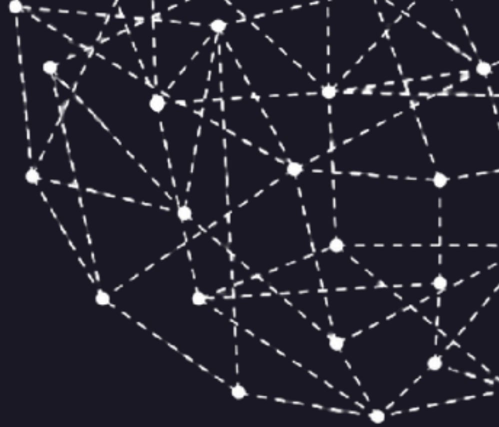
AiCoders



THE TEAM

- Alessio Parato
- Jan Reichle
- Matteo Deò

THE PROBLEM

The assigned problem consists in creating a Neural Network which, through supervised learning, can classify leafs images into two classes.
The assigned images are 96x96 RGB

Class 0 = healthy

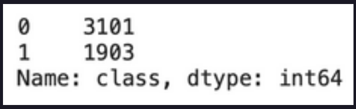
Class 1 = unhealthy

DATA INSPECTION AND CLEANING

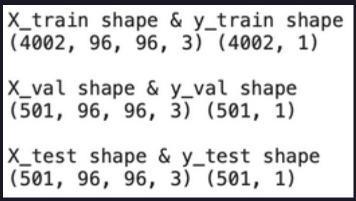
The first thing we did was to normalize the data in [0,1] range, this facilitates convergence during training, speeding up optimization and reducing potential numerical stability issues when calculating gradients.
then we analyzed the dataset provided, containing a total of 5200 images, immediately realizing that some images had nothing to do with the problem, like this one. ⟶
We eliminated them to obtain a dataset with only relevant images for the training of the NN.

PROCESS DATA

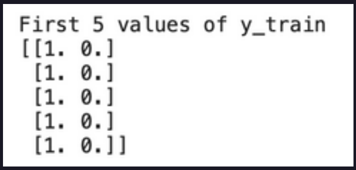
The next step was to visualize the distribution of the images, to do this we used a Pandas Dataframe and it showed us that of the remaining 5004 images of the (clean) dataset, 3001 belong to class 0, 1903 belong to class 1.
This allows us to conclude that the dataset is not perfectly balanced and some techniques, such as k-fold or data augmentation, could be useful to increase the dataset for training, but in any case it is not very small as a dataset.
Then we split the dataset into train_val and test set, the latter being 10% of the total, and then further divided the training set and validation set, leaving the validation set as large as the test set.
The training set will be the largest one that the model will use to train.
We use the validation set during training to see that performance improves in general and not just on the training set (to avoid possible overfitting).
The test set will eventually be used to make inference with the trained model on data it has not yet seen.
Then to have the labels in one-hot encoded format, we used the keras function to_categorical().
This is useful for models that have a softmax output function.



```
0    3101
1    1903
Name: class, dtype: int64
```



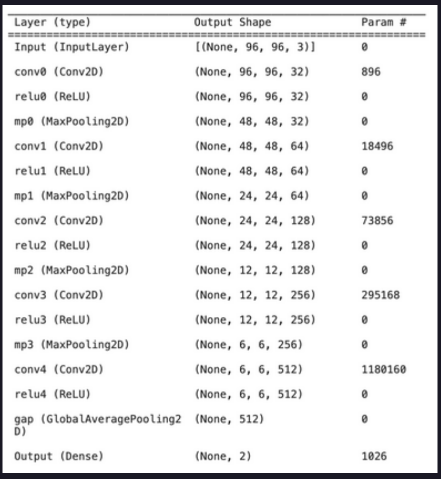
```
X_train shape & y_train shape
(4002, 96, 96, 3) (4002, 1)

X_val shape & y_val shape
(501, 96, 96, 3) (501, 1)

X_test shape & y_test shape
(501, 96, 96, 3) (501, 1)
```



```
First 5 values of y_train
[[1. 0.]
 [1. 0.]
 [1. 0.]
 [1. 0.]
 [1. 0.]]
```

