing one unit of information (e.g. a pixel in an image) are mostly influenced by neighbor units. This translates to a neural network where a node is only connected to a small region of the layer before it, which reduces the number of parameters to be learned, speeding up convergence and avoiding overfitting. A comparison between a fully connected layer and a convolution layer is illustrated in the following figure:

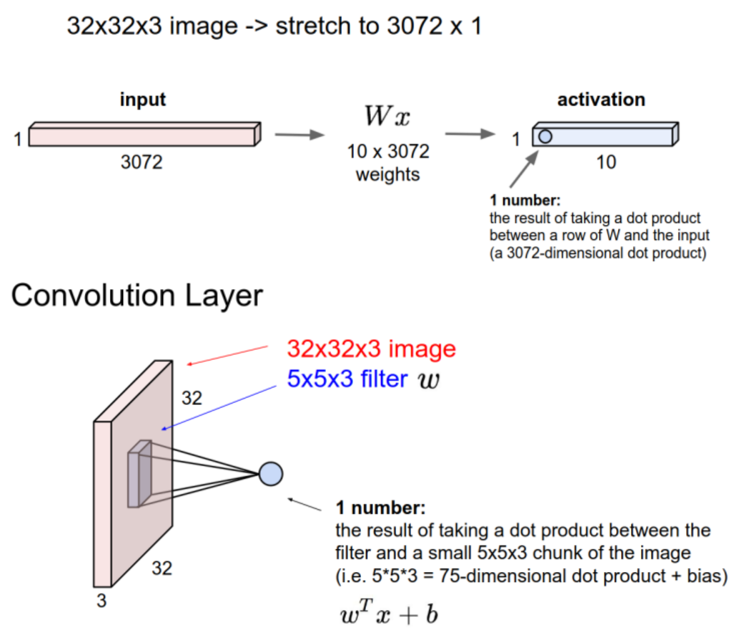

32x32x3 image -> stretch to 3072 x 1
input
1
3072
Wx
10 x 3072
weights
activation
1
10
1 number:
the result of taking a dot product between a row of W and the input (a 3072-dimensional dot product)
Convolution Layer
32x32x3 image
5x5x3 filter w
32
32
3
1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)
$w^T x + b$

Figure 2: Connections in a convolution layer in respect to a fully connected layer. As can be noted, the dot product involves only a small number of nodes in a chunk, differently from a fully connected layers which requires applying it to the complete previous layer. Source: notes of the course "Deep Learning for Computer Vision", Fei-Fei Li et al.

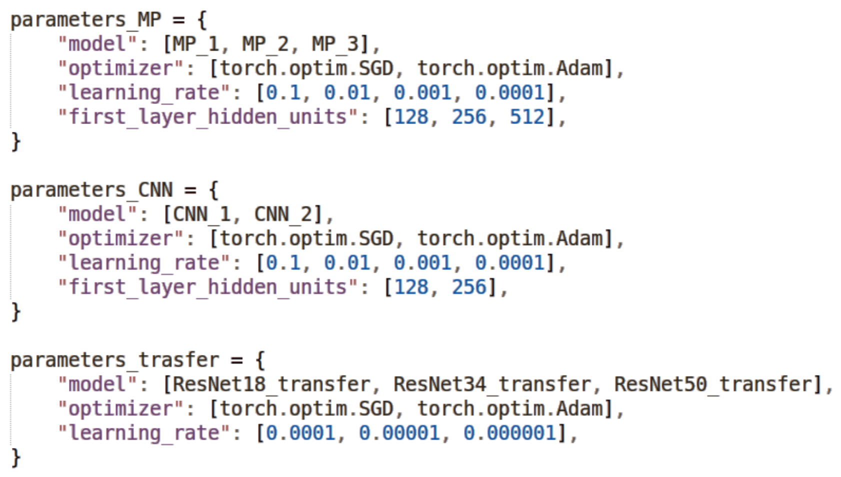
Max pooling and batch normalization have also been used in some convolution neural networks tested. Max pooling should help the robustness of the feature extraction process by down-sampling the feature map of the previous convolution layer, a lower resolution version of the feature can still maintain the structural property and make at the same time the feature map be less sensitive to slight variations. Batch normalization, instead, standardizes the inputs of hidden layers: this way the loss optimization landscape becomes smoother because of this helping the stability of the learning process. The bottom line for these choices is that these kinds of transformations are recognized to be useful when training convolutional neural networks, and so their introduction has been tested during the hyper-parameter tuning of this architecture.

