Dataset Analysis and CNN Models Optimization for Plant Disease Classification

Recognition of Foliar Diseases in Apple Trees

Claudio Moroni | Davide Orsenigo | Pietro Monticone

22 June 2020, *University of Turin*

Abstract

We have attended the Kaggle challange *Plant Pathology 2020 - FGVC7*. In this effort we have trained a convolutional neural network model with the given training dataset to classify testing images into different disease categories. During the training phase we have adopted class balancing, data augmentation, optimal dropout, epoch grid searching and, wherever possible, we have also manually fine-tuned the auxiliary elements of the pipeline. The SVD decomposition of the dataset, the convolutional filters and activation maps have been visualized. We have ultimately achieved a mean column-wise ROC AUC of 0.972 applying the pre-trained Keras model `DenseNet121` and 0.937 applying `EKM`, a relatively shallow CNN defined and trained from scratch .

Data

Both the training and the testing datasets are composed of 1821 high-quality, real-life symptom images of multiple apple foliar diseases to be classified into four categories: `healthy` (h), `multiple_diseases` (m), `rust` (r), `scab` (s).

Altough a leaf labeled as `multiple_diseases` could be affected by a variety of diseases including rust, scab or both, we treated the classes as mutually exclusive because there is no taxonomy: in principle the model should distinguish between all four classes, as none of them is an abstraction of any of the others. The dataset is not balanced, but distributed as follows ($h = 516, m = 91, r = 622, s = 592$).

Here we visualize the first two principal components of a truncated SVD to qualitatively investigate the linear separability of the dataset [1].

As one could have resonably expected given such a high dimensionality, the dataset is not linearly separable.

Later in the report we will describe an attempt using a convolutional autoencoder while in the next section we can appreciate the amplification of the classes performed by `SMOTE` and recognize the clustering of the generated points.

Pipeline and Other Attempts

Class Balancing with `SMOTE`

`SMOTE(sampling_strategy, k_neighbors)` is a class balancing algorithm that operates as follows:

[1] **Assumption**: if the dataset is linearly separable, the direction along which which the classes diverge is one of the principal components with larger retained variance, otherwise the noise would be greater than the signal.

Figure 1: **FIGURE 1.** Sample of training images.

