Artificial Neural Network and Deep Learning

Challenge 1

# NetCom

POLITECNICO
MILANO 1863

Alessia Luoni

Lorenzo Bancale

# INTRODUCTION

The goal of the project is to build a neural network which is able to classify images of leaves. Those images are divided into categories based on the species of plants they represent (e.g. apple, tomato, blueberry).
So, it is a multiclass classification problem whose objective is to predict the correct class label.

# THE DATASET

We have been provided with a labelled training data composed of coloured images (of size 256x256) divided in the following categories:

0. "Apple"
1. "Blueberry"
2. "Cherry"
3. "Corn"
4. "Grape"
5. "Orange"
6. "Peach"
7. "Pepper"
8. "Potato"
9. "Raspberry"
10. "Soybean"
11. "Squash"
12. "Strawberry"
13. "Tomato"

# INITIAL MODEL

We started our work by importing the necessary libraries among those that were allowed, setting the seed and loading the dataset.
We then created a dictionary and two numpy arrays: one array for the images and the other one for the labels. During this process we discovered that the dataset was containing a picture whose size was 256x256x1 and we found out that completely black photo. So, we decided to delete it and to look if others "broken" images were present, but it was it was the only one with this feature.

In the meanwhile, we changed the shape of the images in order to have pictures of size (50, 50, 3) (after having tried (32,32,3) that was at a very low resolution), and we saved them.

## PRE-PROCESSING

The first thing that we did in this phase was to split the dataset into train, validation and test set.
We decided to split it in the following way: train set 80%, validation and test set 10% each one.
Then, we normalized the data in [0;1] and we converted the labels from integer to categorical (i.e. one hot encoding) since the problem was multiclass.

## BUILDING AN INITIAL MODEL

We started by building the following initial model:

| Layer (type) | Output Shape | Param # |
|---|---|---|
| Input (InputLayer) | [(None, 50, 50, 3)] | 0 |
| conv2d (Conv2D) | (None, 50, 50, 16) | 448 |
| max_pooling2d (MaxPooling2D) | (None, 25, 25, 16) | 0 |
| conv2d_1 (Conv2D) | (None, 25, 25, 32) | 4640 |
| max_pooling2d_1 (MaxPooling2D) | (None, 12, 12, 32) | 0 |
| conv2d_2 (Conv2D) | (None, 12, 12, 64) | 18496 |
| max_pooling2d_2 (MaxPooling2D) | (None, 6, 6, 64) | 0 |
| Flatten (Flatten) | (None, 2304) | 0 |
| dropout (Dropout) | (None, 2304) | 0 |
| Classifier (Dense) | (None, 128) | 295040 |
| dropout_1 (Dropout) | (None, 128) | 0 |
| Output (Dense) | (None, 14) | 1806 |

Total params: 320,430
Trainable params: 320,430
Non-trainable params: 0

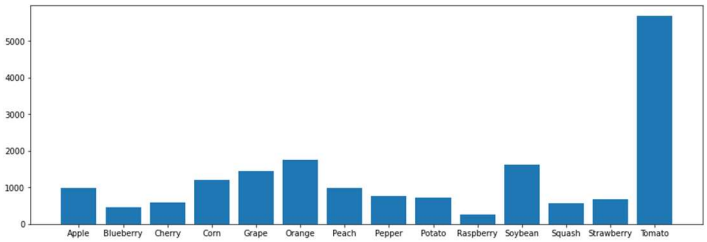
We have decided to use the ReLu activation function in the convolutional layers, while in the output layer we used the softmax since we have a multiclass classification problem. We also inserted two dropout layers to prevent the overfitting.
Then, we proceeded by compiling the model with a categorical cross entropy

function and with Adam as the optimizer algorithm.
Afterwards, the training is completed by means of the "fit" method.

Here we needed to specify the batch size that we set to 8 and the number of epochs that we set to 1000. This latest parameter is very high because it represents how many times the training repeats, but this is not a problem since we also added the callback for the early stopping which monitors the process and stops it when the accuracy does not increase for 10 epochs (i.e. patience=10).
We then plotted the results of the training and by looking at them we found out that overfitting was not present and that the network was effectively learning so we agreed to submit our first model, but the outcome was not good.

