A2NDL - Homework 1 - NeuroNauti

Objective

The objective of the first Homework is to build and train a convolutional neural network able to classify pictures of "healthy" and "unhealthy" leaves. The homework material included a set of images, each labeled accordingly.
The trained model is then validated against a dataset (hidden to us) via the CodaLab interface.

Dataset Analysis and preprocess

We started our work by carefully analyzing the provided dataset structure and content.
The dataset contains 5200 images, each of them labeled as either “healthy” or “unhealthy” and the total number of images in each category is:

- 3199 for the “healthy” category
- 2001 for the “unhealthy” category

Out of them we decided to remove 348 images corresponding to 2 different groups:

- Duplicates: removed to avoid having the same images in both test and validation datasets
- Anomalies: images that did not contain plants, and consequently, deemed not pertinent to the challenge.

After this data cleaning phase we partitioned the remaining images into three sets, employing a stratified approach based on the distribution of labels, with a 60%/20%/20% split for Training, Validation and Test set respectively. Since the classes are unbalanced, we took in consideration the weights for each of them during the training phase to develop an accurate model.
The preprocessing code is contained in the `dataset_to_folder.ipynb` notebook.

Data Augmentation

In order to increase the overall quantity of data for better training, various techniques of data augmentation have been used. We proceeded in steps, each time refining the process and improving the quality of augmented data:

- Initially, the images were statically (but randomly, in each training session) zoomed in by a factor that was later fixed to 20%
- Later on, random image rotation, flipping, and brightness were added; this proved to be crucial to prevent overfitting in the model at train time. At the same time, random shear, contrast, and cropping were removed as they turned out to be poisoning the data without helping us in our goal
- Finally, we were able to augment data in the training set via MixUp and CutUp techniques; this further allowed us to reduce overfitting

Overall, data augmentation has been a significant step in the development process as it allowed us to reach a level of accuracy that we would not be able to otherwise achieve.

Approaches

In order to tackle the problem and find a solution, we decided to split the group into two sub-groups, each of them trying a different approach.
We later joined back to cooperate on the solution that better achieved our goal.

Approach 1: Custom CNN Architecture

The first approach consisted in trying to build a custom architecture, leveraging both dense and convolutional layers.
Initially, the network hidden layer was composed by two components:

- a set of convolutional layers, each followed by a global pooling layer

- a set of dense layers

Both the convolutional and the dense layers are activated via the "ReLU" (rectified linear unit) activation function.
The initial idea was to mimic a simpler version of the original AlexNet CNN, which featured 5 convolutional layers, 3 pooling layers, and 3 dense layers; we modified it by:

- reducing the number of convolutional layers and their hyper parameters (lower stride, lower kernel size) to account for the smaller images in our dataset compared to the ones used in the original paper (96x96x3 versus 224x224x3)
- reducing the number of units in the dense layers, since we only needed two categories as output (compared to the 1000 in the original paper).

To further improve the accuracy of the model, we used the module "keras-tuner" to accurately and systematically find the best hyperparameters, including:

- the amount of Weight Regularization to be applied to the model
- the correct dropout for the training of the dense layers
- the ideal amount of data augmentation for the training and the parameters associated to data augmentation

After training the model for a few sessions, each with up to 200 epochs and variable learning rate starting at 0.1, we found ourselves unable to get an accuracy higher than ~80% in local testing and ~75% CodaLab testing, leading us to favoring the other approach that was being developed at the same time.

Approach 2: Transfer Learning and Fine Tuning

The second approach, that later turned out to be the most suited one for our goals, was utilizing transfer learning and fine tuning techniques. These techniques do not require as much data as building a complete new model and so were naturally less affected by the small size of our database.
Initially we tried to use a variety of different models made available by keras like "VGG16", "EfficientNetB3" and "ResNet50V2", we imported each model without its final fully connected layers and attached our own final layers that we then trained while freezing the keras base model.
The resulting models were then fine tuned with a much lower learning rate with no frozen layers.
All the models we tried gave good results with accuracies of more than 80% but the best one was by far the "ConvNeXtLarge" model which we decided to use moving forward.
To complete the Fine Tuning process, we unfroze all the layers of the base model except the batch normalization ones (as pointed out in the TensorFlow documentation about Fine Tuning).
The early experiments with this approach clearly showed a higher overall accuracy over our datasets (upwards to ~85%) leading us to explore this path rather than the one leveraging the custom architecture path.

