Artificial Neural Networks and Deep Learning Homework #1 Report

Alireza Nadirkhanlou | alireza.nadirkhanlou@mail.polimi.it
Ghodrat Rezaei | ghodrat.rezaei@mail.polimi.it
Riccardo Tomada | riccardo.tomada@mail.polimi.it

Introduction

The first homework of the course was a classification task, in which students were asked to define and train an artificial neural network model to classify images of leaves into one of the 14 categories provided. Each model was then evaluated based on its mean accuracy metric on an unknown test set.

The strategy followed by our group can be subdivided into three separate phases:

- Definition of a basic convolutional neural network (as one seen during the exercise sessions), looking how it behaves with changes in the hyperparameters
- Using the knowledge gained, implementation of the transfer learning + fine-tuning techniques in a new model
- Searching in the literature for further improvements to better tune our model

In the next section(s), we will enter into the details of these three steps. Note that all the notebooks are run on kaggle.com using the GPU accelerator. The amount of free GPU time per week was limited, so we searched for a way to improve the performance of the code also from a computational point of view.

Code Efficiency

As stated by the TensorFlow official documentation, “Achieving peak performance requires an efficient input pipeline that delivers data for the next step before the current step has finished”.

For this reason, we firstly decided to define two objects for the training and validation sets as tf.data.Dataset objects, which are recommended by TensorFlow for their efficiency, using the `image_dataset_from_directory` function.

Note: in all our attempts, we split the data into two sets: training and validation, with ratio 5:1.

Then we applied on them the prefetch transformation, which allows the new images to be loaded in the background while the GPU is still performing computations on the previous batch of images.

Finally, we did data augmentation as it has been demonstrated from literature that this practice can dramatically improve the ability of the network to generalize on “unseen” test images. Giving particular attention to the efficiency of the process, we decided to do data augmentation directly as part of our model, making use of the Keras image augmentation layers. In fact, with this configuration, the data augmentation is performed synchronously with the rest of the model execution, meaning that it will benefit from GPU acceleration.

After having implemented all the above efficiency improvements on the simple CNN architecture seen in class, it has been possible to notice that the new updated version was 7 times faster on a single epoch with respect to the default model.

First Attempt: A simple CNN

The model chosen for the first attempt was the same as one used during an exercise lab.

In particular, as with the configuration, it was made up of 5 blocks of convolution + ReLU + batch normalization + max pooling layers, plus a flatten layer and two fully connected layers on top of them (the last one being the classifier one) with dropout equal to 0.3. As a callback Early Stopping has been used to avoid overfitting, with a patience of 10 epochs.

With this simple model different hyperparameters has been tested:

- The dropout amount
- The learning rate value
- The batch size value
- The number of units in the first fully connected layer

To make the analysis we followed both a train and error approach, as well as using the `KerasTuner` library.

It has been concluded that the most relevant hyperparameters that affects the validation accuracy are the batch size and the learning rate:

- Batch sizes of 8, 16, 32, 64, 128 were tested. The best result was obtained with a batch of size 64, enough large to be representative of the train dataset and at the same time less RAM demanding than a 128 sized one.
- A high learning rate does not allow the optimizer to reach the minima of the loss function, but constraints it to oscillate around it. At the same time a low learning rate may not be enough for the network to learn, for example stopping the optimizer iterations on a local minima. A trade-off approach has been followed for the Transfer Learning + Fine Tuning notebook, which will be discussed in the following section.

Second Attempt: Transfer Learning + Fine Tuning

Since the first introductory lesson on these two techniques we were fascinated by their huge power, so we decided to implement them quite early in the competition.

The need for a great validation accuracy score combined with a good efficiency measured in terms of number of parameters and FLOPS operations led us to choose the `EfficientNet` network, as it can be observed in the Figure 1

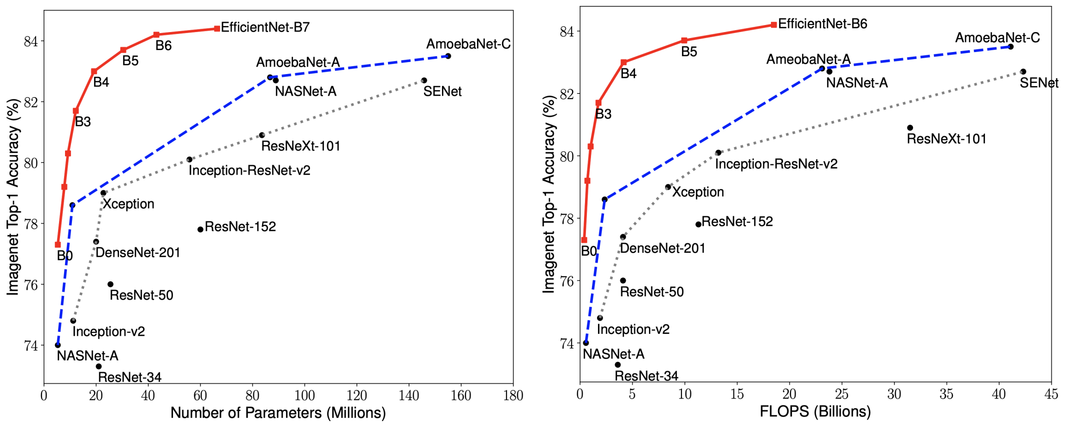

Figure 1 - The number of parameters and FLOPS for the `EfficientNet` models compared to others

In particular we adopted the `EfficientNet-B4` model, as it already reaches great performance with a computational effort comparable with `Inception-V2`, `DenseNet-201` and `Xception` models. We could choose even a better accuracy model like `B5`, `B6`, `B7` or `B8` ones, but we preferred to keep working following the efficiency philosophy already described above.

