Report

Gianluca Ruberto, Riccardo Pazzi, Tommaso Brumani

Summary:

For this project, we decided to start by building a custom convolutional neural network based on those seen during exercise sessions in class, in order to familiarize ourselves with the programming language and tools available.
We then iterated on this model in multiple ways, by monitoring its performance both locally and through submission, to see which modifications would result in an increase in performance and would be worth pursuing further.
After having achieved what we deemed an acceptable result through our custom network, we decided to see if its performance could be surpassed by a network constructed through transfer learning.
Seeing positive results, we endeavored to try different starting models as well as fine tuning the best, until we obtained our final result.

Model Name	Description	Score
MK II	Model seen in class	22%
MK X	Added data augmentation	50%
MK X S	Added shear to data augmentation and increased picture size from 64x64 grayscale to 128x128 rgb, increased batch size from 8 to 64	68%
MK XVI	Added batch normalization after each convolutional layer	77%
MK XXII	Brand new model with a very deep network	79%
MK XXIX	Transfer learning of EfficientNetB7, previous data augmentation is used, input pictures are 256x256 rgb and it is also used the data oversampling	89%
MK XXIX FT	Fine tuning of EfficientNetB7	92%

Preprocessing:

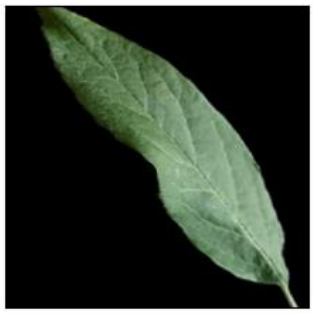
We first tried by using smaller (64x64, 128x128) and grayscale images to improve training time, the speed improved greatly for the first networks we submitted. Also pixel rescaling was adopted with a factor of 1/255.

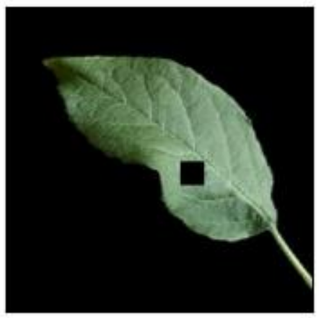
We however saw improvements in the network performance at test time when the images were bigger and rgb, this made sense since color information for this kind of task might be useful and a higher resolution allows for more features. So after a while we moved to process entire 256x256 rgb images with the networks.

Data augmentation experiments:

The first augmentations we tried were taken directly from the exercise sessions, in particular rotation, flip and zoom. The fill mode used was constant black due to images having a black background.

Starting image


Shear


Cutouts

However we then moved to fancier transformations like shear, which to us made sense due to the particular shape of leaves: shear provides elongated leaves with smaller width which is a common occurrence in nature.

A promising augmentation which yielded a worse performance was the use of random cutouts, this seemingly was interesting since holes in leaves where present in the dataset and we thought adding artificial black rectangles in the image would provide a better accuracy at test time, this wasn’t the case and considering the long time this transformation took during training (it wasn’t GPU optimized) we abandoned the idea of using cutouts. Another problem was that many times cutouts would be in the black region and give no benefit.

We also tried adjusting brightness, channel and random noise augmentations which did not yield significant improvements.

Regularization:

We started with custom networks made from scratch and experimented with different types of layer compositions.
One big problem was that the validation error was really different from the test time error. One layer that greatly helped in this sense was batch normalization, which is supposed to improve generalization and therefore reduce the difference between validation and test accuracy.

To avoid overfitting we also used L2 norm regularization and early stopping, these also contributed to better results along with dropouts layers in the dense part of the network.
Finally Global Average Pooling yielded good results both in reducing the number of parameters to train and in improving the test error.

Dataset imbalance solutions:

As previously mentioned, one of the main concerns was the accuracy difference between validation and test sets. One problem was surely the dataset imbalance between classes (raspberry had only around 200 images and tomato more than 3000) but also the relatively small size of the dataset (17k).

Many big models would risk overfitting and provide worse results, so we tried three different strategies to overcome the problem: undersampling, oversampling and class weights.

Undersampling and adding weights to classes provided underwhelming results. We tried different strategies of class weights (e.g. with the maximum weight equal to 1, the minimum weight equal to 1, etc.) but the performance worsened considerably. Undersampling didn’t improve performance either, probably because the raspberry class was so small and this would severely limit the other classes' training sets, class weights weren’t sufficient to balance the network.

So in the end we tried oversampling by replicating the pictures of the less represented classes in order to have the same amount of samples for each class, which greatly improved both performance and accuracy difference between local and server accuracy (went down to 6%), this can be explained both by the ability to use bigger networks on the larger dataset and better generalization due to the larger amount of training samples.

Network structure:

We tried also to change the topology of the network: we tried to add dense layers, to increase the number of neurons, to increase the size of the filters, to increase the number of convolutional layers, to start with a large filter and then decrease its size with each successive convolutional layer (like in AlexNet), we also tried to build a “parallel” neural network with filters of different sizes like GoogleNet/Inception. By the end we had built more than thirty different models, but we failed to see any improvements most of the time.

The final approach we tried was TF with fine tuning. We tried a couple of networks (ResNet, VGG, DenseNet, Inception) but we obtained the best results on EfficientNetB7. The training was cut short due to extremely long training times and hardware limitations, but was still enough to beat the other networks. We then unlocked the last block for fine tuning with a learning rate of 1e-5 and ran for 20 epochs, which improved the test result by a further 3.5%.