Artificial Neural Networks and Deep Learning 2022

Challenge 1

Prof. Matteo Matteucci

Lorenzo Trevisan (Personal Code: 10687901)
Giuseppe Urso (Personal Code: 10628602)
Matteo Venturelli (Personal Code: 10629913)

POLITECNICO
MILANO 1863

1 First Model

Several competitions concerning classification of plants images have been organized in the last decade. One common approach to this problem involves transfer learning, which consists in exploiting the convolutional part of a pre-trained network used for a similar task. This allows to adapt the feature extraction part of a well-established network (such as the ones used to classify ImageNet dataset) to a specific case study instead of designing and training a convolutional architecture from scratch. Even though the implementation of transfer learning implies that the extracted features are not tailored for our specific task, fine tuning can be implemented in order to slightly modify the weights of the pre-trained network and improve the overall performance. In addition, using a convolutional architecture which performed extremely well on images similar to the ones contained in the plants dataset is a solid base to solve the plant classification problem. For these reasons we implemented transfer learning from the beginning.

First of all, the 3542 images dataset was split into training, validation and testing subsets (with proportions 0.8, 0.1 and 0.1 respectively) using the `splitfolders` package. The images were then loaded into batches of dimension 32 by means of either `ImageDataGenerator` and `flow_from_directory` methods, or by means of `image_dataset_from_directory` (we used both methods for different models because each has its own advantages over the other). In this first model no preprocessing or resizing was implemented. The actual model consisted in:

- input layer (`input_shape = (96, 96, 3)`);
- VGG16 network (loaded with `include_top=False` and `weights="imagenet"`, freezed weights);
- flattening layer;
- first dense layer (`units = 256`, `ReLU` activation function and `GlorotUniform` initialization);
- second dense layer (`units = 128`, `ReLU` activation function and `GlorotUniform` initialization);
- output dense layer (`units = 8`, `softmax` activation function and `GlorotUniform` initialization).

VGG16 was the first network we used for transfer learning because several authors already implemented it for plant classification problems. A common practice in transfer learning is to not include the dense part of the pre-trained network and build it from scratch. So we set `include_top=False` and implemented a flattening layer in order to transition from the convolutional part to the dense part of the network. `ReLU` activation function and `GlorotUniform` initialization for the dense layers were chosen because they are the most widely used in literature. The output layer has a number of neurons equal to the number of possible classes and `softmax` activation function in order to be able to interpret the output values as probabilities.

We set `CategoricalCrossentropy` loss function since we are dealing with multi-class classification and `Adam` optimizer with learning rate equal to 10^{-3}. We also implemented `accuracy` as metric (since it is the one used in out-of-sample testing) and `EarlyStopping` callback with `patience = 15` on the validation loss.

No countermeasure for overfitting was implmented yet, so we expected to see much better performance on the training set than on validation and testing sets. As a matter of fact the training stopped after 20 epochs showing `accuracy = 0.9657` and `val_accuracy = 0.7293`. `restore_best_weights = True` allowed to recover the model weights after the fifth epoch which yielded `accuracy = 0.8618`, `val_accuracy = 0.7066` and `test_accuracy = 0.6575`.

2 Further Improvements

First we needed to implement some regularization methods to reduce overfitting. Dropout layers (`rate = 0.3`) were inserted after the first and second dense layers and improved validation and testing accuracy by roughly 2.5%. We also tried to implement L1 and L2 regularization with different `lambda` values, but with minor improvements (or even worsening in some cases). On the basis of the experiments results, which simply consisted in trying out different kinds of regularization and evaluating the obtained models on the test set, we discarded L1 and L2 in favor of Dropout, which happened to be the most consistent technique.

Data augmentation was another fundamental step towards consistent improvement of the metrics. Because of the small number of training images, we expected data augmentation to play a fundamental role in improving performances. As a matter of fact this technique increases the diversity of the training set by applying random (but realistic) label-preserving transformations: in this way we can virtually increase the number of training images and the generalization capabilities of the network. Several tests were performed in order to establish the optimal transformations and parameters. Among the transformations we tried out that did not significantly improve the metrics we mention brightness, zoom and shear, whereas the ones that seemed promising were shift, rotation and flip. The transformations were implemented by means of either `ImageDataGenerator` or by applying a `tf.sequential` containing the transformations to a batch created by `image_dataset_from_directory`. Both methods allow to feed the network with new augmented images each epoch. Data augmentation caused the convergence to be slower, but allowed to have a validation loss to be much more similar to training loss (meaning that overfitting had been successfully reduced). Furthermore, we obtained significantly higher performance on the local test set (accuracy roughly increased by 4%).

