UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

MASTER DEGREE IN COMPUTER SCIENCE

FINAL ASSIGNMENT FOR THE COURSE STATISTICAL METHODS FOR MACHINE LEARNING

ACADEMIC YEAR 2022/2023

Image classification with convolutional neural networks

Author:
Giulia Pais

Student Number:
02014A

December 20, 2022

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

Contents

1 Motivation and results

In the context of this project we tried to obtain a suitable image classifier, having an eye of regard for both the trade-off between resource availability and performance of the final model, and the general methodology of the approach.

We followed a series of logical steps to reach our goal:

- We firstly inspected the data set we were provided with, which was composed of images of cats and dogs, to extrapolate relevant metadata
- We identified some mislabeled items and proceeded with their removal
- We then proceeded to identify a suitable baseline convolutional network model through a series of subsequent trials
- The baseline was then used for hyperparameter tuning
- The average performance of the model adjusted with the best parameters found was thus evaluated with 5-fold cross validation
- Finally, we used the best out of the 5 models in the cross-validation phase to produce predictions for 10 images which were not included in the original data set

As the very first step in our approach, we examined the given data: we were interested in particular in obtaining descriptive statistics (max, min, average, etc.) of width, height and aspect ratio values as an overview on the data we were working with. In collecting this information, we incurred in some unreadable files and mislabeled objects that were therefore excluded from the data set entirely, as described more in detail in Section 2.3, in the hope to decrease potential noise and confounding factors that would negatively impact the model performance. We observed that a significant portion of the images had width and height values in the range of 500 px, averaging on the whole data set at 350-400 px. Even if choosing 500x500 px as dimensions for the inputs of our model could seem an appropriate choice, we assumed that this would prove too challenging of a task for our machines with limited resources. Moreover, famous convolutional neural network architectures such as VGG-16 (Zhang et al. [2015]) and AlexNet (Krizhevsky et al. [2017]) use smaller dimensions, more precisely 224x224 px, so we deemed these values more appropriate for our purpose. Unfortunately also these values produced an excessively high demand of computational resources, so we opted for further lowering the input dimensions to 128x128 px.

Furthermore, it was important to assess if our data set was balanced or not, meaning we were interested to see if both classes contained approximately the same amount of examples. Indeed, having an imbalanced data set is often something that needs to be addressed appropriately before training since it can compromise the model performance: generally, if the data is skewed towards one class, a re-sampling strategy is applied to obtain the same amount of examples for each class, sacrificing however the data set size. If re-sampling is not applicable, appropriate metrics should be

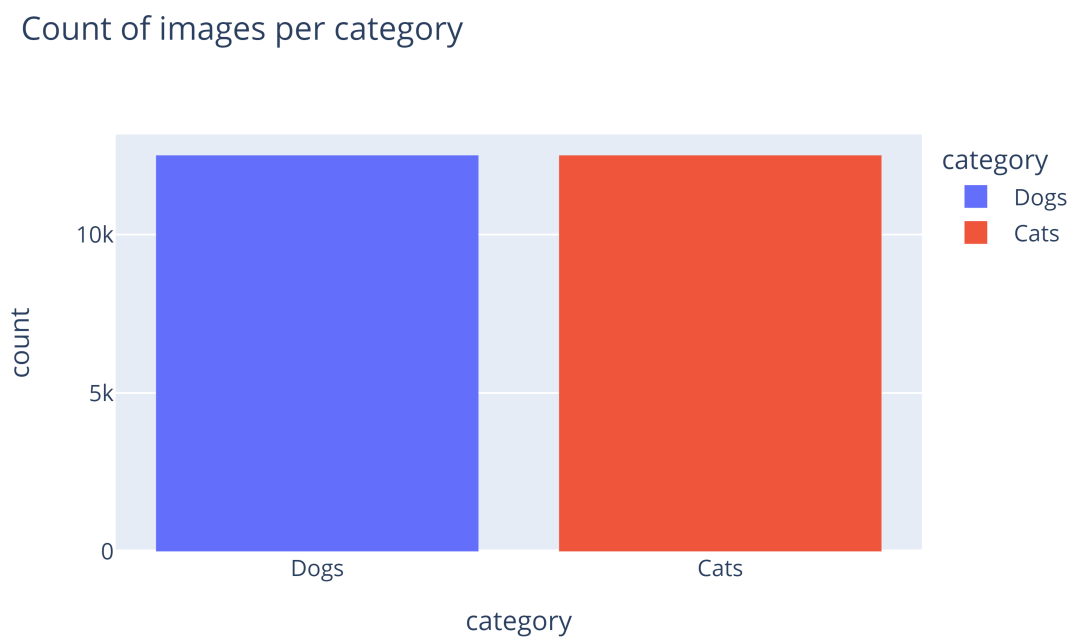

Figure 1: *Counts of examples per category - the data set doesn't show any sign of skewness*

monitored instead of accuracy, such as recall and precision. Luckily, we found that our data, after the previous data-cleaning steps, was perfectly balanced as shown in **Figure 1**, so no further action was required.

We then needed a first building block to start our research for a good classifier model: we began building a very basic convolutional network architecture, commonly used in easy classification tasks that can be found also in Tensorflow 2 documentation and tutorials. After training however, it was clear the model was underfitting the data, since both training and validation accuracy remained steadily around 50%. We then improved the architecture in 2 subsequent steps and we were able to reach a peak accuracy of 94% (**Figure 2**), the process is explained in greater detail in **Section 2.4**.

In the hope of further improving the performances of our baseline model, we decided to perform hyper-parameter tuning to assess the best choices of values for at least some of the parameters. We initially designed this step with a nested cross-validation approach, since it is the formally correct way to estimate both parameter values and model performance avoiding the bias of a random split of the data set in train set and test set. However, this approach proved infeasible in terms of time required: with the limitation of only

