AN2DL Challenge 1: Plant Classification
Artificial Neural Networks and Deep Learning - a.y. 2022/2023
Group: Beck'sPropagation

Davide Franchi
10614321
Codalab: DavideFranchi

Daniele Ferracuti
10580840
Codalab: DanieleFerracuti

Marco Ferrero
10613076
Codalab: marcoferro23

***Abstract*—The purpose of the first AN2DL challenge is to create a CNN model that is able to classify images of plants into 8 species.**

1. Introduction

1.1. Database Structure

The given data-set is composed of more than 3500 images with a 96x96 resolution. It is strongly unbalanced, specially on classes "Species1" and "Species6". Due to the low quantity of data and low resolution quality, we faced difficulties to obtain a good set of images to train models. We solved only partially the problem of re-balancing using the function `class_weight` from the `sklearn.utils` library and passing the weights directly to the `model.fit` function, making training more sensitive to the least represented classes.

We divided the given zip file using the `splitfolders` library, creating two different data-set, respectively with and without test set. Dataset with test set has been splitted with a `80/15/5` ratio while the dataset without the test set has a `80/20` ratio. The dataset with test has been used to firstly evaluate the model accuracy, but the more accurate (and submitted) models have been trained using the test-less dataset.

2. Data Processing

2.1. Images Pre-processing

We noticed that the images are very "poor" in color and in many cases not very sharp. For this reason we decided to try and use different combinations of pre-processing functions with arbitrary saturation, contrast and brightness values.

2.1.1. Saturation, Contrast and Brightness. We have modified contrast and saturation using the TensorFlow `tf.image` library with functions `adjust_brightness` and `adjust_saturation`. Brightness adjustment is performed using the `ImageEnhance` library from `PIL`

- **Saturation** : 3.5
- **Contrast** : 1
- **Brightness** : 1.2

Figure 1 shows the effect of this pre-processing.

2.1.2. Transfer Learning Pre-processing. In the pre-trained networks we used for transfer learning we often used `keras.applications` library functions because each model had a specific preprocessing method which needed to be applied to each picture.

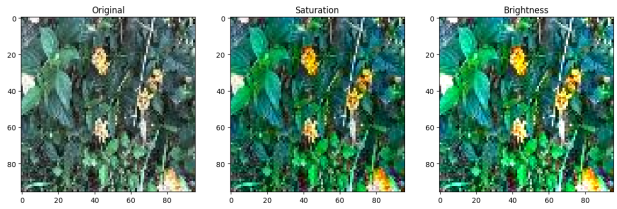

Figure 1. Saturation and brightness preprocessing.

2.2. Data Augmentation

Due to the small number of images provided, it was crucial to practice data augmentation. To obtain a batch of augmented images, we chose to employ `ImageDataGenerator` method in combination with a custom preprocessing function.

ImageDataGenerator parameters	
width_shift_range	50
height_shift_range	50
horizontal_flip	True
vertical_flip	True
fill_mode	'nearest'

In some cases, however, we have noticed that using the classic transformations, training performance decreased. This probably happened because images were deformed too much considering the low resolution of the original ones (e.g. rotation). This resulted in a loss of features in augmented data, which made model accuracy worse.

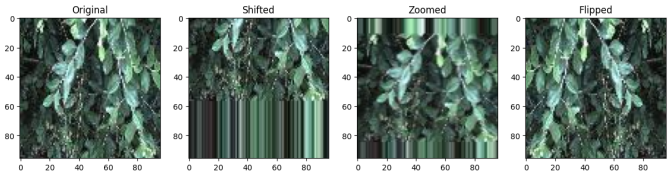

Figure 2. Transformation used for augmentation.

3. Convolutional Neural Network

Our first approach to the challenge was to derive as many models as possible and reason about which one might be the best to focus on. In particular, we initially created ad-hoc CNN modes, using basic 2D convolutions and max pooling, and then, exploiting pre-trained models, we used techniques such as Transfer Learning and Fine Tuning to obtain more accurate results.

We tried using different types of loss functions (such as CategoricalCrossentropy and CategoricalFocalLoss) and optimizers (such as Adam, LazyAdam and RMSprop), but in all the models analyzed we found the following to be the most suitable:

- **Loss function** : CategoricalCrossentropy
- **Optimizer** : Adam

4. Ad-Hoc CNN

In this first approach we tried to use two different network models, both based on 2D convolutions and an increasing number of filters as the model grows.

