AN2DL Challenge 1

Francesco Colabene, Andrea Gerosa, Christian Lisi, Giulio Saulle

1 Introduction

The goal of the project was to build the best model of a neural network that classifies low-resolution images of different types of leaves into classes indicating their health status. A leaf can be either classified as *healthy* (0) or *unhealthy* (1).

2 First Approach

Input/output handling

Input and output have been handled by adjusting our model to the project request: the output ought to be a tensor of dimension [BS], with BS being a batch size of 2, and the input had to be in accordance with what the inner model stated (e.g. the ConvNeXt accepts a value in the range of [0-255], and then pass it through an internal normalization layer).

Dataset analysis

The initial dataset contained 5200 images, 3199 of which were classified as *healthy* and 2001 were classified as *unhealthy*. Each one of them was a square image 96 pixels wide and 96 pixels high, for each of the three RGB channels (shape is 96x96x3).

Cleanup

By extracting the images from the dataset and by doing a quick inspection, we identified some outliers. We removed those images and their labels from the dataset since they could harm the training and the learning of the network. We also noticed duplicates in the dataset and we were able to identify and remove them by hashing the images. We ended up with 4850 images: 3060 labeled as *healthy* and 1790 labeled as *unhealthy*.

Solve imbalance

The next step was to tackle the problem of imbalance of the cleaned dataset. We considered 3 different techniques that allowed us to rebalance the dataset.

1. **Oversampling** increases the number of instances of the under-represented class or classes in the training dataset. This can be achieved by duplicating existing examples from the minority class or by generating synthetic examples using various methods. We ended up using a mixed approach aimed at duplicating 540 random images from the minority class and applying some light geometric transformations to them. The goal was to experiment and build a very simplified version of the more complex **Synthetic Minority Oversampling Technique (SMOTE)**, hoping to improve our results. Each of the randomly chosen images has thus been augmented and doubled. At the end of the process, the resulting dataset had almost the same number of healthy and unhealthy images.

2. **Undersampling** reduces the number of instances in the over-represented class or classes to achieve a more balanced distribution.

3. **Class Weights** is a technique that defines weights for each class. Higher weights are assigned to the minority class and vice versa. In this way, we can give more importance to the minority class.

In the end, we opted for the Class Weights technique, which was the easiest to implement and gave us good results, while we discarded the simplified SMOTE because we didn't notice reasonable improvements in the early results when using an augmented dataset. Deeper augmentation was later implemented at training time.

Sets

When deciding how to divide the dataset into train, validation, and test sets, we tried a plethora of values, ranging from (0.7, 0.15, 0.15) to (0.9, 0.05, 0.05). We also tried to remove the test set in order to use every image of the provided dataset, but the results were practically identical to the ones with the three divisions. The best results came with values (0.8, 0.1, 0.1).

3 "From Scratch" CNN

We started by hand-crafting a standard CNN and then we iteratively made some improvements over it to obtain better results. The first implemented FEN had 2 blocks, each containing a series of 2 convolutional layers followed by a max pooling one, followed by a GAP; the FC part consisted only of one dense layer and the two neurons output layer. We adopted the Early Stopping callback since the start, initially with higher values to spot the overfitting, and

we gradually reduced it to save computation time. We explored different paths: increasing the number of convolutional layers, increasing the number of filters, and increasing the number of convolutional + max pooling blocks. While the first implementation and its variants reached a decent accuracy on the test set (75% to 80%), it didn't perform well on the CodaLab test set ($< 0.45\%$). So we deepened the net: we added an inception layer after the 2 blocks of 3 convolutional + max pooling layers. The inception layer consists of 5 branches: 4 convolutions (1x1, 3x3, 5x5, 7x7) and a max pooling one. We added 1x1 convolution in series in order to reduce the number of parameters and we concatenated the branches at the end. This increased the accuracy of the network, but the best results came by adding data augmentation (0.62%) and adding more FC layers at the end and a LeakyReLU layer after the GAP (0.67%).