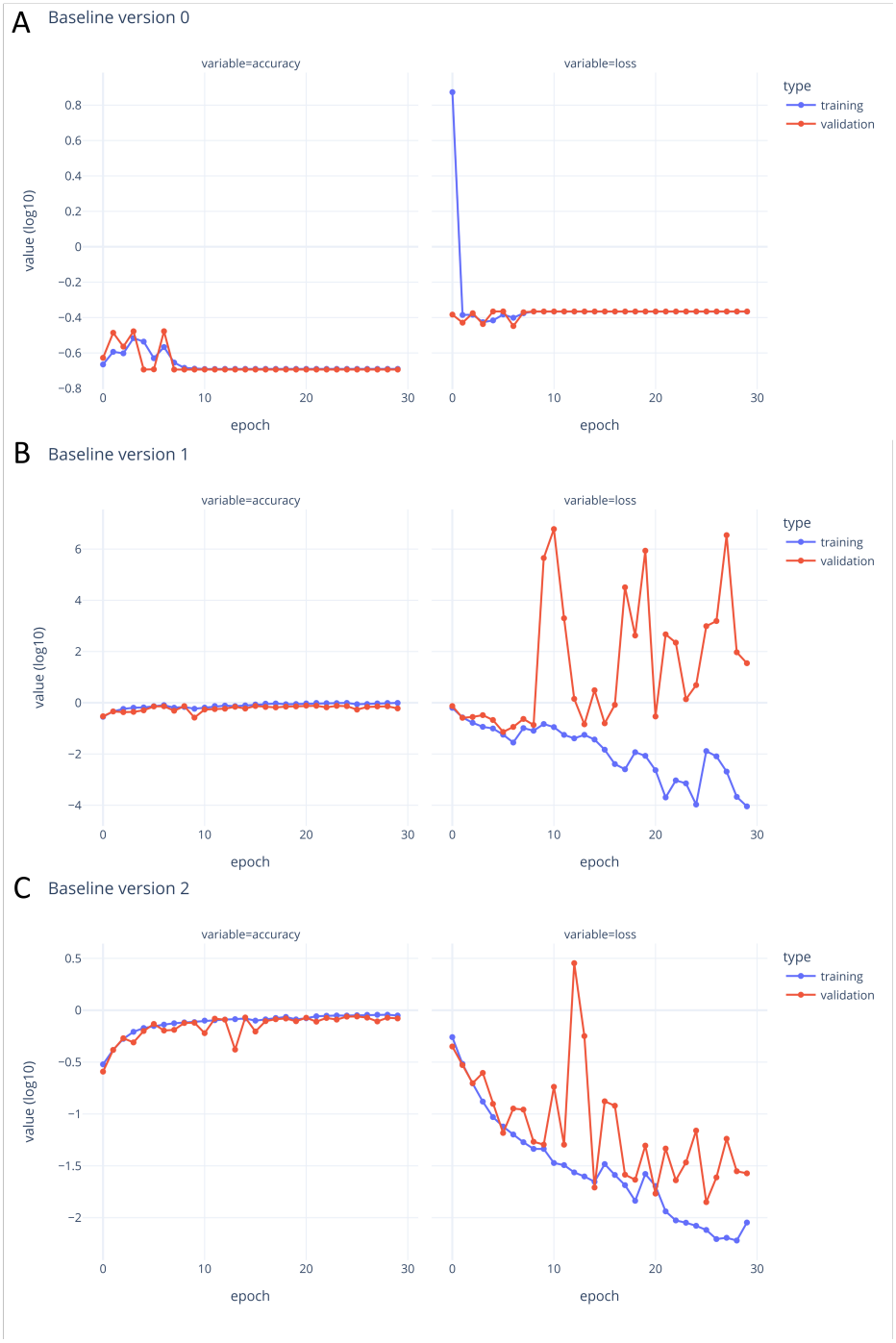

Figure 2: *The curves derived from training histories of the 3 baseline models. Measures are plotted in logarithmic scale for better visualization.*

4 choices for the value of the learning rate to supply to the Adam optimizer, the entire process would take approximately 5-6 hours to run. Nonetheless, for clarity and completeness, we provided the rationale and code of this approach as additional material in **Appendix 3.2**.

Thus, to be able to perform this tuning step, we decided to perform it outside of cross-validation, as a standalone step. We tried to minimize the potential issues with random splits by coding a function that produces a stratified split: both train and test sets contain, in proportion, the same amount of examples from each of the two classes and the images are shuffled at random before sampling. In this way we were able to tune 3 separate parameters: the learning rate, the number of units of the dense layers and the dropout rate. The entire tuning process required 1-1.5 hours and found that the model had a better performance when these values were chosen:

- Dense units: 1024
- Dropout rate: 0.2
- Learning rate: 0.001

The parameter search was performed with the package `keras-tuner` using the Hyperband algorithm (Li et al. [2018]), which is the most recently published algorithm for hyper-parameter search. Hyperband is an improvement of the random search technique that leverages on adaptive resource allocation and early-stopping.

Even if we can observe a bit more regular training curves (**Figure 3**), the optimization process did not produce an outstanding improvement of the model, since similar results were already achieved with the baseline version 2.

	loss	accuracy	zero_one_loss
count	5	5	5
mean	0.110	0.972	0.028
std	0.059	0.014	0.014
min	0.073	0.949	0.016
25%	0.080	0.974	0.020
50%	0.086	0.974	0.026
75%	0.095	0.980	0.026
max	0.214	0.984	0.051

Table 1: *Model performance statistics derived through 5-fold cross-validation*

fold	loss	accuracy	zero_one_loss
1	0.095	0.984	0.016
2	0.086	0.974	0.026
3	0.073	0.980	0.020
4	0.080	0.974	0.026
5	0.214	0.949	0.051

Table 2: *Detail of metrics for each cross-validation fold*

With the final tuned architecture we then proceeded to assess the average performance of our model on the data by setting up 5-fold cross validation. The procedure is by all means similar to k-fold cross validation for any other machine learning model: we used `StratifiedKFold` from `sklearn` to produce 5 stratified splits from the entire data set and looped over the different splits - a newly instantiated model is trained with the training split (4 out of 5 folds) and then validated on the test set (1 out of 5 folds). At each epoch during training the model is saved when a new best value of accuracy is reached - by loading and evaluating the best model for each fold, we were able to collect 5 set of values, each containing a measure of the chosen training loss (binary cross entropy), the accuracy and the