4.1. 3 layers with single convolutions

Each convolution layer is composed of 2DConv, ReLu activation and final MaxPooling2D.

- **1st layer** : filters = 25, kernel_size = 3
- **2nd layer** : filters = 50, kernel_size = 3
- **3rd layer** : filters = 100, kernel_size = 3
- **Global Average Pooling + Droupout(0.3)**
- **Dense layer** : neurons = 256, kernel_size = 3, activation = 'relu'
- **Droupout + Output layer** : neurons = 8, activation='relu'

Trainable parameters = 85,012

4.2. 3 layers with double convolutions

This network is very similar to the previous one. The only difference is that we used 2 covolutions sequentially for each layer and a reduced number of kernels (1st layer = 16, 2nd layer = 32, 3rd layer = 64).
Trainable parameters = 91,416

Training the two models we reach very similar results in both cases, at their best around 0.75 in accuracy. In figure 3 is shown a 200 epochs training on the first model with EarlyStopping to avoid overfitting. Considering the lack of robustness in these models we decided to focus on pre-trained model with transfer learning, that helped us to get better results.

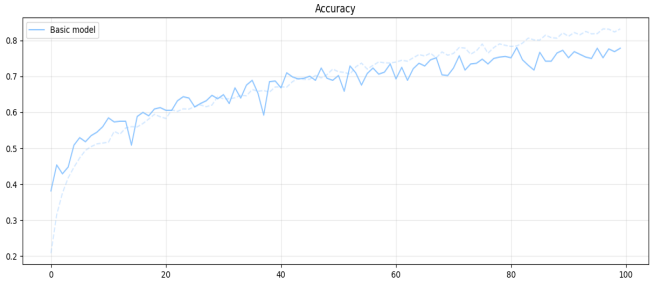

Figure 3. Accuracy for ad-hoc models

5. Transfer Learning

5.1. Pre-Trained models

To navigate all the different models provided in the `keras.applications` library, we decided to test them with our dataset, without employing data augmentation or other techniques. In this way, we decided which model was best fitted for our goal.
Because of time and computation power, we had to decide between a set of models, and in particular we decided to test VGG16, EfficientNetB0, InceptionV3, Xception and DenseNet.

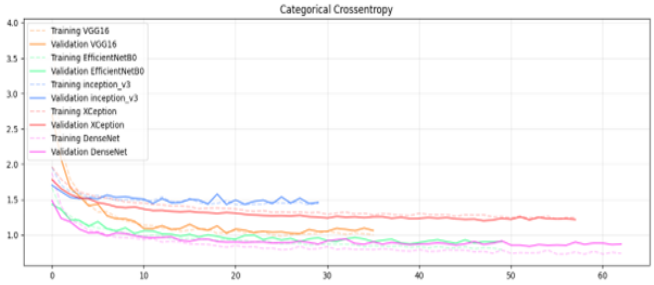

Figure 4. Validation loss plotting for each model

EfficientNetB0	**0.91**
DenseNet	**0.81**
VGG16	1.14
InceptionV3	1.45
Xception	1.27
ResNet	23.21

The picture and the table above show the models' performance measured with validation loss metric.
We found that the best-fitted models for our dataset were DenseNet and EfficientNetB0, so we focused on those for our next experiments.

5.2. DenseNet

The first model we experimented with was DenseNet.
To begin, we employed the data augmentation described above, and modified the dense layer in this way:

Layer type	# of neurons
GlobalAveragePooling	-
Dense(ReLU)	256
Dense(Softmax)	8
Dropout(0.3)	-

We trained the model with all the layers frozen, except of course the dense layers we added at the end.
After 40 epochs of transfer learning, we unfroze some layers (starting from 313) and proceeded to fine-tune the network with a lower learning rate and EarlyStopping callback monitoring val_loss.
The results of this experiment are shown in the table below, in this case we looked at the test accuracy to have an idea of how the classificator was performing.

val_loss	0.78
test_accuracy	0.74

Using this configuration, we didn't appreciate a significative improvement from the control experiment, so we knew we had to tweak the hyperparameters to find a better model.
We decided to add a BatchNormalization layer and introduced class weights to deal with class imbalabce. The improvements were noticeable, as seen in the next table.

