AN2DL: First Homework Report

Les Choristes

Pietro Marco Gallo Michele Guerrini Gabriele Maucione

Introduction

The aim of the first homework of the AN2DL course is to design the architecture of a neural network and train it to perform as accurately as possible a multinomial classification task over a dataset of $96 \times 96 \times 3$-sized RGB plant images, divided in 8 classes.

By starting from the concepts seen during class and going through incessant experimenting, our group has designed a deep convolutional neural network which achieved a mean weighted accuracy of 89.671% (90% over the first 30% of the dataset, 89.53% over its second part).

Neural Network Architecture

Our best model is characterized by a high level of complexity, since it is composed of over 24 million parameters. More specific details about their distribution in each layer can be found in Fig. 1. Its design comprises a number of techniques used to enhance accuracy as much as possible, whilst avoiding overfitting accidents: data augmentation, transfer learning, fine tuning, Quasi-SVM application, learning rate decay and K-fold cross validation.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
resizing_3 (Resizing)        (None, 192, 192, 3)       0
_________________________________________________________________
xception (Functional)        (None, 6, 6, 2048)        20861480
_________________________________________________________________
global_average_pooling2d_2 ( (None, 2048)              0
_________________________________________________________________
dense_6 (Dense)              (None, 512)               1049088
_________________________________________________________________
random_fourier_features_1 (R (None, 4096)              2101249
_________________________________________________________________
dropout_4 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_7 (Dense)              (None, 8)                 32776
=================================================================
Total params: 24,044,593
Trainable params: 20,969,881
Non-trainable params: 3,074,712
```

Figure 1: Summary of the model architecture.

Firstly, images are inputted as tensor image data with `tensorflow.keras.preprocessing.image.ImageDataGenerator`, a stage wherein they are divided into three sets (training, validation, test) with respective ratios [0.8, 0.1, 0.1], all their values are rescaled by a factor 255 - the maximum pixel value of an image -, and data augmentation is performed on the training set: images within this set are subject to batch-wise random transformations (rotation, shift in height, width and zoom; flip). Afterwards, a `Resizing` layer is applied in order to double the size of the inputted image. Then, the resized images are fed to the pre-trained `Xception` network, which was used for transfer learning and fine tuning by freezing the first 32 layers, in combination with a custom-designed top-end net, in such a way to make the architecture suitable for this classification task. The custom-designed top end network starts with a `GlobalAveragePooling` layer, which is followed by a sequence composed of a `Dense` layer, a `RandomFourierFeatures` one approximating a Support Vector Machine (in other words, a Quasi-SVM), a Dropout one and a final `Dense` output layer. The number of non-trainable parameters (see Fig. 1) depends only on the first 32 layers of the `Xception` Functional layer, because the `RandomFourierFeatures` layer is set up

in such a way that all its weights are trained. Finally, the output layer, which has 8 units, returns a vector of probabilities to belong to each of the eight classes from 'Species1' to 'Species8' and the maximum probability among them determines the label assignment to the input image.

The described model is compiled with Categorical Crossentropy as a loss function, and with `Adam` optimizer, as they seemed to be the most suitable for the faced task.

The previously saved testing set is used to evaluate accuracy and calculate a confusion matrix. Used callback functions are model checkpoint, early stopping and learning rate scheduler (which implements a slow learning rate decay after the first 10 epochs). More details can be found in the notebook. After the model is trained, it passes through K-fold cross validation, in order to check its behavior in different conditions of the input dataset.

The Steps Behind the Final Model

In order to approximately evaluate in advance different architectures without doing a submit on the challenge platform, we divided the dataset in three subsets: train, validation and test. In such a way, we kept a portion of data (the test set) to obtain an accuracy score beforehand and a confusion matrix for each architecture we tried in the development phase.
In our first approach this was done by means of a custom function to create the dataset array, and of the `train_test_split` function to divide the array in the aforementioned subsets.

The First Model: Vanilla CNN

Our starting point was a quite simple tailor-made convolutional neural network. It was made up of 5 convolutional layers, each followed by a `MaxPooling` layer, and a top end with a `Flatten` layer, followed by a `Fully Connected` layer and the `Output` layer with 8 units. `Dropout` layers were also present after the `Dense` ones, as a way to prevent overfitting.
The same loss function and optimizer of the final architecture were already used here too. This starter model reached an accuracy score of about 67% on Codalab's test set.