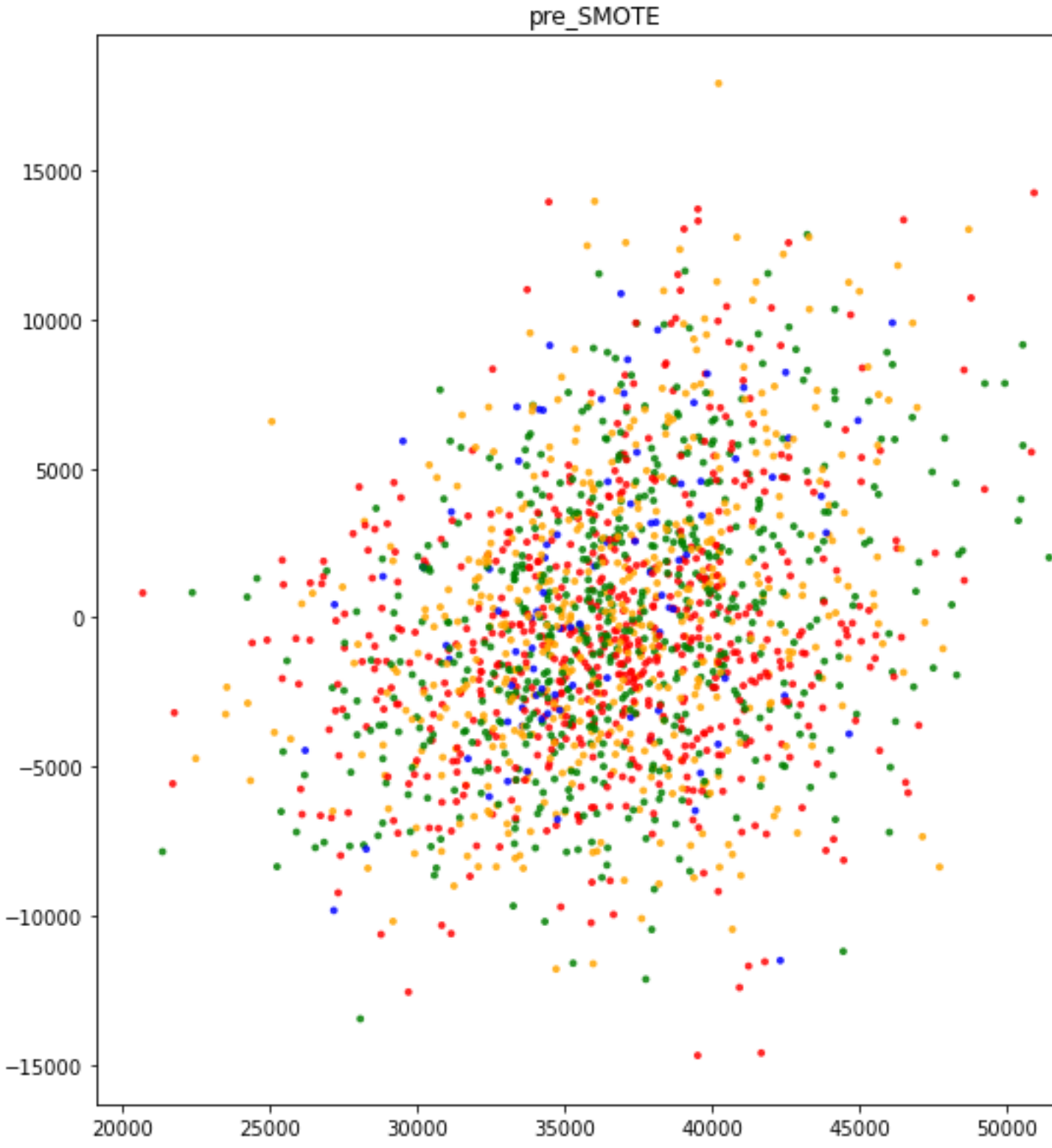

Figure 2: **FIGURE 2.** Pre-‘SMOTE‘ truncated SVD.

1. (one of) the minority class(es) is considered ;
2. a point is randomly chosen and its first `n_neighbors` nearest neighbors are found ;
3. one of those nearest neighbors is then randomly selected, and the vector between this point and the originally selected point is drawn ;
4. this vector is multiplied by a number between 0 and 1, and the resulting synthetic point is added to the dataset.

Besides the baseline variant, `SVMSMOTE` and `ADASYN` have been tried too:

- `SVMSMOTE` starts by fitting an SVM on the data, identifies the points which are more prone to misclassification (i.e. those on the border of the class cluster) via its support vectors and then will oversample those points more than the others.
- `ADASYN` instead draws from a distribution over the minority class(es) that is pointwise inversely proportional to their density so that more points are generated where the minority class(es) are sparser, and less points where they are more dense.

The best perfomance was obtained with baseline `SMOTE`, with some fine-tuning on the `sampling_strategy` [2] and the `n_neighbors` parameters. For further details see Platform Limitations.

Data Augmentation with Keras `ImageDataGenerator`

We have adopted the Keras `ImageDataGenerator` and, after a manual inspection of the images, we found that the best data augmentation technique was a random planar rotation combined with random horizontal flip. For more details see the Keras image pre-processing API.

Model Architecture Exploration

An extensve exploration of all models has been performed. here we report the *gridsearch* that ultimately proved to be better.

- Some fine-tuning/exploration of the models' layers and parameters, in particular a dropout layer (EKM only)
- optimizer variations (EKM)
- optimal dropout and epoch number search
- checkpointing

We couldn't implement Early Stopping both in the EKM model and the DenseNet, as fluctuations in either validation loss, categorical accuracy and column-averaged ROC where too high to set proper `min_delta` and `patience` parameters in TensorFlow's Early Stopping implementation. With some trial and error, the best optimizer choice for the EKM proved to be the **RMSprop**, while the standard **adam** performed quite well with the DenseNet. The manual implementations of the dropout and early stopping searches acted simultaneoulsy, so they performed like a grid search. The dropout, epoch values and weight corresponding to the best column-averaged ROC were saved and used during testing phase. Later, in order to establish the quantitative impact of stocasticity in weights initialization on the EKM, an EKM model with the best drop is trained and validated, and the best epochs of this model and its analogon trained and validated previously are compared: there was a small difference, so we decided to make three submissions: one with the baseline model re-trained on all data and with th best drop, one with a model formed from the bst weights founfd before, and one with the DenseNet. Besides fluctuatios, we noticed that the DenseNet tends to occasionally reach higher submission scores. Considering the fact that optimal epoch number varies with trains et size, a possible third attempt would have seen the best epoch number to use in test phase, when the model is retrained on all train data, extrapolated from a (best-epoch) vs (training set size) plot (given stocasticity

[2] The value `all` means that all classes are resampled to match the size of the majority class.

