Artificial Neural Networks and Deep Learning : Homework 1
Politecnico di Milano
fall '22

Galimberti Claudio, Mileto Alessandro, Staiano Enrico

1 Problem statement

We were asked to develop a Deep Learning model to classify different species of leaves. It can be stated as a multiclass classification problem. We were provided with a dataset made up of eight different leaves species. The dataset is highly imbalanced as species 1 and 6 samples were half of the samples of the other species.

2 Our work

2.1 Data loading and management

We wanted to develop an efficient solution flexible enough to easily perform dataset splitting and preprocessing operations. At first, we split the dataset on the disk to produce two directories in a stratified manner: training and validation set. Loading images with the ImageDataGenerator.flow_from_directory method, we found out that GPU utilization dramatically dropped. There was a bottleneck in the processing done by the CPU to fetch images and process them. So, we switched to another method. We decided to build a different processing pipeline for training leveraging the tf.data.Dataset class (in all the notebooks). This allowed us to simplify splitting, preprocessing and augmentation and to improve the overall efficiency of training thanks to pipelining. We never resorted to a local test set since we used the remote one for 'testing' (even if we used it multiple times, i.e. at each submission). We used 80% of the dataset for training purposes, 20% for validation and model selection.

2.2 Dealing with imbalance

2.2.1 Resampling

To deal with the unbalancing of the dataset, we tried to use different resampling strategies. At first, we implemented from scratch oversampling and undersampling working directly on the images directories before loading them in memory by adding/removing samples according to the number of samples of each species. This solution was not flexible enough, so we tried something else. We moved to a more sophisticated solution named SMOTE (synthetic minority over-sampling technique) from the imbalanced learn Python library. We tried several variants of it and, finally, we used SVMSMOTE, which leverages a SVM to synthesize new samples from minority classes. This was a complete failure but we did not really understand why so we just speculated on it (see SMOTE_imblancedLearning notebook). Then, we moved to our final resampling strategy which consisted in loading the minority classes as a different Dataset object and manually concatenating it to the total training set to perform oversampling. This naive solution yielded acceptable results (in every notebook).

2.2.2 Class weights

We used the scikit-learn library to compute class weights according to the number of samples of each species and, in most of the experiments, we just trained our models (calling method fit on the model) including the keyword argument class_weight for weighting the loss function as this can be useful to have the model "pay more attention" to samples from under-represented classes.

2.3 Data Augmentation

Since the very beginning we understood that data augmentation was the key improve model generalization capabilities. We tried to perform augmentation both online (during the training itself) and offline (before the training). As far as offline augmentation is concerned, we tried to selectively augment species 1 and 6 data by applying transformations to the partial dataset containing only samples from those classes (multiple times in some cases). Then, we put all data together in a single dataset and we augmented it as well (see Baseline notebook for further information). After trying this experimental approach, however, we switched to online augmentation (used to train the best model). One of the most important benefits of online data augmentation is that different random transformations are applied to the data at each training step since the augmentation layer is integrated in the model definition.

2.3.1 Transformations

At first, we augmented our dataset applying random Zoom, Shift, Rotation and Flip transformations trying different parameters for zooming, flipping and so on. To assemble our initial data augmentation pipeline we used tf.keras.layers.Sequential class along with tf.keras ad hoc layers(see Baseline notebook).
We also tried the library keras_cv; in particular, we built an Augmenter made of three layers: a generic one (keras_cv.layers.RandAugment), CutMix and MixUp layers. CutMix randomly cuts out portions of one image and places them over another image, and then MixUp interpolates the pixel values between two images. We used CutMix and MixUp also because they may help to reduce overfitting, improving generalization capabilities as well. The result of this data augmentation approach (see notebook cut-mix) was not much better than the other (simpler) techniques we had applied so we did not include it in the other experiments.

2.4 Data preprocessing

When building our model from scratch, we included in the preprocessing phase pixel rescaling along with Z Score normalization. While, using Transfer Learning and Fine Tuning, they were not necessary because each super network was provided together with its own preprocessing layer.

2.5 Different Network Architectures

2.5.1 From scratch (Baseline notebook)

At first, we tried to develop a model from scratch. This first model was inspired by VGG architecture, since we made use of couples of Convolutional layers stacked, alternating with Pooling layers. In the last part of the FEN we introduced a Global Average Pooling layer for regularizing. Last portion consists of a shallow dense layer (regularized as well using Ridge Regression) and a softmax output layer. We wanted to explore and analyze the capabilities of a very simple model (less than 140000 parameters) and we needed a baseline for doing comparisons with more complicated models which we were going to develop later on. So, we trained this model leveraging offline in-memory data augmentation and oversampling of Species 1 and 6. As a result, we achieved an accuracy of 77% over remote test set. We noticed from the very beginning that performance on species 1 was pretty poor (near random classifier).

