Image Classification

Cecchetti Francesco, Concina Lorenzo

November 2021

1 Approach

The classification of images is a complex problem that can be solved through the use of appropriate neural networks called Convolutional Neural Networks, which consists of two blocks: the first formed by convolutional and pooling layers necessary for the extraction of features from the images and the second formed by fully connected layers necessary to carry out the classification.
We faced this challenge with the intention of trying to apply all the techniques seen in class, trying to understand how each of them improved or worsened the score on the test set and trying to understand the reasons. To this end, we followed two paths, the first by implementing our own CNN from scratch and gradually improving the hyperparameters to obtain better classifications starting from 14% at the first submission and reaching 74.5% at the last, and the second by exploiting other famous networks via transfer learning and fine tuning to obtain high results and try to climb the ranking, reaching 91.6% as a final score.
Please notice that in the following report the relative paths of most significant notebooks are mentioned. They are all available in /Notebooks subfolder.

2 Convolutional Neural Network from scratch

At the beginning of development of our model we started with a very simple one to test out the functioning of the image classification and to be able to obtain feedback on the codalab test set, aware of the fact that many problems not yet considered by our model would arise.
In details, it is a model that uses 4 convolutional layers starting from 16 filters that double at each layer, interspersed with AveragePooling layers, which function as a block of features extraction. All this is followed by a Flatten Layer and 3 Dense Layers (with 512,1024 and 14 units) which constituted our fully connected block for classification. Furthermore, we immediately employed the use of the Early Stopping regularization technique with patience = 7, to stop the training as the validation loss increases, i.e., in the presence of overfitting.
Unfortunately, the validation loss did not decrease well enough: as a result, we got a disappointing 14% accuracy on test set. Even if we were expecting a low accuracy, there was a clear big discrepancy with the validation accuracy, that we could not explain at the time.

We then proceeded trying out various modifications of the hyperparameters such as batch size, patience of Early Stopping, convolutional layers filters and pooling, keeping the same model architecture except for a dense layer less, to reduce network complexity and number of parameters to train. Also, we fixed some mistakes of the first model: our Early Stopping was monitoring the training loss instead of the validation loss and we weren't restoring the weights. This attempt allowed us to reach 18% of accuracy on test set.

Now we wanted to act on the training data, therefore we setup data augmentation on the training set with some basic transformations such as rotations, dimension-shifts, flips and zooms. The improvement on accuracy test was mild.
Seeing training outputs, we thought our issue was overfitting, so we tried to add 2 dropout layers to tackle the issue, but this ended up worsen the accuracy.
At this point revised the work we were doing and realized there was a substantial oversight: we weren't normalizing the test set images with a rescale the same as the training set.
Adding this pre-process also to the test set data basically doubled our accuracy, reaching 46%.

From this point on we tried different solutions to improve our accuracy on the test set, combining various techniques: the process consisted in progressively modifying the architecture of our network, hyperparameters, data-augmentation parameters, with a "trial&error" mindset, to see how accuracy changed.
The big step up was achieved adding one more convolutional layer (5th), replacing the AveragePooling layers with MaxPoolingLayer, and adding BatchNormalization layers after the latters in order to normalize data across the network and improve training learning capabilities, keeping just one dense and dropout layer.

The accuracy reached was a decent 60% *(/Notebooks/CNN_2.ipynb)*.

Aiming at overfitting reduction, we also considered other approaches such as L1, L2 and L1-L2 regularizations but once again, we obtained bad results in validation accuracy and decided to not submit these attempts. Other attempts consisted in hyperparameters tuning, such as dropout probability and different numbers of filters in convolutional layers. Nothing improved the accuracy on the test set.

At this point we got back to our CNN architecture and decided to double every convolutional layer, to improve features extraction capabilities of our network. This resulted in a 1% improvement on test set. While experimenting we also tried replacing the Flatten layer with a GlobalAveragePooling layer to reduce the number of trainable parameters, i.e., the complexity of the model, and obtained 1% more accuracy.
During all these trials, we kept considering training set classes cardinality imbalance a major issue for the accuracy of our model, thus tried adding weights balancing with class_weights parameter of model.fit() (normalizing the weights). All attempts made with this technique resulted in poor accuracy both at validation and test.

The next big step up has been achieved making use of the augmented training set we crafted. Testing on the latest architecture allowed us to reach 66% on test set. Lastly, we attempted to use the new training set both increasing of one convolutional block our network, with learning rate decay (exponential), and to reduce of one convolutional block to help our model to generalize better; the latter try granted us 74.5% accuracy on test set (*/Notebooks/CNN_3.ipynb*).

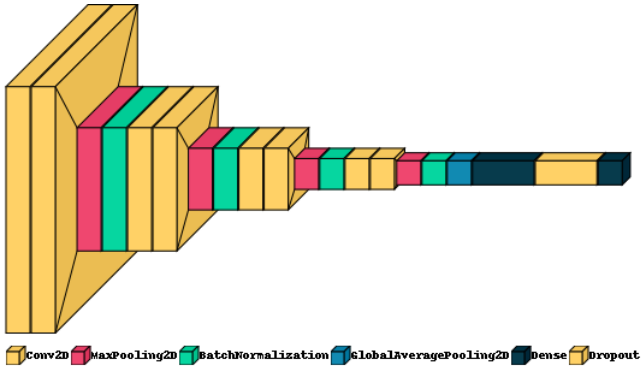

Figure 1: Our best CNN Architecture

3 Transfer Learning and Fine Tuning of pre-trained CNN

In order to improve the predictions on the test set and thus obtain a higher average score, we decided to explore the possibilities offered by transfer learning and fine tuning. These techniques make it possible to exploit other more complex networks trained on much larger data sets to tackle a similar problem. In this challenge in particular, using networks that have been trained for the classification of different images, allowed us to start from models that were already able to extract characteristic features from the images of the classes of our challenge. In the case of transfer learning, only a train is made of the classification block consisting of Fully Connected Layers on top of the convolutional block, while with fine tuning some (or all) layers of the convolutional block can be trained, further improving the classification performance of the network. . It is important to underline that fine tuning must be carried out on a network previously trained through transfer learning, in order to avoid the train of convolutional layers with weights already initialized and fully connected layers that have been initialized randomly.