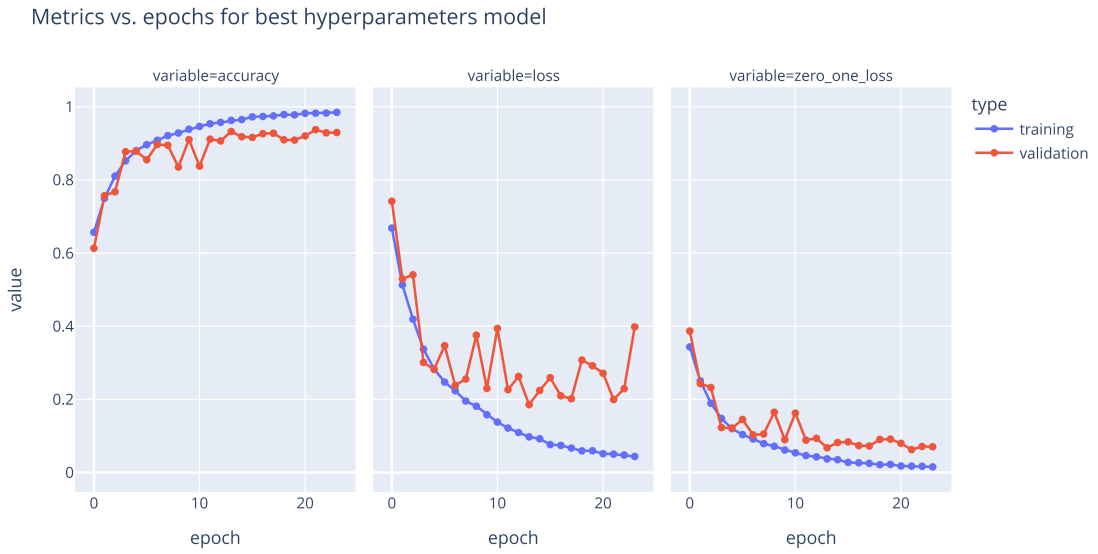

Figure 3: *Training curves for the hyper-parameter tuning phase*

zero-one loss.

We found that on average our model reached an accuracy of 97.2%, with a maximum of 98.4% as shown in **Table 1**. Details of the values for the metrics with respect to the folds are reported in **Table 2** and training history for each fold is shown in **Figure 5**.

As a final step, we selected the best model (in terms of accuracy) out of the 5 produced in the cross-validation phase, more precisely the model relative to fold number 1, and used this to obtain predictions for 10 images which were manually selected and were not included in the original data set. We plotted probability values in output of the final layer and the images with the relative index: we found that on a total of 10 images, only 1 was misclassified - having a probability of 53%, supposing a threshold value of 0.5, the assigned class was 1 despite the true label being 0 (**Figure 6**).

2 Methods

In this section we're going to present in more technical detail the rationale of our approach.

2.1 Tools and code availability

The full code of this project is available at the following GitHub repository `https://github.com/GiuliaPais/SMML_final_project`.

The repository includes:

- An IPython notebook named `SMML_final_project.ipynb`, which contains the code for all the workflow
- An IPython notebook named

Figure 4: *Cross validation metrics results*

`Appendix_A_SMML_final_project.ipynb` which contains the code for the trials of baseline model search

- Subfolders containing tabular files with results obtained, model checkpoints and additional images that can be used directly when executing code

The IPython notebooks can be executed in Google Colaboratory. We recommend, if possible, to use a GPU powered runtime, since the code is optimized for this scenario and computational time for training is greatly reduced. The code is designed to run also without carrying out the more time consuming operations: to re-run from scratch it is sufficient to enable the corresponding options (disabled by default) in the "Setup" section of both notebooks.

The programming language used in this project is Python version 3.8.16.

Relevant packages used:

