Introduction

In this challenge we were asked to classify pictures of plants, divided in two categories according to their health. It was a binary classification problem and the goal was to correctly predict the health status for each sample of the dataset, which could either be 'healthy' (0) or 'unhealthy' (1).

Data loading, visualization, and preparation functions

In order to train the neural network, we first proceeded to do a quick visual analysis of the images in the dataset, during which we noticed the presence of outliers. We decided that the best choice was to remove them, assuming they cannot be found inside the final test set. Since they are divided in two subclasses (`shrek` and `trololo` in our code) containing all equal images, the only thing we had to do was to identify the position of a single outlier for both: at that point we removed them with a simple function doing comparisons through the whole dataset. All of these actions were performed with the functions `load_data` and `delete_outliers`. Moreover, for the training of all models, we performed the usual training/validation split, holding out 500 samples for validation from the original `public_data.npz` dataset.

Transfer learning, hyperparameters tuning

Our first thought was that building a custom network would have led us to spend too much time on picking the right hyperparameters. Since the dataset is quite small, the network would not have probably needed to be too deep, but the right combination of number of convolutional layers, their depth, the filter size through the network, etc., faced us with too many combinations. Therefore, we decided to apply transfer learning. The first network we chose to try was MobileNet, both in the original version and MobileNetV2: from these, we removed the top classifier and added our own. In fact, we picked MobileNet pre-trained models because of its low number of parameters (3.5M) and depth (105) [1]. A first implementation with two output neurons included no augmentations, one dense layer, early stopping and reducing the learning rate on plateau with the following model representation:

```
Layer (type)                 Output Shape         Param #
input_2 (InputLayer)         (None, 96, 96, 3)    0
mobilenetv2_1.00_96          (None, 1280)         2257984
dense (Dense)                (None, 2)            2562
-----------------------------------------------------------------
Total params: 2259265 (8.62 MB)
Trainable params: 1281 (5.00 KB)
Non-trainable params: 2257984 (8.61 MB)
```

The best performance of the two was achieved with the second version MobileNetV2, and from this we obtained as final results on our validation set the following [*val_accuracy*: 0.8200; *val_precision*: 0.8049; *val_recall*: 0.6947], while on CodaLab we were only able to achieve about 60% of accuracy. We all think of this as a clear evidence of overfitting, since the training accuracy was even higher than the validation accuracy.

Data augmentation

To improve the low accuracy and solve the overfitting problem, we introduced data augmentation. At first, we implemented it similarly to what we have seen during the lectures, with a pre-processing layer preceding the network applying some random transformations involving translation, rotation, flipping, change of contrast and brightness, and zoom. Even in this way, no major improvement was reached, probably because of the class imbalance. At this point we had two separate problems to solve, that we decided to address in these different ways:

1. to fight class imbalance, we tried using the function `compute_class_weights`, which adjusts the weights in the model according to the class distribution [2];
2. to reduce overfitting, we tried abandoning traditional data augmentation in favor of `ImageDataGenerator`, that provides a model to be fit on the training data and that can be

directly used during the model fitting to generate the training and validation data, which have the property of changing at every epoch [3].

Given the still not quite satisfactory results ([*val_accuracy*: 0.8280 - *val_precision*: 0.7653 - *val_recall*: 0.7895] for point 1., and [*val_accuracy*: 0.7200 - *val_precision*: 0.5926 - *val_recall*: 0.4848] for point 2.), we preferred not to use these functions. Instead, a different and more elaborate approach was to increase the size of the dataset by cloning and applying augmentation on the training dataset (the `X_train` set, and only that, since we left the validation dataset untouched). In particular, we first created a single sequential layer containing different spatial and color transformations, then we duplicated the original training set by transforming it with the sequential layer, and eventually we appended it to the starting training set to double its size. Furthermore, to solve class imbalance we separately duplicated the samples in the minority class and added them to the new `X_train` array. This does not bring to a perfect 50/50 balance as with `compute_class_weights`; however, this proved to be an apparently efficient choice, improving our accuracy score up to 74% on CodaLab and reducing overfitting, and as a consequence it was later on implemented in all our models.

Other pretrained models

At this point, once consolidated the augmentation pipeline, we also tried different pretrained models (and in particular, EfficientNetV2B0, Xception and ConvNextTiny, all taken from [1]: we chose the models with the lowest number of parameters), but since just adding one final dense layer would not lead to better results, we started working on the classifier part of the network. Our final configuration presents a number of layers between 2 and 3, a number of neurons between 16 and 64, and dropout and batch normalization layers. Every time, we also performed fine-tuning on the last convolutional block of the network. With such a configuration and using ConvNextTiny, for example, we reached an 80% accuracy on CodaLab.