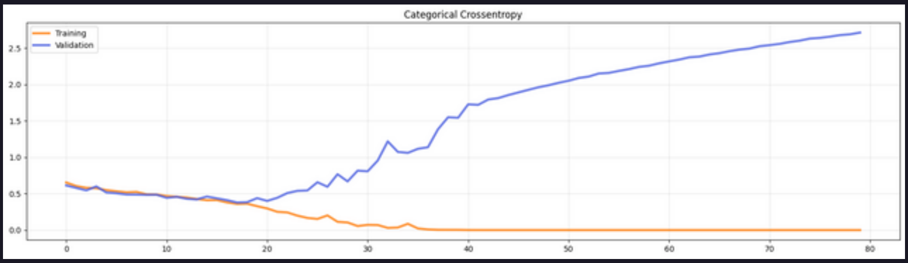
CUSTOM CNN

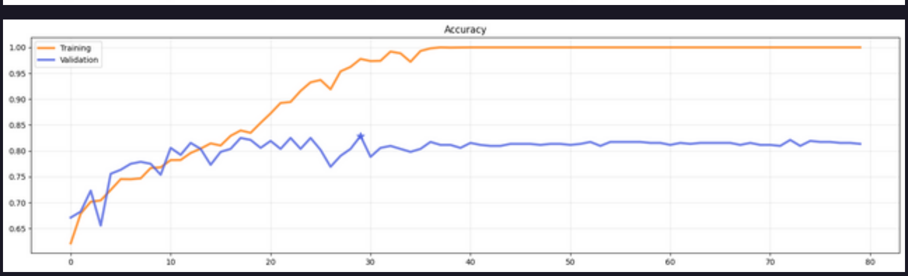
As a first attempt we implemented the CNN seen in class, composed of an input layer, 4x (Convolutional with Relu activation functions and MaxPooling layers), and finally, a convolutional and a Global average Pooling layer and an output layer with two neurons, since we are using the softmax as activation function.
The kernel size is initially set = 3 and padding = 'same' so that the output activations are of the same size as the input image.
For the output neurons we tried both the sigmoid and the softmax function, the latter gave us slightly better results.
As callbacks we initially only used early stopping with patience = 50 and restore_best_weights = True to make sure to save the best weights during training.
As an optimizer we initially used the classic Adam.
The batch size = 50 and 500 epochs.
BinaryCrossentropy as loss function to minimize.



```
Layer (type)                Output Shape              Param #
=================================================================
Input (InputLayer)          [(None, 96, 96, 3)]       0

conv0 (Conv2D)              (None, 96, 96, 32)        896

relu0 (ReLU)                (None, 96, 96, 32)        0

mp0 (MaxPooling2D)          (None, 48, 48, 32)        0

conv1 (Conv2D)              (None, 48, 48, 64)        18496

relu1 (ReLU)                (None, 48, 48, 64)        0

mp1 (MaxPooling2D)          (None, 24, 24, 64)        0

conv2 (Conv2D)              (None, 24, 24, 128)       73856

relu2 (ReLU)                (None, 24, 24, 128)       0

mp2 (MaxPooling2D)          (None, 12, 12, 128)       0

conv3 (Conv2D)              (None, 12, 12, 256)       295168

relu3 (ReLU)                (None, 12, 12, 256)       0

mp3 (MaxPooling2D)          (None, 6, 6, 256)         0

conv4 (Conv2D)              (None, 6, 6, 512)         1180160

relu4 (ReLU)                (None, 6, 6, 512)         0

gap (GlobalAveragePooling2  (None, 512)               0
D)

Output (Dense)              (None, 2)                 1026
```

As it can be seen from the graph, the accuracy reaches a plateau, and the best value is reached before the 30th epoch. This suggested that we could decrease the patience to 20.
Furthermore, from the accuracy graph a slight overfitting can be noticed, in fact, the training set reaches accuracy values close to 100% while the validation set ones are around 80%. This suggested us the use of some regularization techniques and further attempts with variations in the architecture of the model.

