Artificial Neural Networks and Deep Learning - First Challenge 2022

**Image Classification of Plant Species**

Nicola della Volpe, Alessandro Pindozzi. Jana El Khoury

Politecnico di Milano

November 27, 2021

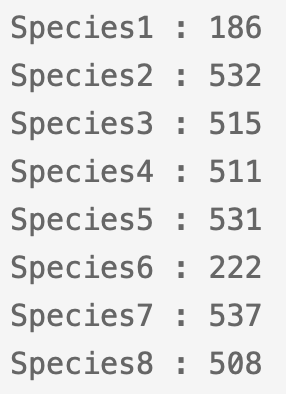
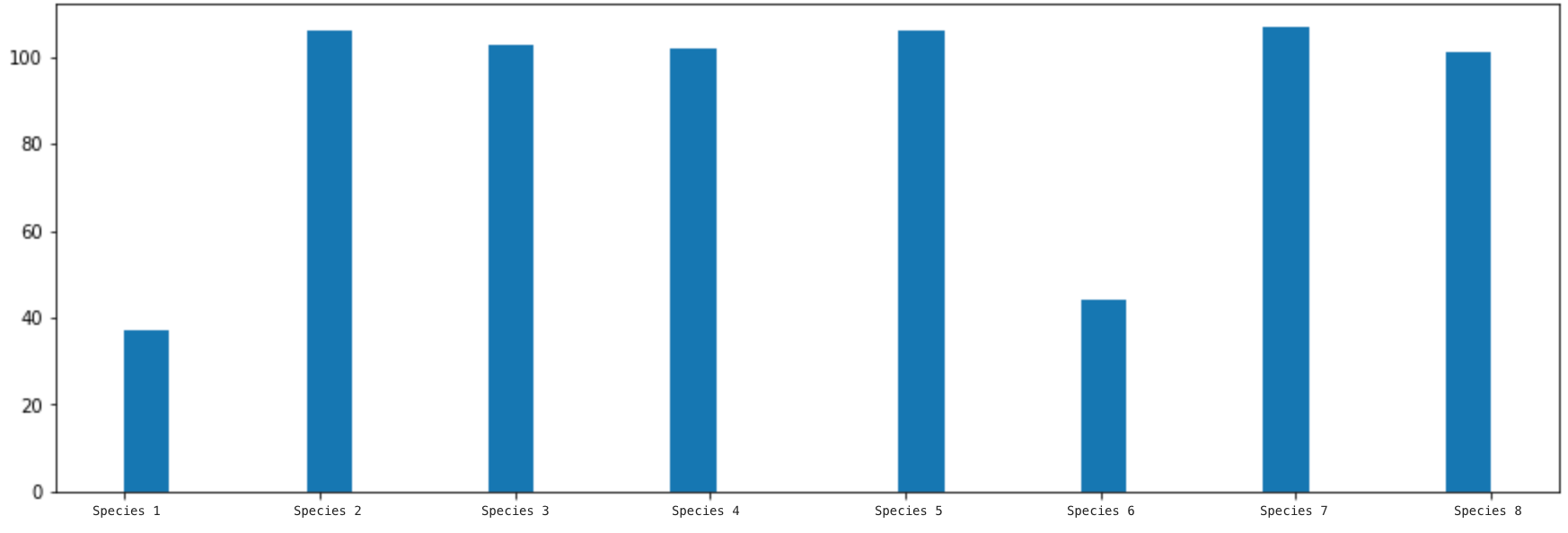
**1. Introduction**

A challenge was set by the course coordinators in order to carry out image classification using Deep Neural Networks on a dataset of images of plant species divided into eight categories (species). The goal of the challenge was to correctly assign a label to an input image, thereby correctly predicting its species.

To implement this, the given dataset was first analyzed in order to capture the information to carry out the classification. Then we experimented with different models and preprocessing configurations including transfer learning techniques and fine tuning as seen in the lectures.

**2. Data Analysis**

The dataset consists of a single folder containing 3542 images already divided into sub-folders each corresponding to their classes/species. The training images’ distribution per class is as follows:

*Image 2.1: Samples per class* 

The images are of size 96x96 within the RGB color space in JPG format. While analyzing the data, it was discovered that the distribution of samples in the dataset was unbalanced with each species having a different number of samples.

**3. Data Preparation**

**3.1. Data Splitting**

The first thing done was to create different directories for training and validation sets. This was done using *splitfolders* function.

Using this function, we decided to create two different training sets to use, in order to see which one could be the best for our models. One was generated by setting the ‘*Oversample*’ attribute of *splitfolders* to *False*, while the other training set, by setting it to *True*, since this attribute generates random samples in the classes in order to balance the set.

Another attempt to face the problem of unbalanced classes was made by giving to each class a weight. Such weights were calculated for all the classes by counting the number of images for each class and dividing by the number of total images.

We decided not to have a split including a test set as this will further reduce the size of our dataset which was already not so large.

Then we used *ImageDataGenerator* from Keras and its method *flow\_from\_directory* to generate batches of tensor image data from the already splitted dataset.

