Artificial Neural Networks and Deep Learning
Homework 1

Nicola Dean
10674826
nicola.dean
@mail.polimi.it

Marco Fasanella
10617541
marco.fasanella
@mail.polimi.it

Raffaello Fornasiere
10790353
raffaello.fornasiere
@mail.polimi.it

Christian Spano
10823764
christian.spano
@mail.polimi.it

1 Introduction

In order to touch all the important aspects of the procedure of finding the best solution to this classification problem, we started from our own self-made model to more sophisticated methodologies.

We can summarize our approaches in: Vanilla network, Transfer Learning, Fine Tuning, Ensemble. Furthermore in all the attempts made, we used two classes, specifically created to automate and support the model creation procedure: **Dataset Helper Model Helper.**

Through the continuous attempts, and support of methods in the two helper classes, we managed to find our best model and reach 0.8691 accuracy in the competition.

2 Dataset Helper and Model Helper

In order to ease the development of our best solution, we thought it would have been useful to create two adhoc helpers. As a matter of fact, a lot of code is often repeated or many lines of code are needed just to plot or to get insightful results (such as the confusion matrix or the accuracy). In particular, what we did was to create two helpers:

- **Dataset helper**: for dealing with everything related to the dataset; for instance (just for mentioning the most relevant functions) we created functions to perform *augmentation* and *cut-out/cut-mix* of the dataset. Other functions were intended for processing data (e.g., normalization)

- **Model helper**: for dealing with everything related to the models; this helper was intended for easing the construction of models. For instance, we create some functions for plotting insightful graphs (e.g., confusion matrix, residuals, predictions, etc.) and some other functions for managing the models (e.g., save the model in a specified directory or create checkpoints during training)

At the end, these helpers indeed turned out to be really helpful. Just with a line we had everything we needed. We firmly think they will turn out to be crucial also in other Deep Learning projects.

3 First try: vanilla network

As first attempt we simply designed a basic CNN with 3 layer of Conv+Pool+Relu and a final flatten plus 128 neurons in the Dense layer (along with dropout). The initial result was pretty bad, 44% on codalab but we managed to experiment a lot of different variation of this network, by adding multiConv layers, Batch Normalization, multiple dense layers and so on.

We also learned a lot of what make sense and what was ”excessive” and cause ”vanishing gradient” like putting too much convolutions of same size consequently.

3.1 Batch Normalization

A first attempt was also adding a Batch Normalization + ReLu Activation Layer before our Pooling layers. This lead to poor result due to the fact that the network was too small.

3.2 Our homemade CNN

After some attempts on small vanilla networks with poor results we tried to improve the network by adding some more layers with more filters.

We combined different parameters: kernel size, number of filters, number of layers and units of the dense layers. After some trials we noticed that we were not reaching any significant improvement (very low test scores, around 0.5/0.6 of accuracy). All of these trials were made with a small dataset (4000 images), but after the work done on augmentation (section 3.3) the same model performed much better reaching 0.80/0.85 on test set.

We noticed though the model was not much reliable since on Codalab it performed around 0.78. We attach to this report an image of this architecture.

3.3 Augmentation

Since vanilla networks were not providing good results we decided to focus on data. First of all we decided to generate a number of images so that the ratio $images_{specie_i}/tot_{images}$ was almost the same for each species $i = 1, \ldots, 8$.

Then we started exploiting all the functionalities of **ImageDataGenerator** in order to generate as much images as possible so that the network could extract them without overfitting. By doing this we started reaching higher accuracies, deducing that the image dataset should not be smaller than 10k images. With further search we noticed smaller – but significant – improvements until 25k images.

After that we decided to improve the generalization ability of the network by augmenting the images with **Cut-**

Mix, **MixUp**, **GridMask** and **Grayscale**. This turned out to be another small but significant improvement on the accuracy, reaching 3/4 percentage points on test accuracy, depending also on the model used.

