Homework 1 - Image classification

Amabili Laura - 10598083

Bavaro Marcello - 10597421

Maniscalchi Luca - 10583576

Simone Sarti - 10580595

# **Introduction:**

The task consists in the classification of different types of leaves. The dataset provided includes 17728 images splitted in 14 folders, one for each type, and a relevant characteristic is that the folders don’t contain the same number of elements. The dataset is unbalanced.

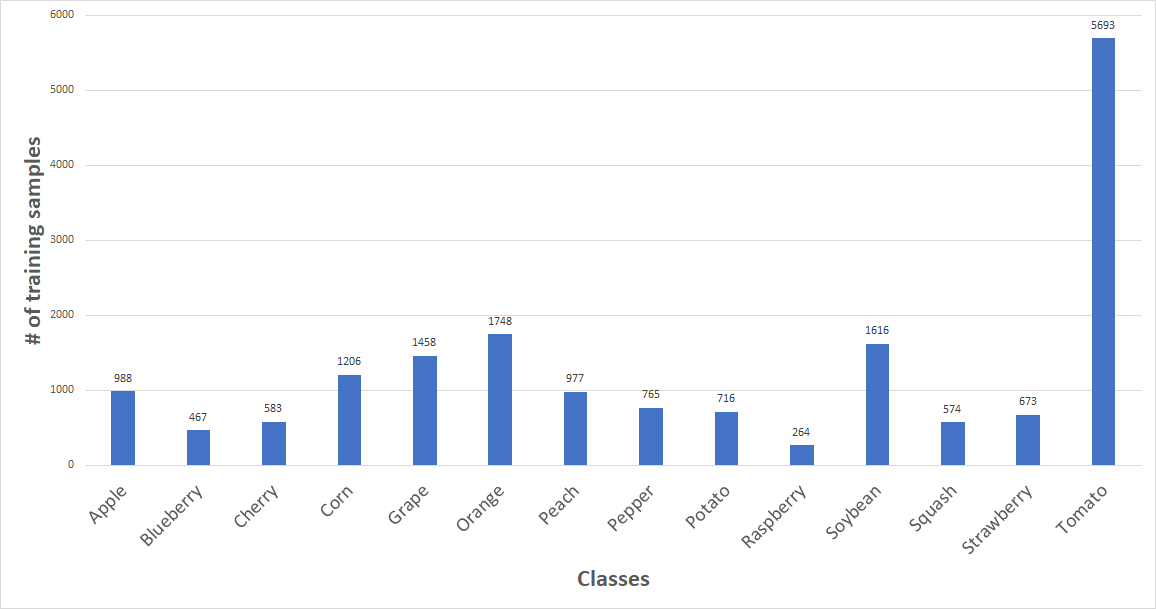


Image distribution over classes

For instance, the ratio between raspberry and tomatoes is more or less 5:100, i.e., for every 100 tomato leaf images, there are only 5 raspberry leaf images.

# **Dataset:**

Images are saved in the .jpg format and all have the same shape and resolution (256x256).

Due to the different number of elements in each class, the algorithms tend to get biased towards the majority values present and don’t perform well on the minority values. To fix this we implemented class weighting. This can be achieved by giving different weights to both the majority and minority classes. The difference in weights will influence the classification of the classes during the training phase. The whole purpose is to penalize the misclassification made by the minority class by setting a higher class weight and at the same time reducing weight for the majority class.

The dataset provided is relatively small, so we decided to overcome this problem using data augmentation. In particular, we set the generator to modify the images in order to produce rotations, zooms, shifts and flippings, moreover we also worked on the calibration of the brightness and of the shear factor. We also set fill mode to constant because in our opinion it’s important to preserve the original shape of the leaves.

We used an 80/15/5 split for training, validation and testing and only used the dataset we were given, no additional images were added.

# **No transfer:**

At the beginning we worked without using transfer learning and the result is a model formed by 6 x (conv. layer + batch normalization layer +MaxPool layer) followed by a globalAveragePooling which performs better than flattering during our tests. The last part is a quite standard fully connected network in which are included two dropouts, in addition to all the layers implementing regularization. The training of this model is the longest among the ones we created, because it hasn’t previous intrinsic knowledge. We trained it for 250 epochs and the final results showed a score greater than 85%.