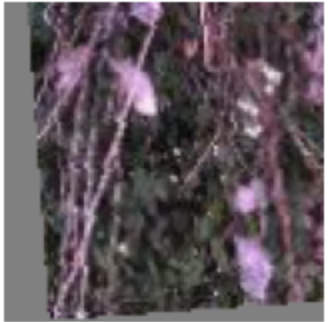
**3.2. Data Augmentation**

We decided to use data augmentation to increase the training samples and to avoid overfitting and strengthen the network.

Also in this case we moved in different directions.

We exploited the *ImageDataGenerator* functionalities which allowed us to perform real-time data augmentation. Rotation, vertical and horizontal flips as well as height and width shift were the translations used to arrive at this new augmented dataset. The empty spaces were filled with the reflect mode which gave the best results. The addition of other translation parameters only seemed to worsen our model so we went through with these. Furthermore, the preprocessing function corresponding to the models used in the transfer learning models was used as parameter of the ImageDataGenerator’s istance in order to obtain consistent results.

In addition to that, we tried different augmentation techniques using the library *Keras-cv*, such as RandAugment, CutMix and MixUp.

a. RandAug b. CutMix

*Image 3.1: Augmentations techniques examples*

**4. Models and techniques**

It was decided to train different models with the different splitted datasets we created, with and without augmentation and then with the two different techniques used in order to augment the data, in order to find the best configuration and the best model.

**4.1. Standard Model**

The first model used was a StandardCNN, a simple network with four convolutional layers, each one followed by a MaxPooling layer, in order to extract the features, then a GlobalAveragePooling to flatten, three Dense layers to classify and Dropout to deal with overfitting .

We trained this network in different ways. The dataset used was the one with no oversampling. We combine this network with augmentations and class weighting techniques in order to see how effective they are. A summary of the obtained results is shown below.

| **Augmentation** | **Class\_weights** | **Val\_accuracy score** |
| --- | --- | --- |
| No | No | 0,772 |
| Yes | No | 0,4703 |
| No | Yes | 0,7989 |
| Yes | Yes | 0,4008 |

*Table 4.1: StandardCNN val\_accuracy summary*

Then the network was trained using the same dataset, but applying different augmentations. Instead of using the attributes of ImageDataGenerator, we wrote some functions (referring to Keras documentation) for CutMix, MixUp and RandAug which were applied through the map function of the instance Dataset(from Keras). The obtained results were not so good, so much so that we decided not to take this way.

**4.2 Transfer Learning and Fine- tuning**

After experimenting with several standard models and pre-processing, we resorted to transfer learning and fine tuning, which was the beginning of a series of advanced deep models. We employed transfer learning using some well-known pre-trained networks for image classification.

We employed several of them, such as VGG, GoogLeNet, ResNet, EfficientNet and so on, trying to change the parameters in order to obtain the best result possible.

In this sense we set up only a few augmentations(some rotations, width and height shift, horizontal and vertical flip with ‘reflect’ as fill mode). We also tried with different fully connected layers for the classification and different numbers of freezing layers in fine tuning.

Finally the networks which gave us the best results are ResNet101 and ConvNextLarge.

**4.2.1 ResNet101**

The training of this model was done by freezing a few layers and then using a custom model using a couple of dense layers with dropout and ‘Relu’ activation. Training this model on the augmented training set, using the oversample dataset.

With this configuration we reached 0,83 in val\_accuracy and 0,82 in test\_accuracy.

**4.2.2 Convnext Large - Best model**

For this model, we used fine tuning to freeze the first 50 layers. The input was flattened and the model had 2 dense layers with 1024 neurons and a dropout layer in between. After several implementations and trial and error, this was the best configuration we reached. The model was then compiled using categorical cross entropy for the losses and stochastic gradient descent as optimizer.

**5. Training of the best model**

To carry out the training efficiently, we used the accuracy of the model to keep track of the performance. The model was trained for 100 epochs which was enough given this training set and models. The validation accuracy was monitored and this way we were able to trace if overfitting occurred. The training on large and deep models took a lot of time so we used early stopping to monitor the validation accuracy with a patience of 10 epochs.

**6. Results**

This section shows the results of the most successful models used for the classification task,

and the accuracy values are reported in the table below.

| **Model** | **Score** |
| --- | --- |
| StandardCNN | 0.7507 |
| VGG16 | 0.6682 |
| ResNet101 | 0.8286 |
| Model Ensemble\* | 0.8477 |
| ConvNext Large  (Best model) | 0,8903 |

*Table 6.1: Results of trained networks*

*\*Model ensemble of StandardCNN, ResNet101 and ConvNext Large*

**7. Conclusion**

From the experiments carried out, the model of fine-tuning with Convnext Large was able to perform best providing the highest test accuracy. A final trial with a model ensemble of the StandardCNN, ResNet101 and ConvNextLarge was done but it didn’t seem to provide any improvements so we stuck to using Convnext Large.

Higher results could certainly be obtained with better pre-processing, hyper-parameter tuning and suitable deep learning models. However, with limited training time and restricted GPU usage, these were the best results we could reach.

**8. Tools Used**

* Google Colab
* Tensorflow
* Jupyter Notebook
* Keras