Artificial Neural Networks and Deep Learning Challenge 1

Introduction

We are about to illustrate the process that brought us to build a model that reached 95% accuracy on tests. All our work has been done on Google Colab, exploiting Keras callback BackupAndRestore to be able to flawlessly train despite Colab usage limits.

First approach

To begin with, we tried building a custom-made neural network, taking inspiration from the architecture of Xception, a pre-trained convolutional model offered by Keras. The custom neural network consisted of:

1. A rescaling layer.
2. 15 convolution blocks with various filter sizes:
    a. Two serial entry blocks with 32 and 64 filter size.
    b. Four parallel blocks of two convolution blocks, a MaxPooling layer, and a convolutional block. Each of the parallel strains having a different filter size: 128, 256, 512, and 728.
3. The last convolutional block with filter size 1024.
4. A GlobalAveragePooling2D layer
5. And two Dense layers of size 512 and num_classes, with 0.5 Dropout applyied to both of them.

We used “relu” as the activation function and “softmax” for the last Dense layer. After taking away 5% of the dataset for testing, we used basic data augmentation like rotation, horizontal and vertical flip, having a 10% validation split. After training till convergence, we we’re not being satisfied by the results we decided to move to transfer learning in hope of better results.

Transfer learning

We first tried VGG16, then Inception, Xception and finally Nasnet.

For the top of the network, after several iterations and inspired by our previous tests with the custom network, we decided to put:

1. a Global Averaging Pool 2D
2. Droupout of 0.2
3. A couple of fully connected layers (with 1024, 512, and 512 neurons respectively)
4. Another Droupout of 0.8
5. And finally, a fully connected layer with num_classes neurons as the output

We used LRelu as activation function as it turned out to perform slightly better than normal Relu. And we tried several optimizers but, in the end, we used Adam.

With those parameters set we obtained these results: we were able to reach 70% test accuracy, with VGG16, 90% Inception and Nasnet, but finally, confronting the results of +90%, Xception turned out to be the best model for the job.

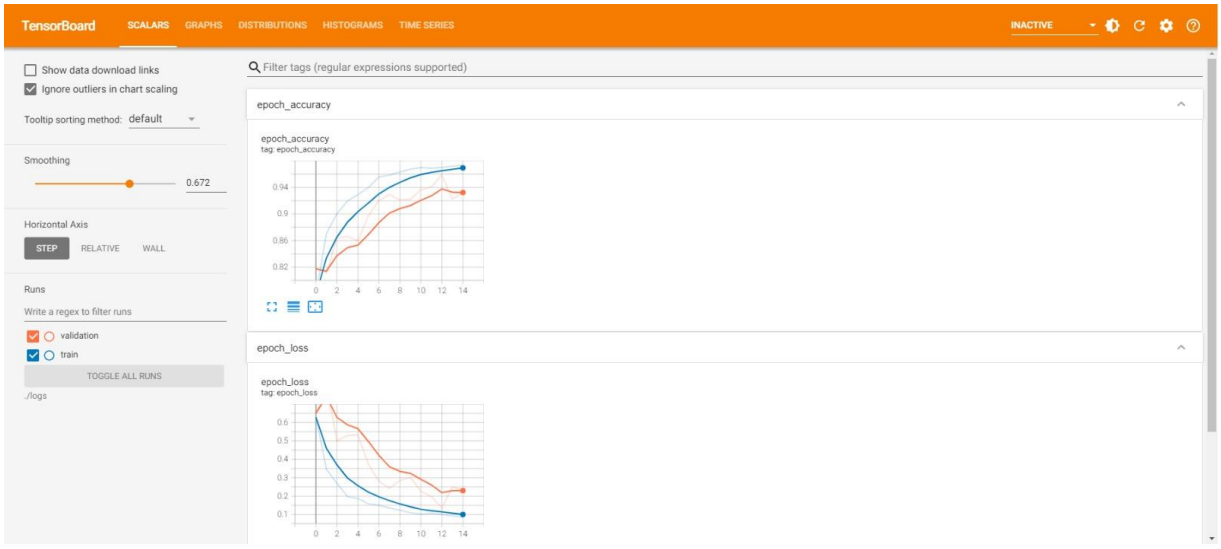
Having chosen the pre-trained with the most promising behaviour, we turned our focus to further improving the training process.

We first tried adding a GaussianNoise as an input layer to better generalize the data, but the results were mostly unsatisfactory or at best, the same.

Then, following some best practice guidelines, we started fine-tuning the network. We would freeze the base model to train only the randomly initialized layers. Then, we would proceed unfreezing the deeper layers of the network and training them to convergence. What we found out, was that for the second step, freezing the first 50 layers produced good results (this was a bit unexpected as we supposed that we had to freeze the first circa 32 layers, being the first Xception blocks). As a final step we would unfreeze the whole net with a very low learning rate (~5e-6) to reach the best convergence.

Furthermore, to help us with the general training process and shortcomings of the resources available through Google Colab, we used several Tensorflow callbacks. As previuously mentioned, we used BackupAndRestore to recover training progress, we also used ReduceLROnPlateau, which was very useful as it adapts the learning rate during the training, and we used EarlyStopping, as taught in the lectures.

To visualize and follow the training process, we used Tensorboard, which when connected to a training process provides real-time statistics such as the training and validation loss/accuracy plots.



A problem of Colab's RAM overflow, which we encountered in the early stages of our training, sprung trying to train the model from a numpy array. To battle that, we used Tensorflow's flow_from_directory function, to automatically read from directories during training, and split between train and validation. As previously stated, we choose a 90/10 training/validation split. We also put aside 5% of data to be used as a test (by moving them to a different folder to have consistent tests between runs). This made us capable of confront the different models, without the risk of overfitting the validation set and not noticing it.

We tested several image augmentations, in the end we used most of the augmentations available by the ImageDataGenerator. We noticed that a high shear range was particularly effective on the test accuracy. We used Keras' Xception original pre-processing method, but we also tried to build our own to add noise. This turned out to be very harmful to the training process and final result.

Moreover, for our batch size we mostly used 128 images, but we had to lower it in some parts of the training because of memory overflow, mostly those with very low learning rate.