**VGG16**

The base model we used for transfer learning was the VGG-16 architecture with pre-loaded ImageNet weights. Our first base implementation simply provided for class weighting together with a basic fine-tuning, in which only the last 5 layers of the VGG-16 architecture were set to trainable. This led to a score of 0.7. A big step ahead was made when we introduced data pre-processing (using VGG-16 preprocess\_input function) and a kernel regularizer that allowed us to reach a score of 0.91. We then obtained little improvements by tuning Data Augmentation parameters to improve our dataset. We eventually decided to perform one final test by making use of a different optimizer, so we replaced Adam with AdaBelief which turned out to be a winning choice. Our best VGG16 implementation scored 0.93.

**XceptionNet**

After some time working with VGG16 as a testbench for transfer learning, we decided to go with a more advanced architecture like XceptionNet, as it offered one of the best performance/parameters tradeoffs (excluding EfficinetNet).

The first iteration of this network only had transfer learning, no fine tuning, using as top part a custom FC network (see next paragraph). The next step was the addition of fine tuning. Looking at the structure of the network we decided to keep the first 86 layers frozen and allow the network to retrain the rest. We had to pay attention to the fact that XceptionNet has branches in its structure, so we made sure to make trainable only the layers after the one which reunited the branches of the fifth repeated block. The batch normalization layers were kept frozen as indicated by the Keras website (more on this later).

Due to the facts that most of the information we found suggested using a learning rate between 1e-4 and 1e-6 when fine tuning a pre-trained network, that in class we used a learning rate of 1e-4 for the VGG16 example (a lot less layers unfrozen), and that we had a not too small number of training examples, we went with a learning rate of 1e-5. Successive experiments showed that 1e-5 was a better choice than 1e-4.

**Fully Connected Network**

The FC network we concatenated to the transfer learning networks is composed of 3 hidden layers and evolved over time according to the performance of the submissions (mostly of iterations of XceptionNet, but also taking in improvements from parallel testing on VGG16), however it seemed to be well built right from the beginning, therefore we refined it through a series of adjustments rather than complete changes in the structure. The main changes involving this FC network were:

* Increase the number of units in the first dense layer from 256 to 512, to avoid creating a bottleneck at the exit of XceptionNet (and EfficentNet), both outputting 2048 features. Reducing the number of units to less than 1/4 could have hurt performance. The successive layers saw an increase in number of units accordingly
* Move from ReLU to LeakyReLU (alpha=0.2 seemed common), and later to ELU
* Replacement of the Flattening layer with a GlobalAveragePooling layer, which not only increased performance by 1%-2%, but also slashed by almost 5 times the size of the output files of the network we were using at the time
* Removal of the dropout layers at the flattening layer and at the last layer before the output of the network. We kept 2 dropout layers, at the first hidden layer with a rate of 0.3, and at the middle layer with a rate of 0.6 (sources suggested a good rate being between 0.5 and 0.8)
* Removal of Kernel regularizes, as they hurt performance in XceptionNet

**EfficinetNet**

At last, we decided to use all the good things we learned from VGG16 and XceptionNet into an EfficientNet transfer learning network. Fine tuning was implemented immediately.

We tested many different combinations of fine tuning and learning rates, and the results were mixed. The interesting (and unexpected) result was that unfreezing the whole network was the best option in the first phase of the competition. Not only that, but we could not match the performance we obtained when also the batch normalization layers were unfrozen, which was explicitly written NOT to do in every fine-tuning page on Keras website (2-3% improvements when unfrozen). Using this network we achieved our top score of 96.23 in the first phase of the competition. In the second phase, the same network performed almost 3% worse, and the best EfficientNet version was the one with fine tuning of just 3/7th of the network with frozen bath\_norm layers, our top (and overall best) score here was 94.34. This particular network was the one we were expecting to be the best during the first phase as well, but wasn’t.

**EXTRA - Optimizers**

We performed several training sessions using different optimizers. In particular we have tested: Adam, RMSprop and AdaBelief. Each of them has been initialized with the same learning rate for a fair evaluation. The first comparison we made was between Adam and RMSprop. The former outperformed the latter, in all tests the models trained using Adam generalized well. With regard to Adam and AdaBelief we surprisingly obtained different results depending on the adopted architecture for transfer learning. The implementation of AdaBelief is nearly identical to Adam. The only difference is on the step size prediction. The authors of AdaBelief pointed out a significant problem with Adam prediction and showed that on many models AdaBelief outperforms Adam, SGD, and many other optimizers across a wide variety of applications. Unfortunately that was not the case in our environment. For instance, AdaBelief outperformed Adam when it worked together with VGG-16. The opposite happened when these two optimizers were compared working with the EfficientNet architecture. We eventually converged on the use of Adam because on the final model it was the one that led to the best performance on new unseen data.