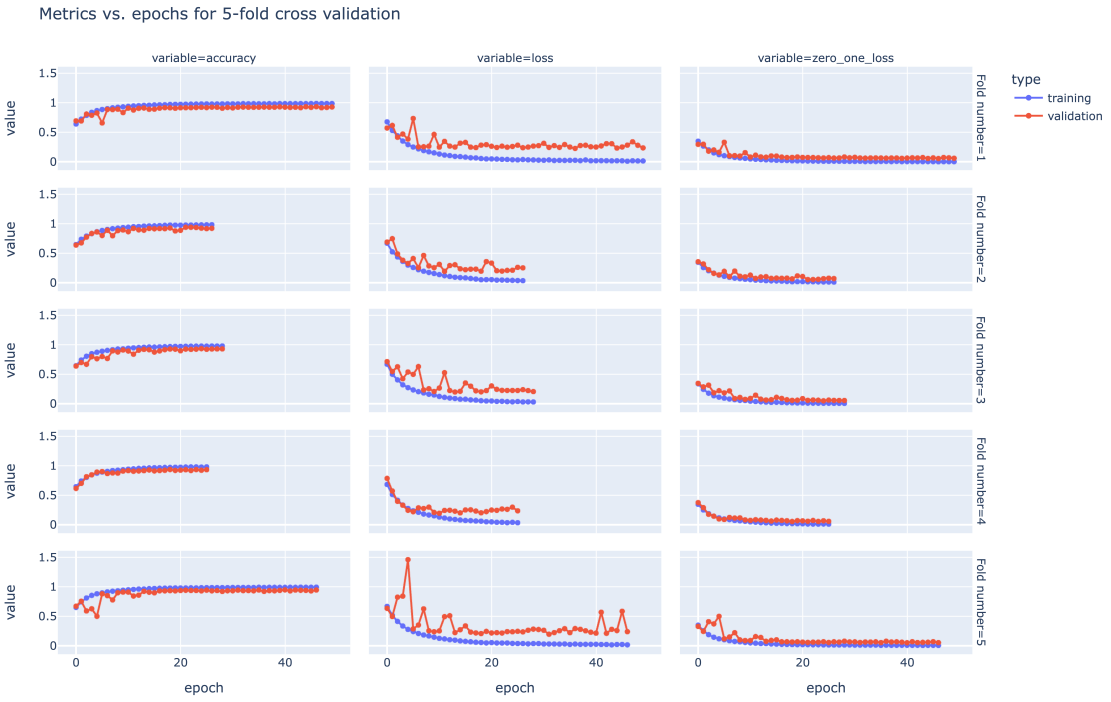

Figure 5: *Model metrics vs. epochs for 5-fold cross-validation*

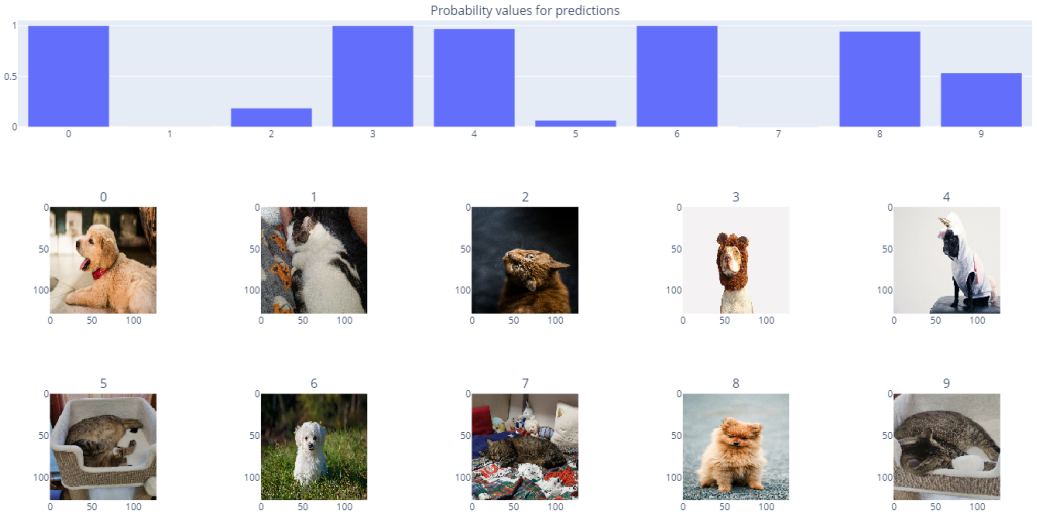

Figure 6: *Results for predictions of 10 images of cats and dogs not originally included in the data set*

- TensorFlow version 2.11.0
- keras-tuner for hyper-parameter tuning
- sklearn for producing stratified k folds during cross validation
- plotly, dash, numpy and pandas for data visualization
- ipywidgets for interactive widgets

2.2 Defining data source

When designing code, we took into account that data provided might come from different sources according to user preferences.

We implemented a class called `DataFetcher` which performs all the required actions to make data available in the session in 3 different cases.

When the method `fetch()` is called, the object looks up the value of a user defined option specifying the source of data: the user can specify the choice by selecting one of 3 options, "drive", "url" or "folder" by a dropdown menu. The "drive" option allows the mounting of a Google Drive folder through the tools provided by Colaboratory, "url" tries to download the dataset from the provided url and "folder" simply asks for a path on the local file system where the images are located (compatible also with manual upload).

Finally, by accessing the object attribute `data_path` it is possible to retrieve the path to the correct folder containing the categories of images to classify.

2.3 Data exploration and pre-processing

We were provided with a data set of 25000 images of cats and dogs, divided in folders corresponding to the relative category.

As a first step, we inspected the data set and collected basic information, available in **Table 3**. Although it wasn't possible to open and read 2 files, we observed the following:

- The average width of an image is 404.4 pixels, while the average height is 361 pixels, with an average aspect ratio of 1.15
- The minimum width and height present is 4 pixels for both, indicating that there might be some corrupt or inadequate elements in the dataset
- The maximum width and height is 500 pixels for both
- Minimum and maximum aspect ratio is 0.31 and 5.94 respectively

We decided to plot width and height of each image in a scatter plot visualization: to make this more informative, we coded a simple dashboard with the package `dash`,

	Width	Height	Aspect Ratio
count	24998.0	24998.0	24998.0
mean	404.45	360.99	1.16
std	109.0	97.01	0.29
min	4.0	4.0	0.31
25%	323.0	302.0	0.93
50%	448.0	375.0	1.27
75%	500.0	421.0	1.33
max	500.0	500.0	5.94

Table 3: *Summary statistics for the image data set*

where it is possible to click on points to inspect the associated image and isolate the categories by clicking on their icons in the legend (**Figure 7**). Through manual inspection we were able to identify some images that were either incorrect (did not depict a cat or a dog) or had other kinds of issues (for example, images whose dimensions were 4x4 px).

Many different approaches to identify and remove mislabeled data are available today (Brodley and Friedl [1999], Muller and Markert [2019]), however due to the relatively small size of our data set and the context of this project, we deemed sufficient a manual inspection of the images to identify such data. We were able to identify 6 wrongly labeled images for cats and other 6 for dogs: together with the 2 images that we were unable to read before, we removed 14 images in total from our starting data set, more in detail:

- I/O errors: Dogs/11702.jpg, Cats/666.jpg
- Mislabelled as cats: Cats/4688.jpg, Cats/5418.jpg, Cats/7564.jpg, Cats/8456.jpg, Cats/10029.jpg, Cats/10712.jpg
- Mislabelled as dogs: Dogs/1895.jpg, Dogs/8736.jpg, Dogs/9517.jpg, Dogs/10797.jpg, Dogs/10237.jpg, Dogs/10161.jpg

After cleaning, we plotted counts for each category to assess if the dataset was balanced, as mentioned in Section 1.

2.4 Obtaining a baseline model

The code for this section is available as a separate Jupyter notebook (*Appendix_A_SMML_final_project.ipynb*).

Before proceeding further it is necessary to start building a baseline model architecture. We used a trial-and-error incremental approach to find a suitable model, starting from simple commonly known convolutional networks architectures.

To evaluate each model we used the same method for each trial: we divided the whole data set of images in train set (80%) and test set (20%), ensuring a stratified split, that is we distributed examples of each category in the 2 splits in a way to have balanced training and test set, and we used the test set as value for the argument `validation_data` of the `fit()` function for 30 epochs, monitoring both loss and accuracy. All the images were re-sized automatically to 128x128 pixels and the data set was created via the function `tf.keras.utils.image_dataset_from_directory`.

Details of training metrics for each of the baselines is reported in **Tables 4, 5, 6**.

2.4.1 Baseline version 0

We defined the very first baseline model as follows:

- 1 Rescaling layer
- 1 Conv2D layer with 64 3x3 filters
- 1 MaxPool2D layer
- 1 Conv2D layer with 128 3x3 filters

