Artificial Neural Networks and Deep Learning - Challenge 1

Authors: Alessandrini Luca, Ronzulli Ruggiero, Venturini Luca

1 Description

We faced our first challenge with different approaches, starting from the ones more naive and simple, to understand the basics, and ending with methods more efficient and complex. When we saw the dataset for the first time we immediately noticed that there was something a little bit strange with respect to the samples, in fact we noticed that the number of samples per classes wasn't uniform at all, since classes like tomato, for example, has more than 5 thousand of images, while others, like raspberry, no more than three hundreds. We immediately decided to try different ways to face the problems, splitting among us the different solutions. First of all we decided that would have been a good idea splitting our dataset in three parts (training, validation and testing). This has been done because we wanted to train our model relying on the validation set, and then compare different models, if needed, with a test set in order to have a more unbiased estimation of our performances. One of the first approaches we tried to solve the challenge was the use of two separate networks, one used just to recognize the tomatoes, and one for all the other types of leaves. We decided to follow this way to exploit the entire dataset and, at the same time, avoid the overfitting due to the unbalanced datas. The basic idea is to fill the first network, ables to distinguish from two classes (tomatoes and others), with the sample, if the sample is classified as tomato the result is decided, instead, if the result is the other class we fill the second network, ables to distinguish from the 13 classes left, and the result of this second network is the output. The two networks have been trained with the same dataset, but organized in two different ways. The network used to recognize only tomatoes has been trained with a folder containing all the photos of tomato leaves, and another with all the photos of other leaves. While for the second network we simply removed the folder containing tomatoes. The folder of the dataset has been divided with a python script in training - validation - test by using percentages of 75%, 15%, 10%. Our first attempt has been done using the network we saw during the exercise session, composed of 5 convolution/pooling layers and dense layers, with dropout to regularize and avoid overfitting. The performances on our validation set were quite encouraging, but when we submitted our first model we realized that instead it was performing very badly on the "hidden test". We decided to apply Data Augmentation on our training set in order to produce different samples at each epoch. We considered that a leaf can be rotated 360°, and we applied shifts in order to train the network also on a small part of a leaf. Looking at the photos already present in the dataset we decided also to apply some zoom. Data augmentation produced a more acceptable result, we were around 65%. After this we did different attempts training and combining the two different networks. Since the network trained to recognize only tomatoes was performing quite well looking at the accuracy on codaLab, we focused more on the network used to recognize the other leaves. We tried to change a little bit the background color of the images for one of the two networks, using a custom function in preprocessing, decreasing the performances. After that we decided to remove this type of preprocessing and we decreased the learning rate, doubling also the number of raspberry's samples. This time our performances increased. Then we tried to change the network layout and maintain a low learning rate, achieving in this case worse performances.

At the same time, to solve the problem of unbalanced dataset, and not knowing what was the real distribution of the test on codaLab, we decided to follow another strategy: we decided to upsample/downsample the classes with a script in order to have a more balanced dataset and then train the networks with it. We performed an oversampling or undersampling on all classes, in order to have the training set composed of 550 elements for each class. We performed the same kind of over/under sampling with the validation set, by setting the threshold at 250. To the obtained sets, we applied data augmentation when training, in order to avoid overfitting. With this balanced dataset, and the simple network seen at lesson, we reached 65% of accuracy. We tried decreasing the learning rate of one order of magnitude (from $0,001$ to $0,0001$), and we

reached 70%. Then we thought that it could be a good idea to change the background of the images on the same balanced-augmented dataset, since they were all black: this could have led to a more "focus" on the leaf, in our opinion. By error we set the new background color at $[1; 1; 1]$ to random images (instead of $[255; 255; 255]$ which is what we wanted to do). For brevity, the condition with $[1; 1; 1]$ will be called "A". This allowed us to do further considerations: with condition A, from the reached 70% we decreased to 68%: we thought that this could be explained with a slower learning due to more variations, hence we decided to increase a bit the patience, from 10 to 20. With this we reached in fact a score of almost 74%. Then, we noticed the error, and we corrected the code: we randomly set (with one third of probability) the background to either black, or gray, or white. In our mind, this could have increased our performance a little bit more, but resulted in $72, 26\%$, even with the patience augmented to 20 epochs. This could be due to the fact that increasing the variation in the background, we need more epochs and more computations to make our network learn actually more. We tried to do another attempt with this last network, and we tried to change the type of pooling, computing the average pooling: we went back to almost 64%, hence we discarded this option.

Since we realized that it was very difficult to reach really good performances with our "homemade" networks we decided to switch to already well known and pre-trained networks, focusing more on *VGG16*, but also trying different solutions like *Resnet*, *Xception* and *EfficientNet*. On VGG16 we tried to resize the input image in different ways. We tried to decrease it a lot, thinking (wrongly) that this may have generated a more general model. Then we trained the network on resized images 224×224, like the original VGG16. We also tried to increase the image size from the original one. Finally we discovered that the original resolution was the better one, as shown also in *Figure 1*. Then we tried to fill the pixel during the shift operation in data augmentation with different options and "constant" black resulted to be the better (as we were thinking, since all the images have a black background). We also tried to change the batch sizes and increasing it seems to help in the recognition.

