Plant Pathology 2020 - FGVC7

Identify the Category of Foliar Diseases in Apple Trees

Claudio Moroni | Davide Orsenigo | Pietro Monticone

19 June 2020, *University of Turin*

1. Remember to cite the Dataset Paper;
2. Separate official from attempted solutions;
3. Stress the difference between `SMOTE` and `ImageDataGenerator`;
4. Model architecture diagram with `tensorflow`;

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 foundfd 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.

The evaluation metric is the column-wise ROC.

Data

Both the training and the test datasets are composed of 1821 high-quality, real-life symptom images of multiple apple foliar diseases, with variable illumination, angles, surfaces, and noise have been manually captured, expert-annotated to create a pilot dataset for apple scab, cedar apple rust, and healthy leaves. 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.

- categories/ classes
- mutual exclusivity
- class balancing

PCA

Pipeline Description and Model Training

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.

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

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 foundfd that the best data augmentin technique consisted in a random planar rotation, mixed with random horizontal flip.

Model Exploration

- Some fine-tuning/exploration of the models' layers and parameters, in particular a dropout layer (EKM only)
- learning rate scheduling (DenseNet121 only)
- 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.

Results

- **ROC performance**
- **Confusion matrix**
- **Training history**

Appendix

Column-wise ROC

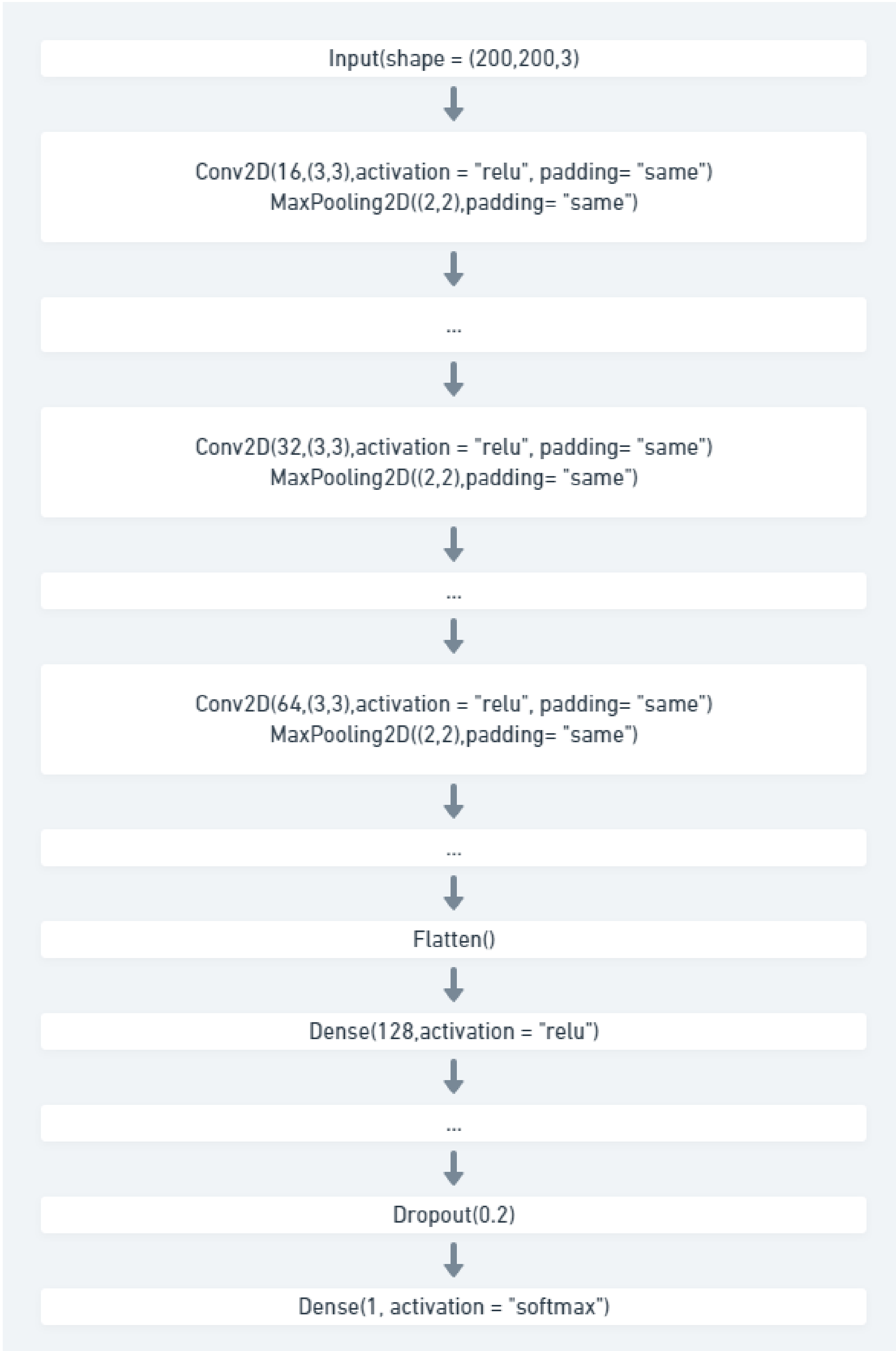
From Challenge Overview on Kaggle: >Submissions are evaluated on mean column-wise ROC AUC. In other words, the score is the average of the individual AUCs of each predicted column.