4 Transfer Learning and Fine Tuning

We then noticed that we needed a big change on the approach to use, because the homemade CNN was performant, but not enough. So **we started using transfer learning and Fine tuning.**

4.1 Approach: Freezing Layers

The *modus operandi* that will be used from now is: freezing all layers while training on our augmented dataset the **keras.application** CNN, and then unlock a small amount of layers to the second phase of training (fine tuning) as near as possible to the output.

4.2 VGG16

The first network we decided to utilize with this approach was VGG16. We started from this network mainly because, generally speaking, it has given great results in many different Deep Learning tasks; so, we thought it was a good network to start with. Indeed, at the end, it did not deceive us.

More in details, at the very first we trained this network neither applying augmentation nor fine-tuning. By the way, the network architecture was equal to the standard VGG16 one, with 256 neurons in the Dense Layer at the top. However, as expected, we got poor results (an accuracy of about 60%). Applying data-augmentation, the accuracy risen up to roughly 70%, that is an improvement in accuracy of 10%.

At this point, to boost up even more the accuracy, we decided to do fine-tuning treating the number of freezed layer as an hyperparameter to be learned. Indeed, we trained the network with k freezed layers and we pick up the one such that

$$k^* = \underset{k}{\operatorname{argmax}}\{test\_accuracy_k\}$$

4.2.1 Results

The best number of freezed layers turned out to be $k = 8$. All the results based on k are reported in table 1. Setting this parameter to this 'optimal' value, we got a test accuracy of 0.8249, against 0.8325 on Codalab. This has been our first best result. That's why further tests have been done. For instance, we tried to change the number of neurons, add more Dense Layers, and change some parameters of the augmentation but all these tests failed since they made the test-accuracy dropping down.

The only test that passed successfully – even though with very little improvement – was to apply the more advanced augmentation we mentioned in subsection 3.3. As a matter of fact, this improved dramatically the test accuracy boosting it up to roughly 0.94. However, it provided very little improvement on Codalab, i.e., an accuracy of 0.8381. No other best results have been received with this network and since we were stuck with these results, we thought it was reasonable to explore other pre-trained networks.

Freezed Layers	Test accuracy
6	0.7881
8	0.8249
10	0.8051
12	0.7994

Table 1: Results with Transfer Learning and number of freezed-layers for VGG16.

4.3 VGG19

VGG19 is a convolutional neural network that is 19 layers deep, so we kept the freezed layers in the range of the first 8-14.

Different data augmentations (with different seeds and augmentation parameters) were performed between the two phases, to even increase randomisation in the two training processes.

4.3.1 Results

Freezed Layers	Accuracy	Precision	Recall	F1
8	0.8169	0.7989	0.7651	0.763
9	0.8225	0.8181	0.7682	0.7776
10	0.8338	0.8161	0.7929	0.8001
11	0.7577	0.7109	0.715	0.7048
12	0.7944	0.766	0.7504	0.7489
13	0.8028	0.7806	0.754	0.7596

Table 2: Results with Transfer Learning and number of freezed-layers for VGG19.

4.3.2 Considerations

One of the most particular observation that we can made after experimenting the first attempts of freezing, is that freezing the net until the pooling and not between convolutions leads to a better accuracy.

4.4 Xception

Xception is the evolution that Google proposed to their **Inception-v3** model, this model contain **131 layers** of **depthwise separable convolution**.

We decided to try this model to see if this particular kind of convolution would have extracted better features for our model and if this would have increased Specie1 and Specie6 F1.

The best result was given, surprisingly, by unlocking the full network and using a dense of 256 with global avg pooling. However, unfortunately, the performance obtained was not so different from our best model at the time, VGG16.

4.5 Other Models

In parallel to fine tuning we had tried to apply some changes to the "home-made CNN"; one of them was to add skip layers to it to avoid **vanishing gradient**.