In order to boost even more the accuracy of the model without increasing the computational effort, as suggested by the Keras documentation we initialized the weights using the latest improvements performed by the developers (`Noisy-Student` weights instead of `ImageNet` ones).

Transfer Learning

The `EfficientNet` network has been imported in the model right after the augmentation layers, without its top dense ones. The layers have been set as non-trainable, as we wanted to preserve the feature extraction capabilities of the pre-trained model. The dense layers have the following configuration:

- `GlobalAveragePooling2D`: it allows to reduce overfitting with respect to a simple flatten layer, acting as a regularizer
- `BatchNormalization`: it is a regularizer too, normalizing the output and centering it
- `DenseLayer`: added to increase the depth of the network
- `Dropout`: it reduces the overfitting as well randomly changing the number of units of the previous layer
- `OutputLayer`: the classifier itself.

The learning rate value chosen is `1e-2`. It is a relatively high one, but as written in the Keras documentation it is a valid option for this phase (warming up phase). Learning rate scheduler has been tried as well (`TriangularCyclicalLearningRate`), resulting in better results for the Transfer Learning part but worse overall.

Fine Tuning

The `EfficientNet-B4` network is made up of hundreds of layers subdivided into 7 blocks. During fine tuning a number of layers get unfrozen and then the model is fitted using a smaller learning rate (`1e-4`) to avoid losing the beneficial results obtained during the Transfer Learning phase.

Note:

- We unfroze only the last block of layers (31), since it is recommended to freeze/unfreeze one or more full blocks. In a second notebook also the 6th block as been unfrozen, leading to worse results so we chose to continue with only one unfrozen block for the future attempts
- As suggested by keras, the `BatchNormalization` layers has been kept frozen

Results on the test set: mean accuracy

The results achieved in the first test set were definitely promising, showing the power of the improvements adopted. You can see the results in **Table 1**.

Model Name	**Simple CNN**	**Transfer Learning**	**TL + Fine-tuning**
Mean accuracy	0.598	0.802	0.936

Table 1 - The mean accuracy of the three different models

Further attempts

In order to increase the accuracy provided by the model, various attempts have been performed. In the following lines the main ones are going to be explained:

- Implementation of additional augmentation layers: `RandomCrop`, since the test images could be cropped as well, and GaussianNoise, since the test images may have not a black background as the training ones or may contain some noise
- Increasing the image size, from `256x256` to `380x380` using bilinear interpolation, since the `EfficientNet-B4` network was initially developed for this size
- Changing the dense layer activation function from ReLU to Mish. The Mish activation function is a slightly modified version of the ReLU, non monotonic, that in some cases provided a better result.
- Completely removing the 512 units dense layer, as it may lead to overfit, leaving just the `GlobalAveragePooling2D` and the BatchNormalization before the classifier
- Changing the optimizer, from the standard Adam to the Ranger, which is a combination of the Rectified Adam with the Lookahead optimizer: it should converge faster, avoiding local minima
- Changing the loss function to be minimized during training, from the standard `CategoricalCrossEntropy` (**Eq. 1**) to the `SigmoidalFocalCrossEntropy` (**Eq. 2**).The latter introduces an adjustment to the cross-entropy criterion and in general is useful once we have imbalanced classes, as in our case, or in object detection problems. It assigns a loss value much higher for a misclassified sample with respect to a well-classified ones:

$$CE(p_t) = -\,log(p_t) \qquad \text{(}\textbf{\textit{Eq. 1}}\textit{ - Categorical cross-entropy)}$$

$$FL(p_t) = -\,(1 - p_t)^{\gamma} log(p_t) \qquad \textit{(}\textbf{\textit{Eq. 2}}\textit{ - Sigmoidal focal cross-entropy)}$$

- In the Fine Tuning part, it has been applied a ReduceLROnPlateau callback which after a given patience (no val_accuracy improvements) it reduces the learning rate of a given factor, to help the optimizer converge
- In the Fine Tuning part, it has been applied the Triangular2CyclicalLearningRate function, which returns an exponentially decaying triangular shaped learning rates, which combines the reduction of the learning rate with the cyclical temporary increase (to avoid local minima)
- For the `CategoricalCrossentropy` loss function, we tried enabling label smoothing in order to avoid overconfidence. The results were promising, but due to slow convergence and limited remaining time, we only trained a model with fine-tuning epoch numbers equal to 35. **Nevertheless, this model which did not fully converge gave us the best score in the final phase (0.9377).**

Unfortunately our trial and error procedure, which implemented combinations of the above attempts, didn't give us better validation accuracy results (except label smoothing), thus only few of them were submitted also for the test evaluation. Probably a more rigorous approach could have led us to increase the test accuracy.