ANNDL2022: Homework 1

Giovanni
Giuseppe
Luca

Introduction
The goal of this project is to produce a Convolutional Neural Network (CNN) and the reasoning behind its structure, to perform an 8 classes plant species classification based on colored 96×96 pixels images.
Our team chose the scholastic approach of developing two separate networks, one from scratch and the other exploiting transfer learning.

Preprocessing data
We used the Python package Split-folder to split the dataset into train, validation and test set. This has been done because we wanted to train our model relying on the validation set, and then compare different models, if needed, with a test set in order to have a more unbiased estimation of our performance. After a first attempt with percentage split, we used a fixed values split to obtain a balance test of our performances because we do not have reason to believe that the distribution of the sample could be not uniform.

Network from scratch
Starting from the CNN model seen at the lab with 6 convolutional layers, made up of 3x3 filters, and a flattening layer, we calibrated the numbers of convolutional layers and the numbers of filters. We observed the optimal number of convolutional layers being 4, spaced out by Max Pooling layer, starting from 32 filters doubling them up to 256.

To catch more general patterns in the dataset, we exploited a bigger filter size (5x5) in the first convolutional layers, but this approach did not lead to a better solution.

After putting a Flattening layer, we tried to improve the performance by adjusting the number and the size of the Dense layer (in figure 1A we can see one of our first fails). Although we thought that increasing the number of fully connected layers could improve the performance by means of more non-linearity with little extra parameters, we saw the best results were reached with a single hidden layer of 256 units: a possible interpretation may be that the net is able to perform adequately just by elaborating almost linearly (i.e., using a single hidden layer) the extracted features from the convolutional layers, and at the same time the limitedness of the training data penalizes deep feed forward neural networks with high nonlinear modelling capabilities. The accuracy of this model on our local test set was 0.55.

To improve this model, we substituted the activation functions of the Convolutional layers from ReLU to *LeakyReLU* and the Flattening layer with a Global Averaging Pooling (GAP) reaching 0.62 of accuracy. Observing in the training phase that the model was unstable, we used a learning rate approach consisting in reducing the learning rate through the *ReduceLROnPlateau* callback. We submit this model on the platform reaching an accuracy of 0.65 (figure 1B).

Then we tried to add a *BatchNormalization* layer after the GAP, but we obtained a drop of accuracy.
At this point, we noticed in the training phase that the accuracy was 0.99 on the training set, to avoid overfitting we added to the already present Dropout the Elastic Net regularization. This improved the performance on Codalab verifier to 0.73.

As the last step, we employed Data Augmentation to balance the dataset's classes and to do oversampling, paying attention to maintaining the labels of the sample. We got to 2000 images per class in our most successful attempts. We retrained our best model with this new dataset, and we achieved 0.8 of accuracy on the verifier (figure 1C).

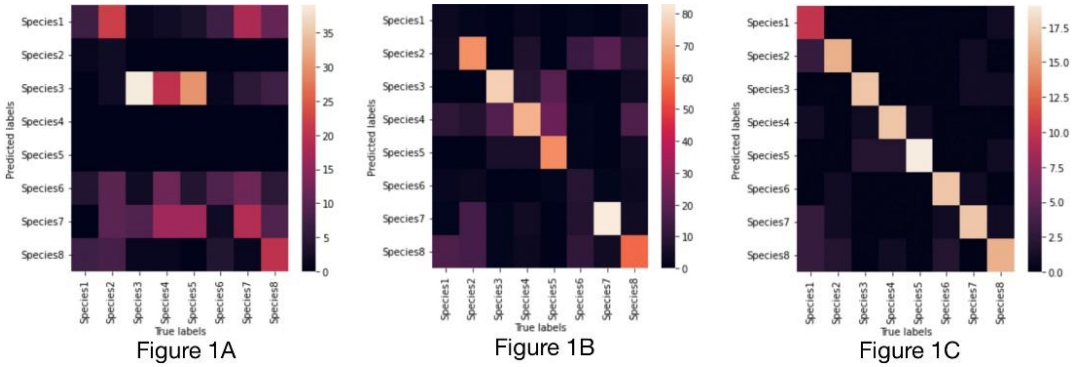

Figure 1A Figure 1B Figure 1C

Figure 1: network from scratch evolution

As shown in the picture, the first species is the weak point of our model: we tried to fix this aspect using class weight of fit method, giving more importance to the under-represented classes, but it failed. Our last attempt was to delete from the dataset all the images with relevant shadows, but even this led to a worse solution.

Transfer Leaning

To improve the performance of the classifier, we decided to switch to a transfer learning approach. At the beginning every member of the group tried different architectures such as *EfficientNetB0, InceptionV3, Xception, MobileNet, and VGG16*. We thought that architectures with less parameter could perform better due to the nature of our dataset (few samples with low resolution), but at the end the *VGG16* architecture has proven to have the better performance among the first guesses.

Thanks to the acquired knowledge on the CNN from scratch we decided to keep the number of hidden layer small with not so many neurons, after some attempts we noticed that optimal configuration was the same of the network from scratch: 1 hidden layer with 256 units. Furthermore, we noticed that CNN from scratch takes advantages of *LeakyRealu* instead of *ReLu* Activation function and a GAP layer instead of a flattening one, so we decided to keep them also in the initial phase of transfer learning. In the Fully Connected part we implemented two kinds of regularization: Dropout($rate = 0.4$) and Elastic Net ($\lambda = 5 \cdot 10^{-4}$). The training phase was split in two parts: transfer learning and Fine Tuning. In the latter part we decided to unfreeze different percentage of layers to select the best performing option. So far, we got 0.74 of accuracy in our local test set unfreezing the 65% of the VGG16 layers, not a good result!