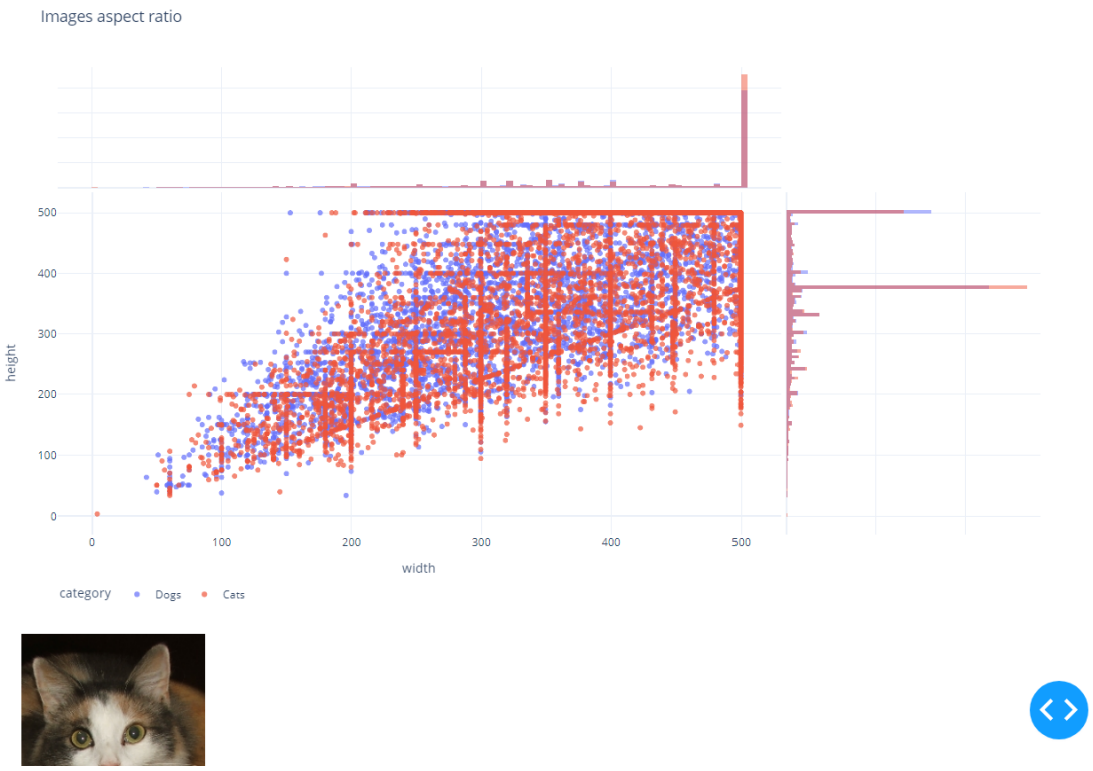

Figure 7: *Scatter plot visualization of width and height of images in the data set. The plot includes the marginal histograms on both axes and the possibility of inspecting the associated image by clicking on the corresponding point. Clicking on the category in the legend in the bottom portion of the plot can deactivate/reactivate the visualization for the category.*

- 1 MaxPool2D layer
- 1 Conv2D layer with 256 3x3 filters
- 1 MaxPool2D layer
- 1 Flatten layer
- 3 Dense layers with 512 neurons each
- A final Dense output layer with 1 neuron

The rescaling layer is used to map rgb values (0-255) in the range 0-1, while convolutional and max pooling layers are responsible for the extraction of features from images.

All convolutional and hidden layers use the ReLU (Equation 1) activation function, while the output layer uses the sigmoid (Equation 2) activation function. The model is compiled using the Adam optimizer with a learning rate of 0.01 and the binary cross entropy loss function. Binary Cross Entropy is the negative average of the log of corrected predicted probabilities and it's expressed in the Equation 3.

By using the sigmoid function in the output layer, we can interpret the results as

probabilities: if we set a threshold at 0.5, we assume that every value above 0.5 corresponds to class 1.

$$f(x) = x^+ = max(0, x) \tag{1}$$

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

$$\begin{aligned} Log\_loss = \frac{1}{N} \sum_{i=1}^{N} -(y_i * log(p_i) \\ + (1 - y_i) * log(1 - p_i) \end{aligned} \tag{3}$$

2.4.2 Baseline version 1

With respect to the version 0 of the model, we operated the following changes:

- Added 1 Conv2D layer with 512 3x3 filters and ReLU activation
- Added 1 MaxPooling2D layer
- Introduced BatchNormalization layers after layers that use non-linear activation functions

Batch normalization layers (Ioffe and Szegedy [2015]) were introduced to contrast the so-called covariate shift problem, that can be defined as the change in the distribution of the input variables present in the training and the test data. They achieve this by fixing means and variances of inputs through normalization after each mini-batch rather than the entire training set. Moreover, batch normalization layers offer many other advantages, among them we can count a reduction of the dependence of gradients on the scale of the parameters or initial values, thus allowing the use of a higher value of learning rate, a reduction for the need of dropout layers and allows the use of saturating nonlinear activation functions without the risk of getting stuck (Gu et al. [2015]).

2.4.3 Baseline version 2

With respect to the version 1 of the model, we operated the following changes:

- We duplicated the first 2 sets of convolutions (Conv2D + BatchNormalization + MaxPooling2D)
- We added Dropout layers after MaxPooling2D layers with a drop rate of 0.3
- We removed one of the 3 Dense hidden layers

2.5 Hyper-parameter tuning

Hyper-parameter tuning is the process of finding a set of values for some parameters relevant for the model in a way that the model performance is (locally) optimized.

For this step we used `keras_tuner` (O'Malley et al. [2019]), a tool provided by Keras that allows the search of the best hyper-parameters with several different algorithms, including the well-known random search and Bayesian optimization. As previously mentioned, we chose the Hyperband algorithm for our purposes.

We implemented a class, `ModelTuner`, that can perform the search procedure: a class instance must be initialized by providing a baseline model provider and other optional arguments that define folders on file system where different kind of files will be

epoch	accuracy	loss	val_accuracy	val_loss
0	0.51	2.39	0.53	0.68
1	0.55	0.68	0.62	0.65
2	0.55	0.68	0.57	0.69
3	0.60	0.65	0.62	0.65
4	0.59	0.66	0.50	0.69
5	0.53	0.68	0.50	0.69
6	0.57	0.67	0.62	0.64
7	0.52	0.69	0.50	0.69
8	0.50	0.69	0.50	0.69
9	0.50	0.69	0.50	0.69
10	0.50	0.69	0.50	0.69
11	0.50	0.69	0.50	0.69
12	0.50	0.69	0.50	0.69
13	0.50	0.69	0.50	0.69
14	0.50	0.69	0.50	0.69
15	0.50	0.69	0.50	0.69
16	0.50	0.69	0.50	0.69
17	0.50	0.69	0.50	0.69
18	0.50	0.69	0.50	0.69
19	0.50	0.69	0.50	0.69
20	0.50	0.69	0.50	0.69
21	0.50	0.69	0.50	0.69
22	0.50	0.69	0.50	0.69
23	0.50	0.69	0.50	0.69
24	0.50	0.69	0.50	0.69
25	0.50	0.69	0.50	0.69
26	0.50	0.69	0.50	0.69
27	0.50	0.69	0.50	0.69
28	0.50	0.69	0.50	0.69
29	0.50	0.69	0.50	0.69

