Plants Binary Classification Report

Artificial Neural Networks and Deep Learning - a.y. 2023/2024

Michael Fiano
Codalab Name: choil
Team Name: MicGPT
10676595 - 232790

Francesco Palumbo
Codalab Name: frapalu
Team Name: MicGPT
10711027 - 221919

Luca Negri
Codalab Name: lucanegri17
Team Name: MicGPT
10674684 - 222048

Federico Lamperti
Codalab Name: fedelampe11
Team Name: MicGPT
10680961 - 233462

***Abstract*—The goal of this challenge was to correctly classify images of plants into 2 classes, 'Healthy' and 'Unhealthy'. In order to face this problem and to learn as much as possible, our team decided to organize the workflow, doing things step-by-step. In this report we discuss, explore and compare our development and design process.**

I. Dataset Analysis

A. Original Dataset Characteristics

The dataset was made of 5200 samples, the image and the associated label. The image format was 96x96 and encoded using RGB, thus having 3 channels. Furthermore, we were able to discover some dataset's characteristics:

- 3199 samples were healthy while 2001 were unhealthy,
- 348 types of duplicates were present,
- 2 different types of outliers were present.

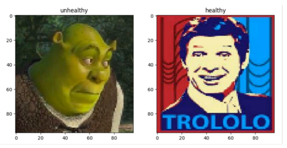

Fig. 1: Outliers identified

Since the outliers could affect the results of the analysis, we decided to remove them, reducing the size of the dataset to 5004 images (each one of them was present 98 times in both classes). In order to balance the difference between the elements of the two classes and to avoid misclassification during the training phase, we introduced a weight for both classes: this allowed us to give higher importance to the unhealthy class. Weights were calculated based on the overall number of samples for each category. We first decided to split the dataset only into training (80 %) and validation (20%), but then we changed to a 60-20-20 split to be able to test the models on never seen data.

B. Data Augmentation

We decided to include data augmentation techniques to improve the overall model performance, accuracy and to make the model more robust. Different techniques were applied at first, starting with an augmentation algorithm on the entire dataset (as soon as the outliers were removed), then we proposed a partial augmentation to be applied only to the 'unhealthy' class, with the aim of balancing the dataset. As final technique, we proposed our best performing one which consists in adding a preprocessing layer within the model, aiming at translating, zooming, rotating and flipping our images. During this process, we observed that applying brightness and contrast led to under-performance in our models. Random brightness in particular produced the worst outcome, in fact, upon reviewing the confusion matrix, we noticed that the model predicted all elements as unhealthy.

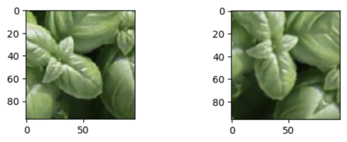

Fig. 2: Side by side view of an original image with its rotation

II. Hand Crafted CNN Model

Initially, we manipulated various CNN models shown during the lab sessions. In particular, we achieved the best outcomes by using a model with a sequence of five convolutions, with an increasing number of filters (32, 64, 128, 256, 512) each followed by a max-pooling layer. In order to avoid over-fitting, we added 3 dropout layers with a rate of 0.25 inside the model. We also implemented early stopping and a learning rate scheduler during the training process. Attempts to add more layers, change the activation functions of different layers, or alter the optimizer did not lead to any improvement.

Evolution	Local_Test_Accuracy
Base Model	0.80
Outliers Deletion	0.82
Data Augmentation	0.82

Although those result could seems to be good, after we submitted in the Codalab platform we reached a worse score, with a drop in the performance of 10-15%, at this point we decided to proceed with more sophisticated models and techniques.

III. Transfer Learning and Fine Tuning

We started using some of the most famous CNN architectures examined during the lectures through Transfer Learning, followed by a Global Average Pooling and a Dense Sigmoid Layer as the simplest classifier. We will report for the sake of brevity only the three models on which we spent most of our time.

Net	Codalab_Test_Accuracy
VGG16	0.78
EfficientNetB5	0.89
ConvNextLarge	0.92

Starting from these basic configurations, we wanted to optimize and then compare the different models results.

A. VGG16

VGG was the first pre-trained model we tried, and it was our most tested one. It served as the baseline for comparing different techniques, such as data augmentation, removing outliers and balancing classes. With this model, we aimed to become familiar with these different techniques by mixing them up. The accuracies obtained varied according to the specific configuration. We proceeded through trial and error to acquire wisdom with layers' structure and also began experimenting by changing the different parameters and using different optimizers. For example, we realized that using the ImageDataGenerator class from Keras was not optimal in our case. Moreover adding a ResizeLayer in order to manage the input size of the images wasn't bringing benefits. Thanks to this model, we gained a better understanding of what we were exactly doing and decided on how to approach the next more complex models.

Evolution	Validation_Accuracy
Base Model	0.81
Augmentation	0.81
Augmentation + class weight	0.80
Fine Tuning (15 frozen)	0.86

B. EfficientNetB5

After training several Keras pre-trained models, we noticed improvements in performance when predicting our validation and test samples with EfficientNetB5. In the Transfer Learning phase we re-trained this model by adding the pre-processing layer for augmentation, followed by the EfficientNet pre-process of the input. As final layers, instead, we proposed a Dropout-Dense-Dropout structure before the output, in order to prevent over-fitting. The Fine Tuning phase, instead, was implemented by freezing the first 397 layers out of 456: this provided us with a better performance with respect to the first version with 250 frozen layers. Surprinsigly, we noted that the partial augmentation technique, applied to the unhealthy class, achieved even better results if combined with the class weights integration.