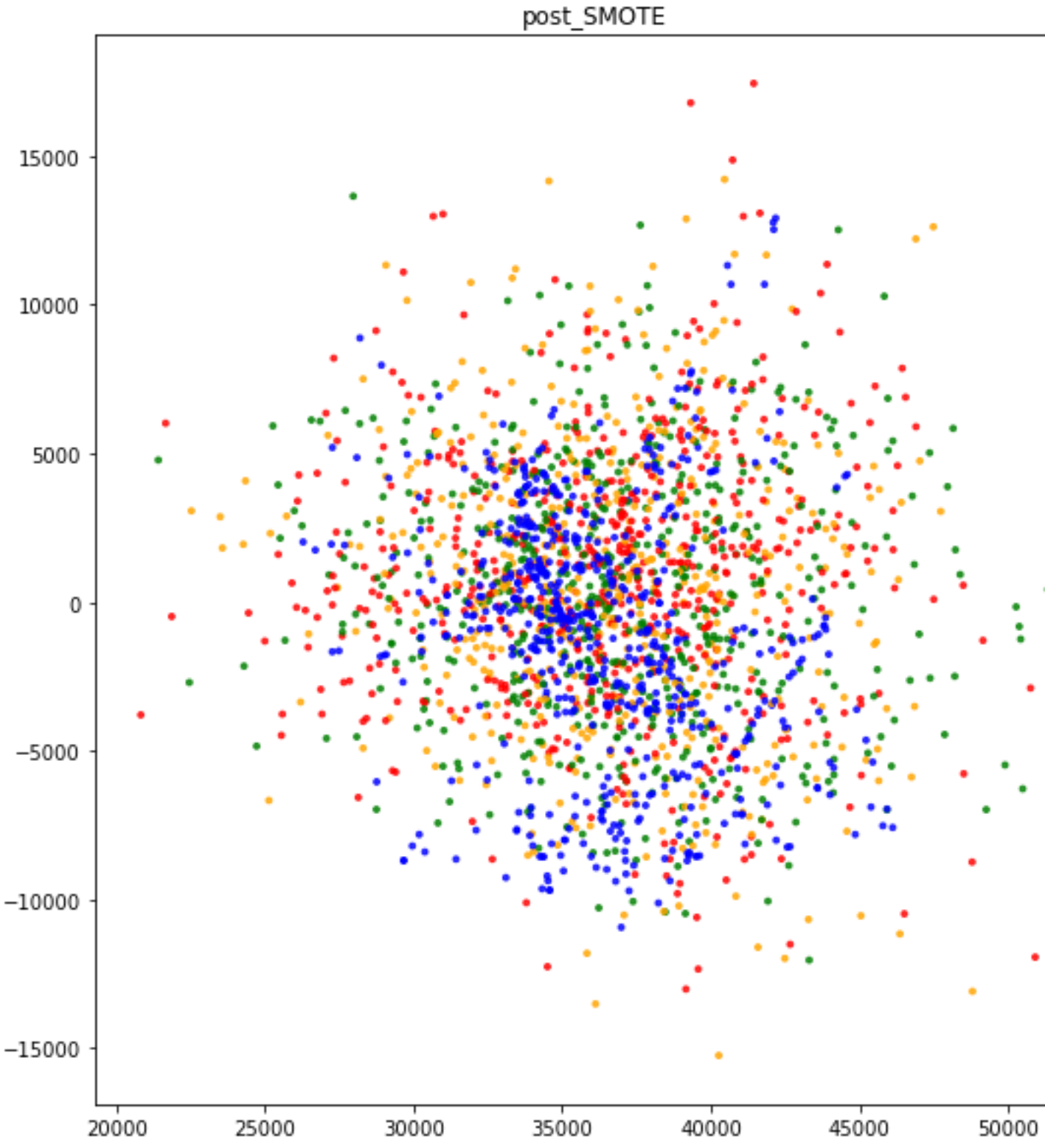

Figure 3: **FIGURE 3.** Post-‘SMOTE‘ truncated SVD.

was not relevant), but this has unfortunately been impossible due to two reasons: a techinical difficulty in combining scikit's Leraning curves with a era smodel necessarily trained with generators, and platform RAM limitations.

