POLITECNICO MILANO 1863

PROJECT REPORT

Image recognition using CNNs for flower species classification

FIRST CHALLENGE IN ARTIFICIAL NEURAL NETWORKS AND DEEP LEARNING

Authors: LUCAS FOUREST, ACHILLE GELLENS, SIMON LESOUEF

Academic year: 2022-2023

1. Introduction

The following is the report of our work on AN2DL's first challenge of the semester. The goal of the challenge was the classification of images belonging to 8 different flower species using Artificial Neural Networks, namely CNNs. We will try to describe the different steps we went through when solving the challenge, the different problems we encountered and the choices we made. There is also a point to be made about the things we did not consider during the homework but could represent leads both for improving upon the submitted solution and, hopefully, perform even better during the second challenge.

In addition to the final model's notebook, you may find other notebooks we used on the following GitHub repository: `https://github.com/achgls/flower-species-classification`.

2. All about the Data

2.1. Summary

The data is made up of 3542 pictures encoded in JPEG format. It is relevant to note they are low-definition pictures, being only 96×96 pixels. Conveniently, each class is divided into its own subdirectory, which makes it easy to directly flow data from disk. However, the data does not come with a training / validation split, *i.e* we will need to split the data ourselves. As for the test data, it is kept separate and inaccessible, for evaluating submitted models on the `CodaLab` server.

2.2. Dealing with imbalanced classes

We notice that there's class imbalance in the given data. Namely, species 1 and 6 are provided with less than half as many images as the other species are. If we do not compensate for this problem in any way, we're at risk of performing poorly for these 2 classes, undermining the overall performance on the model. Furthermore, if the test set had a different class distribution, where species 1 and 6 were not as rare as in the provided data, then we would have an even stronger drawback on the overall test accuracy. Unfortunately, no information was given about the test set class distribution. Luckily, though, there are some ways for us to handle this problem and make up for it to some extent.

Loss weighting - The first thing in our toolbox, is something that comes built-in `TensorFlow`'s `Model.fit(...)` method through the `class_weight` parameter. It allows one to define weights for each class contribution to the loss function, in such a way that a false negative on a rare class would hurt the objective function much more than a false negative on a more common class.

Following the guidelines in the tutorial [3], we set each class weight to be $w_i = \frac{N}{n_i \times K}$ with K the number of classes, n_i the number of samples in class i and $N = \sum_j^K n_j$ = the total # of samples.

Although this method allows the loss function to better represent the model's performance given the data distribution, it does not prevent overfitting in these classes nor the performance drawbacks in the test set, as the amount of training data is still low.

Balanced data pipelines - The better solution would be to deal with the root of the problem and actually train on balanced data. To do this we need to create balanced data pipelines, by either **oversampling** or **undersampling**, as summarized below.

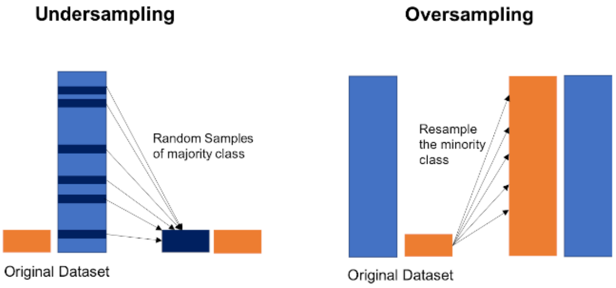

Figure 1: Oversampling vs undersampling

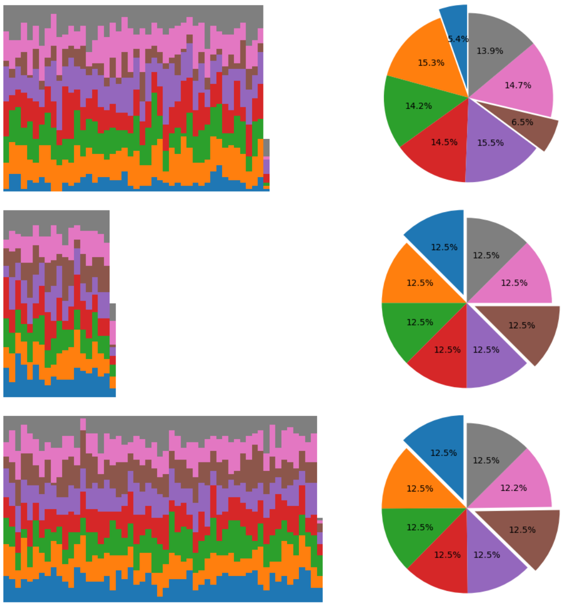

Figure 2: Batch-wise distribution and overall distribution of data pipes; top-to bottom: base dataset, subsampling-balanced, oversampling-balanced