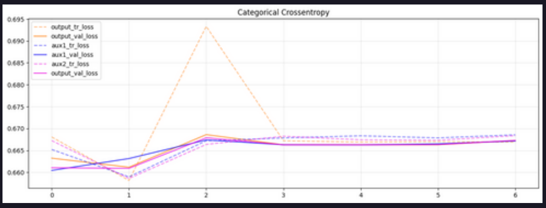


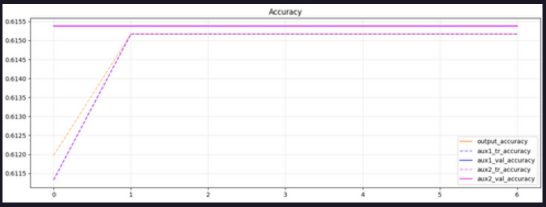


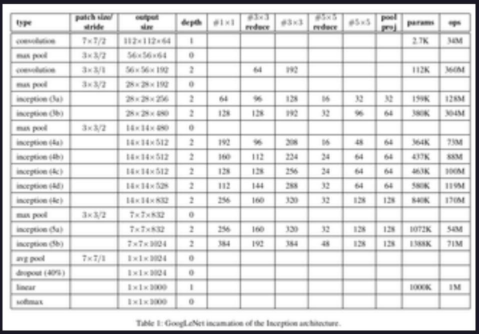
As a first modification we tried to eliminate the last Conv and Max pooling layers (conv4, relu4 and mp3) in order to see if 512 channels of the final relu4 were too many for our problem. We got more or less the same result as before.
Furthermore, with the input images shape being 96x96, we tried increasing the dimensions of kernel_size to 5 and then further to 9, to see if we could capture larger patterns in the input images. Unfortunately the accuracy worsened slightly.
So the initial model was ultimately chosen, but with the addition of the dynamic learning rate and the introduction of drop out to resolve overfitting.
In practice, we have implemented drop out layers after the Max pooling layers, their function is to turn off the neuron with a specified probability ensuring that the network does not adapt too much, helping its generalization. We tried various drop_out_rates between 0.17 and 0.35 observing an improvement also when the last drop out layer (after mp3) was removed.
As a last change we tried to use AdamW as an optimizer in order to try to adjust the learning rate manually (with values ranging from e-3 to e-4) and use weight_decay (values around 5*e-4) to decrease the contribution of larger weights and further prevent overfitting.

GOOGLENET

As another attempt, we tried to implement the GoogLeNet as seen in class, following the specifications present in "Going deeper with convolutions" 2015's paper(https://ieeexplore.ieee.org/document/7298594), implementing first the Inception building block, to then stack 9 layers together interwoven with occasional max pool layers.
To be noted is the presence of auxiliary outputs branching early from the network, there to mitigate the dying neurons problem.
The scarce result shown during its training, including with preprocessed input and dynamic learning rate, suggested that such deep network is not apt for the classification images smaller than the original reported use case (256x256).

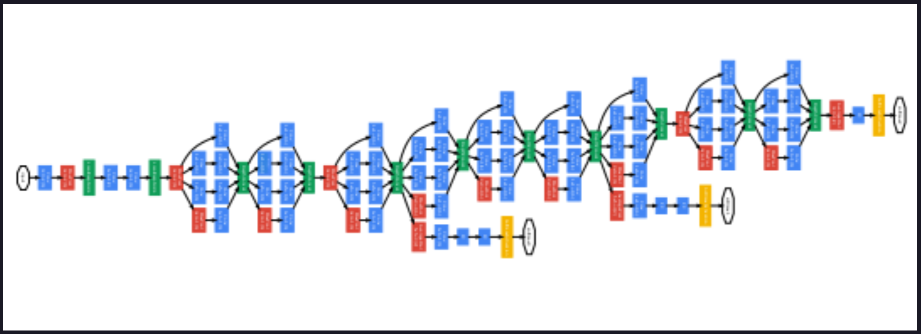






type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Table 1: GoogLeNet incarnation of the Inception architecture.



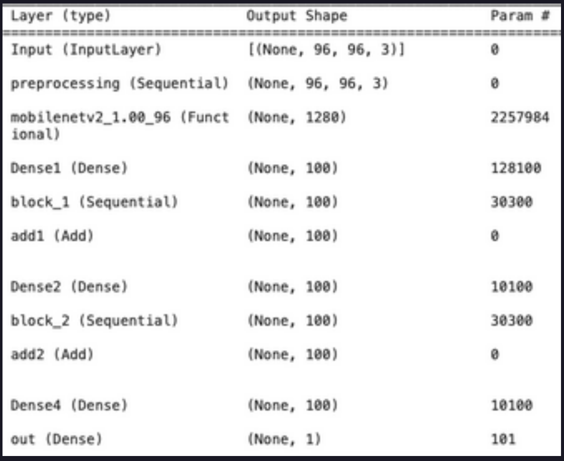
MOBILENETV2

As second architecture we tried to use MobileNet with a single dense layer with two neurons appended. As an activation function we used the softmax-function. Thanks to transfer learning we can deploy this network already trained by Google and later adapt it to our needs later with Fine tuning.
We use the preprocess_input that will scale input pixels between -1 and 1 as required from the documentation.
Initially, the classic Adam and the MobileNet were tested with all the layers frozen to see how the network initially behaved; then, in the fine tuning phase, we tried keeping frozen only up to the 133rd layer, to then up to the 125th one. This has brought about an improvement as the final part of the network is able to "focus" better on the problem under consideration.