Table 4: *Summary training history of baseline model v0*

saved. The baseline model provider is an object of class `BaselineModel` that builds the sequential model architecture adopted for baseline v2, but sets place holders for some parameters that will be tuned: in our case we chose to tune the following hyper-parameters with these choices of values

- `dense_units`: [1024, 1536, 2048]
- `dropout_rate`: [0.2, 0.3, 0.4]
- `learning_rate`: [0.001, 0.005, 0.01]

The values of possible choices for each of these parameters can be specified by the user, keeping in mind, however, that the more choices the more the combinations that need to be evaluated by the search algorithm, potentially increasing a lot the processing time (a single trial takes between 1 and 1.5 minutes). Furthermore, additional metrics can be monitored besides accuracy and they can be specified in the appropriate argument, following docs directions.

epoch	accuracy	loss	val_accuracy	val_loss
0	0.58	0.82	0.59	0.88
1	0.71	0.56	0.71	0.56
2	0.79	0.46	0.69	0.57
3	0.82	0.39	0.70	0.62
4	0.83	0.37	0.74	0.51
5	0.88	0.29	0.87	0.32
6	0.91	0.21	0.87	0.39
7	0.82	0.37	0.73	0.53
8	0.85	0.34	0.88	0.42
9	0.79	0.44	0.56	284.36
10	0.83	0.39	0.77	880.21
11	0.88	0.29	0.78	27.07
12	0.89	0.25	0.79	1.16
13	0.88	0.29	0.85	0.43
14	0.90	0.24	0.80	1.63
15	0.94	0.16	0.88	0.45
16	0.97	0.09	0.85	0.92
17	0.97	0.07	0.83	90.61
18	0.94	0.15	0.86	13.82
19	0.95	0.13	0.87	378.35
20	0.97	0.07	0.89	0.59
21	0.99	0.02	0.89	14.45
22	0.98	0.05	0.84	10.45
23	0.98	0.04	0.89	1.15
24	0.99	0.02	0.88	1.99
25	0.94	0.15	0.77	19.90
26	0.95	0.12	0.85	24.30
27	0.97	0.07	0.87	694.10
28	0.99	0.03	0.87	7.17
29	0.99	0.02	0.80	4.69

Table 5: *Summary training history of baseline model v1*

The `ModelTuner` object is able to automatically recover files and checkpoints without re-running the whole procedure (to re-run from scratch the relative option should be set in the "Setup" section of the notebook). Besides the actual hyperparameter search, the class offers some utility methods for plotting the training history of the model with best parameters, recovering the best parameters found as a pandas data frame and retrieve the configuration of the best model in json format.

2.6 5-Fold cross validation

The step of 5-fold cross validation was implemented as a class, `CrossValidator`, which once again is designed to recover automatically model checkpoints to avoid re-running time consuming tasks. The class can be instantiated by specifying the folder containing the images, the model checkpoints folder and the value for k for k-fold cross validation (5 by default).

When launched, cross validation cre-

epoch	accuracy	loss	val_accuracy	val_loss
0	0.59	0.77	0.55	0.70
1	0.68	0.60	0.68	0.59
2	0.76	0.49	0.76	0.49
3	0.81	0.41	0.73	0.55
4	0.84	0.36	0.82	0.41
5	0.86	0.33	0.88	0.31
6	0.87	0.30	0.82	0.39
7	0.88	0.28	0.83	0.38
8	0.89	0.26	0.88	0.28
9	0.89	0.26	0.89	0.27
10	0.90	0.23	0.80	0.48
11	0.91	0.22	0.92	0.27
12	0.91	0.21	0.91	1.57
13	0.92	0.20	0.68	0.78
14	0.92	0.19	0.93	0.18
15	0.90	0.23	0.81	0.42
16	0.92	0.20	0.90	0.40
17	0.93	0.19	0.92	0.20
18	0.94	0.16	0.92	0.20
19	0.92	0.21	0.90	0.27
20	0.93	0.18	0.93	0.17
21	0.94	0.14	0.90	0.26
22	0.95	0.13	0.93	0.19
23	0.95	0.13	0.91	0.23
24	0.95	0.13	0.94	0.31
25	0.95	0.12	0.94	0.16
26	0.96	0.11	0.93	0.20
27	0.96	0.11	0.90	0.29
28	0.96	0.11	0.93	0.21
29	0.95	0.13	0.92	0.21

Table 6: *Summary training history of baseline model v2*

ates an unbatched dataset from the specified folders and uses `sklearn StratifiedKFold()` to obtain the indices for each of the k splits. Through methods offered by the TensorFlow 2 dataset API, it is possible to generate new train and test datasets with the given indices. In particular, due to issues with high demand of memory for the generation of these datasets, we leveraged on `tf.lookup.StaticHashTable`: we initialize an hash table where the keys are the selected indices and the values are all set to 1 (True), providing as a default value 0 (False). We perform a hash of the index while filtering the entire dataset: if the index is found in the table 1 is returned and the element is included in the new dataset, 0 is returned otherwise and the element is skipped. For each fold a new blank instance of the tuned model architecture is instantiated and compiled with the optimal value found for the learning rate. Subsequently, training is

performed on the corresponding training set for the fold and validation is done on the corresponding test set. During training model checkpoints are saved monitoring the maximum accuracy: at the end of the training, the best model is loaded and evaluated with the test set and metrics recorded. At the end of the whole cross-validation procedure statistics for the metrics such as mean, max, min can be produced by calling the appropriate methods.

We were interested in monitoring, besides accuracy, also zero-one-loss: since this function is not provided by default by TensorFlow, we implemented it in a way that was suitable for the `metrics` argument of the `compile()` method. Just as a reminder, zero-one-loss is defined, for a single example as

$$f(x) = \begin{cases} 1 & \text{if } y \neq \hat{y} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

and this is implemented in our code with the function `zero_one_loss(y_true, y_pred)`.
We can obtain the average zero-one-loss by making use of TensorFlow `tf.keras.metrics.MeanMetricWrapper`.

The class allows also to plot automatically the training history for each fold and to plot the obtained cross-validation results as a box plot.

2.7 Predictions

To obtain prediction from our trained model we selected 10 images which are not included in the original data set, 5 images of cats and 5 images of dogs respectively. We provide these images on the GitHub repository under the folder "pred_images".

To obtain predictions it would have been sufficient to feed the dataset obtained through `tf.keras.utils.image_dataset_from_directory` to the `predict()` method of the model, however we wanted to show the predicted probabilities together with the associated images: the `predict()` method, for this reason, is called on each single image inside a for-loop.

