Image Classification

Introduction and data exploration

The challenge consisted of an image classification problem. The provided dataset included 3542 images (RGB, 96x96) representing 8 different species of plants, labelled with the code of the corresponding species.

The first step performed has been the inspection of the dataset, both visual and by means of statistical indicators. By visual inspection we couldn't identify any clear feature to differentiate the classes, mainly due to the low resolution of the images and the poor knowledge about the proposed plant shapes characteristics. Neither the statistical analysis managed to highlight any difference among classes. Hence, we moved forward passing the buck of feature extraction to convolutional layers.

Before diving into the model design, we inspected the dataset's target distribution to check for class imbalance. It emerged that species 1 and 6 were the most under-represented classes in our dataset. Thus, we examined possible solutions to this problem. From the beginning we excluded down-sampling, since we wanted to exploit all the information we were provided with, and we decided to assess the performance of the remaining options, namely over-sampling and weighted loss function, during the training phase for multiple models. Hence, in this phase we figured out both a rationale for the definition of the weights and the over-sampling approach to be used. Regarding the former, we tried to tell the model to "pay more attention" to samples from an under-represented class by means of weights inversely proportional to the number of elements in each class. We started from a simple $1/N_i$ (being N_i the number of elements in each class) and moved towards slightly more complex functions, including the sklearn method, which was the one that performed best in the end. While, as for the class balancing, we increased the number of images until all classes were equal, creating multiple alternative training sets that resulted to be richer and balanced. To do that, we built a function that generates repositories with balanced training sets with exactly X images for each class (being X an input parameter), both using simple over-sampling (image replication) and augmentation. Having envisaged to include augmentation also in the data generator, in this phase we decided to play mainly with brightness and contrast with reasonable values, to be sure to have no effects on the label.

Besides, having noticed that in many images there where spots with significant shadowing, we tried to include also cut out, an augmentation technique that introduces rectangular black blocks of random dimensions in random positions (inside a given range) on the image.

General approach

To address the image classification problem, we opened the search for the best model to multiple options, including both CNN designed from scratch and pre-trained NNs to perform transfer learning. However, in designing the training phase in every case we employed pretty much the same general approach.

First, we defined the set of images to be used in each phase by splitting the provided dataset. We decided to keep the 90% of the dataset for training (80% effective training and 10% for validation), and 10% for final testing and we ensured that all sets maintained the original target distribution throughout the split. To have the 3 sets well divided and available to the image generator we decided to directly split the folders.

Then we addressed the problem of augmentation. At first, we investigated all transformations offered by the image generator and for each option we analysed whether it would have affected any label in the dataset, and we considered as valid only those that for sure wouldn't. In this phase we decided to go light and to include only transformations that might simulate the variety of real conditions in which the images can be acquired, somehow what we expected to find in other portions of the dataset. Hence, among the valid transformation, we mainly considered the most realistic ones, setting reasonable ranges. Then, during training, we played a lot with valid transformation ranges to assess how those changes influenced the performance, also by means of hyperparameter tuning.

Besides augmentation, we also searched for the best pre-processing technique. While designing from scratch the CNNs we tried both simple rescaling and normalization. In the first place, no relevant differences emerged between the two approaches in terms of performance, and anyway as soon as we moved to transfer learning, we relied on the specific pre-processing of each pre-trained model, included as a layer in the transfer-learning model.

Being the problem about image classification, in each designed model we used a Categorical Cross-Entropy Loss Function, eventually weighted to compensate for class imbalance, as explained before, in case the used training dataset was the original one (not one of those we balanced). Moreover, for our training process we decided not to use the classical stochastic gradient descent procedure, but to base the stochastic gradient descent method both on first- and second-order moments, by means of the Adam optimization algorithm. This choice was mainly led by the fact that it is "computationally efficient, with little memory requirement, invariant to diagonal rescaling of gradients, and well suited for problems that are large in terms of data/parameters" (Kingma et al., 2014) and because it proved to be the most effective option. Regarding Adam parameters, we used default values for almost every model, since they resulted to be the ones providing the best performance, except for models including Inception and Xception architectures, where a higher epsilon proved to be a more appropriate option.

Choosing appropriate hyperparameters plays a crucial role in the success of our neural network architecture and we knew that to get better results we needed to use some optimization method. Since is unfeasible to find the best values manually, we decided to address the problem of hyperparameter tuning performing a random search using the Keras Tuner module, which offers methods to easily perform the search and retrieve the best values found.

As a general training approach, for each model architecture in the first fitting procedure we set many epochs, no regularization, and a very low dropout rate to let the model overfit data, just to understand the network's learning capabilities. Then once we were satisfied with the performance on the training set, we used to introduce anti-overfitting methods such as regularization (L2 or L1L2) and a higher dropout rate. Once we identified the best strategy, we fine-tuned the models, setting a smaller learning rate starting point, eventually with richer training datasets.

In practice, in model fitting we always employed early stopping to control the training time execution, but we played with the patience according to the aim of the training (high patience to let the model overfit or medium patience during regular training and low patience during last phases of the fine tuning).

For model evaluation we employed a customed function that included as metrics accuracy, f1 score, precision, recall, and a normalized confusion matrix.