We failed to implement them in a performant way, but

Freezed Layers	Accuracy	Precision	Recall	F1
No Tuning	0.5734	0.5942	0.5344	0.5376
20	0.6328	0.5903	0.5855	0.5852
1	0.8136	0.7869	0.7751	0.7763

Table 3: Some of the best results of FineTuning using Xception (other tests were done but they provided similar results).

this bring us to the next model of this FineTuning section: **Resnet50-v2**.

We thought that maybe we were doing something wrong with skip layer, so to prove it we decided to give a try to Resnet.

4.5.1 Resnet

Like for VGG and Xception we had started by trying different dense layer dimensions and learning rates. We found out that by changing the final dense layer from 256 to 512 to 1024 was not affecting significantly the results. Therefore, we kept the lower one for faster training.

That said, we could not find satisfying results, no more than 0.7401 on local test after fine tuning. Maybe, with more trials, it would have improved but we did not want to lose precious time on non-promising models. Thus, we moved on.

4.5.2 GoogleNet

Just to have a broad exploration and not leaving any doubt on which could be the most power pre-trained network, we decided to give a chance to GoogLeNet. At first, just for clarity, we did not analyse this network in deep and the reasons are here explained.

As usual, we trained the standard GoogLeNet architecture but we noticed immediately a bad behaviour: just after a bunch of epochs the model started to overfit. Besides this point, the accuracy on the validation test fluctuated abruptly passing, for instance, from 0.40 to 0.80 at each epoch. This make us a little distrustful of this network, also because we already found other more promising networks. Thus, we abandoned any further tests on this network.

4.6 EfficientNet

As last option we decided to give a try to one of the 'big' models on keras.application. Initially we were discouraged by the huge number of parameter (e.g., on Google Colab we hadn't managed to load it due to RAM issues) but we wanted to give it a try since its architecture was different from the already tried models beside the fact that on keras.application was the best on Top-1 accuracy.

As already said, it was a little hard to work with this model given our resources so we used an approach not 100% correct. That said, to select the best hyperparameter, we started to train only few epochs and keep running *only* if it was promising (probably not the best approach). The model resulted to be promising, since the simple transfer learning managed to get 0.71 of accuracy locally.

Afterwards, we started experimenting fine tuning. We discovered that with a big number of freezed layers during fine tuning there was no significant improvements. Improvements come when the number of freezed layers was between 200 and 100. Precisely this last one was the best with an accuracy of 0.85/0.86.

After this, since both Resnet50 and Xception was performing well with '*full* unfreezed', we tried, for curiosity, to apply 30 more epochs with full unfreezed layers and discovered this increased model performances of 2/3% locally.

So, the best option was: **Transfer - 100 layer run - 1 layer run** with 0.88 accuracy in local and 0.849 on Codalab. We tried also the opposite, i.e., (**Transfer - 1 layer run - 100 layer run**). However, this was not effective; it performed even worst.

5 Ensemble

The ensemble technique consists into combining multiple models together to create a more robust and accurate one.

The idea behind it is that we can take our best models, and apply a majority voting on the predicted species. This will grant more robustness, since wrong choice come from majority of models predicting wrong species.

The improvement was as expected, pretty high, but locally we get a "biased" result since the EfficientNet was trained using a different random seed (and so different test/train split). Moreover, was too late to retrain such a huge model. Thus we will use CodaLab accuracy as metrics, obtaining 0.8789 during the **Development phase** and 0.8691 on the **final phase**.

This ensemble combine together our best 7 models: Xception, VGG (either 16 and 19), Resnet, EfficientNet, good Home-made CNN, and First Home-made CNN.

5.1 Better approach

At this moment the time was end, and we could not do more tests but a thing we would have wanted to do was to apply ensemble with some more "intelligently" chosen models. For instance: choose models with a particular confusion matrix to maximize the concept of majority voting, as CNN that work well on certain species to avoid false positive/false negative on that species.

6 Conclusions

The best technique was Ensemble that grant us the best result. Initially, we did not feel like this was the best way but when our models get stucked in a plateau of accuracy we decided to try it because we thought that put our best models together would have covered each other weaknesses.