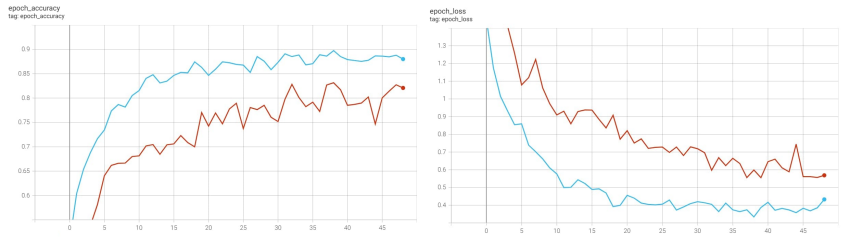

Figure 1: Resizing images in VGG: accuracy (left) and loss (right) after transfer learning for inputs in original size (*blue*), and resized (*red*). Already after transfer learning the difference is quite visible.

Since the balanced dataset applied on our network made from scratch increased our performance, we wanted to try this on the VGG16: by applying the same over/undersampling (550 training samples and 250 validation samples per class), and by applying data augmentation, we reached a good score: we were at $89, 6\%$. Then, conscious of the fact that we could overfit a little bit (due for example to the class of the raspberry with very few samples), we wanted to try a "*more aggressive*" over/undersampling on the initial dataset: we changed the threshold to 1100 for training and 500 for validation. This results in slower training (hence, more time needed), but in a higher result: with this attempt we reached $93, 77\%$. After, we tried also to take away the last level of dropout in the network, but this resulted in a decrease of performance, we went down to arount 90%. After this, we inserted back in the network the last level of dropout, and what we did was to modify the learning rate of the best VGG result up to now: we set the transfer learning part's learning rate equal to $0, 0001$, and the one of the fine tuning equal to $1e - 5$. This resulted in a lower performance, little more than 92%. Unfortunately, we would have liked to try to increase the patience in this last model, but since already this took us much time, we could not perform this last test.

While we were working with VGG16, we tried to use transfer learning also on other popular networks to try to understand which one was actually the best for our case. One of the first attempts after VGG16 was with *InceptionV3* which, however, showed from the beginning worse results than the VGG16's ones, although it kept the accuracy on the hidden test over the 80%. Having regard to the results of Inception we

moved to *Xception*, that is an extension of the previous module. Since with VGG16 we had good results with the balanced dataset with 550 elements for each class we decided to use it with Xception from the beginning and use two dropout layers for the regularization. This time our results were very close to the results of the VGG16 network which is why we decided to utilize this network to do some more tests. First, we tried to modify slightly the network. First try was adding some noise to the input images trying to increase the robustness of the network, to do this we added one gaussian noise layer, initially set with the standard deviation of the noise distribution at $0,3$ (we also tried to randomize the value of the standard deviation but that just worsen the performance of the network) , however we didn't get the expected results because the network has lost few percentages points on the hidden test accuracy, probably due to some underfitting during the training phase caused by too much regularization. Due to this we tried to remove one dropout layer to avoid this underfitting. In doing so, our accuracy increased but has kept worse performance than the ones with the 2 dropout layers. But still, before dropping the idea of putting some noise on the input images of the network we gave it one last try, that was working on the dataset. The idea was to increase the number of elements inside the dataset, since we already used data augmentation this time we wanted to try another approach, increasing the dataset's images generating brand new images writing a *Generative Adversarial Network*. We use as input of the GAN one class per time of the balanced dataset (the one created with the data augmentation) and due to time constraint of the challenge we train the network on 200 epochs, but the results we obtained were not those we hoped for, unfortunately the images generated could not be used in our dataset due to very poor resolution, which may have been caused by a training too short compared to the big input resolution of the images. After the failed attempt with the GAN and having chosen the network with 2 dropouts over the one with the gaussian noise layer, tried last time with Xception, this time decreasing the network learning rate and using an Adam optimizer with an L2 regularization. This attempt gave us an accuracy of $93,2\%$, that is still lower than the one we got with VGG16.

Our last attempt has been the use of EfficientNetB7 trained with a very low learning rate. We did just one attempt since the training required a lot of time and the competition was almost over. This last model performed more or less like one of our best VGG models.

2 Further considerations

While working, we tried to analyze our outputs with confusion matrices, and various metrics, but locally, even with a dedicated test set generated at the beginning from the dataset (and never shown to the network until the test phase), we always reached good results as shown in the *Figure 2*, not even close to the one reached in codaLab. We understood that the dataset given to us, and the dataset used in codaLab for testing were quite different. For this reason we used the metrics to compare our models and to be sure of having not decreased too much our performances. This also shows that VGG16 generalizes better with respect to our model made from scratch, because even if locally they don't show substantial differences, on codalab one is performing really better with respect to the other.

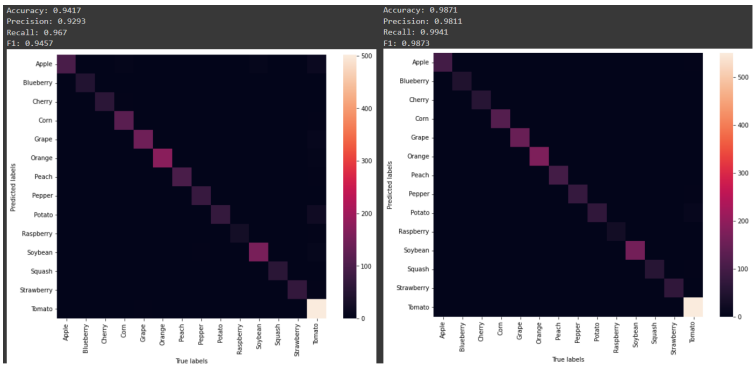

Figure 2: On the left, confusion matrix and metrics of the model which gave us a score around 70%, while on the right, confusion matrix and metrics of our best model, which gave us a score around 94%.