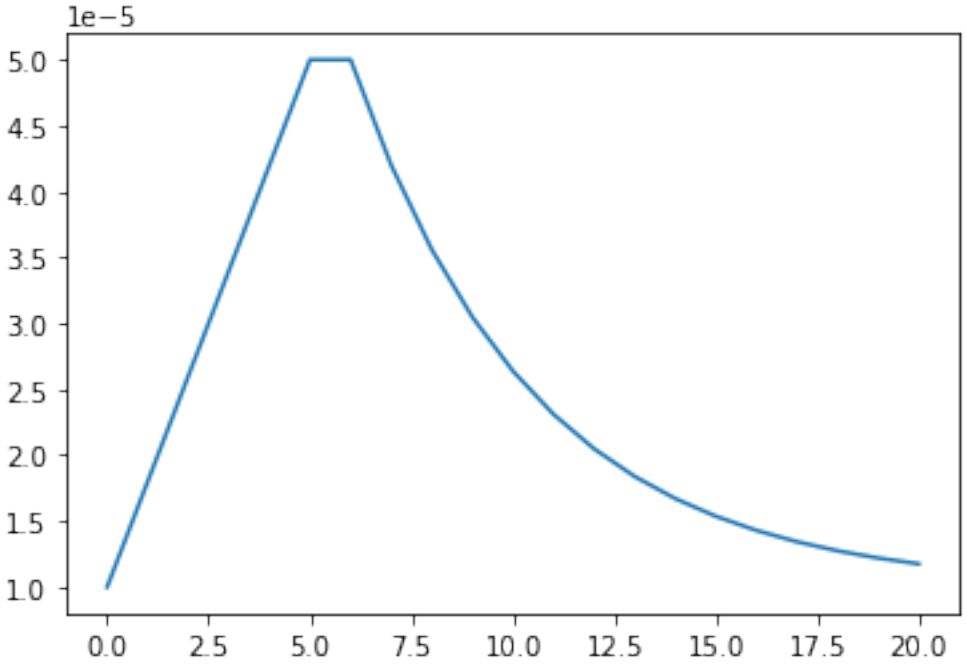

Figure 4: **Figure 2.** Convolutional Autoencoder

Convolutional Autoencoder

Putting a Covolutional Autoencoder between the Smoted & augmented data and the model training, despite the multiple configurations tried, the best we could get is a 0.7 column-averaged 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 and see some loss drop 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)

