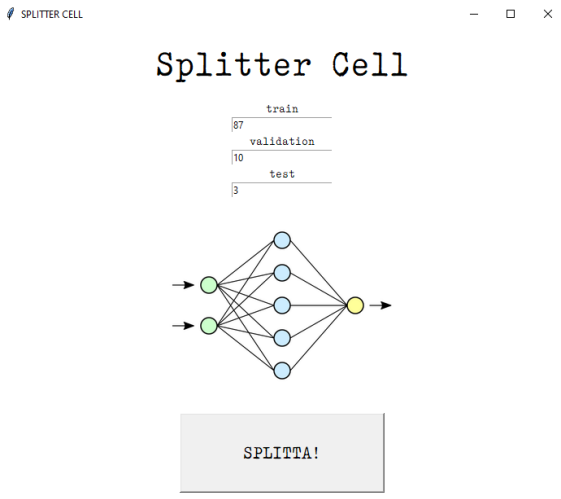
ANNDL 1st CHALLENGE REPORT

Team ColleDellOrso
Alberto Tavini 10578559 - Simone Reale 10616980

Dataset Preparation

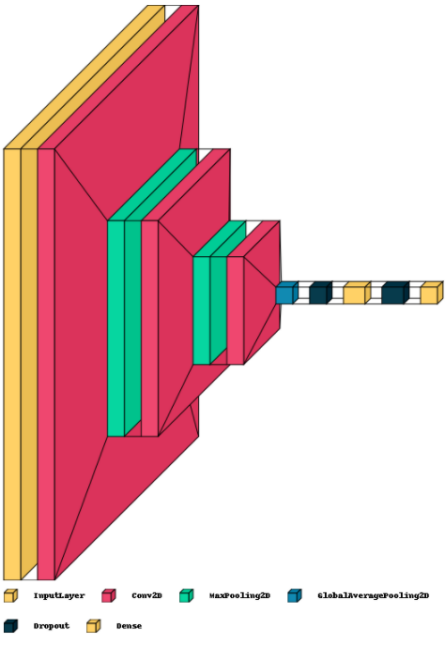
The first thing we did after inspecting the Leaves Dataset was write a program (Splitter Cell) that on our own terminals would split the folders in a fashion that suits the ImageDataGenerator object: it would divide the in three folders “train”, “validation” and “test”, while maintaining the subfolders as they were supposed to be, one for each class. The user can select the percentage he prefers for the proportion of the splitting, and these values are applied on the 14 categories, and not to the entire dataset, in order to maintain class balance. The files are also shuffled to introduce some randomness that could help not overfitting or learning a single pattern when different ones are present for the same plant. These also helped at the end, when we came out with the idea to use *bagging* to ameliorate our performance.



Be careful, our notebooks are written to work with this configuration, and they won’t work if the dataset isn’t split with this procedure.

First Experiments

At the beginning we tried to have a look at the images and to apply some random transformations to them using different parameters for Data Augmentation. Then we tried to write some simple models that would not use preexisting valid architectures, because we wanted to explore the amount of accuracy that could be reached without the use of something known to be powerful. Some trials can be seen from the section “Model Definitions” of the notebook “ANNDL homework 1”. We understood from the first moment that 5 convolutions was too much to learn the structure of the leaves, and the best results were obtained with 3 convolutional blocks. We also saw pretty soon that GlobalAverage Pooling performed better than Flattening as a connection



between the first and the last part of our models, due to its capacity to act as a structural regularizer. We did some variations on models that would integrate three convo blocks (16,32 and 64 filters), the GAP and two or three Dense layers, including the output. We tried different kernel sizes and also to increase strides, but these variations only worsened our results. In this case our accuracy rating on the Submits were not very high, around 0.5. That's why we decided to try some Fine Tuning. The first model we trained was not well thought and it performed only 0.61, after this we repeated the procedure with few variations and after a long training, first with the last part of vgg frozen and then allowed to variate we got our first good result of 0.79 accuracy. This was very similar to the model seen at laboratory number 4, and from this one we started our search for the best model for the task.

