Plants classification with CNNs

Calligaro Nicola, Castiglia Danilo, Cimpeanu Vlad

1 Introduction

In this task, we are required to classify species of plants using neural network architectures, which are divided into 8 categories according to the species of the plant to which they belong. Being a classification problem, given an image, the goal is to predict the correct class label.

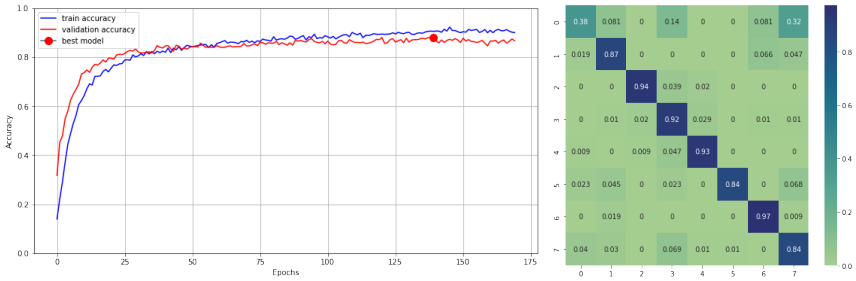
2 Data preprocessing and data augmentation

First of all, we split the dataset into training set (80%) and validation set (20%)[1] with stratification, so that the proportions between classes are the same in the training and validation set.
By looking at the distribution of the classes, we notice that the dataset is unbalanced, more precisely Species1 and Species6 are minority classes, thus, we may expect some difficulties while learning to recognize plants belonging to those species.
To decrease the issues provided by the unbalanced dataset, we use oversampling method to increase the number of training samples for Species1 and Species6.
Since neural networks need a huge amount of data to get good performances, before feeding our architectures with the training images, we augment them through data augmentation layers such as random vertical and horizontal flip, random rotation and random translation layers, so that our architecture can see more images during the training process and avoid overfitting.
We tried also to add a custom data augmentation function that drastically reduces the brightness of the images in input so that our architecture becomes shadow invariant, indeed, some of the images are very dark due to the photographer's shadow. We believe shadows are partially responsible for poor performance, as matter of fact, by applying this filter only to the validation set, the validation accuracy dramatically drops, still, we were not able to increase performances in none of our experiments by applying this transformation to the training set.

3 Model from scratch

In all the experiments described in this section, the input is rescaled, dividing all pixels by 255.
Our first approach is to build a convolutional neural network from scratch. We start with a very simple CNN with 3 convolutional layers alternated by max pooling layers. Each convolution has a kernel size of 3x3 and the number of filters per layer is respectively 16, 32, and 64 with Swish activation function. The convolutional part is connected to the classifier through a flattening layer. The classifier has only one hidden fully connected layer with 100 neurons and Swish activation function. Finally, the output layer is a fully connected layer with 8 neurons and a softmax activation function.
From our experiment we notice very low training accuracy, thus the model is too simple concerning the problem we are facing. To increase the complexity, first, we try new configurations of the classifier by adding more hidden layers and more neurons, without achieving significant improvements.
This outcome leads us to suspect the convolutional part cannot extract enough meaningful features.
As the input images are very small (96x96), we can not add new convolutional layers alternated by max-pooling layers, since after few max-pooling layers, the input image loses all its dimensions. To increase the complexity of the convolutional part without losing the dimensions too early, we increase the number of convolutional layers before each max pooling layer. Notice that this approach makes sense only if we apply a non-linear activation function such as Swish on the output at each convolutional layer.
After some experiments, the best configuration we find is the one having in sequence: 2 convolutional

[1] We do not need to create a test set, because the server hosting the challenge already provides us with a hidden test set.



(a) (b)

Figure 1: (a) Accuracy for transfer learning ConvexNeXt Large (b) Confusion matrix for Transfer learning ConvexNeXt Large

layers with 32 filters, max pooling layer, 3 convolutional layers with 64 filters, max pooling layer, 2 convolutional layers with 128 filters, max pooling layer, Global average pooling layer and finally the output layer. All the convolutional layers have a 3x3 kernel size and Swish activation function, this configuration reaches 77% accuracy on the validation set.
The model presents some variance, but still, this is the best result we were able to achieve, indeed, by applying some regularization techniques, such as weight decay, dropout, and batch normalization, the accuracy dropped.

