Artificial Neural Networks and Deep Learning - Homework 1 Report

Alberto Aniballi, Andrea Bosisio, Flavio La Manna (Group: F2ANN_Polimi)

November 28, 2022

1 Dataset inspection and split

We started the challenge analysing the dataset[1]: we immediately noticed that it was unbalanced. All classes had approximately 500 samples except `Species1` and `Species6` that had less than the half of the images with respect the other classes. Moreover, we noticed that images had a small resolution (96×96 pixel) and that there were some "noisy" images with predominant background and some outliers such as images with a large presence of external entities and their shadows. Anyway, we firstly decided to not remove any image from the dataset.

We used the library `split-folder`, which allows *stratified sampling*, to split the dataset in *training*, *validation* and *test set* with rate 80%, 15% and 5% respectively. Training data is used to learn the model parameters while validation data is used to have an estimate of the performance of the model and perform *early stopping* to avoid *overfitting*. For every model we plotted graphs to better understand the trend of the losses during training phase and a confusion matrix to check easily how the model performed on the classification of each class based on our test set.

2 Development

We started working together on CNNs built from scratch as explained in section 2.1. Then we started working in parallel on pre-trained models (section 2.2) and on finding different techniques to achieve better results.

2.1 First custom models

The first model we built, $\mathcal{M}_1$, was a simple neural network similar to the ones seen in class: 5 convolutional layers with 2×2 *max-pooling* and 1 dense layer for classification. We added *dropout* to avoid overfitting, we set learning rate to `1e-3`, 200 as the number of epochs and we normalized the input dividing it by 255. We never reached 200 epochs for a training because we used early stopping to prevent overfitting. We tried different values for the *patience* and we had the best results with patience equal to 10. Also, we tried to reduce the learning rate to `1e-4` but we didn't get any improvements in the accuracy. With $\mathcal{M}_1$ we reached a test accuracy of 0.56 on *Codalab*.

Given that the dataset was not so big, we applied *data augmentation* to add more training data. Firstly, we analysed the images to understand what were the possible transformations that could be applied to the input. Obviously, the images could be *rotated* of 360 degrees, horizontally and vertically *flipped* without losing the label. In a lot of images, the plant occupied a small portion of it, so we applied height and width *shift*, and *zoom*. Applying the set of possible transformations to some train images, we assessed that this combination produced results where in most of the cases the plant was visualized better. As seen in class, we used `ImageDataGenerator` to generate new augmented images at each epoch. We set the `fill_mode` to `reflect` because the images are in a natural environment and so they can be filled with the same content of the image itself without losing information. We built the second model, $\mathcal{M}_2$, with the same architecture of $\mathcal{M}_1$ but adding the *data augmentation* part just described. We reached an accuracy of 0.64 on our test set but only 0.59 on *CodaLab*. On one hand we had improvements because *data augmentation* made the model more robust to transformations; on the other hand, the improvements were not as much as we expected, probably because the hidden test set was much different from the dataset provided. *Data augmentation* helped us, but the problem of the unbalanced classes was still there. The imbalance in the data might have made the models more biased towards the majority classes prediction ignoring the minority classes. This hypothesis was confirmed by looking at the confusion matrices: the number of samples classified correctly for those two classes was much smaller than the others.

To handle the imbalance in the dataset, we firstly performed *oversampling* of the two classes with less images, `Species1` and `Species6`. *Oversampling* consists of randomly selecting samples from the minority class with replacement and duplicate them in the dataset. It was applied on the training and