Evolution	Validation_Accuracy
250 frozen layers	0.80
397 frozen layers	0.84
Partial augmentation	0.89

C. ConvNextLarge

ConvNextLarge is a very powerful and complex model capable of reaching a very high accuracy score on the training set. Having the knowledge gained training previous models, we removed all the outliers, added a weight to balance classes (better outcomes than performing augmentation only on unhealthy class) and performed data augmentation by flipping and rotating images, then we used all the methods to avoid overfitting as much as possible by implementing early stopping, learning rate scheduling and two different dropout layers. The model contained a pre-processing layer to perform data augmentation as above described and two dropout layers with rate respectively 0.5 and 0.15 with a dense layer in the middle with 256 units. Using this structure we achieved a score of 0.91 after performing fine tuning. Adding also a random translation in the pre-processing allowed us to reach 0.92. We tried to change something in the net as dropouts' rate, the number of units in the dense layer or the number of frozen layers during fine tuning, but unfortunately they did not improved our effectiveness.

Evolution	Validation_Accuracy
Dropout+Dense+Dropout	0.89
Fine Tuning	0.96
Adding translation	0.90

IV. Additional techniques

Before moving to the detailed description of our best model, it's convenient to highlight some practices that have been adopted in order to improve the performances and the training speed of our architectures. Firstly, the global policy has been set to 'mixed-float16' precision, that considerably sped up the training phase, leading us to 4 seconds per epoch. Furthermore, it has been used the Test Time Augmentation in the prediction function of the model, which slightly improved our accuracy on the local testing set. In order to have a balanced split in training, validation and test sets, we applied a shuffle to the original dataset. In addition to these techniques, it was furthermore implemented an ensemble model putting together 3 of our most performing architectures: ConvNext-Large, ConvNext-XLarge and EfficientNetB5. This ensemble model includes an average layer on top of the three models' outputs, which computes the output as the average of the previous layers' classifications; while the final prediction is computed by a Dense layer with a Softmax activation. This model reached a validation accuracy of 0.9698, a test accuracy of 0.9859, and a final accuracy on the evaluation test set of 0.84, classifying itself as our second best model.

V. Final Submission Model

A. Model

The model we selected for submission during the final phase is based on the ConvNext-Large network. While it wasn't our top-performing model in local testing, it outperformed all others when assessed on the Codalab platform. Leveraging the knowledge obtained during the challenge and last explorations in adjusting hyperparameters, we achieved our best result, composed as:

- Batch size = 16
- Frozen layers = 252
- Loss = Binary Crossentropy
- Optimizer = AdamW
- Activation of last layer = sigmoid

Layer (type)	Output Shape	Param #
Input	(None, 96, 96, 3)	0
Preprocessing	(None, 96, 96, 3)	0
convnext_large	(None, 3, 3, 1536)	196230336
global_average_pooling2d	(None, 1536)	0
Dropout	(None, 1536)	0
Dense	(None, 256)	393472
Dropout	(None, 1536)	0
Dense	(None, 2)	514

Total params: 196,624,322
Trainable params: 71,535,362
Non-trainable params: 125,088,960

B. Local Performance

We trained the model using a 60-20-20 dataset split, implemented early stopping with a patience of 12 based on validation accuracy and a learning rate scheduler with a patience of 5 on validation loss, along with class weights for dataset balancing. The model achieved its best value at epoch 20.

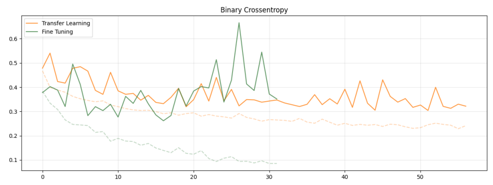

Fig. 3: Binary Crossentropy during training

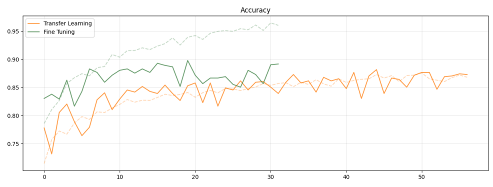

Fig. 4: Accuracy during training

Subsequently, we computed the confusion matrix to evaluate its performance, resulting in the following metrics:

- Accuracy: 0.9021
- Precision: 0.899
- Recall: 0.8921
- F1: 0.8953

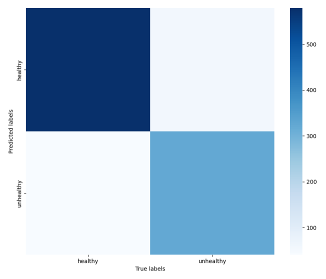

Fig. 5: Confusion Matrix on the Local Testing Set

C. Leaderboard Performance

Development Phase Accuracy: 92%
Final Phase Accuracy: 86,40%

VI. Contributions

At the beginning of the challenge we decided to work together in order to familiarize with the platform and all the new knowledge gained during the lectures since no one had previous experiences. When we decided to test the different models of Keras Applications we split the work equally, everyone tested different nets and shared the results including operations that lead to improvements and things that worsen the performance.