After these tests, we tried to maintain the fine tuning with the first 125 layers blocked, to then adding AdamW, trying different values, and finally we added the l2 regularizer, together with the drop out layer. We tried different values to see which ones were best for us
A test was also carried out by increasing the dataset using as transformations those that seemed most sensible for the problem: horizontal flip, 20% rotation and 20% translation. As can be seen from the graphs, the training is more "noisy" as expected, however significantly better results were not achieved.

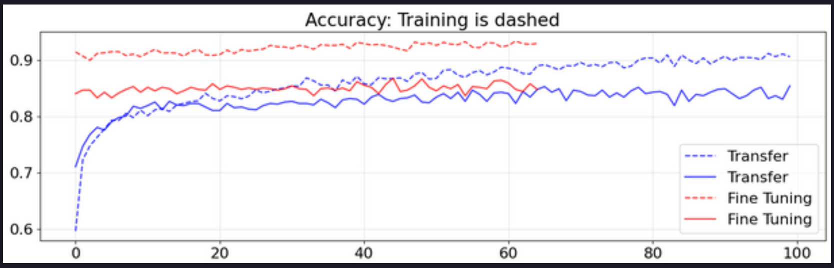
MOBILENETV2 WITH CLASSIFIER

Our next step was to use again MobileNet model with only one output and a sigmoid function in the last layer. This is possible since it is a binary classification task.



```
Layer (type)                 Output Shape              Param #
==============================================================
Input (InputLayer)           [(None, 96, 96, 3)]       0

preprocessing (Sequential)   (None, 96, 96, 3)         0

mobilenetv2_1.00_96 (Funct   (None, 1280)              2257984
ional)

Dense1 (Dense)               (None, 100)               128100

block_1 (Sequential)         (None, 100)               30300

add1 (Add)                   (None, 100)               0

Dense2 (Dense)               (None, 100)               10100

block_2 (Sequential)         (None, 100)               30300

add2 (Add)                   (None, 100)               0

Dense4 (Dense)               (None, 100)               10100

out (Dense)                  (None, 1)                 101
```

As the classifier we created a relatively deep model with ten dense layers. To avoid overfitting, we also used 6 Dropout-Layers with a dropoutrate of 1/6. On the left, the model specification can be seen, in which block_1 and block_2 are significant, each consisting of 3 Dense and 3 Dropout layers. The two Add layers are also worth mentioning, as we wanted the model to be able to access more calculated information in order to classify more accurately and generally.
Since the amount of data is contained, we used image augmentation right from the start. First, we only used RandomFlip and RandomTranslation-layers.
At the first Fit we trained the model with the Adam optimizer for 100 epochs with a freezed MobileNetV2. Then we unfreezed from the 133rd layer to the 154th one. In the fine tuning phase we used early stopping with patience=20 and restore_best_weights=True to avoid overfitting.



In the graphs on the right we can see an increase of accuracy on the training set to 92% and on the validation set to about 85%.
To get a better accuracy we decided to add a dense layer and a block of dense and dropout layers. After multiple training and fine tuning sessions the model finally overfitted. The test accuracy dropped down to 72%. We tried to restore its generality by adding a dropout layer after the MobileNetV2, though to no positive results.

In the third Fit we added a RandomBrightness layer and a concatenate layer to combine what we had already learnt and use it again. The concatenate layer was followed by a new block, but this time the dense layers contained a L2-Regularization with its parameter set to 2e-6. The validation accuracy increased up to 85%, with also the test accuracy increasing again up to 85%.
Next, we added a RandomZoom layer to be more general. As the optimizer we now used AdamW. In this situation AdamW performed better than Adam. When the accuracy didn't improve, we tried a manual learning rate=1e-5 and weight_decay=5e-4. Unfortunately this didn't help us and the validation and test accuracy stuck at 85%.
As the last change we added a RandomRotation layer and another concatenate layer. The concatenate layer also was followed by a new block with L2-Regularization with the parameter set to 2e-5 in the dense layers. Even with repeated training (with Adam or AdamW as optimizer), we were unable to achieve an accuracy over 85% in the validation or test set.

Finally we tried to use data augmentation also on the test set and we observed the results with:

- not modified test set
- horizontally flipped test set
- vertically flipped test set
- right-down shifted test set
- right-upper shifted test set
- left-down shifted test set
- left-upper shifted test set

We also calculated the mean of the first three predictions, the mean of the shifted predictions and the mean of all predictions. The accuracy didn't increase significantly by using modified data or taking the mean of all. The differences were between 1 and 2 percent.