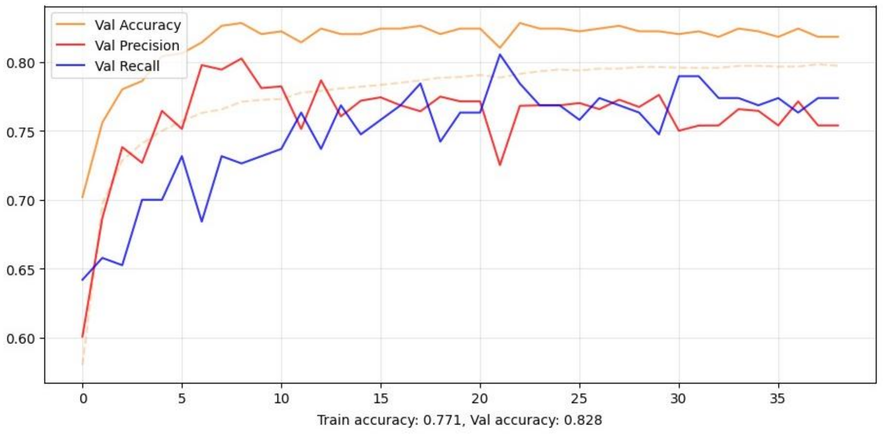
Ensemble of models

Not being still quite satisfied, as a final improvement we decided to merge the best models we were able to build, i.e., MobileNetV2, EfficientNetV2B0, Xception and ConvNextTiny. To all of these we applied fine tuning and for all of them we used similar versions of the feature classifier. The rationale behind this choice is that, assuming the single classifiers as independent and when their error rate is less than 0.5, making them vote and taking the majority voting decreases the total error rate. We approached the problem in a different but quite similar way: in fact, since we noticed that the network ConvNextTiny had a slightly better performance with respect to the other ones, both when fine-tuned and when not, we decided to include two models derived from ConvNextTiny (one fine-tuned and one not), and one for each other (for a total of five models). Moreover, our model does not predict the majority voting of the ensemble, but it outputs the average of the five predictions of the individual models. In such a way, it is as if a higher weight was given to models which are very sure of their choice, while more uncertain models, which present more balanced predictions, do not influence as much the final output since they contribute with a similar quantity to both probabilities (that of belonging to the 'healthy' class, and that of belonging to the 'unhealthy' class). A few details were taken into consideration while doing this:

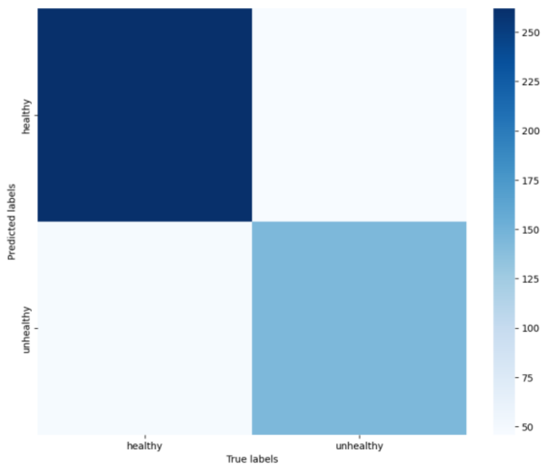
- each model was trained independently, making sure to change the seed for randomness and performing slightly different transformations on the training set, in order not to train on the same set of images and to aim for a higher robustness;
- when merging the models, each of them would receive the input in the same format, so (when needed) an extra layer containing the correct pre-processing function was added before the model. This was done using a so-called 'lambda' layer, which can implement custom functions; the same was done after the 'add' layer, by including a 'lambda' layer that would divide the sum of the predictions by 5 in order to get normalized results.

Graphs and metrics analysis

During the development we kept track of all the most important metrics (mainly accuracy, but also precision and recall) that we analyzed through the use of graphs to see their evolution during the training. The case for the MobileNetV2 follows:



An important observation has to be made about the training and validation values: as we can see from the image above, the training accuracy is generally smaller compared to that of validation. We think the main reason for this is that since the training set is much more varied than the validation set, due to all the augmentation process it goes through, the network finds more difficult to learn all the new features but can obtain better results thanks to this. The relative confusion matrix for this case is the one on the left, and it was computed on a separate test set of 500 samples. The accuracy is not perfect, and that is the main reason why we adopted an ensemble classifier at the end, but it has an overall good performance. Moreover, the FP's and FN's are quite balanced, presumably thanks to the class balancing we performed, while of course the TP's and TN's follow the class distribution of the test set we used (in which we maintained the same class distribution as in the original `public_data.npz`)



Final observations

As a final note, we would like to stress that the analysis presented for the MobileNetV2 network was also developed for the remaining models to better understand where the problems lay, but they are not reported here because of their high similarity with what was already shown here, and hence their low significance. Furthermore, during the design of our final submitted model, configurations both with one output neuron (with sigmoid activation function) and two output neurons (with softmax activation function) were used, and to be more specific, the solution with one output neuron was necessary in order to analyse also precision, recall and all related quantities. These two choices are not perfectly equivalent, since this changes the number of trainable parameters, however the difference is so small (the number of neurons in the second to last layer) compared to the total number of trainable parameters (as an order of magnitude, about 16/50000=0.03%) that the results we obtained were much more sensitive to the random fluctuations due to the seed, rather than to this choice.

References

[1]. https://keras.io/api/applications/

[2]. https://scikit-learn.org/stable/modules/generated/sklearn.utils.class_weight.compute_class_weight.html

[3]. https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator

Contributions

- Francesco Barabino: data preparation and outlier treatment, MobileNetV2 analysis, EfficientNetV2B0 analysis.
- Luis Riaz Ahmed: `ImageDataGenerator`, `fit_generator` and `compute_class_weights` study/impementation (all excluded from the final model), MobileNetV2 analysis, ResNet50 analysis (excluded).
- Lorenzo Iori: Xception analysis, ConvNextTiny analysis, ensemble model development.