[1] see `dataset_visualization_split.ipynb`

validation sets to have an equally balanced dataset and we developed a script[2] to perform it on all the classes specifying the number of images we wanted to add. We tried different types of *oversampling*: levelling all classes to approximately 400 samples for training and 70 for validation, 1000 for training and 200 for validation, adding 100 and 20 samples to only the minority classes or other similar combinations. We built $\mathcal{M}_3$ with the same architecture of $\mathcal{M}_1$, using the same *data augmentation* of $\mathcal{M}_2$ and trying different oversampled dataset. We had the best result with the $400/70$ balanced dataset reaching 0.629 accuracy on *CodaLab*. Looking at the confusion matrix, we increased the number of correctly classified images in the two minority classes though they were less than the half of the test images (using our test set). We also noted that the number of correctly classified images in the other classes was smaller than the results we had in $\mathcal{M}_2$. For example, in $\mathcal{M}_2$ we had 100% correctly classified test images of `Species8` while in $\mathcal{M}_3$ they were reduced to 75%. Maybe with the balanced dataset the model couldn't distinguish precisely between the species due to the larger presence of `Species1` and `Species6`. So, we tried to make the new model $\mathcal{M}_4$ a bit more complex adding more filters, another dense layer for classification and using some regularization techniques to avoid overfitting. We applied *Ridge* and *Lasso regularization* both in the convolutional layer and in the dense layers. We also tried to remove the classifier and substitute it with a *global average pooling* layer, but this didn't make any improvements. At the end we discovered that without any *regularization*, even without *dropout*, $\mathcal{M}_4$ reached a better accuracy: 0.646 on *CodaLab* and 0.688 on our test set.

Before going on pre-trained models, we wanted to build a more complex model, $\mathcal{M}_5$. It was composed by two convolutional layers with 2×2 *max pooling*, three inception modules like those implemented in *GoogLeNet* followed by 2×2 *max pooling*, GAP and one dense layer with *dropout*. The first part aimed to convert the input image in low level features, the second to extract high level features and the third to classify the input. Two inception modules have a 3-path structure: two paths are composed by a 1×1 convolution followed by one or two 3×3 convolution while the third path is a 1×1 *max pooling* followed by a 1×1 convolution. The third inception module is composed by two paths equal to those in the previous modules with two layers. The outputs of each inception module are concatenated depth-wise and before being given in input to the next inception module, there is a 2×2 *max pooling*. Unfortunately this network, using *data augmentation* and the balanced dataset, didn't give any improvements obtaining 0.59 accuracy on *CodaLab*. So with all the information gathered while working on our simple model, we switched to more complex pre-trained networks trying to improve a lot the accuracy.

2.2 Transfer Learning with pre-trained models

For the models explained in this section, we used both *data augmentation* and *oversampling* as described in the previous section.

The first pre-trained model we chose was *MobileNet*, since it is suitable for the classification of low-resolution images in resource-constrained scenarios. Indeed, as the name of the network suggests, it is meant to be deployed on mobile devices, thus, its architecture is simple enough to perform inference with limited memory and computational power. Even though we knew that this was not our case, we wanted to give it a try. Therefore, we built the first model with *MobileNet*, $\mathcal{M}_1^{MN}$, by freezing all its backbone weights and adding on top of it 2 dense layers with *dropout* for the classification task. We immediately noticed while training that the validation accuracy was around 0.8, most probably because these pre-trained models, being trained on million of images (in this case on the *ImageNet* dataset), can extract meaningful low-dimensional features. Then, we performed *fine tuning* by setting the last blocks of the CNN to be trainable and by reducing the learning rate to `1e-4`. $\mathcal{M}_1^{MN}$ obtained an accuracy of 0.757 on *CodaLab*.

At this point, we came back on the problem of *outlier images*: we deleted some of them by going through the dataset, in order to avoid "confusing" the model. Also, a quick analysis with the library `fastdup`[3] helped us finding others "confusing labels" images that we then deleted. This improved the accuracy on our test set and therefore we submitted a new version of the previous model but trained on the "clean" dataset, $\mathcal{M}_2^{MN}$, which obtained an accuracy of 0.764.

[2] see `oversampling.py`

[3] see `fastdup_analysis.ipynb`

Later, we implemented k-*fold cross-validation*[4] to evaluate different configurations for the classifier considering layers with 256, 128 and 64 neurons and including different *regularization* techniques such as *Lasso*, *Ridge* and *dropout*, each of them with different rates. Unfortunately, given the limited computational resources and time, it was unfeasible to complete the validation on all the desired configurations even with a small k (e.g. $k = 5$). Anyway, it seemed that the configuration composed by 2 dense layers with 256 and 128 neurons regularized by *Ridge* with $\lambda = 10^{-4}$ and the learning rate set to `1e-4` was the best one. We submitted $\mathcal{M}_3^{MN}$ with this configuration achieving an accuracy of 0.787 on *CodaLab*.