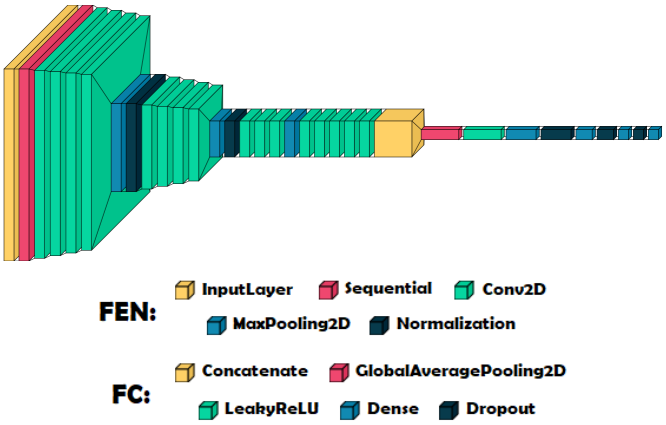

Figure 1: Final custom CNN

4 Transfer Learning and Fine-Tuning

After assessing the potential of our hand-crafted model, we concluded that we needed to adopt more advanced techniques in order to improve the accuracy of our network. The natural evolution was to do Transfer Learning with established pre-trained models to save computational resources and reduce overfitting, together with Fine-Tuning to adapt the model's knowledge to the task at hand.

Test on different models

We then proceed to experiment with different models following the typical training workflow:

1. We instantiate a base model and load pre-trained weights into it
2. We then attached our classification layers to the base model previously imported
3. We train our new network on the dataset, leaving the weights of the base model untouched
4. Lastly, we select a number of layers of the base model to unfreeze and compute our last round of training

The models that have been employed are four: **MobileNet** was the starting point, we experimented with Transfer Learning and Fine-Tuning, but decided to upgrade the inner model. **EfficientNet** (B2,B3,B4) was chosen because of the low number of parameters and quite good accuracy based on the Keras documentation. **Xception** was mostly used for trying out how the augmentation and hyperparameters setting would affect our local accuracy, given the smaller number of parameters this was feasible enough to do the testing more rapidly. **ConvNeXtBase** is the best one out of all, having a high number of parameters and a normalization layer included.
From the comparison of the empirical results, we found that an increase in the complexity of the base model translated to better performance and network behavior during the training phase.

MobileNet	0.75
EfficientNet	0.78
Xception	0.78
ConvNeXt	0.89

The ConvNeXt allowed us to achieve the best scores, so we decided to further develop the network, trying to achieve an even better result.

Augmentation Choices

We experimented on the augmentation methods, and we determined that the techniques that had the most influence on our dataset were the following:

Horizontal Flip	True
Vertical Flip	True
RandomRotation	0.2
RandomZoom	0.2
Contrast	0.2
Brightness	0.2

We also added a bit of noise to our training set. We found that the model can generalize better the test set when the noise is applied during the training, giving us better results on the platform.

Following the TA suggestion, we applied the RandAugment provided by **KerasCV** only on one of our last models, effectively increasing the accuracy. This augmentation technique provides a set of 10 different transformations applied directly on our test set, that involves: *AutoContrast*, *Equalize*, *Solarize*, *RandomColorJitter*, *RandomContrast*, *RandomBrightness*, *ShearX*, *ShearY*, *TranslateX*, *TranslateY* applied randomly on each image.

5 Final Model + Hyperparams tuning

Layer (type)	Output Shape	Param #
Input (InputLayer)	[(None, 96, 96, 3)]	0
data_aug (Sequential)	(None, 96, 96, 3)	0
convnext_base (Functional)	(None, 1024)	87566464
leaky_re_lu (LeakyReLU)	(None, 1024)	0
dense1 (Dense)	(None, 1000)	1025000
dropout (Dropout)	(None, 1000)	0
dense2 (Dense)	(None, 550)	550550
dropout_1 (Dropout)	(None, 550)	0
dense3 (Dense)	(None, 205)	112955
dropout_2 (Dropout)	(None, 205)	0
dense (Dense)	(None, 2)	412