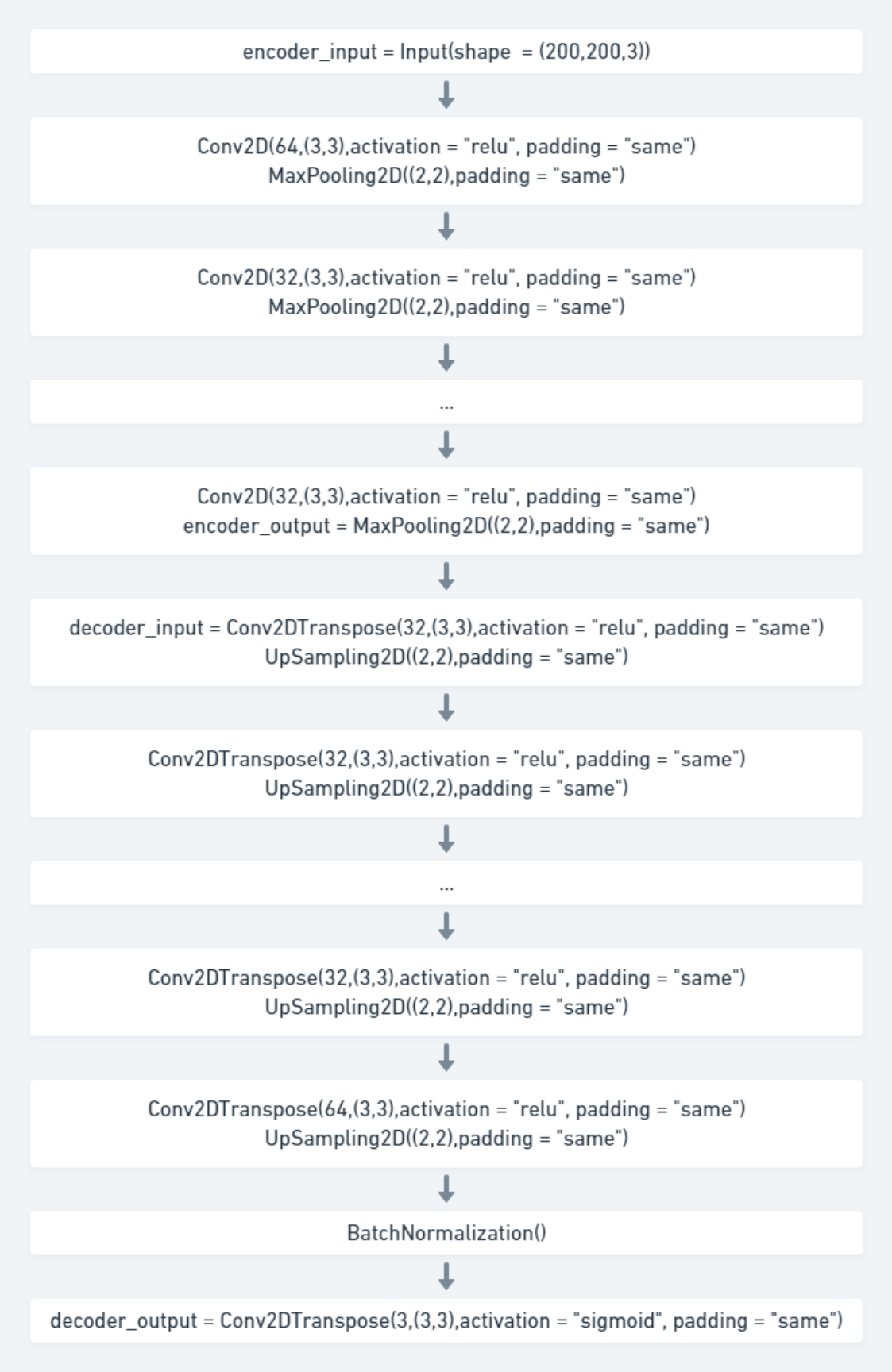

Figure 5: **Figure.** Convolutional Autoencoder

Selected 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.

Results

The performance of the models has been evaluated on **mean column-wise ROC AUC**: 0.972 for `DenseNet121` and 0.937 for `EKM`.

