Convolutional Neural Network for Plant Disease Classification

Claudio Moroni
University of Turin
claudio.moroni@edu.unito.it

Davide Orsenigo
University of Turin
davide.orsenigo@edu.unito.it

Pietro Monticone
University of Turin
pietro.monticone@edu.unito.it

16 June 2020

Abstract

In this effort we train a CNN model in order to take part in the Plant Pathology 2020 - FGVC7, an image classification task where we ultimately achieved a categorical roc score of 0.956 using denseNet121, and 0.915 using a relatively shallow CNN defined and trained from scratch. The train and the test datasets are both composed of 1821 images, showing various leaves which are to be classified in 4 categories: "healthy", "multiple disases", "rust", "scab", where we founfd no indication that "multiple_diseases" refers to both "rust" and "scab", it just seems to indicate the presece of other (different) kinds of illnesses. The metric used is the categorical roc. In the end, we also output a visulization of the filters and activation maps of the layers, and an SVD decomposition of the dataset. The techniques we used during training are balancing classes with SMOTE, data augemntation with Keras ImageDataGenerator, optimal dropout and epoch grid searching. Where possible, also auxiliary elements of the pipeline (e.g. SMOTE) have been manually fine tuned.

Keywords: deep learning; convolutional neural network; image classification

1 Model training

The Challenge consisted in classifying leaves images into four categories: HEALTHY, MULTIPLE_DISEASES, RUST, SCAB. Altough a MULTIPLE_DISEASES leaf could be affected both by rust and scab, or by rust and another disease or by scab and another disease, because there is no taxonomy we treated the classes as mutually exclusive. This is to say that in principle the model should distinguish between all four classes, as none of them is an abstraction of (some of) the others. The evaluation metric is the column-wise ROC. We implemented an explicit keras model (EKM in the following), then also a pre-teined model - DenseNet121 - was used. In order to build a more robust model, we tried/implemented the following techniques (in chronological order):

1. Class balancing with SMOTE.
2. Data augmentation with keras ImageDataGenerator.
3. Some fine-tuning/exploration of the models' layers and parameters, in particular e dropout layer (EKM only) , Early Stopping (DenseNet121 only), learning rate scheduling (DenseNet121 only).
4. An attempt to put a convolutional autoencoder autoencoder between point 2 and point 3.

Secondary information may be found in the Appendix.

2 Class balancing with SMOTE

`SMOTE(sampling_strategy,k_neighbors)` is a class balancing algorithm that operates as follows: (one of) the minority class(es) is considered, a random point from it is picked and its first `n_neighbors` nearest neighbors are found. One of the latters is then randomly selected, and the the vector between this point and the originally selected point is drawn. This vector gets then multiplied by a number between 0 and 1, and the resulting synthetic point is added to the dataset. There exist many variants of **SMOTE**, so besides the standard one also the **SVMSMOTE** and **ADASYN** have been tried. **SVMSMOTE** is a variant of **SMOTE** that first of all fits an SVM on the data, and uses its support vectors to identify points more prone to misclassification (i.e. those on the border of the class cluster): these points are later oversampled more than the others. **ADASYN** instead draws from a distribution over the minority class(es) that is pointwise inversely proportional to their density. That is, more points are generated where the minority class(es) are sparser, and less points where they are more dense. Anyway, the class balancing algorithm that ultimately performed better is baseline **SMOTE**, with some fine tuning on the `sampling_strategy`(the `all` value means that all classes are resampled

to match the size of the majority class), and the `n_neighbors` parameters. See Platform limitations

3 Data augmentation with Keras' ImageDataGenerator

Click the following link for an introduction to keras Image preprocessing API. Using Keras' ImageDataGenerator, after a manual inspection of the images, we founfd that the best data augmentin technique consisted in a random planar rotation, mixed with random horizontal flip.

4 Some fine-tuning/exploration of the models' layers and parameters, in particular e dropout layer (EKM only) , Early Stopping (DenseNet121 only), learning rate scheduling (DenseNet121 only) and optimizer variations (EKM)

The explorations reported in the title of this section have been performed. We couldn't implement Early Stopping in the EKM model, as fluctuation in either validation loss, categorical accuracy and row-wose ROC where too high to set proper `min_delta` and `patience` parameters in TensorFlow's Early Stopping implementation. (We also explored some optimizers) *(Visto che l'early stopping di tensorflow non va, usiamo quello manuale già implementato? Quali altri optimizers abbiamo provato? Abbiamo una submission per questi?).* The manual implementations of the dropout and early stopping acted simultaneoulsy, so they performed like a grid search. The dropout and epoch values corresponding to the best column-wise ROC were saved and used during testing phase.

5 An attempt to put a convolutional autoencoder autoencoder between point 2 and point 3.

Despite the multiple configurations tried, the best we could get is a 0.7 column-wise ROC. The reason behind it could be the fact that on one hand an autoencoder with no pooling on the encoder side makes little sense in terms of dimensionality reduction,s on the other hand even a single bidimensonal maxpooling caused the output image to be too little for last EKM layer to classify. See Platform limitations.

6 Appendix

6.1 column-wise ROC

From Challenge Overview on Kaggle: >Submissions are evaluated on mean column-wise ROC AUC. In other words, the score is the average of the individual AUCs of each predicted column

6.2 Platform Limitations

The newest unstable version of TensorFlow with GPU support is needed to run the code. Unfortunately, we haven't been able to set proper kernels up on our local machines, so we had to rely on publicly available cloud interactive environments like Kaggle, that provided free out of the box kernels for our purposes. The only limitations are in terms of cpu RAM, which forced us to downsize the images to about $200 * 200$ pixels.

6.3 Visualization

6.3.1 PCA

6.3.2 Convolutional Filters and Features Maps