Predicted probabilities and associated images are shown in a dedicated plot (**Figure 6**).

3 Appendix

3.1 Baseline models details

```
Model: "baseline_v0"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 rescaling_3 (Rescaling)     (None, 128, 128, 3)       0

 conv2d_9 (Conv2D)           (None, 128, 128, 64)      1792

 max_pooling2d_9 (MaxPooling  (None, 64, 64, 64)       0
 2D)

 conv2d_10 (Conv2D)          (None, 64, 64, 128)       73856

 max_pooling2d_10 (MaxPoolin  (None, 32, 32, 128)      0
 g2D)

 conv2d_11 (Conv2D)          (None, 32, 32, 256)       295168

 max_pooling2d_11 (MaxPoolin  (None, 16, 16, 256)      0
 g2D)

 flatten_3 (Flatten)         (None, 65536)             0

 dense_12 (Dense)            (None, 512)               33554944

 dense_13 (Dense)            (None, 512)               262656

 dense_14 (Dense)            (None, 512)               262656

 dense_15 (Dense)            (None, 1)                 513

=================================================================
Total params: 34,451,585
Trainable params: 34,451,585
Non-trainable params: 0
_________________________________________________________________
```

```
Model: "baseline_v1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 rescaling_4 (Rescaling)     (None, 128, 128, 3)       0

 conv2d_16 (Conv2D)          (None, 128, 128, 64)      1792

 batch_normalization_27 (Bat  (None, 128, 128, 64)     256
 chNormalization)

 max_pooling2d_16 (MaxPoolin  (None, 64, 64, 64)       0
 g2D)

 conv2d_17 (Conv2D)          (None, 64, 64, 128)       73856

 batch_normalization_28 (Bat  (None, 64, 64, 128)      512
 chNormalization)

 max_pooling2d_17 (MaxPoolin  (None, 32, 32, 128)      0
 g2D)

 conv2d_18 (Conv2D)          (None, 32, 32, 256)       295168

 batch_normalization_29 (Bat  (None, 32, 32, 256)      1024
 chNormalization)

 max_pooling2d_18 (MaxPoolin  (None, 16, 16, 256)      0
 g2D)

 conv2d_19 (Conv2D)          (None, 16, 16, 512)       1180160

 batch_normalization_30 (Bat  (None, 16, 16, 512)      2048
 chNormalization)

 max_pooling2d_19 (MaxPoolin  (None, 8, 8, 512)        0
 g2D)

 flatten_4 (Flatten)         (None, 32768)             0

 dense_16 (Dense)            (None, 1024)              33555456
```

```
 batch_normalization_31 (Bat  (None, 1024)             4096
 chNormalization)

 dense_17 (Dense)            (None, 1024)             1049600

 batch_normalization_32 (Bat  (None, 1024)             4096
 chNormalization)

 dense_18 (Dense)            (None, 1024)             1049600

 batch_normalization_33 (Bat  (None, 1024)             4096
 chNormalization)

 dense_19 (Dense)            (None, 1)                1025

=================================================================
Total params: 37,222,785
Trainable params: 37,214,721
Non-trainable params: 8,064
_________________________________________________________________
```

```
Model: "baseline_v2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 rescaling_8 (Rescaling)     (None, 128, 128, 3)       0

 conv2d_38 (Conv2D)          (None, 128, 128, 64)      1792

 batch_normalization_59 (Bat  (None, 128, 128, 64)     256
 chNormalization)

 max_pooling2d_38 (MaxPoolin  (None, 64, 64, 64)       0
 g2D)

 dropout_30 (Dropout)        (None, 64, 64, 64)        0

 conv2d_39 (Conv2D)          (None, 64, 64, 64)        36928
```

```
batch_normalization_60 (Bat  (None, 64, 64, 64)       256
chNormalization)

max_pooling2d_39 (MaxPoolin  (None, 32, 32, 64)       0
g2D)

dropout_31 (Dropout)         (None, 32, 32, 64)       0

conv2d_40 (Conv2D)           (None, 32, 32, 128)      73856

batch_normalization_61 (Bat  (None, 32, 32, 128)      512
chNormalization)

max_pooling2d_40 (MaxPoolin  (None, 16, 16, 128)      0
g2D)

dropout_32 (Dropout)         (None, 16, 16, 128)      0

conv2d_41 (Conv2D)           (None, 16, 16, 128)      147584

batch_normalization_62 (Bat  (None, 16, 16, 128)      512
chNormalization)

max_pooling2d_41 (MaxPoolin  (None, 8, 8, 128)        0
g2D)

dropout_33 (Dropout)         (None, 8, 8, 128)        0

conv2d_42 (Conv2D)           (None, 8, 8, 256)        295168

batch_normalization_63 (Bat  (None, 8, 8, 256)        1024
chNormalization)

max_pooling2d_42 (MaxPoolin  (None, 4, 4, 256)        0
g2D)

dropout_34 (Dropout)         (None, 4, 4, 256)        0

conv2d_43 (Conv2D)           (None, 4, 4, 512)        1180160

batch_normalization_64 (Bat  (None, 4, 4, 512)        2048
```

```
 chNormalization)

 max_pooling2d_43 (MaxPoolin  (None, 2, 2, 512)        0
 g2D)

 dropout_35 (Dropout)         (None, 2, 2, 512)        0

 flatten_8 (Flatten)          (None, 2048)             0

 dense_30 (Dense)             (None, 1024)             2098176

 batch_normalization_65 (Bat  (None, 1024)             4096
 chNormalization)

 dense_31 (Dense)             (None, 1024)             1049600

 batch_normalization_66 (Bat  (None, 1024)             4096
 chNormalization)

 dense_32 (Dense)             (None, 1)                1025

=================================================================
Total params: 4,897,089
Trainable params: 4,890,689
Non-trainable params: 6,400
_________________________________________________________________
```