4 Transfer learning

Now, we try different well known pre-trained networks available in `Keras Applications` and compare the accuracy obtained to fine-tune only the most performing ones.
First, we replace the top layers, which are the fully connected ones, with some combinations of layers designed by us. For all the architectures taken into consideration, we notice that by just adding a simple classifier, therefore a `dense` layer with softmax activation function, all the models are affected by underfitting, this suggests us there might be some margin of improvement if we increase the complexity of the classifier.
Thus, our strategy is to iteratively increase the complexity of the model, without letting it overfit. We want to keep the training accuracy the same as the validation accuracy, otherwise, we risk losing any benefit from fine-tuning, which is prone to overfitting.
The best combination of layers we find is ConvexNeXt Large as supernet, followed by a `Flatten` layer, `Dense` layer with 256 neurons, a `Dense` layer with 128 neurons, a `Dense` layer with 64 neurons, a `Dense` layer with 32 neurons, and finally a `Dense` layer with 8 neurons and activation function Softmax. All the dense layers, except the last, use Swish as activation function and before each of them, we add a `Dropout` layer with a rate of 0.1. This configuration leads to an 87% validation accuracy.
In our experiments, we considered also other pre-trained architectures such as VGG16, Resnet50, and EfficientNetB0 reaching respectively 73%, 75%, and 77% of accuracy, which do not look very promising.

5 Fine tuning

In the following experiments, we always set all the `BatchNormalization` layers to inference mode.

5.1 EfficientNetB0

For the EfficientNetB0 we followed some recommendations from the keras guide. First of all, we decrease the batch size from 512 to 128, then we decrease the learning rate from 10^{-3} (used for the transfer learning procedure) to 10^{-4}. As stated in the guide, EfficientNetB0 is composed of 7 blocks, and to achieve better performances with fine-tuning, each block needs to be all turned on or off. This is because the architecture includes skip connections from the first layer to the last layer for each block. After various experiments, the best accuracy we achieve is 89% on the validation set, with the first block frozen.
This result meets our expectations, indeed, our dataset is very small, thus we need some layers frozen, but,

since from our point of view, these images are very different from the ones belonging to the ImageNet (the one used to pre-train EfficientNet), most of the layers are not able to extract the correct features, therefore they must be retrained.

5.2 ConvexNeXt large

Due to the limited number of Efficientnet's blocks(7), it was possible to try all the possible combinations of frozen blocks in a reasonable amount of time.
In contrast, ConvexNext is composed of 295 layers, furthermore, it is too expensive to try freezing all possible combinations, indeed, running in parallel 2 GPUs (14Gb each), 1 epoch with 512 batch size takes 12 seconds during the transfer learning phase. Given that, we decide to try only a few candidates, thus we try to freeze the first 150, 200, and 250 layers. As a common practice, even in this case, we decrease the learning rate from 10^{-3} to 10^{-4}.
We reach 91% of validation accuracy by freezing the first 150 layers. Since this configuration is the one with the highest accuracy on the validation set, we select this model as the final one, achieving on the test set 88.3% of accuracy.

6 Facing unbalanced dataset

By analyzing the confusion matrix related to the validation results, it is clear that some classes are harder to classify than others. In particular, species 1 is the most difficult to recognize: this can be due to the number of samples available in our dataset, which is significantly smaller compared to most of the other classes, however, this can not be the only cause of bad performance since, for instance, the number of samples available for species 6 is similar to the one available for species 1, nevertheless, species 6 is much easier to recognize.
Facing up with this issue, first, we try introducing a set of weights to be applied when computing the loss function to make "weigh" more wrong predictions on species that have high weight values. In this way, we force the model to learn more carefully how to recognize the most challenging species. In our implementation, weights $W = w_0, w_1, ..., w_n$ are computed as:

$$w_i = \frac{1}{8 * c_i} \sum_{i=0}^{n} c_i$$

where c_i is the number of samples belonging to a single class (specie) inside the training set[2].
Another approach proposed is to build a classifier *1vsAll*, whose goal is to solve a (possibly) "easier" problem: discriminate the species 1 against all the others; then this is combined with the already discussed model able to discriminate all the species.
Our strategy is to try first to discriminate whether the input belongs to the species 1 class using the *1vsAll* model, if so, then output species 1 as the final result and stop the data flow, otherwise, output the prediction made by the complete model as the final result (this method works well in case of few false positive predictions made by the *1vsAll* model).
To improve the *1vsAll* model performance in the scenario of an unbalanced dataset, we set the initial value of the final layer bias to $\log(\frac{pos}{neg})$ where *pos* is the number of species 1 samples and *neg* is the number of non-species 1 samples in the training set. The idea that lies behind this value is that we want to set the bias on the logits to get a probability of predicting the positive class (specie 1) equal to $\frac{pos}{neg}$ at the initialization of the model (considering *sigmoid* as activation function on final layer neurons).
With some disappointment, none of these experiments lead us to successful results.

7 Conclusions

In conclusion, the best performance achieved is with ConvexnNeXt Large, nevertheless, according to Keras webpage, ConvexNeXt presents about 197.7 million parameters, in contrast, EfficientNetB0 is composed by only 5.3 millions of parameters with only 2% accuracy difference.
As a result, we think one should choose EfficientNetB0 when memory and energy are limited but the accuracy is not critical, whereas ConvexNeXt if there are "unlimited" resources.

[2] In the case of the oversampled training set, we still compute the weights accordingly to the original training set.