If you randomly over/sub sample at each epoch, then the two options only differ in the fact that for oversampling, you would see the whole data in each epoch, as opposed to over many epochs for subsampling. The steps-per-epoch will also be much lower on the subsampled dataset. If you sample in a fixed way, however, then oversampling is without discussion the better option, as subsampling would throw away so much authentic, valuable data.

We tried implementing a proper class for creating balanced data pipelines using the `tf.data` module, which happened to be a little less easy-of-use as we'd expected, so we ended up not using it in the submitted models although Figure 2 shows the batch-wise distribution of the resulting data pipes. In fact some models were trained on a fixed-oversampled dataset whereas the final submitted model was trained on the raw imbalanced data. Nevertheless, the code is available on the repository in the `balanced_data_pipe.py` file. We hope to make it work for the second challenge, if relevant.

2.3. Data loading and splitting

We used several methods to load and split the data. The easiest way to do both is by using `ImageDataGenerator.flow_from_directory(...)` that allows both automatic splitting of the data and flowing it into training. Some of us also decided to have a script create temporary, manually oversampled `train` and `validation` sub-directories on disk. In that case, the oversampling was thus fixed.

The `image_dataset_from_directory` function from `keras.utils` also allows to create a `Dataset` object from the directory, and was used to build our customized data pipelines.

2.4. Data augmentation

Additionally to the imbalance problem, we can safely say that the dataset is rather small, having between 186 and 537 images per class, not to mention the low quality of the pictures. As a result, we're at a high risk of **overfitting**. To increase the generalization power of the model, we can either regularize the trained weights in some way (see 4.3) or take action at the root of the problem, *i.e* the data itself, by applying **data augmentation** to virtually expand our dataset.

Image data augmentation goes as easy as inputting the range of the different transformations we wish to apply as parameters of the `ImageDataGenerator` class. These transformations include rotating, shearing, flipping, shifting or zooming in or out the image, but it also allows to randomly alter the brightness and the color information. Figure 3 showcases examples of augmented versions of an original image.

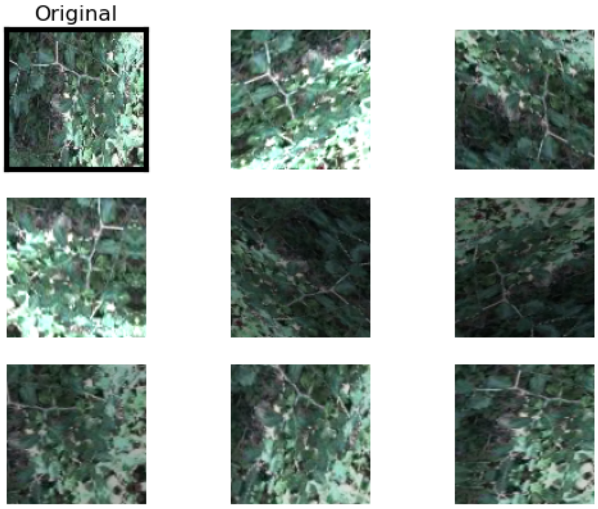

Figure 3: Original image and augmented variations

3. Handmade CNN

The first model that we developed aimed at obtaining initial results for a baseline. For this purpose, we built a model similar to the one seen during the laboratory session, being made up of 5 sequential convolution-then-pooling blocks, followed by a flattening layer. On top, we added one intermediate fully-connected layer with dropout and then the final output FC layer, with `softmax` activation. This baseline model resulted in a 67% accuracy on its validation split but was not submitted for scoring on the test set.

4. Transfer Learning

The usual way to go with image classification problems is to use pre-trained CNNs for extracting features from the image. Furthermore, these models were trained on immense image datasets, which can somewhat make up for the fact that *our* data is small.