The first experiment was carried out using a VGG with a Flatten and two Dense layers on top that carried out the classification (one of 512 and the other of 14 neurons). The layers of the VGG were freezed and therefore in the training only the weights of the Dense layers were trained, thus implementing transfer learning. The result obtained on the hidden test set was 56%.
To reduce the number of trainable parameters and increasing the ability of generalization of the network, we replaced the Flatten layer with a GlobalAveragePooling, leaving all the other hyper parameters unchanged, achieving 73% on the test set (*/Notebooks/Transfer_Learning_VGG.ipynb*). To try to further improve the results on the test set we have increased the number of Dense layers in the classification block to 3, hoping to increase the classification capacity. However, the result of the submission was not satisfactory and lower with respect to the previous submission.
The next step was to perform a Fine Tuning of the VGG, also training the last 3 convolutional blocks of the

latter, obtaining a further improvement on the test set and reaching 77%

Driven by the need to further improve the score, we decided to use a deeper and more complex network, choosing the Xception. Also in this case the first attempt was transfer learning, using only GlobalAverage-Pooling and two Dense layers for classification and obtaining 76.6% on the test set, a result similar to that obtained previously with the fine tuning of the VGG. So we re-trained the model just discussed, unlocking the last 60 layers of the Xception to fine-tune, obtaining 87%, therefore a considerable improvement on the test set.
In all previous attempts, the normalization of the images was carried out by dividing the value of each pixel by 255 using the 'rescale' attribute of the ImageDataGenerator. We therefore decided to preprocess the images using the preprocessInput layer of the Xception and this further improved the results on the test set, bringing them to 89%.
Having reached a high score by playing with the hyperparameters of the network, exactly as we did on the CNN created by us, we decided also in this case to use a dataset with an augmention of the data as explained in the following chapter, thus increasing the number of instances for the fewer classes. By fine-tuning the previous Xception with this dataset, we were able to obtain a score of 91.5% *(/Notebooks/Fine_Tuning_Xception.ipynb)*, then subsequently brought to 91.6% with a fine-tuning of the last 80 layers of the Xception.

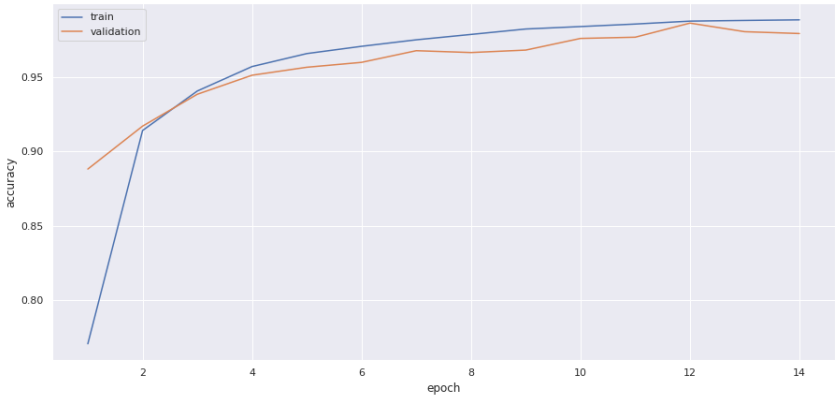

Figure 2: Accuracy on train and validation for the train of Xception with fine tuning

4 Training set Data Augmentation

As we hinted during the explanation of the journey in building our first image classification models, we considered the imbalance in cardinality of our training set classes a major issue for achieving good results in test. Approaching the limits of our models' capabilities we felt the urge of tackle the problem. Some attempts consisted in simply tuning the parameters of the ImageDataGenerator of Keras which didn't end up in significant contribution to the accuracy. We figured out we needed to literally increase the number of images our models could train on. One idea could have been to use an online accessible dataset with plenty of similar images to expand our training set, but we felt it would have been a sort of "cheating", being that in a real-case scenario we cannot always have access to such large sets for each class of what we are trying to classify. Therefore we decided to expand our training set starting from itself: we developed a simple python script (*/Scripts/main.py*) that exploited the library Augmentor, which basically allowed us to perform operations on images similar to the ImageDataGenerator of Keras, but deciding how many images to generate per class, doing it in an "offline" modality. As a result, we ended up with a new training set of 24.378 images. This way we re-balanced the training set, increasing the number of images available, especially for those classes with only few instances.

5 Conclusion

In the Final Phase of the competition, we submitted the two best models we described so far: the CNN_3 and Fine_Tuning_Xception.
Both of them had only minor decrease in accuracy on the new hidden test set, achieving 72.5% the former and 90.6% the latter.

Overall we are personally satisfied with the results we got, especially for the CNN considering we built it from scratch.