Image Classification - Homework 1

Team: gliANNidoroDLgrandereal

Enrico Sarneri - 10805002
Luigi Umana - 10614228
Giovanni Rizzo - 10681879

November 28, 2022

1 Introduction

This report aims to show the results of the Image Classification Project of Artificial Neural Networks and Deep Learning course. The aim of this project is to solve an image classification problem by building a Convolution Neural Network and exploiting the knowledge learned in class. The dataset is composed of 3542 images of plant species, divided into 8 classes. In particular, the images provided are characterized by a 96 x 96-pixel size, RGB color format, and JPG file format.

2 Solution

We immediately noticed that the dataset was not particularly numerous and not equally distributed among classes. It mainly presented imbalances in class 1 (Species 1 - with 186 elements) and class 6 (Species 6 - with 222 elements), while all the other classes were roughly converging to 500 elements. However, we decided to face this fact in a second step.

We started by reading the raw dataset exploiting the ImageDataGenerator.flow_from_directory function (since all the classes were already divided in the corresponding folders), performing an 80% - 20% split among training and validation sets, and scaling all the images to the same range [0, 1]. Then, we began building a basic CNN model: a feature extraction part consisting of 5 Convolutional Layers joined by 2x2 MaxPooling, followed by a Flattening Layer and a Classifier Layer. Notice that in this phase we didn't apply any modification to the initial set, just to see how the model and its settings behaved.

Given the scarce effectiveness of this network, we went on by applying data augmentation. In particular, we did this in order to face the problem of data scarcity, to create variability and flexibility in the dataset (since all the images were quite standard), to increase the generalization ability of the data model, and also to help in resolving the class imbalance (as a first approach). We performed it by taking advantage of ImageDataGenerator and executing geometric transformations (shifting, flipping, zooming, etc.). After having added the augmentation the model improved considerably, going from a validation accuracy of $\approx$0.25 to $\approx$0.45.

2.1 Deeper Models

For what concerns the next step, we wanted to make the model more complex by adding 2 Dense Layers and a Batch Normalization Layer after each Convolution, each Pooling, and each Dense Layer. With this model, the validation accuracy increased to $\approx$0.50/0.55 and then to $\approx$0.60 by tweaking the location of the Batch Normalization Layers, putting them only between Convolution and Max Pooling. We also tried to change the number of Dense Layers, increasing it up to 3, to see if further complicating the model we would achieve an improvement. Obtaining no notable improvement we decided to keep the simpler model with 2 Dense Layers.

2.2 Transfer Learning and Fine Tuning

At this point we thought it would be difficult to further improve starting from our convolution model, so we decided to tackle the path of Transfer Learning, using pre-built models. Before that, we quickly tried to implement VGG19 Keras Application from scratch to see the performances, and we reached a 0.62 validation accuracy. In Transfer Learning, we used the ”imagenet” weights and, on top of the feature extraction frozen layers, we added new trainable layers in order to find the best setting for the classifier by testing different combinations. We started by using 1 Dense Layer, but we moved to 2 seeing that it was improving the performance. We also tried 3 Dense Layers but we saw that the performance was not affected by the third one, so we kept the two Dense Layers set. Furthermore, instead of the Flattening Layer, we used a Global

Average Pooling Layer, and we also added BatchNormalization Layers after each Dense Layer as well as Dropout Layers in order to mitigate overfitting. At this stage, the validation accuracy was reaching values around $\approx$0.73.

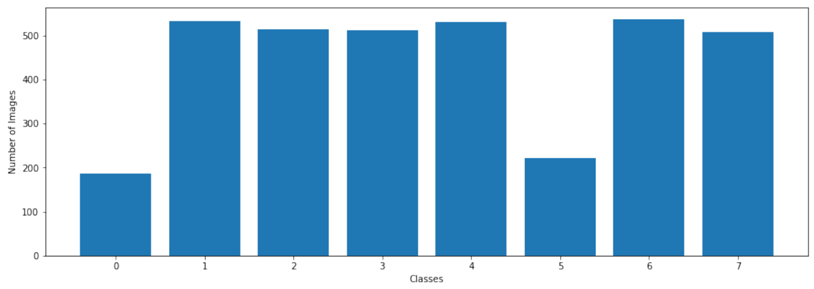
To further improve this model we then added Fine Tuning. In particular, we made different trials by unfreezing and setting as trainable blocks of Convolutional Layers in cascade by starting from the deeper ones. The best performance has been registered by leaving frozen the first two pairs of Convolutional Layers and their respective MaxPooling Layers, reaching a validation accuracy of $\approx$0.84. We then decided to try other Transfer Learning methods followed by Fine Tuning: ResNet152V2, InceptionV3, DenseNet201, MobileNetV2, EfficientNetB0, and EfficientNetB4.