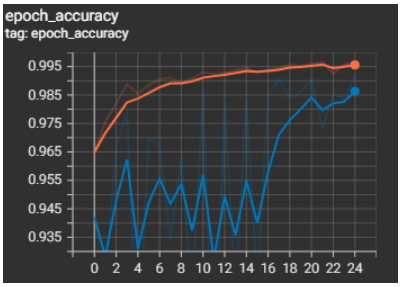
Thoughts on Data Augmentation

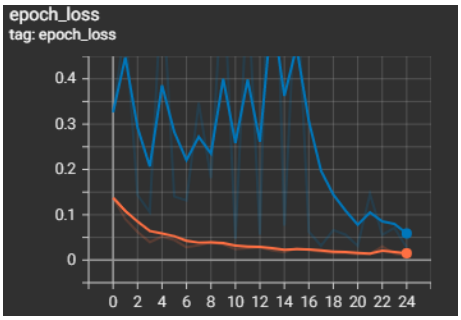
At the beginning we did our data augmentation at random, without thinking much about it, just making the parameters more extreme with respect to the one seen during labs. After a few trials with these we took our time to reflect on it. We first concluded that orientation of the leaves mattered, in particular for certain kinds of leaves, and it was important not to rotate them too much, as it would be if we were learning visages of different individuals. We also saw that the zoom could result in losing parts of the borders of the leaves. So we decided to be gentle to not have too much variance among the batches and let the network learn something consistent. Just a little bit of shift and rotation. We also experimented with zca whitening, but the results didn't justify keeping it. Also we prefered fill mode constant because a lot of the images had black background and we thought reflection wasn't a great fit, as it would be for images of landscapes or aerial photos.

```
aug_train_data_gen = ImageDataGenerator(rotation_range= 50,
                                        #zoom_range=0.1,
                                        #zca_whitening=True,
                                        #zca_epsilon=1e-06,
                                        width_shift_range=0.2,
                                        height_shift_range=0.2,
                                        horizontal_flip=True,
                                        preprocessing_function=preprocess_input,
                                        fill_mode = 'constant'
                                        )
```

Best pure model and Bagging

After a few training sessions with different parameters we found a model that performed very well. This model got a 0.90 and it used VGG, it first trained the Dense part, made by GAP, two dense with respectively 300 and 48 units, dropout after the first one with a 0.4 probability to help avoid overfitting and finally the output layer with 14 units and softmax to determine the posteriors. After this, we unfroze the last convo block to fine tune the network to our specific task (see results in the graphs). We tried to do better with Gaussian Noise Layers with different stddev and Batch Normalization Layers, but the differences were negligible. Some examples of our attempts can be inspected in the different "reduced" notebooks. Another thing we did to help us see through the differences between accuracy on validation and submission time was to implement the f1 metric to the metrics monitored by the





model, but its values didn't differ much from the ones on accuracy, so we didn't persist in using it. We concluded also that more than 25 epochs were not necessary and after them the accuracy would just oscillate back and forth. After this we came up with the idea to increase the performance by applying Bagging to different copies of the same model trained on different random splits of the dataset, keeping 3% of the data out as testing elements. The output is computed from the mean of the confidences of the different models. We tried with 3, 5, 8 and 9 models, and the results with the latter three were pretty much the same. This helped us to obtain 2% more in performance, which is what you would usually expect by such technique. We also wrote a script to estimate the proportion of common data the random splits on which we trained the equivalent models had in the training set, and we got an average of circa 77% among the different classes.

The will to do test time augmentation

Another thing we wanted to implement was test time augmentation. The idea was to help the network become truly invariant with regard to translations and rotations when inspecting new data to be evaluated. Unfortunately it increased too much the time needed for the submissions, and when we tried to implement it in combination with bagging, which already needed a lot of time to execute, our trials failed due to reaching the time limit of the servers.
Nonetheless we saw while running our notebooks that it did help in some misclassifications on the test sets of our splitting. The screen shows the model.py with the code to implement the two techniques described.

```python
import os
import tensorflow as tf
from tensorflow.keras.applications.vgg16 import preprocess_input
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import numpy as np

class model:
    def __init__(self, path):
        self.modelVgg1 = tf.keras.models.load_model(os.path.join(path, 'SubmissionModelVgg1'))
        self.modelVgg2 = tf.keras.models.load_model(os.path.join(path, 'SubmissionModelVgg2'))
        self.modelVgg3 = tf.keras.models.load_model(os.path.join(path, 'SubmissionModelVgg3'))

    def predict(self, X):


        test_generator = ImageDataGenerator(rotation_range= 50,
                                            width_shift_range=0.2,
                                            height_shift_range=0.2,
                                            horizontal_flip=True,
                                            preprocessing_function=preprocess_input,
                                            fill_mode = 'constant' )



        test_data = test_generator.flow(X, shuffle=False)
        tta_steps = 5
        predictions = []

        for i in range(tta_steps):
            results = []
            results.append(self.modelVgg1.predict(test_data))
            results.append(self.modelVgg2.predict(test_data))
            results.append(self.modelVgg3.predict(test_data))
            #sum of all the predictions
            summed = np.sum(results, axis=0)
            predictions.append(summed)

        pred = np.mean(predictions, axis=0)

        res = tf.argmax(pred, axis=-1)

        return res
```

Conclusions

Having reached the final phase of the challenge we think we can be satisfied with our work, in particular because when inspecting some of the images that were commonly misclassified we had difficulties on our own discerning which class they belonged to. The experience helped us also understand that what in theory seems reasonable does not always work in practice, and that every task may have its peculiarities which can only be discovered by trial and error.