Models

Considering the general approach presented above we decided to address the classification problem with a gradual increase of models' complexity to explore different possibilities and to understand if a shallow and simple network could be a valuable solution.

We started with traditional CNNs designed from scratch. We envisaged as fundamental unit of the network a sequence consisting in a convolutional layer, followed by a Batch Normalization Layer and a ReLU activation. Then we decided to base our architecture on *convolutional blocks* consisting in this sequence repeated twice, followed by a max pooling layer and a dropout layer. Then we added a fully connected top, with a flattening and a classification layer before an output layer with 8 neurons and a SoftMax activation function. After many trials and tuning of parameters, the best model from scratch resulted to be a stack of 5 convolutional blocks with increasing depth (32, 64, 128, 256, 516) with a reported accuracy of 62% on the local test set. We understood that a network of that kind could not be suitable for the task due to low depth and complexity so to improve our results, we decided to move to famous pre-trained networks.

The first architecture we investigated is the VGG19, that we chose because of its shallow depth, good performance and fast training. We imported the network without including the top layers, and we added a GAP layer, a classification layer with 256 neurons and an output layer. We also included dropout layers, batch normalization layers and decided to use LeakyRelu as activation function to solve the problem of dying neurons and vanishing gradient. At first, we trained the network setting the convolutional part as non-trainable to perform a simple Transfer Learning using a learning rate of 0.0001, achieving an accuracy of 63% on our test set. So, we decided to fine tune the network using a ten times smaller learning rate, setting as trainable the last 2 convolutional blocks. This led to a large increase in accuracy, in fact we reached 84,1% on our local test set and 80,4% on the hidden test set.

These results were significantly better than the “homemade” model, so we decided to pursue this path, even increasing the architecture’s complexity to further increase the performances. Therefore, we moved from a 19-convolutional-layer-model to a deeper one. We chose to use the EfficientNet class of models, because they can reach higher performances with less parameters. The first model that we tried was the EfficientNetV2S, that can have a 12% boost in the accuracy with about 6 times less parameters than VGG-19. Taking EfficientNet architecture fixed, we played with the FC part configuration, starting from a simple Flatten-Dropout-Dense-Dropout-Dense configuration and moving towards more complex structures, being the two dense layers respectively the classification and the output layer. Employing this first configuration we reached 81% in accuracy, using a low drop-out rate 0.3 and no-regularization, according to the general approach. Then, to reduce overfitting, we increased the drop-out rate to 0.5 and we added regularization to the Dense layer. For the regularization we select the Elastic Net regularization, basically because it solves the problems of Lasso and Ridge Regression by combining them. This helped in reducing the gap in performance between the training and the validation.

To optimize even more this model, we tried to switch the flatten layer with a Global Average Pooling layer, considering its good performances presented in literature. We also moved from ReLU to LeakyReLU activation functions for the same reasons stated above regarding the dying neurons. With this configuration we reached 85% accuracy on our local test. Given the positive results obtained with the EfficientNet, we tried to get better results by exploiting hyperparameter tuning to find the best configuration of hyperparameters by means of random search. We performed the tuning for both the hyperparameters of the FC layer (number of units, learning rate, regularization parameter) and the augmentation transformations for the training images. The best combination obtained 87% accuracy on our local test. Subsequently, we applied the same logic to build a second model using the EfficientNetV2L architecture to boost even more the performance thanks to the higher number of parameters. It emerged that this was partially true since we observed an increase of about 2% in accuracy.

After many trials we understood that our models’ accuracy was on a plateau and so we tried to improve the performance combining the above presented networks using an average ensemble. Having this in mind we evaluated different ensemble models and the top performing was the one with the last two Efficient Nets combined. This configuration reached 86,3% in accuracy on the hidden test set. After having selected our final model, we tried to further improve it trying to use a weighted average between models.

We also tried to combine the output of each model with additional layers to let the model learn how to combine them in an optimal way. We tried two structures: at first, we substituted the Average layer with a Concatenate layer, and we added a fully connected output layer in cascade. By doing so we obtained a final output as a combination of all the 16 outputs coming from the two models. After this first trial we substituted the dense layers, with a fully convolutional structure. We concatenated the output of the two models in depth and we added in cascade a 1-D convolutional block. By doing so the output we obtained for each class was a learnable weighted combination of the output of the ensembled models.

Unfortunately, these configurations and the weighted average ensemble performed better on our local test set, but worse in the hidden test set. So, we decided not to use them and to stick with the “classic” average 2 model-ensemble.

Finally, we tried to use Test Time Augmentation. To do so, we modified the "predict" method achieving a .7% increment in the accuracy.

Final model and conclusions

To conclude, the model that performed the best on the hidden test, achieving an accuracy of 86.98%, was the Ensemble model including the two best performing Efficient Nets (V2S and V2L) with the simple average output combination.

To be honest we ended up quite surprised with this result. We expected that the inclusion in the learning process of the weights, used to combine the models’ outputs at the top of the ensemble architecture, would have had a meaningful impact on the performances, not a decrease in the accuracy. We have related this happening to overfitting in some way, but we believe further investigation would have been required to find out how to improve that solution.

In the end, we are quite satisfied with our achievements, but we are confident that further developments and attempts would have led even to better results, and we are aware that we have still a lot to learn.