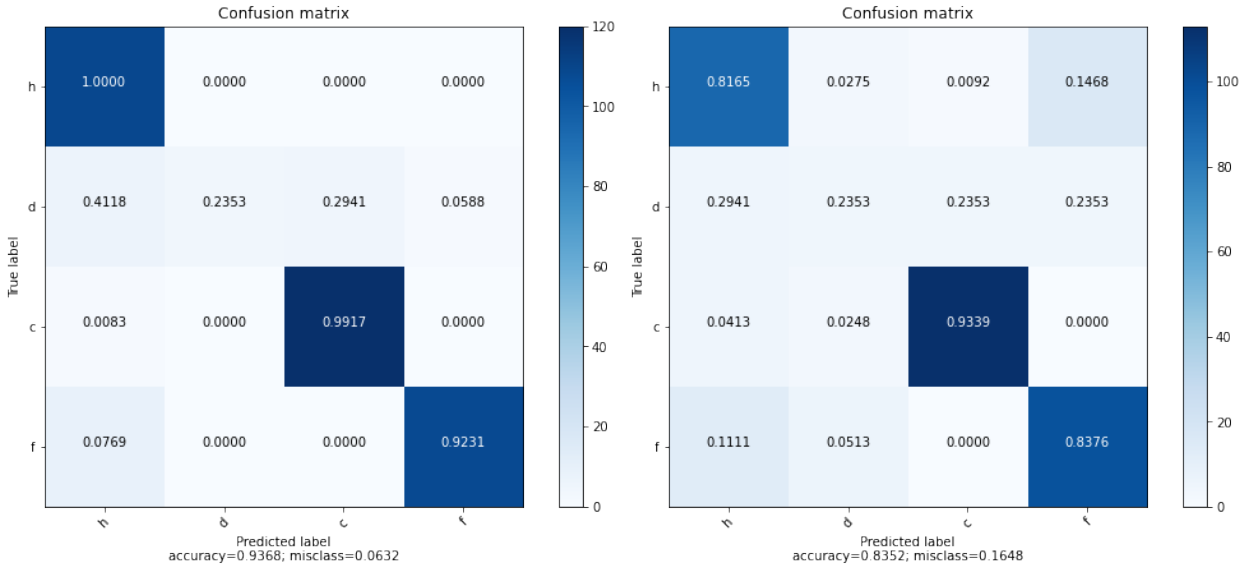

Figure 6: **FIGURE.** Confusion matrices of ‘EKM‘ (left) and ‘DenseNet121‘ (right).

Appendix

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.

Visualization

PCA

Nonostante si sia poi stabilito di non usare la PCA per la dimensionality reduction del dataset può essere interessante visualizzare le prime dieci direzioni principali e confrontarli qualitativamente con una sequenza di direzioni principali con meno varianza ritenuta.

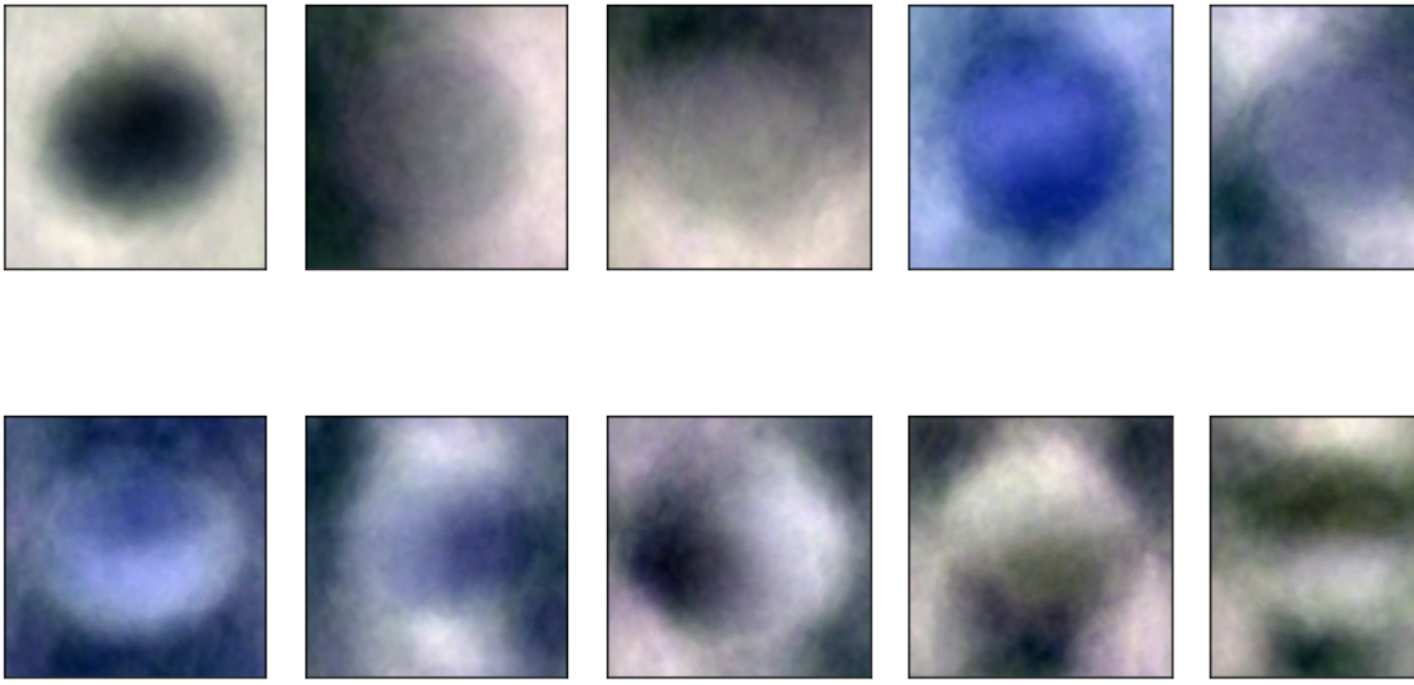

Figure 7: **Figure.** Layer 1-10 of PCA.

