ANNDL- Challenge 1

Adriano Racano – Mattia Siriani – Marco Tramontini

General considerations on the dataset

The training dataset is composed by 8 folders each one containing photos of a different species of plants in their natural environment. The photos have not been processed since they were taken, and so brightness and contrast are very variable, there is not a unique background, and plants are not centered in the images. The images are sized 96x96px and the resolution is very low.
It is also evident that the dataset is not balanced (species 1, 6 and 8 have a lower number of images respect to the others) and that overall, it is small.

Our considerations on the dataset

Since the dataset is not composed of many images, we decided to use them all as training data, leaving only the dataset on Codalab as test.
We thought that with a dataset like this (very small), it would be better to use K-fold cross validation but we got better results using the Hold-out technique and so we opted for that.
At first, we split the dataset in training and validation (0.8, 0.2), but since we wanted a better accuracy of how our model behave, and since we also wanted to compute more metrics (e.g., confusion matrix), we decided to add a test set splitting the dataset as (0.8, 0.1, 0.1).
Once we decided all the hyperparameters, we trained with more data and we obtained the best result splitting the dataset as 0.95 for the training set and 0.05 for the validation set, without a test set.

Dataset balancing

Since the given dataset was not balanced, we tried to use different techniques in order to make the model recognize all the species at the best possible.

- The first approach we tried, was to use class weights. We achieved that by giving different weights to both the majority and minority classes. The resulting weights influenced the classification of the classes during the training, assigning a higher weight to the smaller classes. Considering the given dataset, the class weights retrieved were: {w0 = 2.13, w1 = 0.742, w2 = 0.767, w3 = 0.773, w4 = 0.744, w5 = 1.785, w6 = 0.735, w7 = 3.12}.
- The second approach was to use the under-sampling technique: we achieved that by randomly removing samples from the most populated species until the dataset was well balanced. We obtained a dataset with the same number of images for species as the number of the lowest species.
- The third and last approach was to use the over-sampling technique. We achieve that by randomly duplicating images of the less populated species. We obtained a dataset composed by the following numbers of images per classes: [531, 532, 529, 537, 498, 511, 497, 538].

The first approach gave the best results overall while the second approach gave the worst results since there were too less images in the dataset to let the model learn. The third approach gave us good results, but considering the number of duplicates in some species, the model overfitted on them, even if a lot of augmentation has been applied.

Preprocessing

Although it would be useful, the properties of the dataset make it not suitable for image segmentation techniques (such as thresholding) that may lose some useful properties of the images.
Since we noticed that some images had a very low contrast (especially in the images that had a low brightness), we used the python cv2 lib to slightly improve the contrast but, after some tries, we decided not to modify the brightness.

To avoid overfitting, instead, we added a gaussian random noise to the images in the preprocessing phase that, as expected, did not improve the training accuracy but led us to reach a remarkable improvement on the validation and test sets (on which we did not apply the noise).

Data augmentation

Since the dataset is quite small, we decided to apply data augmentation on our training set to increase the number of images that we could have used for training and to better generalize the model. Since the transformations made in the augmentation process are pseudo-random, we did not augment the validation and test sets to avoid the model to learn on transformed images.
To perform data augmentation, we used the ImageDataGenerator class. In particular, we selected 360° as rotation range (so that the images could be rotated in all the directions) and, to obtain also flipped images, we set the flags relative to the flipping to true. Then we slightly adjusted the remaining parameters to make minor edits to images.
We also tried cutout and cutmix approaches, using custom ImageDataGenerators, which gave us good results in terms of model generalization, but worse than using only the other augmentations.
Lastly, we tried to add contrast and different type of random noise (Gaussian, Salt&Pepper, Poisson and Speckle) to the images in the preprocessing phase, but in the end, we opted to add gaussian noise layers to obtain a similar result, since the cv2 library that we used locally was not working properly on the Codalab remote environment.

