POLITECNICO
MILANO 1863

Artificial Neural Network and Deep Learning
Academic Year 2022 - 2023

Homework 1 Report

Team: KitKat

Christian Grasso 10652464
Filippo Lazzati 10629918
Matteo Nunziante 10670132

Professor: Matteo Matteucci
Tutor: Eugenio Lomurno

Contents

1 Introduction

In this project we had to deal with an **image classification problem**. The dataset is made of 3542 images of plants, divided in eight species. The dataset is an unbalanced one, meaning that for each species there are different quantities of images. The images have a size of 96x96 pixels.

2 Development

2.1 Training, Validation and Testing

The **dataset** has been divided in *training set*, *validation set* and *test set*, with respectively the 80%, the 15% and the 5% of the total dataset. After an exploratory phase of the data, we realized that the dataset is unbalanced, namely some classes contain much less samples than others. To deal with this fact, we have computed the **class weights** in order to let smaller classes give more contribution, at train time, to the loss function. The weight of a class has been simply computed dividing the total number of images by the number of images of the specific class.

2.2 Image Augmentation

Since the dataset is rather small considering the large amount of parameters a strong learner like a Neural Network can have, and taking into account also the partitioning in three sets, very few images were available for mere training. Therefore, a *pre-processing* step of **image augmentation** was mandatory, mainly for two reasons: increasing the number of different images, and thus the number of samples to use at training time, and contrasting *overfitting*.
We have tried two approaches:

Manual augmentation After splitting the dataset, we have manually augmented the images used for training by manually applying different transformations (translations, shifts, reflections, rotations etc.) to each image, and adding the augmented images to the training set.

In-model augmentation Without changing the original dataset, we used the Keras-provided Augmentation layers in the model before the convolutional layers to automatically and randomly apply transformations to each input image.

Initially, we started with the first approach, which improved the overall accuracy but also caused *overfitting*. Instead, using the second approach, since the augmentation layers randomize the augmentation parameters for each epoch, potentially the network has never seen twice the same data, thus it was able to learn without overfitting the data. At the beginning, the augmentations used were zooms, flips, shifts and translations. We didn't use rotations because they negatively affected the performance. Later on, after noticing that the relevant part of the images, i.e. the batch of pixels that actually characterizes the plant species, is located in the center of the images, we decided to try to remove a few pixels on the sides of the images using *cropping*. This effectively increased the performance by around 3%. We noticed that cropping up to 3 pixels had a positive effect, while larger crops lead to a decrease in performance.

2.3 Architectures

2.3.1 Custom architecture

We have approached the problem starting with simple models created by us, made of 4 **convolutional layers** divided by *max pooling* layers, 2 dense layers for classification, a *dropout layer* and the output dense layer. As mentioned above, initially we performed manual augmentation, but this caused severe overfitting: our initial accuracy on the test set was around 60% while the training set accuracy approached 100%. We tried with different **regularization techniques** (lasso, ridge, elastic net) and different values of the regularization rate λ and the dropout rate, but this ended up in just a little improvement. Another attempt was to combine different augmentation techniques, and later switching to in-model augmentation as described above. Also, we replaced the max pooling layers with *average pooling* layers, since the latter achieved better performance. Other tests which did not yield better results were trying different activation functions and layer weight initializers. In order to reduce training time and further reduce overfitting, we have also exploited the EarlyStopping Keras callback. With these improvements, and after performing **hyperparameters tuning**, the model was still overfitting on the training data, but the accuracy in the test set increased to 71%.

2.3.2 Transfer Learning and Fine Tuning

After various attempts at improving our custom model performance by manually and automatically tweaking hyperparameters, we decided to switch to a **transfer learning**-based approach. The first model we tried to use is NASNetLarge with *ImageNet* weights. Because of the size of the model, it could not be trained fully with the resources available to us, so we froze its layers and placed a custom fully-connected classification head on top of it, effectively using the original model as a feature extractor. After an initial round of training, we performed **fine tuning** on the last few layers of the model with a reduced learning rate for a small number of epochs. This attempt drastically reduced overfitting and slightly increased performance, but the accuracy on the training set did not rise above 80%, an indication that the model was not learning properly. This did not change by modifying the learning rate or increasing the complexity of the classification head. We therefore decided to switch to a different, smaller base model that could be fully trained (without freezing its layers) with our data. The model we selected is EfficientNetV2B2 with ImageNet weights, which in the end also turned out to be the best of those we tested. With this model, we reached an accuracy of 80%.

The initial architecture of our model was made of the following blocks: augmentation, base model, flattening layer and 3 dense layers. This model was too complex since it was still overfitting, so we reduced its complexity changing the flattening layer with a global average pooling one, thus reducing the number of learnable parameters. This single change brought an improvement of 5% in the test accuracy. We explored also other base models keeping more or less the same architecture. The models that gave us good performances were (in order) EfficientNetV2B2, Xception, ConvNeXt and EfficientNetB5 (see `https://keras.io/api/applications/` for some indications of the benchmark performances of these models). Since some of the base models we tried were trained on larger images (224x224), we also tried resizing our images from 96x96 to 256x256 (and increasing the cropping size to 16 pixels). After this change, ConvNeXt performed better compared to the case of small images. Instead, EfficientNetV2B2 gave almost the same results with both original images and bigger images. We tried also to use fine-tuning on the above models, but the performance mostly stayed the same or even decreased in some cases compared to the train-all approach, with the only exception of *Xception*.

2.4 Optimizers

According to https://keras.io/api/optimizers/, we tried different optimizers such as Adam, AdaGrad, AdaMax, SGD with momentum. Since we noticed that sometimes the training got stuck in a local minima, we also tried using learning rate schedules such as *restart cosine decay*. We obtained just faster convergences but no accuracy improvements. SGD with Nesterov momentum also did not significantly affect the result. With the train-all approach, AdaGrad with a learning rate of 0.01 is the optimizer that gave us the best results, both in terms of convergence time and final performance. For what concerns fine-tuning, instead, we used AdaGrad with a learning rate of 0.01 for the first phase and Adam with a learning rate of 0.001 for the fine tuning phase.

2.5 Ensembles

Based on the theory of **Bagging**, given two non-trivial classifiers, there exists a way to combine them in order to get a better classifier (with reduced variance). We tried to combine the best models we obtained, both with a simple average approach (by summing the label probabilities and dividing them by the number of models in the ensemble) and with majority voting. We obtained the best solution combining 5 models, all of them trained through *transfer learning*:

- EfficientNetV2B2 with 96x96 images
- EfficientNetB5 with 96x96 images
- EfficientNetV2B2 with 224x224 images
- ConvNeXt with 224x224 images
- Xception with 224x224 images

Combining all of them we get an accuracy of around 90% (our final submission). We were not able to get higher results because all the models we combined were not able to recognize images of species 1 and 8 with a sufficiently high accuracy, and changing the class weights caused a reduction in the accuracy of other classes. A further improvement might be to combine a model focused on recognizing items of species 1 and 8, and use it to discriminate between the two when the above model does not yield a sufficiently high accuracy.

3 Conclusions

We have seen in practice how *augmentation* is a crucial step during the learning process. The quality of the augmentations directly affects the performance of the model, no matter how "strong" the model is. We had the chance of comparing different techniques/models in practice on a *real-world* scenario and relatively low-quality dataset. In the end, even with just a few augmentation techniques and by combining together simple models, we have been able to get good performances (accuracy of 90%) on the test set.