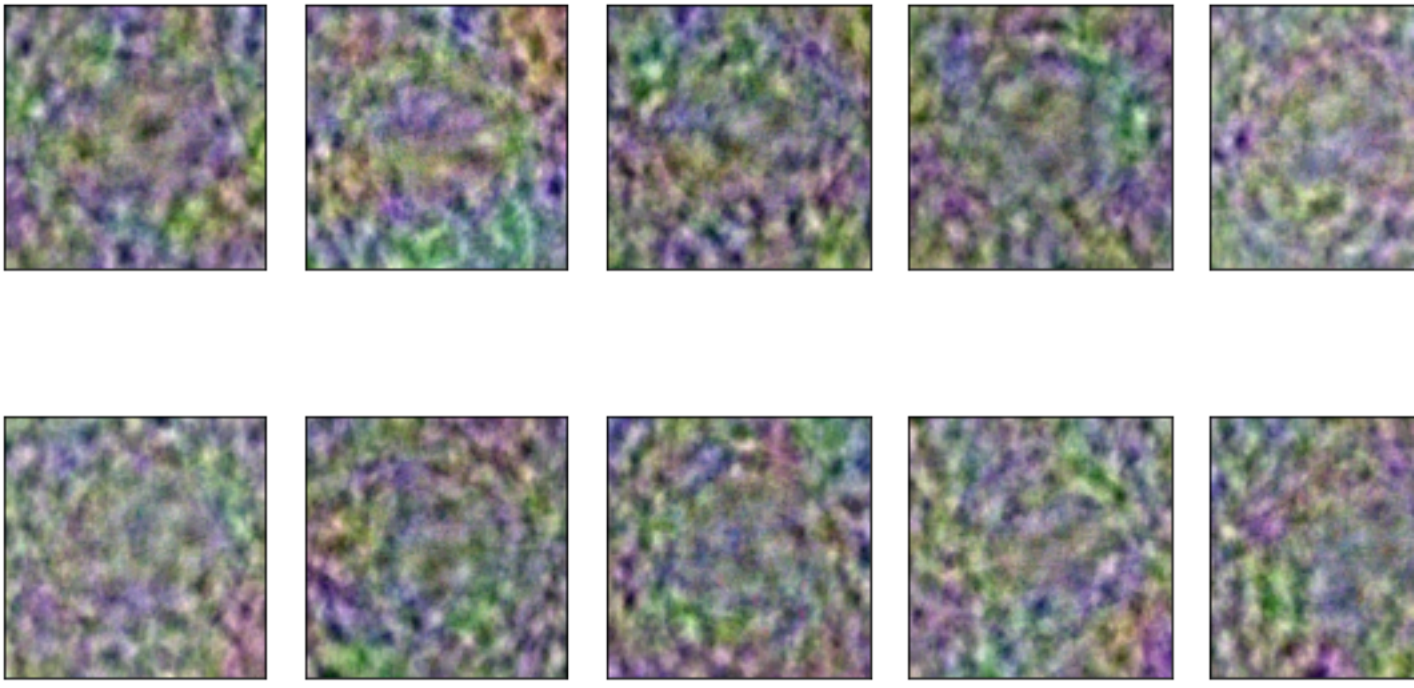

Figure 8: **Figure.** Layer 200-210 of PCA.

Come prevedibile le direzioni principali con meno varianza, se graficate, corrispondono a quasi puro rumore (si veda il contrasto tra Fig. M e Fig. M+1)

Di seguito viene visualizzata l'analisi dell'explained variance dove, utilizzando il criterio di conservazione del 90% della varianza, otteniamo 429 componenti.

