Artificial Neural Networks Homework 1 Report

Residual Sum of Students

Juliette Daniel, Davide Rigamonti, Raul Singh

INTRODUCTION

This is a report aimed at analyzing the work of our group through the first challenge held for the Artificial Neural Networks course at Politecnico di Milano under the supervision of Prof. Mattucci during the month of November 2022.

THE OBJECTIVE

The project assignment consisted in using the tools that were seen in the course lessons in order to solve a multiclass classification problem centered around the recognition of 8 different species of plants.
We decided to use the recommended frameworks (TensorFlow and Keras) in conjunction with Python Jupyter Notebooks, which allowed us to tackle the problem in a familiar and effective way; as a collaborative versioning tool we used GitHub and we also used Google Drive as a shared storage for bigger files.
Lastly, this report was written utilizing $\LaTeX$.

THE JOURNEY

Basic Model

We started our journey for this competition by trying the simplest approach at our disposal in order to familiarize ourselves with the task and the submission procedure: we took the convolutional models seen during the exercise sessions and we tried to adapt them to our use case without changing the underlying structure too much.
The resulting net had 5 convolutional layers utilizing powers of 2 (going from 32 to 512) of 3x3 filters interleaved by Max Pooling layers, a Flattening layer and a classifier composed by a Dense layer of 512 neurons between two Dropout layers with a drop rate of 0.3.
All of the hidden neurons of Dense layers use a ReLU activation function while the 8 output neurons use a Softmax since it fits well the multiclass classification scenario.
The resulting accuracy on the validation was around 50% at best, which is quite expected since the net was prone to overfitting due to its structure.
By adding a rescaling factor of 1/255 as a preprocessing operation for the inputs of the net we noticed an improvement in validation accuracy of around 5%, this is an expected result since having bounded values in a small interval is a wanted feature for a dataset and constraining the RGB 0-255 values to the [0,1] interval exactly does that.

GAP and Standardization

We, then, decided to pursue two parallel lines of development in order to subdivide work more easily: one dedicated to the Global Average Pooling (GAP) layer and data preprocessing, while the other was centered around Data Augmentation. The GAP layer addition strictly improved the results (75% validation accuracy) and cemented itself in all of the latter designs in place of the Flattening layer due to its noticeable advantages such as increased robustness to spatial transformations (mainly shifts) and overfitting reduction due to an overall lighter network.
On the other hand we also experimented with data standardization on the whole dataset (either by manual preprocessing or by inserting appropriate preprocessing Normalization layers inside the network), unfortunately the results weren't too different from the far more easier and accessible rescaling.
Data standardization is applied in order to redistribute data along a normal distribution centered around 0 and with standard deviation equal to 1, in our case each RGB channel is standardized separately.

Data Augmentation

Data augmentation looked like it might have come in handy for our problem given the size of the dataset and the very unbalanced classes. From our observations, prediction accuracy for classes like *Species1* suffered a lot due to the small number of samples. Data augmentation, by shifting, flipping, rotating, zooming and translating samples of the dataset could have brought a lot more variety in the training data. As a blueprint we used the implementation seen during exercise sessions, making use of the data augmentation features of *ImageDataGenerator* from Keras.
However, at first, performance did not seem to improve. We tried to experiment with different image augmentation parameters and, at last, by toning down the transformations, we observed a huge leap in performance, with around 75%-76% in validation accuracy.
After submitting this fresh model on Codalab, we experienced an unusual drop in accuracy between our validation accuracy and Codalab's test accuracy, with a difference of around 10% (way more than the 2% difference we usually observe). After tinkering a bit with the model, we came to the conclusion that the *ImageDataGenerator* was causing troubles. In fact, we used the validation_split parameter to let it automatically split our data between training and validation. However, we found out that the *ImageDataGenerator* applied image augmentation also on the validation dataset. This made our validation set incredibly unreliable, hence the drop in accuracy on Codalab.
With some workarounds we managed to apply data augmentation on the training set only and, as expected, the increase in performance wasn't as prominent as before, with a validation accuracy around 65%-66% (applied on our base model which had a 55% validation accuracy before), but we got much similar results on Codalab, making our validation set more truthful and accurate. In addition, we noticed that classes with fewer samples started getting decent accuracy results.
It came to our knowledge that data augmentation also helped a lot in decreasing overfitting. Given also the benefits mentioned earlier, data augmentation became a staple in our next models combined with GAP.

Batch Normalization

After tinkering and implementing the previously mentioned techniques, we came to a point where we could not improve our model any further. One of our last attempts before changing our model from the ground up was to implement Batch Normalization, making use of Keras' Batch Normalization layers.