Platform Computational 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.

Model Architecture

Some online resaerch and trial and error with the network architecture gave us some clues about how to build from scratch an effective, dataset dependent model for image classifcation tasks. the network should of course start with a Input layer, followed by blocks of Conv2D-Pooling (MaxPooling in our case) layers. The number of these blocks should be such that the last of them outputs a representation of $n \times n$ pixels ($\times c$ channels) where n is of the order of units. This should be then followed by $1 - 2$ dense layers, and a final dense classifier layer. If The classification is binary (sigmoid), then the last layer should be preceeded by a BatchNormlization layer.


Input(shape = (200,200,3)
Conv2D(16,(3,3),activation = "relu", padding= "same")
MaxPooling2D((2,2),padding= "same")
...
Conv2D(32,(3,3),activation = "relu", padding= "same")
MaxPooling2D((2,2),padding= "same")
...
Conv2D(64,(3,3),activation = "relu", padding= "same")
MaxPooling2D((2,2),padding= "same")
...
Flatten()
Dense(128,activation = "relu")
...
Dropout(0.2)
Dense(1, activation = "softmax")

Convolutional AE

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 reductions, while 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 and see Model architecture . The only way way we managed to at least run it was to build a very shallow autoencoder (just a couple of layers besides the input and the output), with the result that the loss didn't decrease much. Anyway, inspired by the work of others and by some trial and error, we had a chance to collect some architectural criteria to build an convolutional autoencoder that at least learns. The folowing is to be intended as an empirical recipe, with no or little theoretical foundation of the reasons behind its ingredients. The autoencoder is divided in an encoder and a decoder. The encoder should of course start with an input layer, followed by some blocks of Conv2D and Pooling layers (in our case it was MaxPooling2D). Deeper layers should have decreasing filter numbers (for images as big as ours, a range from 64 to 32 should work). The decoder should start with a specular copy of the encoder, where Conv2D layers are substituted by Conv2DTranspose,Pooling by UpSampling. The decoder shall then have as its last two layers a BatchNormalization layer and Conv2DTranspose with 3 filters (in order to be able to compare output with input) activated by a sigmoid (this explains the BatchNormalization layer). The unknown numner of Conv2D-pooling blocks in the encoder (that determines the number of Conv2DTranspose-UpSampling in the decoder) has to be jointly concocted with the number of Conv2D-pooling layers of the network (see Model Architecture)

