# APPROACHING THE PROBLEM

## Description of the problem:

This is a binary classification problem concerning the state of health of plants. It is available an initial dataset containing 5200 RGB pictures concerning plants (stored in a Numpy array of shape 5200x96x96x3), and their respective state, being each one of them either 'healthy' or 'unhealthy' (included in a second array of labels). I develop a neural network model to predict the class of belonging of plants' pictures. Moreover, a hidden dataset is provided to test intermediate models before the choice of the best one.

## **Approaching the problem:** 3 main phases.

The problem was approached step-by-step, developing a variety of models, which could fit the data better and better.
I will now give an overview of the chronological steps followed on my way to the final model.
*Phase 1:*

The very first step was loading and visualizing raw data. This led me to detect, through manual inspection, some outliers in the dataset, meaning Shrek and Mr. Trololo pictures, to be excluded from usable data. Here below images from 50 to 59 of the original dataset:

```
# Iterate through the selected number of images
for i in range(50, 60):
    ax = axes[i % num_img]
    ax.imshow(dataNorm[i])
    ax.set_title(f'{data["labels"][i]}')  # Show the corresponding digit label

# Adjust layout and display the images
plt.tight_layout()
plt.show()
```


*Figure 1, visualization of the dataset*

Playing with data, it was straightforward to notice that the two classes of the dataset were not balanced, having 3199 healthy subjects and 2001 unhealthy ones. This was the very first clue that led me to augment the data in a second phase.
The next step was data processing. First, outliers were removed from the dataset. I also opted for converting pictures' labels from strings to integers, associating 1 to healthy samples and 0 to unhealthy ones. Then, dealing with a binary classification problem, I performed one-hot-encoding to build a neural network with a two-neuron last layer. Moreover, to facilitate code crafting, two dictionaries, Labels Name and Labels Count, were created to associate (1, 0) and (0,1) to a state of health and to a number of pictures. Finally, this data was stored as data_clean.npz. This would allow retrieving the new dataset without building it from scratch at each runtime.
In the very first phase, I proceeded dividing the dataset into a training set and a validation set, choosing the 80%/20% split and maintaining the initial class distribution in each subset. However, this was not always true: I tested some of my models on a test set devoting generally 5% of the data. The very first created models were sequences of CNN and FNN layers, designed from scratch and varying parameters and hyperparameters at each training. To do so more efficiently, a first block of code containing global variables was added and modified as needed, to manage the setting of data, model and training

parameters. However, from the training and initial testing, the tendency to overfit was evident. As a matter of fact, the accuracy scores in the training and validation parts didn't match the ones obtained in the test one. This led me to work on the dataset and perform data augmentation.

*Phase 2*:

To perform data augmentation, the following custom functions were defined: Rotation, Flip, Brightness, Zoom and Random. It must be said that the dataset was increased to ensure a balanced class distribution. The augmented dataset is composed of 5000 'healthy' images and 5000 'unhealthy' ones. Eventually the data was shuffled and stored in the file data_aug.npz to be reloaded directly at each model try without rebuilding the dataset at each runtime.

During this second phase, I experimented on the new data file creating and training different models designed analyzing not only the accuracy but also the F1 scores and the confusion matrix of previous ones. I selected these measures to have a more precise impression of the performance of the models for each class. Unfortunately, this didn't lead me much further. Although I worked on overfitting with different techniques such as early stopping and dropout, it was still a major downside of my model as shown in the table below reporting the accuracy scores of few models:

| Name | model_2023-11-11_20.50.10 | model_2023-11-12_11.51.41 | model_2023-11-12_13.29.3 |
|---|---|---|---|
| Accuracy (Train) | 0,9321 | 0,9508 | 0,9188 |
| Accuracy (Validation) | 0,913 | 0,934 | 0,8811 |
| Accuracy (Test - Phase 1) | 0,69 | 0,66 | 0,62 |

*Table 1*

*Phase 3*:

In phase three, I moved from custom CNNs to pre-trained Keras default CNN implementations. Transfer learning was performed trying to exploit a few Keras models which could fit the data the best.

These were the four major factors taken into consideration when choosing the pre-trained CNNs: not too many parameters, reasonable bytes size, pre-trained with RGB images with similar size and from ImageNet.

I played around with four Keras pre-trained models: MobileNetV2, Xception, EfficientNetB4 and EfficientNetV2S. Although the feature extraction part of the network was fixed, many were the parameters and hyperparameters that could be changed: the number of classification layers and the neurons per layer, the batch size and the number of epochs, the patience of early stopping, the split between training set and validation set and so on.

I also implemented fine tuning so that the pre-trained CNN could better fit my problem. During fine tuning training I realized that manually increasing the numbers of epochs little by little (20 epochs, then 10, then 5…) could give me a better overview of how the training was proceeding. It's worth highlighting that I used a very low learning rate, i.e. $1e^{-5}$, to avoid modifying too much the weights of the entire model.

In this last phase a second augmented dataset of 22800 observations was constructed and seemed to have very good results in the transfer learning training; however, RAM availability issues never allowed me to finish the program.