2.5.2 VGG19 (vgg19 notebook)

Our first transfer learning/fine tuning experiment involved the VGG19 network. We only kept the Features Extraction Part and then we attached to it a GAP layer, one dense layer and a softmax output layer. We also included two dropout layers for regularization. We trained the network in two phases: in the first phase we set the VGG layers to not-trainable to the classifier part only (Transfer Learning); in the second phase we set the last 14 layers to trainable, and we retrained the network with a low learning rate (Fine Tuning). This technique gave us 77% of accuracy over the remote test set, not better than our from scratch model.

2.5.3 Inception_v3 (inceptionft notebook)

After trying out VGG19, we decided to move onto Inception_v3 to try out inception layers in our use case scenario. We jumped directly onto fine tuning to better fit the model to our needs. During the first attempt, we unfreezed

the first fifteen and the last fifteen layers of the Inception Features Extraction Network and we unlocked circa 830000 parameters. The custom terminal part of our network was made up, again, of a fully connected layer (128 neurons) ending with a softmax output. We employed dropout layers to regularize since they seemed to be working against overfitting in the previous experiments. The network was trained with a learning rate equal to 0.001 and for 50 epochs circa; data augmentation strategy was the offline one we mentioned previously. Validation accuracy was promising but we noticed that F1 score of the first class was, again, very low. We submitted our model to give it a shot and we got circa 81% of accuracy (overall). Performance was pretty similar to our validation score and so we thought that we were moving the 'right' direction; we had to improve performance on the minority species (1 and 6) whose F1 score was pretty low with respect to other species.

2.5.4 Inception_v3 (inception_ft_moreparams notebook)

We tried to go further with parameters unfreezing sticking to this setting. We incrementally set to 'trainable' more and more layers in both ends of the FEN and we did some experiments with learning rate, trying to keep it quite low for fine tuning purposes. After various attempts, we reached 6 million unlocked parameters. To feed more data to the model, we reduced the portion of data retained for validation purposes, even if this introduced some bias in the validation score computation (since the set was very small and there were very few pictures from minority classes). Validation accuracy was high and, in the end, this model brought us from roughly 81% obtained with slightly less than a million parameters unlocked (see inception_ft notebook) to 87% on the remote test set. There was still something missing.

2.5.5 Final Solution : Inception V3 + quasi SVM (inception_svm notebook)

In the very last round of experiments, we picked our best model to time (see previous section), we retained the fine tuned FEN since it proved to be pretty acceptable in the previous phases and we plugged into our architecture a more robust classifier, **a quasi SVM**, which was the only trainable portion of the network. New custom classifier was made up of a layer to map the extracted features into a Gaussian space (whose parameters were set after some experiments with various combinations) and of a softmax output. Regularization was carried out by means of a dropout layer and using Ridge Regression. Training leveraged online augmentation and oversampling of Species 1 and 6 performed twice; learning rate was set to 1e-3. This very last model made us reach 88.5% of accuracy on the remote test set in the Final submission round. One last attempt to improve our model was training it from scratch on the whole dataset for a similar number of epochs (circa 50) , but this did not lead to the hoped result since we cannot rely on any estimates of its generalization capabilities and so we kept using the version trained on the split dataset, which is our best model (see models/inception).

3 Further experiments

While doing experiments on our network from scratch, we also used an approach inspired by ResNet architecture; in particular, we increased model depth and we introduced skip connections implemented by means of tf.keras.layers.Add layers to try to stabilize and improve training (see resnet notebook). However, accuracy with respect to our model built from scratch did not change significantly, so we dropped it.
Another interesting tool we used is KerasTuner. The script we wrote (see hypersTuningScript notebook) produces the 10 best parameters combinations, id est the ones resulting in the highest accuracy on the validation set. We tuned the images size (experimental), the dropout rate of the two dropout layers, the number of dense layers to put in the network and their number of neurons, the learning rate and the activation function. We tried to train the network using several combinations of the parameters yielded by KerasTuner, but none of the trained networks resulted in remarkable scores over remote test set even if validation score was promising (see Inception-SchedulerAndHypersTuning notebook). We even tried exponential decay in the former experiment. Results were pretty much in line with what we had done before. In the end, we tried to exploit tensorflow abstraction Strategy to train one of our early models on a GPU cluster made up of two cards. We employed MirroredStrategy and results were similar to the one obtained in the single GPU training version (see inception-ft-MULTIGPU notebook, compare it to inception-ft) but we noticed training was numerically less stable. We avoided exploring this realm as GPU hours are too valuable to be wasted doing this kind of experiment (we used platform-as-a-service services Kaggle and Google Colab for carrying out experiments).