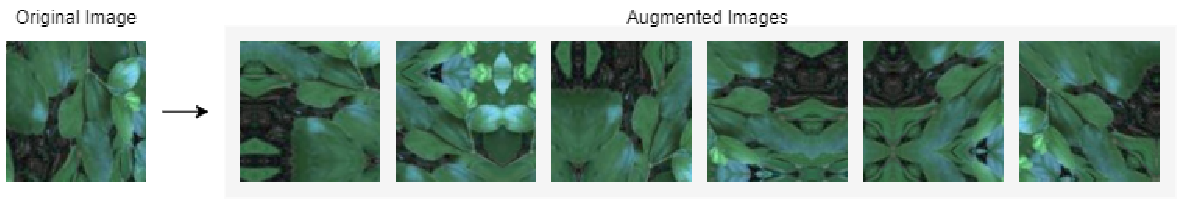

Figure 1: Example of data augmentation starting from a sample image.

In addition to containing a small number of images, the training dataset also presents class imbalance. The least represented class contains only 186 images, whereas the most represented one contains almost three times the images. To avoid having the network perform much worse on some classes with respect to the others, we trained the model with `class_weight` argument, which allows to ”pay more attention” to examples from an under-represented class, improving their accuracy. We also tried to implement oversampling for the least represented classes; at some istances this technique showed minor improvements with respect to `class_weight`.

To further improve performances by a few percent we needed to adapt the images to VGG16 architecture and weights. To do so we implemented VGG16's `preprocess_input` function in the augmentation stage of the network and at inference time. In addition, we also resized the images from 96×96 to 224×224 (which is default input size for VGG16).

After some hyperparameter tweaking we realized we had to implement major changes to have substantial increase in accuracy (test accuracy almost reached 80% so far). With fine tuning of the pre-trained network we were able to obtain significantly better results. At first we tried to fine tune only VGG16's last convolutional block, then we trained with the whole network unfreezed. The learning rate was set to 10^{-4} during fine tuning, 10 times smaller with respect to the transfer learning process. We also introduced `ReduceLROnPlateau` callback with `patience = 5` on the validation loss to reduce the learning rate once the learning stagnates. With all these improvements we were able to reach `test_accuracy = 0.8276` on the local test set.

After reaching this plateu with VGG16 architecture we tried to implement different models with pre-trained weights. Our choice fell on DenseNet, which has seen some use for other plant classification tasks. Alongside DenseNet we also implemented a global average pooling layer to substitute the flattening layer. These updates yielded very good results: with these edits only, the accuracy on the test set improved by roughly 3%. Since this architecture seemed more promising than the previous one, we spent time in tuning the hyperparameters (the model will be described in the next section). To further improve the performances we opted for a `(0.9, 0.1)` splitting of the dataset (where 0.9 indicates the training fraction and 0.1 the validation one). This allowed to have more images to use for training, at the cost of not being able to locally test the prediction capabilities.

3 Final Model

First of all, the images were resized to 224×224 which is the standard for DenseNet. Oversampling has been implemented to triplicate `Species1` images and duplicate `Species6` images to roughly obtain an equal number of images per class. Splitting ratio of the dataset was set to 0.9 for training and 0.1 for validation. The augmentaion transformations consist in:

```
- tfkl.RandomRotation([-0.4, 0.4], fill_mode="reflect");
- tfkl.RandomTranslation(height_factor=0.3, width_factor=0.3, fill_mode="reflect");
- tfkl.RandomFlip("horizontal_and_vertical").
```

The architecture of the final model is the following:

- input layer (`input_shape = (224, 224, 3)`);
- preprocessing layer (layer that applies DenseNet's `preprocess_input` function to images)
- DenseNet network (loaded with `include_top=False` and `weights="imagenet"`, freezed weights);
- 2D GAP layer;
- first dense layer (`units = 256`, `ReLU` activation function and `GlorotUniform` initialization);
- first droput layer (`rate = 0.5`);
- second dense layer (`units = 256`, `ReLU` activation function and `GlorotUniform` initialization);
- second dropout layer (`rate = 0.5`);
- third dense layer (`units = 256`, `ReLU` activation function and `GlorotUniform` initialization);
- output dense layer (`units = 8`, `softmax` activation function and `GlorotUniform` initialization).