We tried different combinations between layers of our models and Batch Normalization layers. At first, we tried alternating Dense layers of the classifier with Batch Normalization layers and, after running the model a couple of times, we were not able to observe any difference in accuracy. We then tried to alternate Convolutional + MaxPool layers with Batch Normalization layers, but to no avail. Sometimes we even experienced lower performance in validation accuracy. Lastly we tried to combine both Batch Normalization in Convolutional layers and Dense layers, but the results were the same if not even worse sometimes. We decided to give up on Batch Normalization and move on.

Validation/Training Ratio Experimentation

The last attempt to squeeze more performance out of our current model framework was to change the ratio of samples between validation set and training set. We know that a larger number of samples usually leads to better performance, therefore we thought that increasing the number of training samples would lead to an accuracy improvement. Up to this point, our validation/training ratio was 0.2. We tried lowering it first to 0.15, then to 0.1 to get a larger training set. This led to an apparent boost in performance, with a validation accuracy of 80%. However, when submitting the model, we experienced a significant drop in accuracy on the Codalab test set, around 7%. We came to the conclusion that while having more samples surely has benefited our model, the validation set, by being smaller, became much more inaccurate. Therefore, choosing the best weights based on validation accuracy became less and less reliable, leading to mediocre results when submitting the model.

We came to the conclusion that reducing the validation/training wasn't worth it, so we decided to stick with a 0.2 ratio.

Transfer Learning

When we realized that improving on our own designs was getting harder and harder, we decided to start implementing transfer learning by utilizing pre-constructed and pre-trained networks as feature extractors by removing the "top" part of the supernet and training/using a custom classifier in its place.

The main models that we decided to focus on were *VGG16*, *InceptionV3* and *ResNet50* (out of these 3, *VGG16* was definitely the one that gave the best results, mainly due to its shallowness which gave less room for overfitting), which, in combination with data augmentation, GAP and data rescaling reached a validation accuracy of around 78% (for *VGG16*).

Without the help of previous techniques the transfer learning models were too prone to overfitting to give any satisfying result.

Fine Tuning

After transfer learning we also experimented with fine tuning, which gave results that were only marginally better, this time focusing mainly on *VGG16* and *InceptionV3* architectures (the latter didn't give any meaningful result, probably due to its complex and deep structure, prone to overfitting).

By freezing some layers of the supernet and allowing for the rest to adapt we obtain a much more flexible and adaptable model, at the cost of more parameters to train and therefore more room for overfitting.

We found some peculiar combination of *VGG16* with 12-13 layers frozen which gave satisfying results breaking the 80% barrier on the validation accuracy.

Class Balancing

By implementing a quick script able to evaluate the performance of our model on the various classes (employing the *classification_report* function from *scikit_learn*) we noticed that, even while performing data augmentation, our performance on some of the unbalanced classes (in particular *Species1* and *Species6* have an extremely low number of total samples) was the bottleneck for improving the validation accuracy of our models, therefore we tried two different solutions to solve this problem.

At first we tried to apply some rudimentary upsampling by performing data augmentation on the samples of species 1 and 6, however it didn't really work out as expected since the results were even worse than before, this is probably due an implementation error on our part and we still believe that it should be the right technique to apply in this case.

After this attempt we resorted to experimenting with class weights, which gave our model a small edge over the critical species which behaved well with our weaker models but resulted in an overall drop of performance for the stronger ones.

Cutmix

At this point we found the suggestions inside the Logbook, which led us to experimenting with new techniques such as CNN-SVM classifiers, cutmix, tuning and ensemble learning.

As far as the cutmix is concerned, we tried to insert it inside our data augmentation pipeline using a custom script and we obtained some surprising results; however, in the process of tuning the data augmentation hyperparameters we quickly realized that cutmix only hindered the transformations that we were already doing, probably due to some problems in label preservation for setups that heavily relied on data augmentation.

In the end we opted to intensify the data augmentation parameters leaving behind the cutmix transformations.

Tuning

We also ran some experiments using the *keras_tuner.RandomSearch* and *keras_tuner.BayesianOptimization* in order to optimize model hyperparameters and net structure, however we reckon that our models were a bit too premature to fully benefit of an automated hyparameter tuning technique; undoubtedly, with more time on our hands, this would have been a useful refining tool.