Finally, residual neural network are a kind of deep neural network, where layers are aggregated in blocks called "residual blocks" such that each single block passes to the next the sum between its modification of the input values, and the input values themselves. In general, this helps in solving the problem of having to learn the identity function for a layer, which is helpful because a more shallow network could be more accurate for the problem at hand. For this reason, in a residual neural network, each block, instead of performing a transformation $F(x)$ where x is the input, performs $F(x) + x$. This can help a deep neural network reaching higher accuracy because gradients can follow a shorter path in the first phases of training during the backward pass reducing the actual depth of the network. At the same time, each block has a simpler task to learn because it doesn't need to convey all the information to the next block immediately and can just represent a slight variation of the identity function in the first phases of the training process. The tested residual neural networks in the performed experiments were ResNet18, ResNet34 and ResNet50 with the weights and biases already initialized by their training on ImageNet. To achieve the transfer to the required task, the last linear layer with 1000 nodes has been substituted with a layer with just two nodes. In addition, an important sanity check has been performed to exclude the possibility that ResNet could be able to solve the classification task without any fine tuning, since ResNet already has in its category many dog species including Chihuahuas, which would make the training useless. This was excluded by performing two simple experiments: in the first experiment, we used the pretrained ResNet50 with no further training to produce a prediction between the 1000 categories for the images in the test set, and assigned a “Chihuahua” label when the initial label produced was that of a dog species, otherwise we assigned a “Muffin” label; in the second one we did the same but with any animal instead of only dog species for the ”Chihuahua” reassignment: the network already achieved a good accuracy on the task, 91.5% and 92% of correctly categorized images respectively; however as it will be shown in the results section the fine tuning achieved a perfect score of 100%, so continuing to train the network was needed to reach it.

2.4 Hyper-parameters tuning and Cross Validation

The three architectures described in the previous section have been tested with different hyper-parameters to search for better models. While the dataset provided was already divided in a train set and a test set, an additional splitting of the train set was needed to tune the hyper-parameters of the models: a validation set to evaluate the accuracy of different hyper-parameters was used. Here, it was applied a 5-fold cross validation procedure. This is important because if, instead, the hyper-parameters were tuned on the test set itself, the out-of-sample estimate of the accuracy of the model would be probably slightly inflated because the hyper-parameters will be better suited to the folds than to an arbitrary set of examples. While the 5-fold cross validated estimate for the test error was useful to assess which model to use, the final test error was calculated by training the best model, found by comparing those estimates on the entire training set and computing the loss on the test set. A further complication of this procedure was introduced in order to decrease the tuning time: an early-stopper to end the training when in the last 10 epochs no loss improvement on the validation set was made. This is something that needs some care for two reasons: when the early stopper intervenes on a fold, the result from that epoch till the last one won’t be available and so the last part of the validation curve needs to be imputed; secondly, since one of the hyper-parameters to tune is the number of epochs, we need to choose a criterion to consider the optimal number of epochs to use on the test set, also considering the early stopping behavior. Both these issues can be solved coherently by assuming that the last part of the training, cut because of no improvements in the last epochs, will continue like so until the end of the training. This means that we can use the last available validation error for all subsequent epochs, and use this when estimating the 5-fold cross validation estimate. Also, the optimal number of epochs to use on the test set will be the value of epochs which maximizes the validation accuracy on the imputed curve which has the higher accuracy in respect to all other hyper-parameters.

Apart from the number of epochs, some ”sub-architectural” choices as well as some parameters of the learning process were tuned. The following code snapshot contains the values which were varied for each parameter:



```
parameters_MP = {
    "model": [MP_1, MP_2, MP_3],
    "optimizer": [torch.optim.SGD, torch.optim.Adam],
    "learning_rate": [0.1, 0.01, 0.001, 0.0001],
    "first_layer_hidden_units": [128, 256, 512],
}

parameters_CNN = {
    "model": [CNN_1, CNN_2],
    "optimizer": [torch.optim.SGD, torch.optim.Adam],
    "learning_rate": [0.1, 0.01, 0.001, 0.0001],
    "first_layer_hidden_units": [128, 256],
}

parameters_trasfer = {
    "model": [ResNet18_transfer, ResNet34_transfer, ResNet50_transfer],
    "optimizer": [torch.optim.SGD, torch.optim.Adam],
    "learning_rate": [0.0001, 0.00001, 0.000001],
}
```

Figure 3: The image exemplifies the parameters values which were scanned during the hyper-parameters tuning. Notice that the "model" parameter represents different models belonging to the same overarching class. For more details on the architectural choices refer to the previous section and to the code attached to the report. Source: own analysis.