2.3 Hyperparameter Tuning and Learning Rate Reduction

The next step was figuring out what other hyperparameters we could work on. After having made some attempts by modifying parameters such as batch size, dropout rate, network width and a different optimizer, we acted on the learning rate. In particular we used a keras function, ReduceLROnPlateau, which dynamically reduces this parameter when the loss function does not decrease after a given number of epochs. We kept this feature in the final model as well because we noticed a significant improvement.

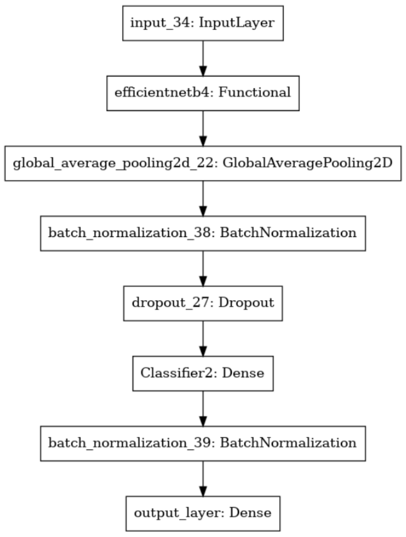
2.4 Oversampling and Class Weights

Analyzing the results by classes, after the first models, we realized that one of the reasons why we struggled to obtain optimal performance, was due to the misclassification of some classes, in particular, Species1 and Species6 that, in the provided dataset, contained few elements compared to the other ones, which had more than double the elements. To solve this problem we thought three solutions, both tested: oversampling, offline data augmentation, and class weights. Through the first one, we manually created random duplicates of the images of the classes with fewer data, in order to reach a (more or less) equal number with respect to the other ones. Through the second one, we performed for each class offline data augmentation, by creating augmented copies for each image in a number such that the resulting number of images in each class was almost the same. With class weights, on the other side, the loss function is influenced by assigning relatively higher costs to examples from minority classes. In this case, we adopted the re-weighting method from scikit-learn library to estimate class weights, using 'balanced' as a parameter. Unfortunately, all the options didn't give us good results for all the models that we trained. We alternated the training of models with the original dataset plus online data augmentation, offline augmented dataset plus online data augmentation, and re-weighted dataset plus online data augmentation without seeing any improvement after the last two. Therefore, we decided to keep the dataset as it was, just implementing the online data augmentation.



3 Final Model: EfficientNetB4

Since all of our models had a very large number of parameters, we tried to reduce them, while maintaining accuracy. For this reason, we decided to keep the model with EfficientNetB4 instead of ResNet152V2 or VGG19, because it had fewer parameters trying to find the best trade-off between accuracy and complexity. We also replaced the Flattening Layer with one of Global Average Pooling. The results were a little bit better and also the number of parameters dropped significantly, so we thought that, with almost the same performance, it would have been better to keep the less heavy model, that was the one with GAP. Furthermore, we tried to change the weights using the 'noisy_students' weights as pre-learned model for transfer learning, but it showed lower accuracy with respect to the 'imagenet'. In order to improve the accuracy, we also thought that it could have been useful to try to apply specific preprocessing to the dataset. In particular, since the dataset was characterizing natural images, it could have been useful to highlight the subject of the images (the plants), by emphasizing them with respect to the rest of the image. For this reason, we implemented both contrast stretching and histogram equalization, adding them to the preprocess function of ImageDataGenerator. However, this tip didn't help us in reaching better accuracy and we kept the original images. At this point, the accuracy of our model was ≈0.90.



input_34: InputLayer
efficientnetb4: Functional
global_average_pooling2d_22: GlobalAveragePooling2D
batch_normalization_38: BatchNormalization
dropout_27: Dropout
Classifier2: Dense
batch_normalization_39: BatchNormalization
output_layer: Dense

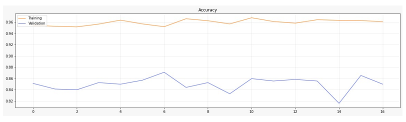


Figure 1: EfficientNetB4 Fine Tune Model Accuracy

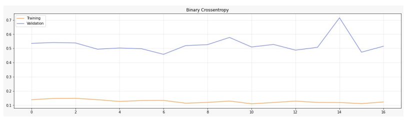


Figure 2: EfficientNetB4 Fine Tune Model Categorical CrossEntropy

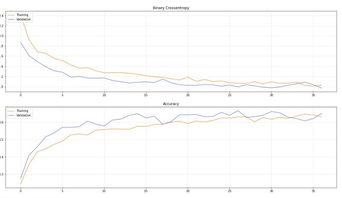


Figure 3: EfficientNetB4 Transfer Learning Model