Our approach at that point, in the extent involving the convolutional network part, was to maintain this core structure, but intervene by editing the number of involved convolutional layers, as well as the number of filters and the kernel size of each layer. After a first period of trial and error in hyperparameter tuning, a more systematic and senseful approach was found with the help of `keras_tuner` library: the function we used allowed to perform a random search of the optimal values of all the most critical[1] hyperparameters in an automatic fashion.

After having tested different configurations, we found the best one in a concatenation of three convolutional layers, resulting in a model with a contained number of parameters (169k). Once the main architecture was set, overfitting revealed to be a serious problem, so regularization was introduced.

On this kind of architecture we also tried to test different types of pooling (`MaxPooling`, `Average Pooling`), dropout rates (between 0.2 and 0.6), regularizer types (`L1`, `L2`, `L1L2`) and values (between 10^{-5} and 10^{-2}), and different combinations of all these parameters.

As well as the convolutional part, the top-end also changed: the first model, that used a `Flatten` layer, counted a humungous number of parameters. With the aim of reducing the network's complexity for practical needs in the fitting operation, the `Flatten` layer was replaced with a `GlobalAverage Pooling`. This reduced significantly the number of parameters, due to the different change in shape between input and output.
Also, it turned out to work better with no intermediate `Dense` layers before the `Output` one.

In the end, this configuration provided the best performance in terms of accuracy so far, with a 78.8% score on *Codalab* test set.

[1] The least immediate to properly tune with a manual approach.

Data Augmentation

The next idea to improve the model performance was to implement *Data Augmentation*. At first, it was done with a custom function operating on `numpy` arrays and applying geometric transformations to the pixels of the input images in the training set. This did not improve the accuracy of the model, probably also because this approach lacked a bit of randomness.
However, since the dataset was quite small, performing data augmentation was vital in our opinion. That is the reason why we tried again with `ImageDataGenerator`, in particular exploiting the `flow_from_directory` method from Keras. In order to create directories in an automated fashion with a single function, we used the `split-folders` library.
Regarding the batch size, we found by searching in the literature that `Adam` optimizer, which was used to tackle this task, works at its best with small batch sizes, which is why we set its value to 8 in the generators.

From CNN To Transfer Learning

Since the earlier standard CNN did not allow us to overcome the 79% accuracy limit, we decided to adopt transfer learning as a solution. Several deep learning models of `keras.applications` were tried (e.g., `VGG16`, `ResNet`, `EfficientNet`) among the most commonly used ones for multinomial classification. We found in the `Xception` architecture the best compromise between network complexity (mainly for time constraints) and performance quality.

In the context of this prefab configuration, at first the model was trained with the whole `Xception` network freezed (with non-trainable weights), in order to extract features of the dataset for the top-end part. Additionally, fine tuning was performed, by making some of the weights of the imported supernet trainable. In this sense, we tried to freeze a variable number of layers during training, and after some attempts we found out that the best hyperparameter was 32, meaning that only the weights from the 32nd convolutional layer on were trained in the `fit()` function. This comparison highlighted this kind of fine tuning to be more efficient than maintaining the whole set of weights pre-trained with 'Imagenet' (pure transfer learning) by a difference of around 15%. In our final architecture, we only perform the second fit (i.e. fine tuning), reaching optimal results in less training time.

Together with the transfer learning architecture, we added the `Resizing` layer still present in the final model. It was introduced so that the higher image resolution (i.e., the double) could facilitate the image analysis in the following convolutional part.

For the custom-designed top-end, numerous configurations have been tested. Each one always started with a `GlobalAveragePooling`, followed by a concatenation of `Dense` and `Dropout` layers. Also here we have tried different hyperparameter settings. In a first step mainly fully connected layers were used (together with dropout layers and `L1L2` regularizers). Such an architecture, with a 256-unit `Dense` layer, a `Dropout` and the 8-unit `Dense` output reached about 84,6%.

Another improvement was achieved by focusing on the arguments of the `fit()` function. It was set for 200 epochs and used a list of callback functions, including a learning rate scheduler, early stopping and model checkpoint. The learning rate scheduler reduces the learning rate by a factor of 10% each epoch after the tenth one.

A further step added to improve the accuracy performance of the model was to change the classifier in the top network from a linear one to a Quasi Support Vector Machine, implemented with the `RandomFourierFeatures` experimental layer.

Together with this upgrade, one last effort to improve the performance was done on this architecture, that seemed to be the best one obtained at that point, by performing K-Fold Cross Validation with 5 folds. The weights obtained after this training procedure ensured the final result reported in the chapter "Introduction".