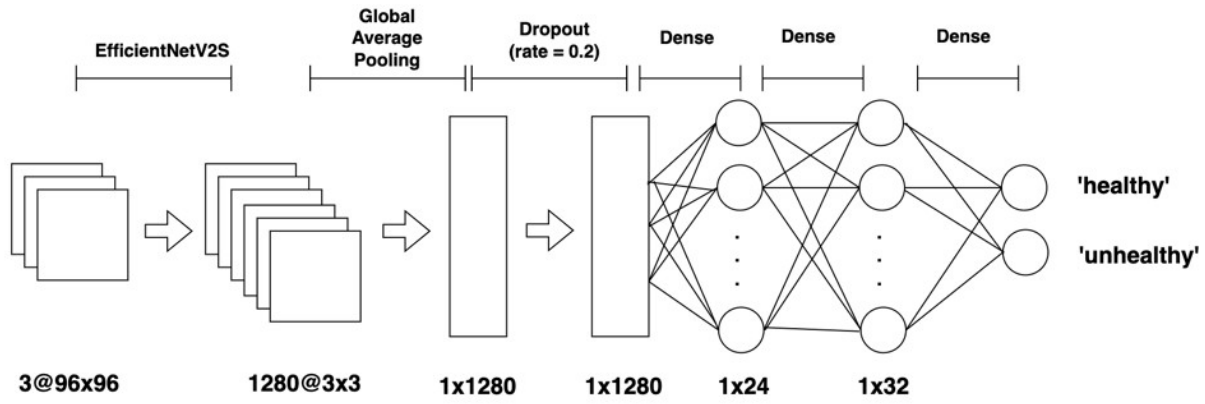
After training different networks and analyzing their performances which were consistently higher than the ones obtained in phase 2 (see table 2), I propose the following as my final best model

| Name | model_2023-11-15_22.20.8 | model_2023-11-15_18.56.21 | model_2023-11-14_22.33.28 |
|---|---|---|---|
| Accuracy (Train) | 0,9178 | 0,9188 | 0,8389 |
| Accuracy (Validation) | 0,887 | 0,879 | 0,8215 |
| Accuracy (Test - Phase 1) | 0,79 | 0,77 | 0,72 |

*Table 2*

## The model:

I will start describing the structure of the network. The accepted input data, in the first layer, are RGB images of size 96x96 pixels, stored in a (BS, 96, 96, 3) tensor, where BS is the batch size. Just after the input layer, my network presents a preprocessing layer, whose role is to perform data augmentation applying, in a random way, the following transformations to the input images: horizontal flip, translation, rotation, zoom. The parameters of these transformations are specified in the *Global Variables* section at the beginning of the code. The preprocessing layer is directly connected to a convolutional neural network, which in turn is tied to a global averaging pooling. This layer converts the output of EfficientNetV2S, which is a four-dimensional tensor of shape (BS, 3, 3, 1280) into a two-dimensional tensor of shape (BS, 1280), which can be fed to the following fully connected network. Before the fully connected network, I have placed a Dropout Layer to reduce the risk of overfitting by turning off randomly some of the inputs. The feature classification part is composed of two hidden layers, respectively of 24 and 32 neurons. It's worth noticing that the activation function, implemented in both layers is the ReLU. It has, in fact, given me better performances than the Sigmoid and the Tanh. Finally, the output layer consists of two neurons with the Softmax as activation function. I choose this configuration, with one output neuron for each class, instead of using a single neuron, because this makes my model more scalable. As a matter of fact, in a real-world scenario it is likely to be requested, at any moment in time, to add new output classes. For example, to distinguish between healthy plants, unhealthy plants and background/no plant, it would be sufficient to add a new output neuron and retrain the model.
The following is a representation of the network:




I will now move on describing the training parameters. I opted for a splitting between training set and validation set of 75%/25%, a batch size of 64 both in transfer learning, to prevent overfitting. The chosen number of epochs in transfer learning training is 120 and the early stopping patience is 25. The learning rate is 0.001. These numbers are mainly the result of a trial-and-error process. I choose not to perform fine tuning because the training and validation accuracies after transfer learning were significantly high and close (around 0.9).

**Conclusion:** To conclude, I would like to outline some further developments. If RAM availability had allowed for it, one could have tried to experiment on the second augmented dataset and on other Keras pre-trained networks, with a higher number of parameters, potentially with a higher number of dense layers.

References:


- *Create date format:* https://www.w3schools.com/python/python_datetime.asp
- *Numpy rotate and flip:* https://note.nkmk.me/en/python-opencv-numpy-rotate-flip/
- *Image zoom:* https://stackoverflow.com/questions/37119071/scipy-rotate-and-zoom-an-image-without-changing-its-dimensions
- *Randomint:* https://numpy.org/doc/stable/reference/random/generated/numpy.random.randint.html
- *Save images:* https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.savefig.html
- *Pandas factorize:* https://pandas.pydata.org/docs/reference/api/pandas.factorize.html
- *Numpy delete:* https://numpy.org/doc/stable/reference/generated/numpy.delete.html
- *Numpy savez:* https://numpy.org/doc/stable/reference/generated/numpy.savez.html
- *Numpy append:* https://numpy.org/doc/stable/reference/generated/numpy.append.html
- *Numpyshuffle:* https://numpy.org/doc/stable/reference/random/generated/numpy.random.shuffle.html
- *Keras layer - Dropout:* https://keras.io/api/layers/regularization_layers/dropout/
- *Kears application:* https://keras.io/api/applications/
  - *Xception:* https://keras.io/api/applications/xception/
    *(Cornell University:* https://arxiv.org/abs/1610.02357*)*
  - *EfficientNetB4:* https://keras.io/api/applications/efficientnet/#efficientnetb4-function
    *(Cornell University:* https://arxiv.org/abs/1905.11946*)*
  - *EfficientNetV2S:* https://keras.io/api/applications/efficientnet_v2/#efficientnetv2s-function
    *(Cornell University:* https://arxiv.org/abs/2104.00298*)*
- *Keras guide to transfer learning & fine tuning:* https://keras.io/guides/transfer_learning/
- *File:* https://www.geeksforgeeks.org/file-handling-python/