# EVOLUTION OF THE MODEL

The first change that we applied was to remove the last max pooling layer and to substitute the fully connected layer with the *Global Average Pooling layer (GAP)* which transforms the dimensions from (12, 12, 64) to (1, 1, 64) by performing the averaging across the 12 x 12 channel values.
The main advantages of the GAP layers are that it reduces the risk of overfitting, it allows to classify images of different sizes (not relevant in our case) and it increases robustness to spatial transformation of the input images. The last benefit listed is probably the one that played a major role in increasing our score (even if only for a few points: ~0.32).
After having tried some little changes such as the use of max pooling layers instead of average pooling layers and using different learning rates with poor results, we analysed the accuracies of the classes one by one. At this point, we realized that some categories had a score of 0.000 and comparing them with the distribution of data made us discover that we were having bad results where we had less data. The following graphs shows the distribution of the images in the dataset and its imbalance:




To deal with this problem we introduced a mechanism called *class weights*. It has been useful for our scope because it gives different weights to the classes accordingly to their distribution:
during training it assigns to the minority classes higher weights in order to penalize their errors and lower weights for the majority class. In this way the algorithm tries to reduce misclassifications on the minority classes. Also this strategy helped us to increase the score a little.

Since the test set was hidden to us and the results were still low, we thought that it could be because of the "secret" pictures had different features with respect to our training dataset. For example, we might have issues in identifying a leaf which was positioned upside down.
So, we decided to apply *data augmentation*. Data augmentation on images is a technique that allows the network to create new training data starting from the initial set of pictures and implement some changes on them. In particular, we exploited the "image data generator" to put in place the following transformations that has been useful for the recognition of leaves: rotation, height/width shift, zoom, shear, horizontal/vertical flip and filling the space left over by the precedent distortions with the nearest pixels.
Putting together all the precedent modifications we went from an accuracy of ~0.32 to ~0.36.

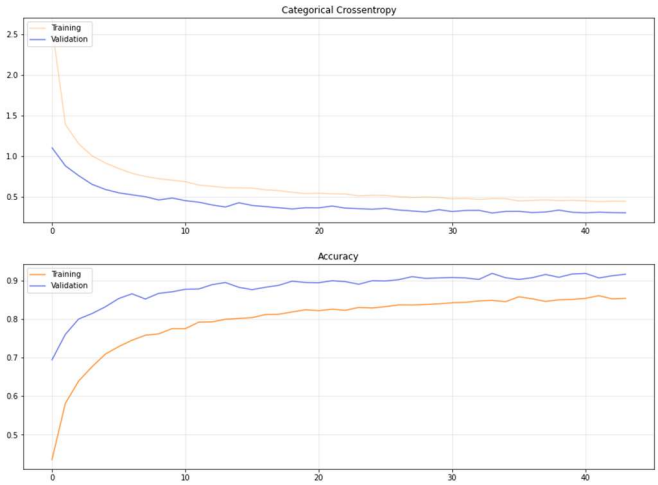
At this point we decided to make some *changes in the structure of the model*, that up to this moment has been left untouched apart from little changes in some layers. We added several convolutional layers and its size duplicated.
This decision was taken in order to help the neural network to extract more complex information from the images. More information means more features which directly impacted on the accuracy of our model. The obtained structure was the following:

| Layer (type) | Output Shape | Param # |
|---|---|---|
| Input (InputLayer) | [(None, 50, 50, 3)] | 0 |
| conv2d (Conv2D) | (None, 50, 50, 16) | 448 |
| conv2d_1 (Conv2D) | (None, 50, 50, 32) | 4640 |
| max_pooling2d (MaxPooling2D) | (None, 25, 25, 32) | 0 |
| conv2d_2 (Conv2D) | (None, 25, 25, 32) | 9248 |
| conv2d_3 (Conv2D) | (None, 25, 25, 32) | 9248 |
| conv2d_4 (Conv2D) | (None, 25, 25, 32) | 9248 |
| conv2d_5 (Conv2D) | (None, 25, 25, 32) | 9248 |
| max_pooling2d_1 (MaxPooling2D) | (None, 12, 12, 32) | 0 |
| conv2d_6 (Conv2D) | (None, 12, 12, 32) | 9248 |
| conv2d_7 (Conv2D) | (None, 12, 12, 32) | 9248 |
| conv2d_8 (Conv2D) | (None, 12, 12, 32) | 9248 |
| conv2d_9 (Conv2D) | (None, 12, 12, 32) | 9248 |
| global_average_pooling2d (GlobalAveragePooling2D) | (None, 32) | 0 |
| Flatten (Flatten) | (None, 32) | 0 |
| dropout (Dropout) | (None, 32) | 0 |
| Classifier (Dense) | (None, 512) | 16896 |
| dropout_1 (Dropout) | (None, 512) | 0 |
| Output (Dense) | (None, 14) | 7182 |