Visualization

PCA

Convolutional Filters and Features Maps

Figure 1: **Figure 2.** Convolutional Autoencoder

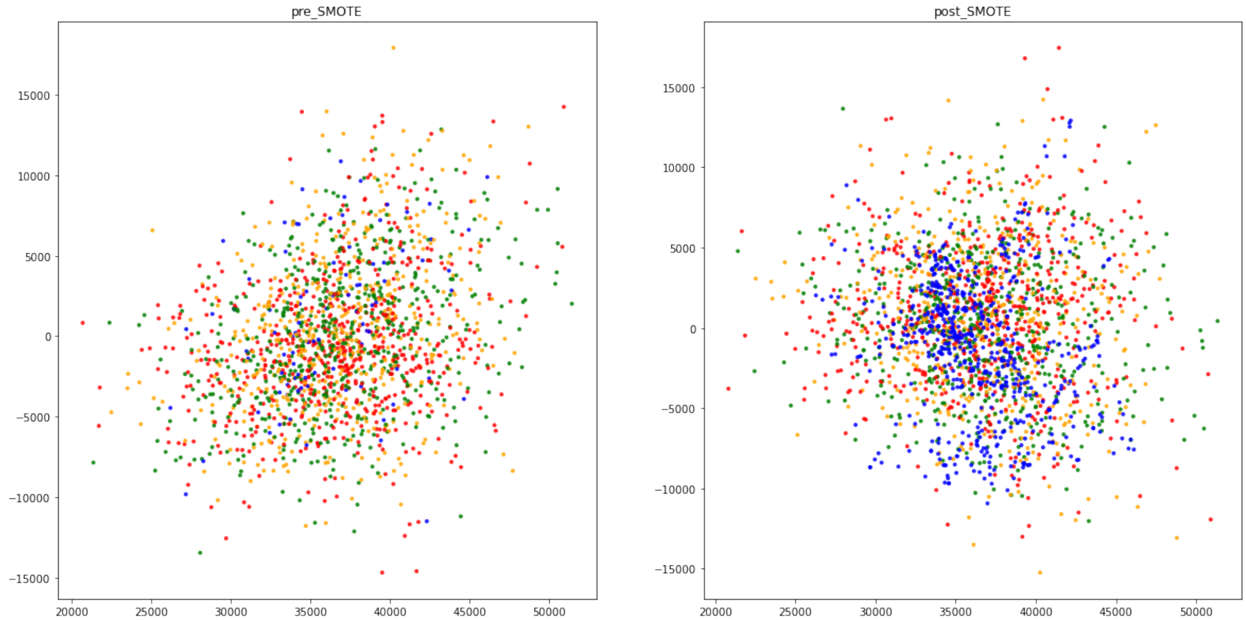


Figure 2: **Truncated SVD**. texttexttexttexttexttexttexttexttexttexttexttexttexttexttext-texttexttexttexttexttexttexttexttexttexttexttexttexttexttexttexttexttext