As you can see, the learning rate was varied in different ranges; the reason is that some more heuristic searches demonstrated that those could be interesting ranges to test for that kind of architecture. Finally, notice that on some architectures it was made a richer hyper-parameters search; this was decided in relation to the training time of the models in question.

3. Results

In this section the results of the hyper-parameter tuning phase and the final test errors for the best models of each architecture[2] examined will be described. Since the overall best model achieved a 100% accuracy on the test set, we will look at the normalized output of the model through the softmax function as an uncertainty measure of the model in respect to the classification task. While this interpretation is problematic when the model probabilities are not well calibrated, as can happen with deep neural network, and should be taken into account by using Platt scaling or other techniques as described in Chuan Guo et al. (2017), we will show that this doesn't pose concerns in our case. Finally, a visual inspection of images classified with more uncertainty will be proposed to try to understand where the model performed more weakly.

The next figure shows the multilayered perceptrons validation error for the models throughout the training, computed as described in section 2.4. The color of the line representing each model error is colored differently in relation to the architecture, namely the "model" attribute inside the parameters dictionary of figure 4. This means that in this case three colors are used each representing a different architecture. See the code attached to the report for more insights into the different choices. The darkness of the color used increases with the complexity of the architecture; this can help to see if a model performs more consistently in relation to a change of the other hyper-parameters. In this case, we can see there is a lot of variation in all classes, interestingly the least complex model which entails only a hidden layer with the ReLu activation function performed more consistently than the other versions and arrived only at a slightly worse result than the best model which instead used also some drop-outs. The best model in this setting achieved a cross validated accuracy of 77.41% at epoch 18 as can be seen in the figure:

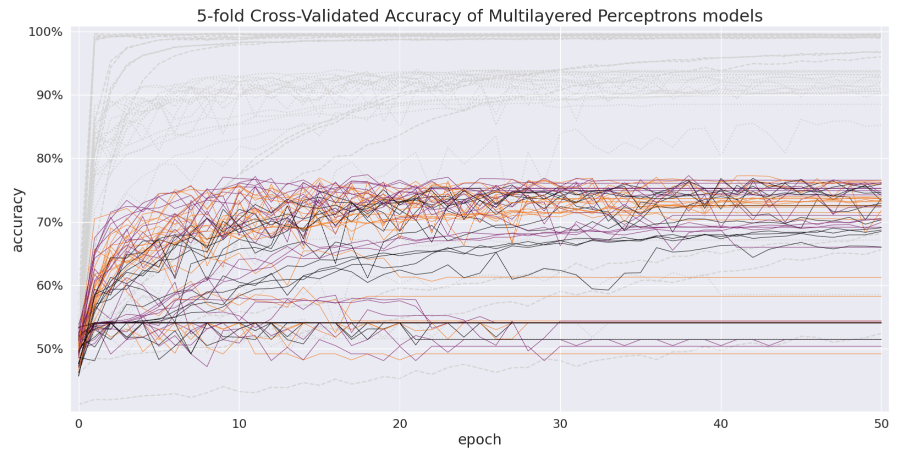


Figure 4: Results for the cross-validated validation error of the multilayered perceptrons models trained during hyper-parameter tuning. Source: own analysis.

The best model in this category was retrained on the all training set to compute the final

[2] the final models are available in the results folder of the attached code

test error, using 18 epochs as per above, it performed slightly better, with an accuracy of 77.83%.

The next figure describes the same results for the convolutional neural network architecture. Here the best model achieved a validation accuracy of 94.06% at epoch 30. The final test accuracy was slightly higher: 94.53%. Also here there is a lot of variation in the performance of the models, interestingly some combination of parameters seems to flatten the learning process entirely: for example just changing the optimizer from SGD to Adam in one of the experiments led to a degradation in performance from nearly 93% to just 53%. This validates the importance of making some hyper-parameter tuning to avoid sub-optimal solutions.