We set `CategoricalCrossentropy` loss function and `Adam` optimizer with learning rate equal to 10^{-4}. The number of epochs was set to 500 as maximum value, but in practice was ruled by `EarlyStopping` callback with `patience = 15` on the validation loss function. `ReduceLROnPlateau` callback with `patience = 5` on the validation loss was also implemented. After the transfer learning, fine tuning was implemented by unfreezing the whole DenseNet and retraining the network with the same parameters, except for `Species1` class weight that was set to 2 in order to compensate for its poor classification accuracy during transfer learning. An accuracy of 90.18% was obtained post-fine tuning with this model on the phase 2 external test set.

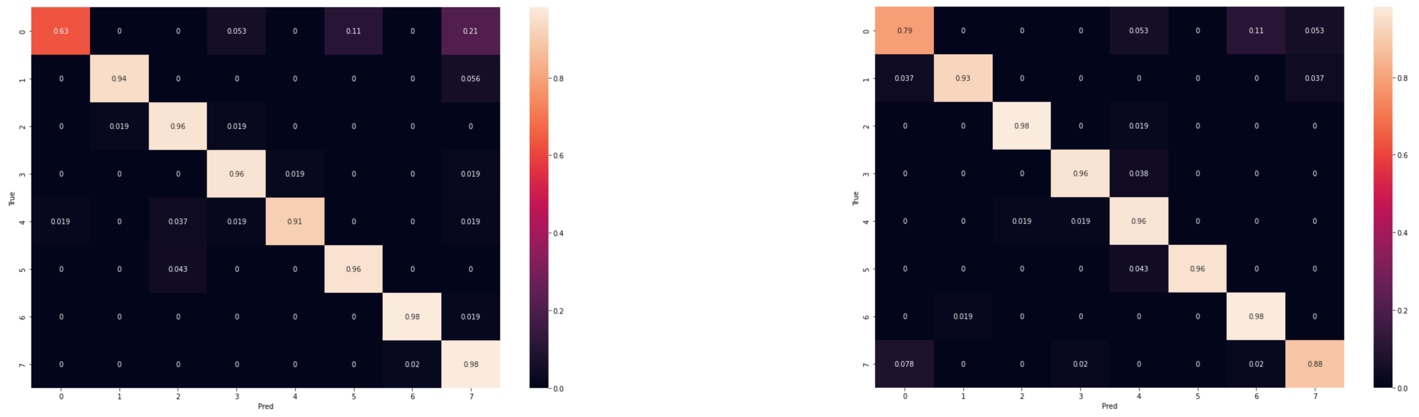

Figure 2: Confusion matrices related to final model after transfer learning (on the left) and after fine tuning (on the right). The matrices were evaluated on the validation set since `(0.9, 0.1)` splitting was used and no test set was available.

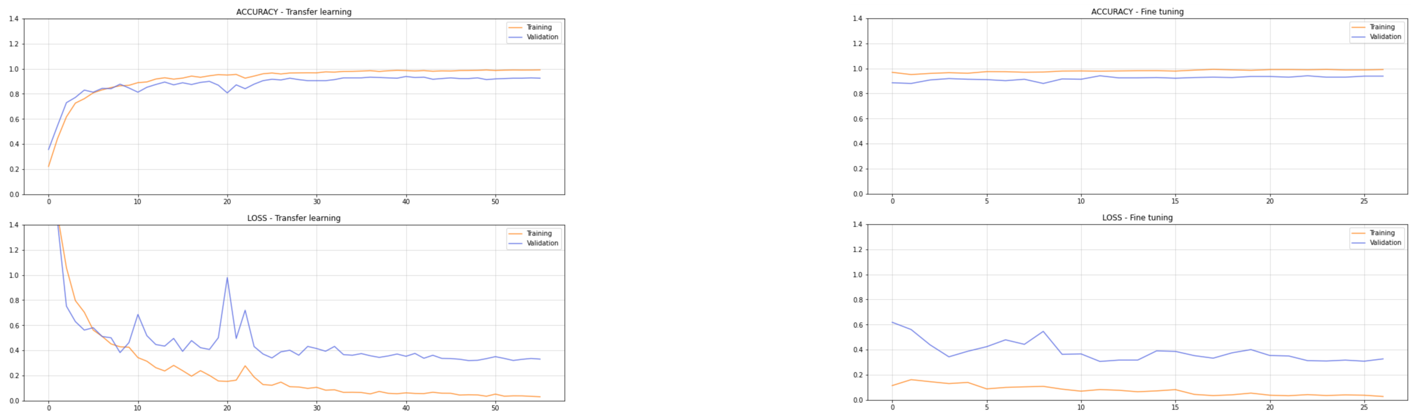

Figure 3: Accuracy and loss function graphs as a function of the number of epochs related to final model during transfer learning (on the left) and fine tuning (on the right). A maximum value of `val_accuracy = 94.15%` was achieved during fine tuning.