val_loss	0.42
test_accuracy	0.87

5.3. EfficientNet

DenseNet was a promising model, but we started experimenting with EfficientNet because the performance was very close do DenseNet and it was computationally lighter. Transfer learning with only BatchNormalization, Dropout and a fully connected layer of 8 neurons gave promising results with test accuracy. The performance increased even more with some fine-tuning (we unfroze the layers after the 360). The fine tuning results are shown in the table below.

val_loss	0.38
test_accuracy	0.79

We obtained similar results with EfficientNetB0, but chose to stick with the B3 variant because the accuracy was slightly better.
After some tinkering, we found this configuration to be optimal:

Layer type	# of neurons
GlobalAveragePooling	-
Dense(ReLU)	128
Dense(ReLU)	64
Dense(ReLU)	32
Dense(Softmax)	8

Using BatchNormalization and Dropout gave us worse results in accuracy, so we decided to remove those layers. This configuration led to good results on the test validation:

val_loss	0.44
test_accuracy	0.85

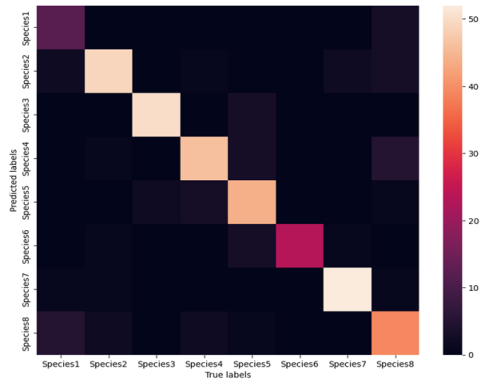

Figure 5. Confusion matrix of last model, showing the accuracy for each class

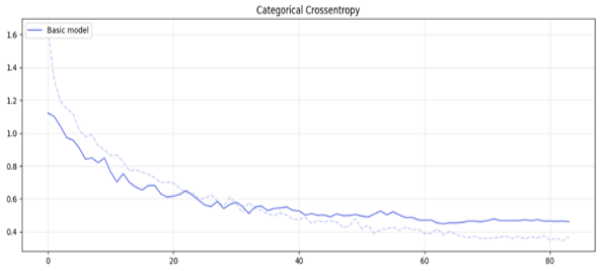

Figure 6. Categorical crossentropy evolution during epochs

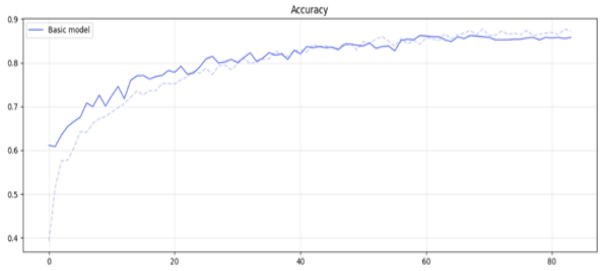

Figure 7. Validation accuracy evolution during epochs

5.4. Training Parameters

5.4.1. Layer regularization. We experimented with layer regularization, in particular with L2 from `keras.regularizers` library. We didn't appreciate a noticeable change in performance, so we ended up not using it in the last model.

5.4.2. Learning rate. We tried using a scheduler, which decreased the learning rate by an exponential factor when epochs number surpassed 40. We then used ReduceOnPlateau keras function to decrese LR when loss validation stagnated for more then 3 epochs around the same value.

5.4.3. Early Stopping. To have a more accurate result in training we used 200 epochs with early stopping monitoring validation loss with 10 epochs patience in transfer learning and 40 epochs patience in fine tuning.

5.4.4. Callbacks. Using checkpointer we have saved the weights of our model considering its best validation accuracy during the training session

6. Conclusion

6.1. Final Model

The final model submitted is the one described in the EfficientNet subsection.

7. Possible Optimizations

Some possible optimization that we could have done to our best model are:

- **More work on preprocessing**: We've noticed that preprocessing images was the key factor to reach better results in this challenge. We've worked hard to find the best solution possible, especially in the last days of work. Unfortunately we didn't have enough time to submit a better score model that reached 0.87 in testing accuracy, using a weaker saturation (1.4) and different brightness (0.9).
- **Ensemble**: We investigated different ensembling options, AdaBoostCNN seemed the most promising. Involving a chain of CNN in training phase and combining them to predict images, leveraging the weakness of the other trained networks to improve the prediction of the hard cases, would probably have given much better performance.