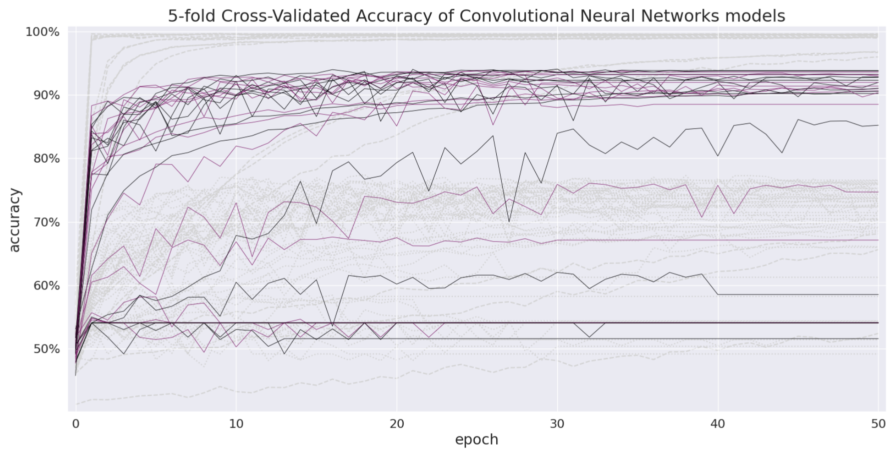


Figure 5: Results for the cross-validated validation error of the convolutional neural network models trained during hyper-parameter tuning. Source: own analysis.

Finally, the residual neural network architecture was much more consistent during training, because most of the variations in the validation accuracy was due to the choice in the learning rate as hinted from the fact that the shape of the validation curves is less erratic. The best model is a transferred ResNet-50 trained with the Adam optimizer and a learning rate of 0.0001. It achieved a 99.66% cross-validated accuracy at epoch 9 and a striking 100% test accuracy when retrained and evaluated on the test set.

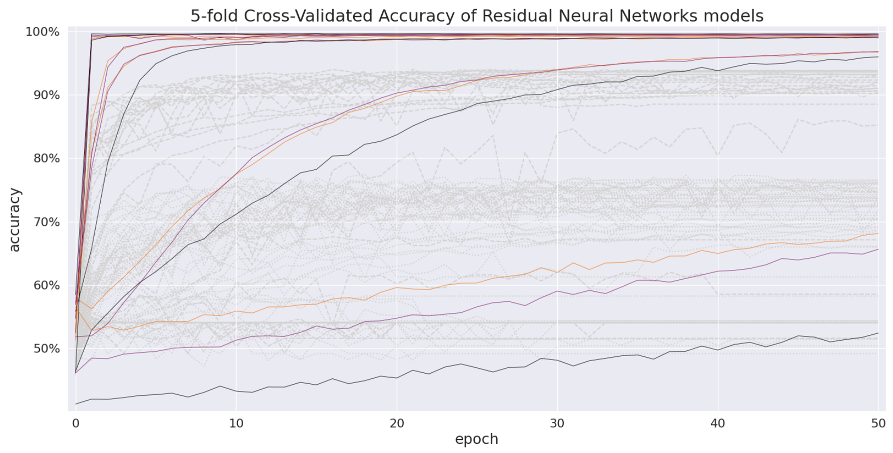


Figure 6: Results for the cross-validated validation error of the residual neural network models trained during hyper-parameter tuning. Source: own analysis.

The fact that all best models achieved a better accuracy on the test set than on the validation sets needs some explanation. One could be that, even though the hyper-parameters used should have delivered a less optimistic estimate for the test error because used to maximize the average performance only on the validation sets, it is also the case that the full training set could have conveyed some more information to the model, so much so that the final error was reduced. Also random variations could have contributed more on the single test set than on the 5 validation sets employed.

Since the best model achieved a 100% test accuracy, it was then tried to interpret a softmax transformation of the output as a measure to quantify the uncertainty of the model on the classification on the images and to identify the ones which the model found harder to correctly classify:

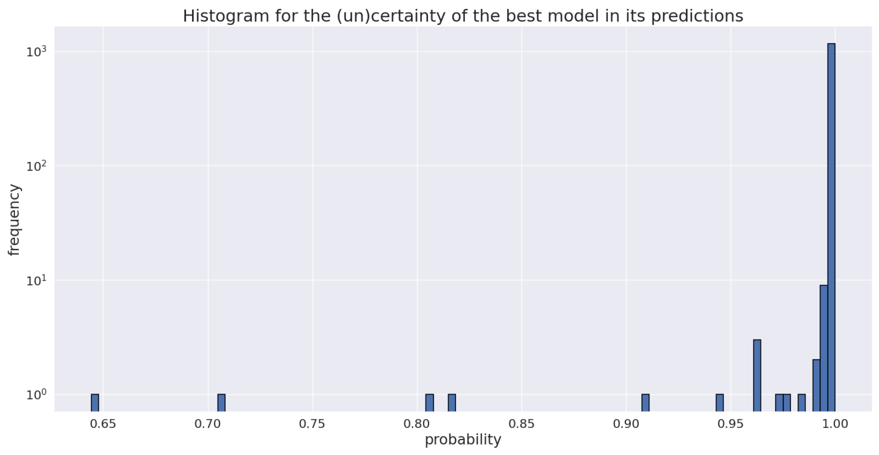


Figure 7: Histogram which shows the assigned probability of the prediction produced by the model. Source: own analysis.