Proposed Model

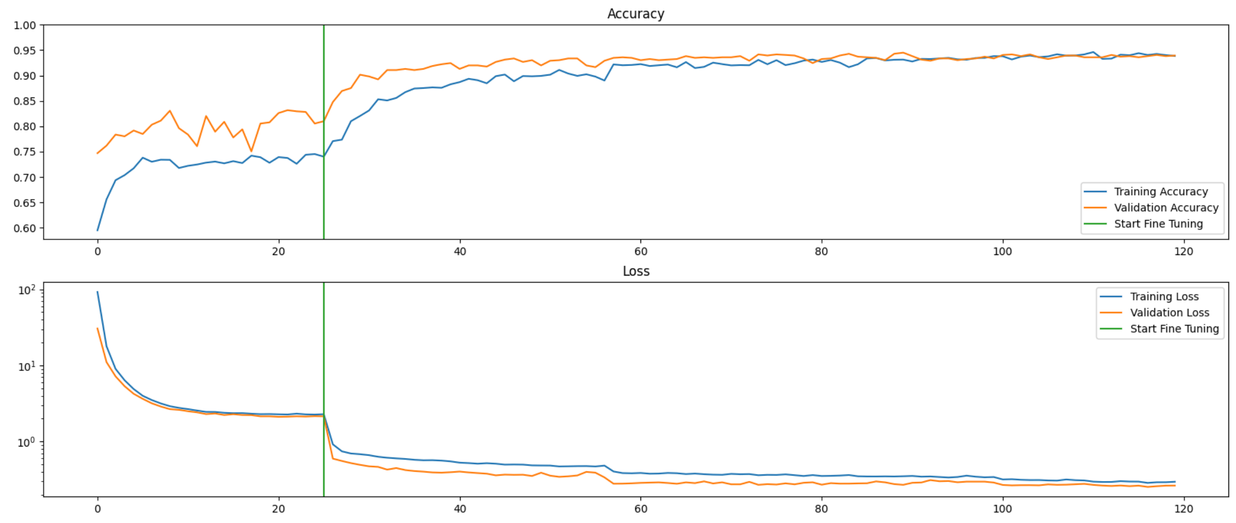
After exploring the two approaches, we started working together on the model that we are proposing; as stated in the previous section, the Transfer Learning method is the one that we chose.
We decided to use the "ConvNextLarge" model for the Transfer Learning part, paired with two dense layers of 1024 units each with ReLU as activation function and a dropout of 0.4 and 0.3 respectively before and after these layers. This approach created a model with an accuracy upwards to ~95% against the local test set and ~91% on the Codalab Dataset.

Training

To speed up the training process, we also introduced callbacks for early stopping and to reduce the learning rate whenever a plateau on the validation value was found.

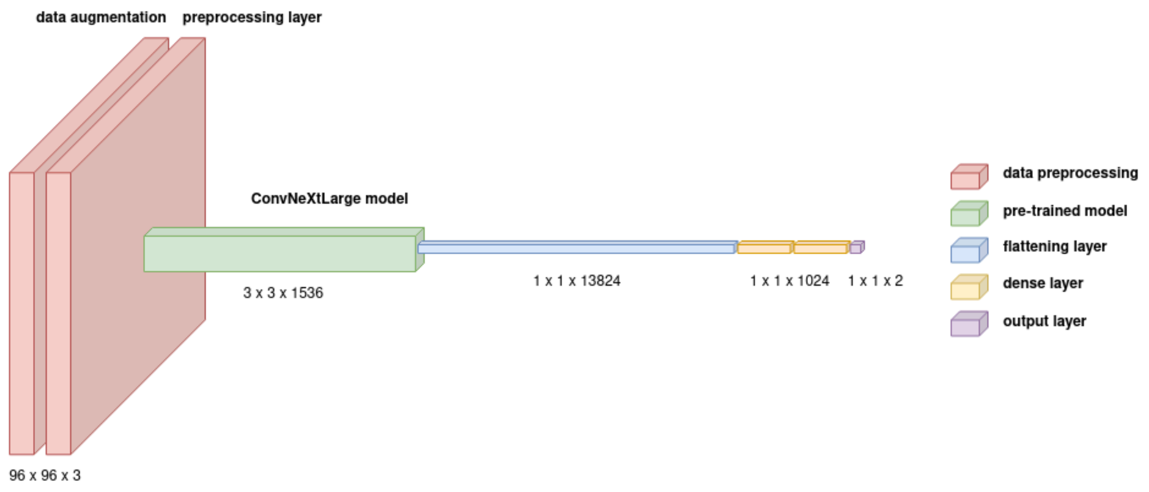
During the Transfer Learning phase, the model was trained for ~200 epochs with a learning rate of 0.001. Fine Tune it, we unfreezed all the layers and trained it for 200 epochs with a starting learning rate of 5e-5. The training produced a model with 211438786 parameters for a total size of ~2.4 GB.

The training part turned out to be quite challenging, as at first we had a significant difference between the local accuracy and the accuracy shown on CodaLab. Furthermore, due to the limited nature of the CodaLab validation (only two submissions were allowed for each person per day), one of the limits of the development phase was trying to develop a satisfying model locally and then choosing the best two to validate on the hidden dataset.
Despite our best efforts, we still introduced some overfitting in the training phase, as the accuracy results on the hidden dataset were slightly lower than the one that we could perform locally. As mentioned before, data augmentation turned out to be fundamental to overcome this problem: tuning it correctly helped us reduce this effect.



System Architecture

The following illustration shows the system architecture.



Other relevant attempts

In order to improve our model we tried a variety of different techniques that did not always work as expected, here we list some of the less fortunate ones that did not make the final implementation:

- Test time augmentation: we tried to improve the final prediction of our model by feeding it multiple copies of the same image, augmented in different simple ways, and then by selecting as result the class predicted the most. Unfortunately it did not produce any noticeable benefits.
- Model ensembles: we tried to use multiple models by removing their output layers and connecting them together with a single dense layer that we then trained.
  We regrettably found that the resulting model was more prone to overfitting than the composing parts.

Who did what

- Lorenzo Rossi: First approach, Creation of the Final Model
- Matteo Beltrante: Second approach, Creation of the Final Model