Callbacks

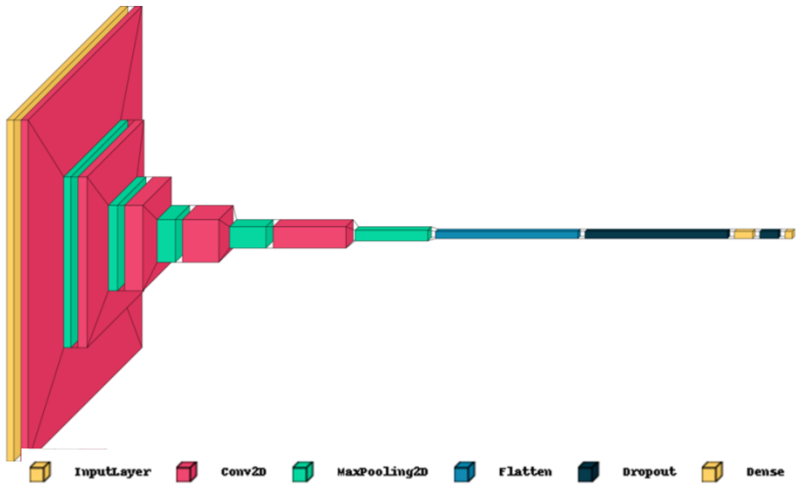
In order to avoid overfitting, we used early stopping: monitoring the "validation_loss" and after the patience ends, it restores the best weights. We also tried to use a scheduling technique (ReduceLROnPlateau) to dynamically reduce the learning rate of the optimizer (we used Adam as optimizer) when there is no improvement for a defined number of epochs. Doing so, the model takes smaller learning steps to the optimal solution of the cost function. At the end it didn't bring relevant improvements in the accuracy of our model, so we discarded it, and we opted for a fixed learning rate.

Models

To perform each training, we chose a limit of 200 epochs, and a batch size of 16, which, after some attempts, turned out to be the best compromise between accuracy and time taken to train.

Custom

We built our custom neural network with the typical architecture of a convolutional neural network: a data-driven feature extraction part followed by a feature classification part. We increased the number of filters learned at every Conv2D layer, in order to have a large number of filters for the classification task.
We stopped after 5 Conv2D layers since the size of the tensor was too low to add an additional layer.



We then tried to resize the images, by doubling their original size (192x192px) in order to add an additional Conv2d layer to learn more filters, but we achieved a lower accuracy.
At the end of our neural network, after flattening the results of the feature extraction part, we put some fully connected layers interspersed with Dropout layers (to avoid overfitting).
This approach made us reach good results on the test set (we got an accuracy of about 75%), but we also realized that with such a network, we would not have been able to achieve better results and we moved towards an approach that uses transfer learning and fine tuning.

We also tried to implement a Quasi-SVM, according to the Keras documentation (https://keras.io/examples/keras_recipes/quasi_svm/), we added a RandomFourierFeatures with a Linear layer on the top of a DNN and approximate it using a Hinge loss function, but since we didn't obtain better accuracy, we discarded this route.

Transfer Learning

Developing our model using transfer learning, we tested all the suggested CNN architectures (VGG, Inception, ResNet, DenseNet and EfficientNet), plus some networks present on the Keras site (such as ConvNext), from which it was mandatory to update the tensorflow version to 2.10. The best result we obtained was using the ConvNext architecture.
While using transfer learning, we didn't include the top of the network and we decided to build our classifier, by using the GlobalAveragingPoooling2D layer followed by some Dense and BatchNormalization layers.

Fine Tuning

When we approached to fine tuning, we initially decided to freeze all the starting layers of our pre-trained model and let only the last convolutional block to be retrained with our dataset. Later, on Convnext, we tried to unfreeze the whole network and retrain it on our dataset (which took us to a better result in terms of accuracy).
We also decreased the learning rate, so it did not have a significant impact on the already adjusted weights calculated in the Transfer learning phase.

Classifier

In Transfer learning and Fine-Tuning models, we decided to use GlobalAveragePooling2D, instead of a Flatten layer, because the latter would have turned out in a larger fully connected layer afterward, which would bring to a more expensive network, increasing the risk of overfitting. Lastly, even if after the GlobalAveragePooling2D we could predict the classes just using a soft-max, we noticed that adding a dense layer between them, the accuracy of our model increased.
Instead of the Dropout layers, we used a LeakyRelu layer followed by a BatchNormalization layer (used to mitigate internal covariance shift, applying a regularization effect).
The order of LeakyRelu before BatchNormalization was determined by the suggestion of the following paper (https://arxiv.org/pdf/1502.03167.pdf). Using this order, we noticed a slight improvement in accuracy. We didn't use both Dropout and BatchNormalization because placing them next to each other creates disharmony between the two, since BatchNormalization normalizes values for each batch while Dropout randomly drops a percentage of neurons to prevent overfitting.
Applying this strategy, the accuracy of our model increased by 2%.
We also tried to apply a low label smoothing's value to the loss function in order to prevent the model from becoming overconfident and failing to generalize.

Hyperparameter tuning, ensemble techniques

We also tried to improve our results with hyperparameter tuning, by means of the Hyperband utility, provided by KerasTuner API. This function trains multiple models generated by different choices of the hyperparameters you want to optimize (learning rate, dense units, dropout rate) and returns the most accurate one as output. This method can be very effective if properly used, but it is very expensive in terms of computations and occupied storage, especially on large models, so we didn't use it that much.
At the end of the process, we tried to mix different models with similar scores in an ensemble model, taking a weighted mean of them, but without considerable improvements.