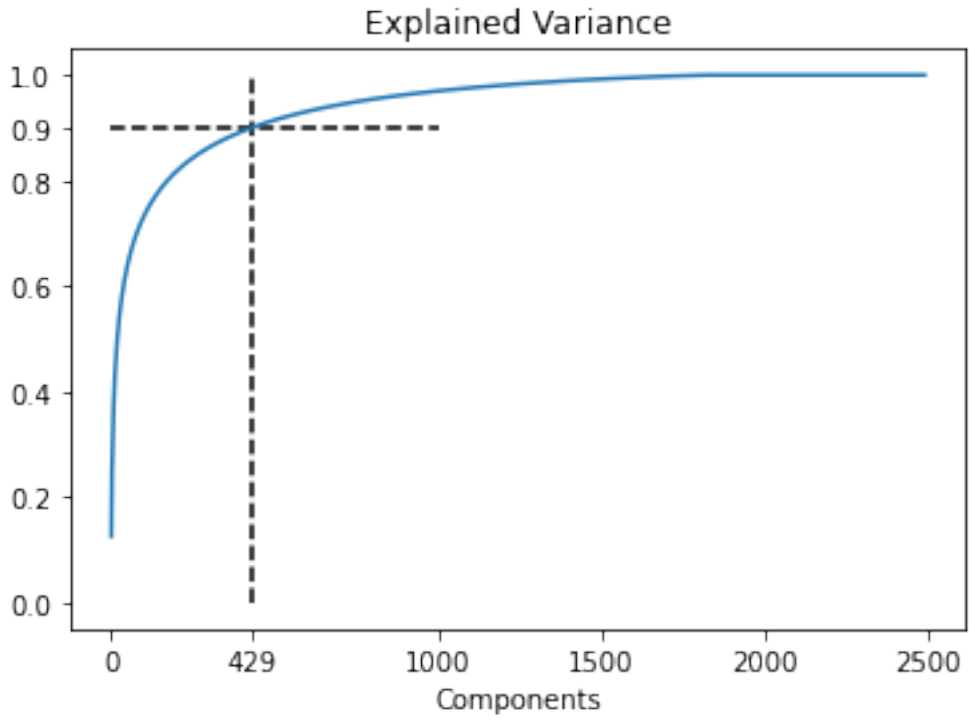

Figure 9: **Figure.** Layer 200-210 of PCA.

Convolutional Filters and Features Maps

References

1. Plant Pathology 2020 - FGVC7: Identify the category of foliar diseases in apple trees, *Kaggle* (2020).
2. Ranjita Thapa et al. The Plant Pathology 2020 challenge dataset to classify foliar disease of apples, *arXiv pre-print* (2020).
3. Gao Huang et al. Densely Connected Convolutional Networks, *arXiv pre-print* (2018).

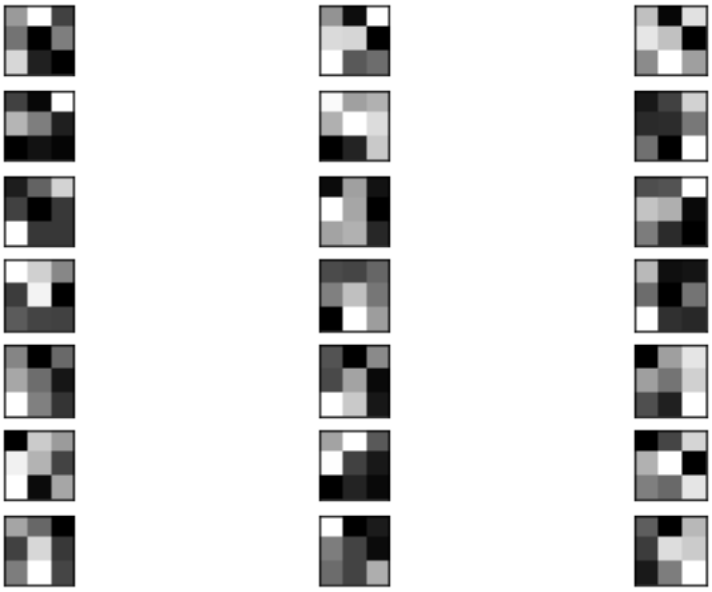

Figure 10: **FIGURE.** Three channels with some of the first 3x3 filters of the ‘EKM‘.