Transfer Learning + Fine Tuning

To further improve the performance of our models, we tried to combine both transfer learning and fine tuning. The main idea was to first train our classifier while keeping a supernet from *keras.applications* frozen as a feature extractor. After that, un-freeze some of the last layers of the supernet making them trainable and training again with a small learning rate. We applied this framework to all the three networks we used before when trying out transfer learning and got a slight improvement in accuracy, with 81% on the validation set on average.

At this point it seemed that the model wasn't going to improve any further. We tried to incorporate data augmentation in the steps mentioned above, with different combinations (only in the first step, only in the second step or in both steps) but our model wasn't really improving.

After hitting this plateau, we tried to swap the two steps: we first train the classifier together with some layers of the supernet unfrozen, then we freeze the whole feature extraction network and train the classifier alone. Against all expectations, we experienced a groundbreaking improvement in accuracy. We decided to apply data augmentation only on the first step, since we had to train a larger number of parameters. With this configuration, we got up to 84%-86% accuracy on the validation set depending on the network chosen as feature extractor.

We tinkered with our current setup to improve it even more, trying even counter-intuitive solutions, like the one mentioned earlier. We tried to modify our first step. Instead of training the classifier plus some layers of the feature extractor, we decided to train with data augmentation the whole model together with a small learning rate as a first step, then as a second step train only the classifier using the training dataset without any augmentation.

We observed astounding results. With this methodology, making use of networks like *DenseNet* and *EfficientNet* as feature extractors, our models reached around 90% accuracy on the validation set. We tried to use this framework with almost every model at our disposal in *keras.applications*, employing them as feature extractors and leveraging their own preprocessing layers. Some of them reached a similar accuracy (around 90% as before), some of them performed a bit worse.

At last, we tried a model available only in the latest versions of Tensorflow: the *ConvNeXt* model. Among the different implementations of the model we first tried the *ConvNeXtBase*, which seemed to be a good middle ground. However, having a staggering amount of parameters, the network struggled to learn anything and in some cases was too prone to overfitting. We then gave a try to the smallest *ConvNeXt* available, the *ConvNeXtTiny*, which stored way less parameters. Surprisingly, using the same framework for training, the *ConvNeXtTiny* scored an incredible 92% in accuracy with the validation set, becoming our best model, obtaining 89% on the Codalab test set.

Ensemble Models

With little to no remaining time on our hands we tried to ensemble some of our best models to see if they would work any better together rather than alone.

Unfortunately the best models we had were much too complex and similar to give any meaningful improvement (although there were some minor overall accuracy improvements); we also tried with some weaker models and, as expected, the improvements were more evident but they still scored an overall lesser accuracy.

After experimenting with some ensemble network setups we found that a 10 neuron Dense layer before the outputs and after an Ensemble Concatenation layer for the various networks worked best for us.

As far as the number of subnets used we experimented with 2 to 5 subnets at the same time but we settled at 3, since the resulting models are heavy and the overall performance is influenced negatively by the less accurate networks.

In the end, the validation accuracy of our ensemble models oscillated between 90% and 93% but we noticed that they were a little bit more consistent on the test set employed on Codalab with respect to the other kinds of models.

THE RESULT

Our final model was an ensemble model composed of a *ConvNeXtTiny*, a *Densenet* and an *EfficientNet*, all trained with transfer learning + fine tuning and data augmentation on the whole training dataset.

Although we are satisfied with our result we believe that with more time we could have developed a much better model overall since some of our approaches were quite unorthodox and could have definitely been improved with more care, for example: when we utilized the non-augmented dataset during transfer learning (second step) we noticed that there was no real improvement over time for our classifier and the only gain in accuracy and loss was at the start of the training, which basically meant that our network was already "ready" and we were just doing extra steps.

Our models could have been improved in many ways and there were a lot of paths that were left unexplored but could have been beneficial for the networks, for example stratified resampling, more tinkering with the parameters of the optimizer, custom loss function and regularization could have made the model more robust and helped in the class balancing problem. In addition, we didn't really expand on custom input preprocessing aside from rescaling and standardizing, which were replaced by the appropriate preprocessing layer for each network imported through *keras.application*. Furthermore, some of the recommended tools such as the *keras_tuner* and the various special data augmentation techniques (cutout, cutmix, mixup,...) gave good results when utilized on weaker models, but when it came to integrating them in our best performing models they always gave underwhelming results; it's not easy to pinpoint a single failure point for this behavior since it was mainly due to multiple design faults.

As a last note, we weren't really expecting for this last model to perform better than others, therefore the jupyter notebook included in the report may not reproduce 1:1 the architecture of the submitted model on Codalab, however, the resulting models have similar performance and the main components are still the same.