This interpretation is valid in this case because the probabilities given by the model are already well calibrated, which intuitively means that the model is not too certain/uncertain about its predictions when it is respectively often incorrect/correct about them. This is so because there are only very few images where the model is not certain about its predictions: almost all predicted probabilities are near 1 which means that the Expected Calibration Error (ECE) as defined in Chuan Guo et al. (2017) is very low, indeed, it is less than 0.001.

Finally, let's look at the 4 images where the assigned probability is less than 0.9, to evaluate if they seem difficult to classify as the model predicts:

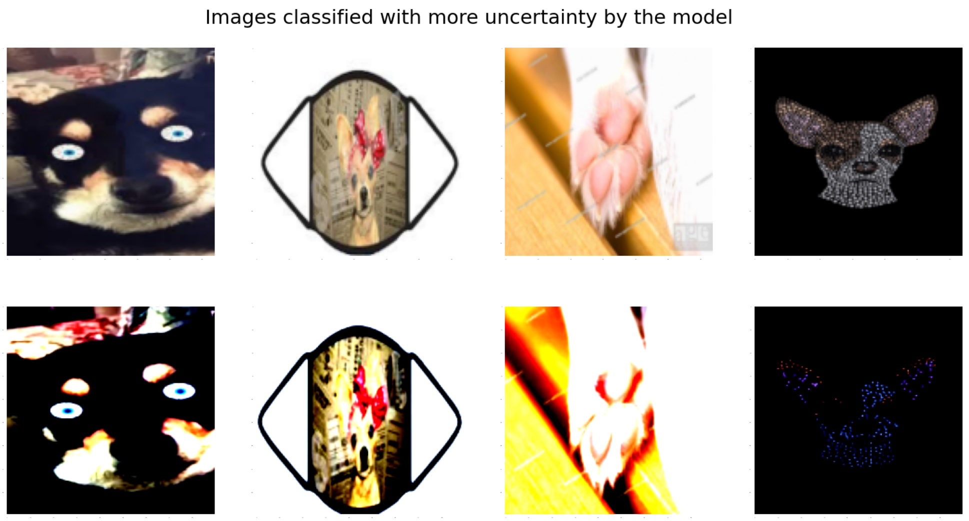


Figure 8: The four images which the model finds hardest to classify, in increasing order of uncertainty. On the first row there are the images before any ResNet required transformation is applied, in the second row there are the same images after those transformations.

By inspecting the images, it seems reasonable that the model is confused by them. In particular, since the model uses for inference the images in the second row it becomes clear why the image at the right side is difficult to classify correctly by the model, and it gives only a 65% chance of correct classification. The second image doesn't appear to be that problematic, but it could be noted that the algorithm could have found difficult to separate the background from the subject because similar in color.

4. Conclusions

In this report we described the methodology employed and the results achieved by different machine learning models during a binary classification task on images. The best model reached a validation accuracy of 99.66% and a test accuracy of 100%. Moreover, the model probabilities appear to be well-calibrated, besides that, a visual inspection of the test images the model finds hardest to classify seems to give more trust in its ability to work correctly with out-of-sample images. At the same time, since the images are not always about ”Muffin” and ”Chihuahuas” in a strict sense, because many of those contains related but different subjects, a maybe interesting extension of this analysis could be to improve the quality of the data by reclassifying the images in ”Muffin”, ”Chihuahuas” or ”Unknown” and train a neural network to classify the images in this setting.

5. Bibliography

1. Notes of the course Machine Learning and Statistical Learning, Nicolò Cesa-Bianchi et al., University of Milan, 2023.

2. Notes of the course Deep Learning for Computer Vision, Fei-Fei Li et al., Stanford University, 2017.

3. On the calibration of Modern Neural Networks, Proceedings of the 34th International Conference on Machine Learning, Chuan Guo, Geoff Pleiss, Yu Sun, Kilian Q. Weinberger, 2017.