3.2 Nested cross validation approach (abandoned)

```
def print_and_get_best_hp(best_hp, fold_number,
                          param_list, verbose):
  hp_dict = dict()
  for p in param_list:
    hp_dict[p] = best_hp.get(p)
  if verbose == 1:
    param_strings = "\n".join(
    [f"- {key}: {val}" for key, val in hp_dict.items()])
    print(f"""
    Hyperparameter search is complete for inner
```

```
    fold {fold_number}.
    Summary:
    {param_strings}
    """)
  return hp_dict

default_hp_list = ["learning_rate"]

# Define a function that performs inner KfoldCV
from collections import defaultdict

model_checkpoints_fold = "/model_checkpoints"

def inner_KFold_CV(dataset, n_folds, num_fold, ck_folder,
                   verbose=1):
  '''
    Computes the inner cross validation step for
    hyperparameter tuning.

    Params:
        - dataset: a tf.data.Dataset object (generator)
        - n_folds: the number of folds to use for inner
                   cross validation
        - num_fold: the number of the trial of outer CV
                    or None if tuning is performed on
                    whole dataset
        - ck_folder: the path to the folder in which
                     the models are saved
        - verbose: either 0 or 1, if 1 prints additional
                   messages to inspect progress
  '''
  if verbose == 1:
    print("Hyper-parameter tuning in progress...")
  inner_kf = StratifiedKFold(n_splits=n_folds)
  if verbose == 1:
    print("_Obtaining validation set...")
  labels = np.asarray([el[1] for el in dataset])
  ind_gen = inner_kf.split(np.zeros(len(labels)), labels)
  best_hps = defaultdict(list)
  best_histories = {}
  for k, gen in enumerate(ind_gen):
```

```
    if verbose == 1 and not num_fold is None:
      print(f"__Inner fold number {k+1} of {n_folds}")
    train_ind, test_ind = gen
    inner_train_set, inner_test_set = get_folds(dataset,
    train_ind, test_ind)
    inner_train_set = inner_train_set.batch(BATCH_SIZE)
    inner_test_set = inner_test_set.batch(BATCH_SIZE)
    tuner_saves_folder = f"tuner_saves/{num_fold}/{k+1}"
    stop_early = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss', patience=5)
    tuner = kt.Hyperband(get_baseline,
                         objective='val_accuracy',
                         directory=tuner_saves_folder,
                         project_name='cats_dogs_classification',
                         distribution_strategy=strategy)
    tuner.search(inner_train_set, epochs=50,
    validation_data=inner_test_set, callbacks=[stop_early])
    best = tuner.get_best_hyperparameters()[0]
    best_hp_dict = print_and_get_best_hp(best, k+1,
    default_hp_list, verbose)
    best_hps["fold_number"].append(k+1)
    for key, val in best_hp_dict.items():
      best_hps[key].append(val)
    model = tuner.hypermodel.build(best)
    model_cp = tf.keras.callbacks.ModelCheckpoint(
        filepath=f"{ck_folder}/{k+1}",
        monitor="val_accuracy",
        mode="max",
        save_best_only=True
    )
    history = model.fit(inner_train_set, epochs=50,
    validation_data=inner_test_set,
    callbacks=[model_cp, stop_early])
    best_histories[k+1] = history
  if verbose == 1 and not num_fold is None:
    print(f"""Hyper-parameter tuning completed
    for outer fold {num_fold}""")
  best_hps_df = pd.DataFrame.from_dict(best_hps_df)
  return (best_hps_df, best_histories)

# Define the function for outer CV
```

```
def outer_KFoldCV(n_folds=5, inner_folds=3,
                  color_mode="rgb", verbose=1):
  if verbose == 1:
    print(f"""Performing {n_folds}-fold cross validation
    estimates on baseline...""")
    print(f"- Creating dataset of images from folders...")
  dataset = create_img_dataset(data_path,
  (IMG_HEIGHT, IMG_WIDTH), mode=color_mode)
  if verbose == 1:
    print(f"- Obtaining folds indices...")
  labels = np.asarray(list(dataset.map(lambda x, y: y)))
  outer_kf = StratifiedKFold(n_splits=n_folds)
  index_gen = outer_kf.split(np.zeros(len(labels)), labels)
  accuracies = []
  losses = []
  for i, gen in enumerate(index_gen):
    if verbose == 1:
      print(f"-- PROCESSING: outer fold number {i+1} of {n_folds}")
    train_ind, test_ind = gen
    train_set, test_set = get_folds(dataset, train_ind, test_ind)
    best_params_df, histories_best = inner_KFold_CV(
    train_set, inner_folds, i+1,
    f"{model_checkpoints_fold}/{i+1}", verbose=verbose)
    # Evaluate each of the models with test set
    eval_metrics = []
    for j in range(1, inner_folds+1):
      if verbose == 1:
        print(f"--- Evaluating hypermodel {j} out of {inner_folds}")
      local_model = tf.keras.models.load_model(
      f"{model_checkpoints_fold}/{i+1}/{j}")
      local_eval_metrics = local_model.evaluate(test_set)
      eval_metrics.append(local_eval_metrics)
    # Take the best one (highest acc)
    acc_hm_fold = list(map(
    lambda x: x[eval_metrics.index("accuracy")], eval_metrics))
    loss_hm_fold = list(map(
    lambda x: x[eval_metrics.index("loss")], eval_metrics))
    selected_hm_fold = np.argmax(acc_hm_fold)
    accuracies.append(acc_hm_fold[selected_hm_fold])
    losses.append(loss_hm_fold[selected_hm_fold])
  avg_acc = np.mean(accuracies)
```

```
    avg_loss = np.mean(losses)
    return (avg_loss, avg_acc, accuracies,
            losses, n_folds, inner_folds)
```

References

Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun. Accelerating very deep convolutional networks for classification and detection, 2015. URL https://arxiv.org/abs/1505.06798.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, may 2017. ISSN 0001-0782. doi: 10.1145/3065386. URL https://doi.org/10.1145/3065386.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018. URL http://jmlr.org/papers/v18/16-558.html.

C. E. Brodley and M. A. Friedl. Identifying mislabeled training data. *Journal of Artificial Intelligence Research*, 11:131–167, aug 1999. doi: 10.1613/jair.606. URL https://doi.org/10.1613\%2Fjair.606.

Nicolas M. Muller and Karla Markert. Identifying mislabeled instances in classification datasets. In *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, jul 2019. doi: 10.1109/ijcnn.2019.8851920. URL https://doi.org/10.1109\%2Fijcnn.2019.8851920.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015. URL https://arxiv.org/abs/1502.03167.

Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Li Wang, Gang Wang, Jianfei Cai, and Tsuhan Chen. Recent advances in convolutional neural networks, 2015. URL https://arxiv.org/abs/1512.07108.

Tom O'Malley, Elie Bursztein, James Long, François Chollet, Haifeng Jin, Luca Invernizzi, et al. Kerastuner. https://github.com/keras-team/keras-tuner, 2019.