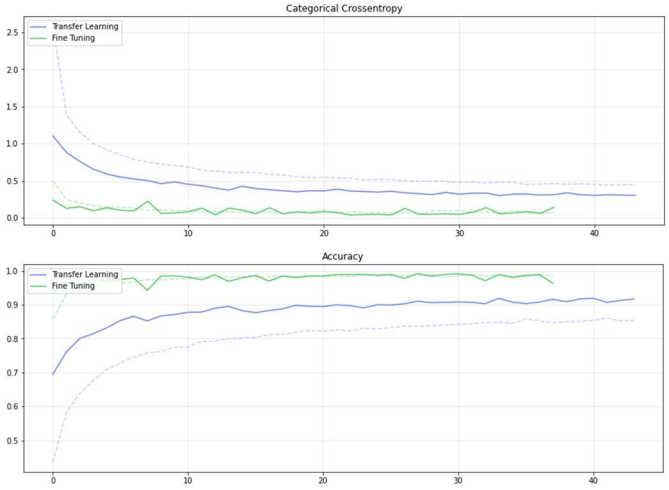
Total params: 103,150
Trainable params: 103,150
Non-trainable params: 0

We trained the model as before and we got better a result than the precedent submits: ~0.40.
At this point we tried to work in parallel on those two different models to see which one was performing better by doing a little bit of hyperparameter tuning. However, those little changes didn't go well, but we came up with the idea of trying to balance the dataset in a different way than using class weights. The new plan involved doing a sort of *oversampling* by copying some images and apply augmentation on the less populated classes so that they could reach the same number of samples of the most populated ones. We tried this approach on both the model, but it improved only the oldest one and we reached the accuracy of ~0.44.
After some other changes that didn't bring us the expected results, we moved to use *transfer learning*. To take advantage of some pretrained features like for example the weights, we utilized VGG-16 as the pre-trained model for transfer learning. The first step was to download the VGG model from the API of Keras with the weights of ImageNet, but since ImageNet has 1000 classes, we needed to adapt the classifier for our problem of 14 classes. So, to avoid losing the effectiveness of VGG, we froze its training layers by making them not trainable and we added the following layers: a flattening layer and two blocks composed by a dropout layer and a dense layer. At this point we inserted the augmentation used also in the precedent models and we trained the new one. However, the score was not as expected and so we decided to repeat this process with the images at a higher resolution (112,112,3) instead of the shape that we initially choose (50,50,3). This change was due to the fact that VGG was implemented with images of size (224,224,3), so we firstly tried this shape but we had troubles with the RAM usage and so we moved to (112,112,3). The reshape made us improve and reach the outcome of ~0.49.
After that, we took inspiration from what we did to the old model and we applied the same alterations. We firstly experimented the *class weight* as before, but this time it was a bad attempt, and secondly, we tried to add a *GAP layer* after the VGG. Adding this layer increased the accuracy to ~0.57.

We had a further improvement by adding a dense layer and a dropout layer to the structure.
Then we decided to modify the last layers of the above model *by substituting the units* of the first dense layer from 256 to 512 and for the second one from 128 to 256. It increased the number of trainable (and total) parameters and in the meanwhile, after having tried different *learning rates* for the Adam optimizer, we choose to set it to 0.0001. Those modifications made us obtain an accuracy of ~0.63.
One of the things that we didn't touch so far was the data augmentation, so we tried to change some parameters and we achieved ~0.64 of accuracy.




At this point the last big change that we could apply to the model was to adopt the *fine-tuning* approach. We started by taking the VGG network from the last model with transfer learning and we froze the first 14 layers, while we let the other 5 of VGG trainable (in fact the last layers are able to extract more specific features than the first ones). In this way the latter model could have half of the parameters trainable and half of them not trainable. Fine tuning helped us a lot to increase the performances (in fact we obtained the outcome of ~0.81) because this technique forced the weights to be adjusted for specific features in the dataset.
We wanted to further improve the performances, so we tried to *make trainable the last 6 layers* of VGG instead of the last 5 layers and we changed the *learning rate* of the Adam optimizer from 0.0001 to 0.00005. This helped us to reach ~0.84 of accuracy.




From the graphs we can notice the improvements given by the fine tuning w.r.t. the transfer learning.
The following image represents the confusion matrix obtained with fine tuning