An important step was made when we implemented *Data Augmentation* and *Oversampling*. We believed that our dataset was not enough for the huge number of parameters of the architecture chosen, so we increased the number of samples for each class up to 1000 applying data augmentation. This let us to archive a major improvement in accuracy: 0.85 on Codalab test set. Trying to furtherly increase the outcome, we search in the VGG16 architecture something to refine, we noticed that no normalization layer is present in it, so we added a *Batch Normalization Layer* after the GAP and this brought us to reach 0.88 on the verification platform.

Even if the result was fair enough, we notice that it was poor in the first and last classes, to fix the problem we tried to increase the number of samples for each class up to 4000 samples (with data augmentation). Unfortunately, it did not increase the accuracy but at least lead to more balanced classifier. On the contrary, class weights in training did not help to solve the issue.

During the design process we learnt that *keras.application* models are divided into blocks, so we changed our fine-tuning script from unfreezing a percentage of the network to a block of the structure. We noticed that the best result in VGG16 was reached unfreezing the final 3 blocks, approximately the 67% of the network, so not so distance from the percentage found during the hyper-parameter tuning phase.

A second attempt to furtherly improve the performance was carried out changing drastically the design of the feed forward fully connected network: we were only partially satisfied with the results and the training phase of the current net was irregular and seemed unrobust to changes in test sets and on initialization seed. Therefore, we found several papers defending opposite theses about regularization and how to deal with it (small values, high values, differentiated values depending on the layer dept and type, …) and we also looked at learning rate “theory” and especially on ideas to furtherly exploit the specific structure of VGG16. The

latter research was the most useful: the VGG16 fully connected feed forward network has three 4096 nodes layers. The new idea was therefore to develop a script for precise hyperparameters tuning including a new design for the fully connected part with a structure more similar to the original VGG16.

The number of tests carried out by the script is massive (more than 150) and showed the following results:

- The most important tuning parameter is the learning rate, with an optimal value for our task of 10^{-4}.
- The optimal elastic regularization is extremely low (10^{-5} and lower, even 0, have the best results).
- The best dropout was as low as 0.25, but it was not that relevant.

We ended up submitting a model with almost all layers freed (4 blocks), little regularization and a 10^{-4} learning rate and *ReduceLROnPlateau* technique, which is reduced by around 2/3 every time it gets "stuck".

Overall, with respect to the first attempt with different parameters and only one hidden layer, the result seemed not to improve on the development test set, but we believed that this model was more robust that the first one: it was proved right during the final phase when we submitted both versions, and the tuned one remained stable on the 0.88 of accuracy, compared to the first that lost 0.2 accuracy points.

It was not as good as hoped, but allowed us to reflect on a little bit of theory: from an optimization point of view, a proper and adapting learning rate is the best option for non-convex functions as the one shown in figure; from a cost-function point of view, a too high elastic regularization incentivize to reduce the values of the parameters of the original VGG16 for the sake of the loss function alone, worsening the performance of the net the more layers are freed; the regularization however is important to allow a net with a huge amount of trainable parameters to avoid overfitting on few augmented samples available. Furthermore, the lower-than-expected improvement with three layers in the fully connected net may be explained by the network exploiting mainly linear combinations of the extracted features.

Due to the limitation of time and computational power, it shall be noted that we decided to skip the transfer learning phase in the hyper-parameter tuning script: the effect of this difference with respect to the "standard approach" may be furtherly investigated in the future.

As a very last attempt, we tried one of the latest pre-trained models available on *keras.applications*: the base version of *ConvNeXt*. For the fully Connected part we took inspiration from the first attempt: a GAP layer followed by a Batch normalization one and then a hidden layer, now with just 128 neurons, which seemed to be the best choice, regularized with Elastic net ($\lambda = 5 \cdot 10^{-4}$) and dropout ($rate = 0.35$). In the transfer learning part, the model had a validation accuracy of 0.83, a very good result as our first attempt. Then in the fine-tuning part we tested to unfreeze various stages of the ConvNeXt architecture: unlocking only the fourth did not significantly improve the performance with respect to the transfer learning part; when we unlocked the third stage, we saw much better results, but we concluded that making all the layers of the pre-trained model trainable gave us an accuracy of 0.914 on the platform verifier, our best result.

We noticed that for all the models used one of the major improvements was to unfreeze a big part of the network. We interpret this phenomenon by noting that the "ImageNet" database has almost 400 categories of animals and more than 500 categories of objects, but not any categories of plants, so a deeper training is needed. Therefore, it may drastically improve the accuracy to perform transfer learning using pre-trained weights on databases already including at least some categories of plants.

Future Works

To improve our work, the best direction is likely to proceed to perform a more precise hyperparameters tuning on ConvNeXt, because we have noticed that different network fit better with different hyperparamenter, of the kind we could only apply to VGG16 for time constraints. Even augmentation may be refined, both in its parameters and on the strategy applied (that is, augmenting only original images or augmenting also augmented images, or even find a better middle ground), since we tested it only at first glance. Moreover, it is likely that an improvement on ReduceLROnPlateau parameters and in general on the learning rates management can still bring perceptible upgrades.