4.1. Choosing the feature-extraction CNN

We have plenty of CNN architectures to choose from, for the feature-extraction part of our network. One way to sort it all out is to look through the available papers dealing with tasks related to ours. For instance, [1] features a great comparison of the performance of different architectures for *automatic identification of plant diseases*.

The different architectures we tried include DenseNet, EfficientNet, VGG, Inception, ResNet... conveniently all available on-the-go with `keras.applications`. Under similar conditions, these architectures provide somewhat similar results. Perhaps the low input-size does not allow the more complex models to demonstrate their full power.

4.2. Adding fully-connected layers

We now have to build a fully-connected feature-interpretation — or *classification* part — on top of our convolutional feature-extraction pipe. There are a number of parameters to play with here: number and size of intermediate layers, dropout and L1/L2 regularization strength which help reduce overfitting.

4.3. Training strategy

The pre-trained weights in the feature-extraction part were trained on immense amounts of data and we assume that they are *very good* weights. In order to get them to adapt to our very specific task, we only want to *fine-tune* them ever so slightly, which translates to using a lower learning rate.

In practice, this is a 2-step process. First, you *freeze* the parameters of the convolutional pipe and train only the dense part. Once converged, you then unfreeze the convolutional part and use a much lower learning rate to fine-tune its weights.

To converge to even better results, we found it useful to reiterate the fine-tuning process using increasingly low learning rates and decreasing the image augmentation. That way, we dedicate most of the training to generalization using deformed images, before teaching the model what *nominal* data actually looks like.

5. Best model: bilinear-CNNs

Our model that provided the best results — with 88.3% accuracy on the final test set — was inspired from [5] which presents bilinear CNNs as a mean to solve fine-grained visual recognition problems, which are cases where the perceptual difference between images is often composed of an overwhelming part of non-semantic *contextual* difference, *i.e* background, point of view, illumination, which makes it hard to pick up on the characteristic features of each object. Another explanation of the subject can be found in [4]. The actual implementation was inspired from [2]. More information can be found at [6].

The submitted model applies bilinear pooling as described in the paper to the outputs of two VGG16 architectures. The resulting bilinear feature vector is then fully-connected to the output layer, with softmax activation. It is to note that neither dropout nor any form of weight regularization was applied and yet, suprisingly, it gave us the best results.

6. Leads for improvement

Unforunately, there are a lot of things that we could not explore in the time frame of the challenge.

Hyperparameter tuning - We did not conduct any proper parameter-tuning, but rather tried a couple of different setups and went with what seemed to work best. We learned about `KerasTuner` a bit too late into the challenge, but for the next one, we would like to use it and build a proper, reliable tuning flow that we could run on our every model.

Ensembling techniques - Another promising lead that we did not have the time to explore is found in the ensembling techniques: boosting, bagging, stacking. The goal is to combine several weak classifiers with low bias-high variance into a low-bias low-variance meta-model.

Further data augmentation - There are augmentation techniques that we did not try to play with, such as `cutout` and `cutmix`, as proposed by E. Lomurno in the challenge's logbook. Also, there are ways to do image augmentation on GPU, instead of CPU as `ImageDataGenerator` does. This could drastically reduce the time between epochs.

References

[1] Justine Boulent, Samuel Foucher, Jérôme Théau, and Pierre-Luc St-Charles. Convolutional neural networks for the automatic identification of plant diseases. *Frontiers in Plant Science*, 10, 2019.

[2] Sebastian Correa. Bilinear cnn models in tensorflow-keras. `https://scorrea92.medium.com/bilinear-cnn-models-in-tensorflow-keras-801121cc8c4d`, 2019.

[3] TensorFlow documentation. Classification on imbalanced data. `https://www.tensorflow.org/tutorials/structured_data/imbalanced_data#class_weights`.

[4] Ahmed Taha. Bilinear cnn models for fine-grained visual recognitions. `https://ahmdtaha.medium.com/bilinear-cnn-models-for-fine-grained-visual-recognition-b25ba24d3147`, 2018.

[5] Lin Tsung-Yu, RoyChowdhury Aruni, and Maji Subhransu. Bilinear cnns for fine-grained visual recognition. 2017.

[6] Papers with Code. Fine-grained image classification. `https://paperswithcode.com/task/fine-grained-image-classification#datasets`.