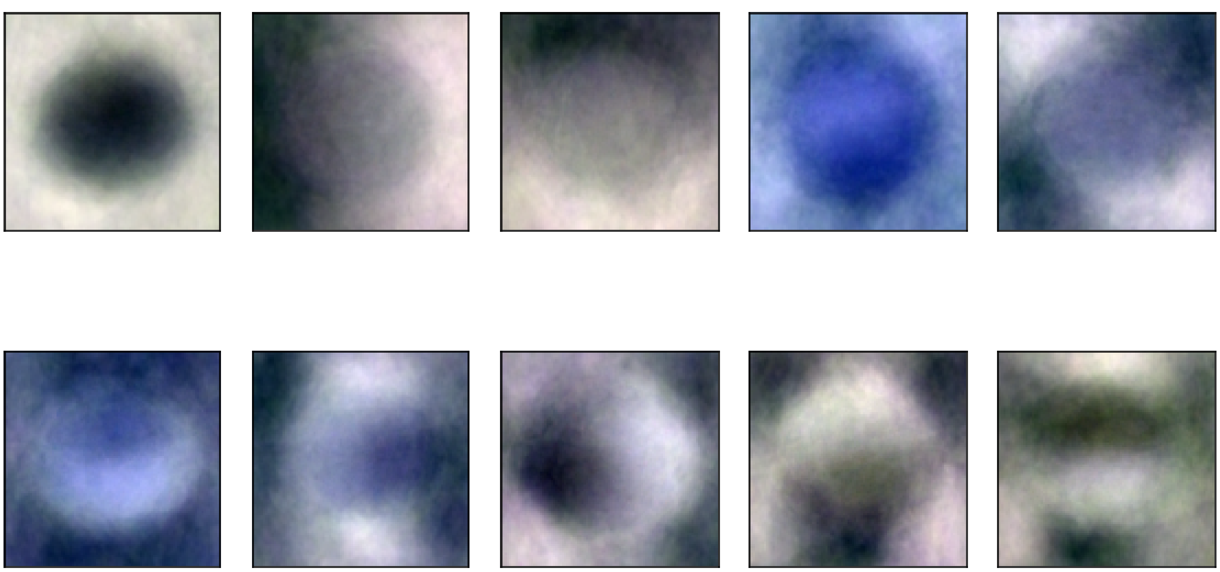


Figure 3: **Figure M.** Layer 1 of PCA.

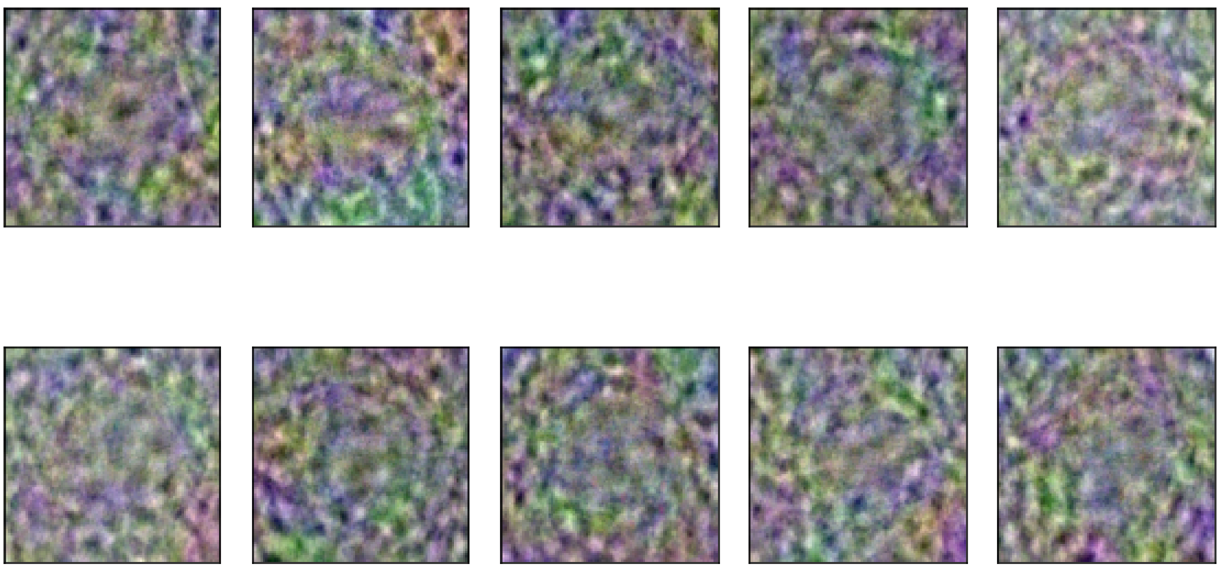


Figure 4: **Figure M+1.** Layer 200 of PCA.