Report on first ANN challenge

Alessio Braccini, Giovanni Luca d'Acierno, Davide Li Calsi
November 2021

Introduction

Nowadays Convolutional Neural Networks (CNN) are widely used in tasks like image classification.
The purpose of this challenge is to create a CNN capable of classifying leaves belonging to 14 different classes.

Data preparation

Dataset is divided into 14 different directories. This division suggested us to use the ImageDataGenerator utility to prepare images for the pre-processing part, which also allowed us to split the dataset into training and validation parts easily.
We also fixed a seed for random to have a reproducible model.

Pre Processing

We created two ImageDataGenerator to split the dataset into validation and training and have data augmentation, essential for increasing training performances. In an initial moment, we used a training/validation split of 80/20, which was then increased to 90/10 after the observation of an increase in the performances on the test set.
We added some transformations like shear range, zoom, horizontal and vertical flip, rotation, height and width shift, and brightness change. We also included a custom processing function that introduced Gaussian noise in input images, twhich has made the model more robust from external problems like a missing pixel and similar. We rescaled the images to normalize the pixels' value in the input.

Checkpoints and Callbacks

We created a method that periodically saves model checkpoints during the training and manages Tensorboard log generation. We also added callbacks like early stopping.
At the end of the training, it restores the best weights to maximize the performance of the network.

Model selection

We tried two ways: custom construction of CNN and transfer learning.
We first tried to create a custom CNN with custom convolution and maxpooling layers. These attempts were done in the first week of competition, we tried a model built with different numbers of convolutional and maxpooling layers with the fully connected layer at the end. As a first approach, we tried a simple model with no data augmentation to test Codalab submissions and learn how to build a CNN from scratch. With this first network, we

found some nice validation accuracy around 95% but after submitting that model Codalab returned a 30% test accuracy, so we understood that our model was overfitting.
After further insufficient submissions, we tried to add some regularization to our network to gain some performance and we arrived at around 50% in test accuracy. In addition, we have used a more intense data augmentation (see above for more detail) and this has increased the performance by a 22% reaching 72% in accuracy on Codalab.
Here's a summary.

- Custom Model: it used some convolutional layers with batch normalization and maxpooling with dropout set to 0.2. It ran with one hidden layer composed of 512 neurons preceded by a global average pooling layer, batch normalization, and dropout (0.5) before the classification layer, it used Adam optimizer (learning rate: 1e-3)
    - Validation accuracy: 95.4%
    - Validation loss: 0.145
    - Test accuracy: 72.2%

Several tries were done but nothing better had been found, so we tried to apply the transfer learning technique.
We built a model composed of a pre-trained CNN, without the top part, connected with a custom dense layer and an output one. First, we set the CNN layers' trainability to false, freezing them, and trained only the Fully Connected part with a normal learning rate, around 0.001. Then, lowering the learning rate (around 1e-5) and unfreezing the last layers of the pre-trained CNN, we retrained the network.

Taking a look at the learning curves, after the first training session the accuracy curve looked edgy, then with the unfrozen CNN layers, the curve became regular and clean.
The testing confirmed the good expectations, resulting in an increase and stabilization of accuracy, leading to a big improvement in performances.
The following of this path led us to very promising results, so we tried this approach with several architectures, obtaining fluctuating testing accuracy:

- Inception-ResNetv2: it ran with one hidden layer composed of 512 neurons preceded by a global average pooling layer, batch normalization, and dropout (0.5) before the classification layer, it used Adam optimizer (learning rate: first 1e-3 then 1e-5)
    - Validation accuracy: 91%
    - Validation loss: 0.31
    - Test accuracy: 75.8%

- XceptionNet: it ran with one hidden layer composed of 1024 neurons preceded by a global average pooling layer, batch normalization, and dropout (0.5) before the classification layer, it used Adam optimizer (learning rate: first 1e-3 then 1e-5)
    - Validation accuracy: 98.7%
    - Validation loss: 0.054
    - Test accuracy: 91.8%

In parallel, we conducted similar experiments with other well-known architectures. However, since we were running out of time, we preferred to skip the first stage (i.e. training with the

entire pre-trained network frozen) and went directly for the last one. We also tried changing the number of unfrozen layers and checked how this hyperparameter affected performance. Such method was applied to the following architectures, providing the following top results:

- EfficientNetB2: it ran with one hidden layer composed of 1024 neurons preceded by a global average pooling layer and followed by a dropout layer (0.4) before the classification layer,
    - Validation accuracy 93%
    - Validation loss 0.2347
    - Test accuracy 92.2 %

- vgg16: one Dense layer with 512 units, 0.4 Dropout layer afterward, L2 regularization
    - Validation accuracy 99%
    - Validation loss 0.052
    - Test accuracy 79.6 %

- Inceptionv3: it ran with one hidden layer composed of 1024 neurons preceded by a global average pooling layer, batch normalization, and dropout (0.5) before the classification layer,
    - Validation accuracy 91%
    - Validation loss 0.293
    - Test accuracy 81.3 %

Worse results (< 70%) on the validation accuracy were reached with resnet50, EfficientNet B5, B6 and B7, and Densenet, so those nets were not uploaded on CodaLab due to limitations on the daily submissions.

For the entire Development Phase, we trained our networks with different optimizers. First, we used the classical SGD, then switched to Adam. The latter proved to be much more efficient and effective, resulting in smaller training time.

Conclusions

At the end of the day, after all the testing, we chose to submit the models which achieved the best test accuracy: two fine-tuning models with EfficientNet (one using B0 and one using B2) and one using XceptionNet. For further details, you can have a look at the code in the notebook file.
Overall, this challenge was a great opportunity to get in contact with the practical side of CNN, and experience the technical details of the subject.
In particular, it showed us the effectiveness of fine-tuning, and its clear superiority to custom models. This was somehow expected since the architectures we exploited have been designed and improved for years by clever researchers. Therefore they are expected to be much more optimized than a custom model designed in a week or more.