Total params: 89255381 (340.48 MB)
Trainable params: 89255381 (340.48 MB)
Non-trainable params: 0 (0.00 Byte)

Figure 2: Final ConvNeXt model summary

We tested the effects of various hyperparameters on overall performance, and we found that the sweet spot was:

- **Batch size** : 128
- **Epoch** : 400
  We left a high number because it's not that relevant since Early stopping will kick in much sooner.
- **Learning rate (with Adam)** : 1e-3→3e-4→1e-5
  The learning rate decreases as the training goes on, from Transfer Learning to Fine-Tuning. The initial value is higher because we need to train the classifier from scratch, and slightly modify the weights inside the FEN to better learn our problem.
- **Early Stopping patience** : 35→20→10
  Just like the learning rate, the patience decreases between phases in order to reduce the computation time.
- **Reduced learning rate on plateau** :
  Patience : 4-5
  Factor : 0.95
  Minimum LR : 5e-4 →5e-5 →1e-6
- **Number of layers frozen** : 152→0
  Our initial value for this parameter was 278, but as the experiment progressed, we were able to decrease it, and finally we settled on 152, achieving better results. The second Fine-Tuning training didn't have any frozen layer.
- **Number of dense nodes in the classifier** : 1000→550→205
  Originally, the values for the classifier were smaller. Then we tried increasing the complexity of the classifier, and in doing so the local accuracy increased. Going even beyond these values doesn't seem to be beneficial for our model.
- **Dropout rate** : 0.2
- **Regularizer** : we used an L2 regularizer (Ridge Regression) with a factor 2e-6.
- **GELU** : the activation function used is the Gaussian Error Linear Unit (or GELU), which gives back a better gradient and permits it to be more effective in learning complex patterns in data.
- **Sigmoid** : it has been used in order to get a binary guessing as output.
- **Label smoothing** : 0.15
  Increased the accuracy of the final net by 1/2%.

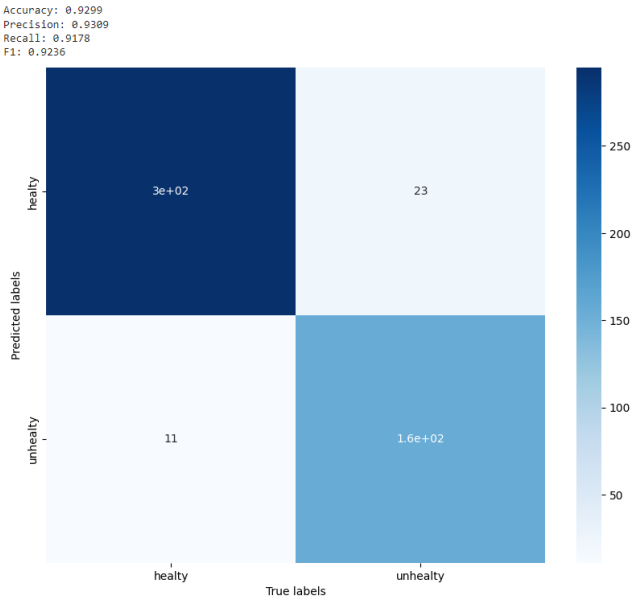

Figure 3: Confusion matrix

6 Conclusions

In the second phase of the competition, we tested again our models, achieving the following final score:

Model	Accuracy-value
ConvNeXt	0.8770

During the competition, we developed a lot of different models, with small and big differences among all the members. We previously described the parameters of the best-performing one, but other models performed just as nicely as that one or even better in the first phase, or vice-versa. For this reason, in addition to the notebook of the best performing one, we included the Custom CNN one and one more that reached similar results with different hyperparameter settings.

Contribution

	Hyperparams tuning	Experiments on augmentation	CustomNet	MobileNet	EfficientNet	Xception	NasNet	ConvNext
Saulle	✓	✓				✓	✓	✓
Gerosa	✓	✓			✓			✓
Colabene	✓	✓	✓			✓		✓
Lisi	✓	✓		✓		✓		✓