Another improvement was achieved by introducing *test-time augmentation*[5], which consists in averaging the output probabilities of the image to be classified, i.e. **x**, with images obtained by augmenting **x** via a certain number of random transformations (the same used in the training phase). A model submitted with this prediction technique, $\mathcal{M}_4^{MN}$, obtained an accuracy of 0.814 on *CodaLab*. By re-training the same model with *deeper fine tuning* phases and by accordingly reducing the learning rate, we improved the accuracy obtaining a score of 0.83 on *CodaLab*.

Once we noticed the early performance limitations of *MobileNet*, we decided to opt for new models: *EfficientNet* and *ConvNeXt*. Several versions were tried for each of the above models, namely: *EfficientNetV2B0*, *EfficientNetV2B1*, *EfficientNetV2B2*, *EfficientNetV2S*, *EfficientNetV2M*, *ConvNeXtSmall*, *ConvNeXtBase*, *ConvNeXtLarge*, and *ConvNeXtXlarge*. Given the numerical constraint on the number of submissions, each model was previously evaluated on our validation and test sets and then we selected the most promising versions: *EfficientNetV2B0*, *ConvNeXtBase*, and *ConvNeXtLarge*. The first result 0.8373 on *CodaLab* was obtained using *EfficientV2B0* on the "clean" dataset by first implementing transfer learning and then two consecutive *fine tunings* first up to 237 layers and second up to layer 149. In order to improve learning on the minority classes, the model was trained on the oversampled dataset reaching 0.8635 on *CodaLab*. By applying *test-time augmentation EfficientNetV2B0* was able to reach 0.8912 accuracy on *CodaLab*. Still not satisfied with the results obtained, we selected a more complex model: *ConvNextBase*, reaching at the first submission 0.9008 of accuracy. The second version of *ConvNext* was *ConvNeXtLarge* with which we reached an accuracy of 0.9134 by applying a first *fine tuning* up to layer 221 and a consecutive fine tuning up to layer 180. Despite the complexity of the model, the result obtained on the first species of plants was still disappointing compared to the other species, so we decided to train the model by setting `class_weights` to the *loss function* that gave more importance to the first and sixth class and an equal, albeit lower, importance to all other classes. By using this `class_weights`, the accuracy became 0.9270 on *CodaLab*. Aware of the complexity of our models, we tried to reduce the variance of the predictions using model ensemble[6]. We took the three best versions of the models trained so far ($\mathcal{M}_4^{MN}$, *EfficientNetV2B0* and *ConvNeXtBase*) and we trained them on the same dataset. We assigned to each sample the class with the highest sum of the three models output probabilities. However, the accuracy obtained in our test set did not exceed the result of the *ConvNextBase*, as expected given the difference in performance of the three models. Furthermore, given the computational difficulty in training so many models characterized by performances similar to our best model so far (*ConvNeXtLarge*) and the difficulty in loading heavy models on *CodaLab*, we realized that ensemble modeling was not a viable option to try improving our results. We tried instead to perform k-*fold cross-validation* on *ConvNeXtLarge*, as done previously for the model $\mathcal{M}_3^{MN}$, but, again, we decided to stop k-*fold cross-validation* earlier due to limited time remaining. Therefore, we used the best configuration among the ones analyzed[7] to choose the final version of *ConvNeXtLarge* using just one dense layer with 256 neurons for the classifier. Finally, we increased the number of random transformations performed in the *test-time augmentation* to improve the accuracy on *CodaLab*. Doing so, we reached our best accuracy of 0.9357. Once the development phase ended, for the final one we submitted the same best model (*ConvNeXtLarge*) but we reached 0.9218 of accuracy.

[4] see `kfold_Xval.ipynb`

[5] see `model_TTA.py`

[6] see `model_ensemble.py`

[7] see `kfold_Xval_convnext_results.xlsx`