Dataset Analysis and CNN Models Optimization for Plant Disease Classification

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

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

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

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 0.937 applying EKM, a relatively shallow CNN defined and trained from scratch, and 0.972 applying the pre-trained Keras model DenseNet121, a CNN whose main feature relies on the connection between layers that are non contiguous (i.e. the output of the first layer is not only of the input of the second layer but also the third, fourth, etc.), which allows for feature reutilization that ultimately improves performance.

1. PROBLEM

Misdiagnosis of the many diseases impacting agricultural crops can lead to misuse of chemicals leading to the emergence of resistant pathogen strains, increased input costs, and more outbreaks with significant economic loss and environmental impacts. Current disease diagnosis based on human scouting is time-consuming and expensive, and although computer-vision based models have the promise to increase efficiency, the great variance in symptoms due to age of infected tissues, genetic variations, and light conditions within trees decreases the accuracy of detection.

2. 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 ($h = 516, m = 91, r = 622, s = 592$).

Even if we have ultimately decided not to apply the PCA to reduce the dimensionality of the dataset, we believe it might be interesting to visualize the first ten principal directions and qualitatively compare them with a sequence of principal directions with lower retained variance. As we can appreciate in the figures reported in the Appendix, the principal components with lower retained variance correspond to almost pure noise and from the retained variance assesment (using the criterion of 90% variance retention) we have obtained 429 components.

Here instead 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 verify the amplification of the classes performed by SMOTE and recognize the clustering of the generated points.

2.1 Methods

2.1.1 *Class Balancing with* SMOTE

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

1. (one of) the minority class(es) is considered ;
2. a point is randomly chosen and its first n_neighbors nearest neighbors are found ;

[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.

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 tested too:

- SVMSMOTE starts by fitting an SVM on the data, identifies the points which are more prone to mis-classification (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.

We have obtained the best performance applying baseline SMOTE with some fine-tuning on the sampling_strategy [2] and the n_neighbors parameters. For more details see Platform Limitations.

2.1.2 *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. [3]

2.1.3 *Model Architecture Exploration*

We have implemented an extensive exploration of all models and here is reported the grid search that achieved the best performance:

1. some exploration and fine-tuning of the layers and parameters of the models (i.p. a dropout layer for the EKM only) ;
2. variations of the optimizer (for the EKM only) ;
3. optimal dropout and epoch number search ;
4. checkpointing .

We couldn't implement early stopping both in EKM and DenseNet121, since the fluctuations in either validation loss, categorical accuracy or mean column-wise ROC AUC where too high to properly set the min_delta and patience parameters in the TensorFlow implementation.

The best choice of the optimizer for the EKM proved to be the RMSprop, while the standard adam performed pretty well with the DenseNet121. 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 highest mean column-wise ROC AUC were saved and used during the testing phase.

Then, in order to establish the quantitative impact of stocasticity in the initialization of the weights on EKM, an EKM1 with the best drop is trained and validated, and the best epochs of EKM and EKM1 are compared. There was a small difference, therefore we decided to make three submissions: one with the baseline EKM re-trained on all the data and with the best drop, one with EKM1 and one with the DenseNet121.

Besides fluctuations, we have noticed that the DenseNet121 tends to reach higher submission scores. See Training Histories.

2.1.4 *Convolutional Autoencoder*

The best we could achieve by inserting a covolutional autoencoder between the smoted data, augmented data and the model training is a 0.7 mean column-wise ROC AUC, despite the large number of the configurations that have been tried. The reason behind this relatively poor performance could be that on the one hand an autoencoder with no pooling on the encoder side makes little sense in terms of dimensionality reduction, while on the other hand even a single bidimensonal maxpooling caused the output image to be too little for the last EKM layer to classify. See Platform limitations and Model Architecture Exploration.

The only way way we have managed to run it and see at least some loss drop was to build a very shallow autoencoder (i.e. just a couple of layers besides the input and the output), with the result that the loss didn't decrease significantly. Anyway, inspired by the work of others and by some active trial and error, we have had a chance to collect some architectural criteria to build a convolutional autoencoder that at least exhibits learning. The following is to be intended as an empirical recipe, with no or little theoretical foundation supporting the choice of its ingredients.

The autoencoder is composed of an encoder and a decoder. Obviously the encoder should start with an input layer, followed by some blocks of Conv2D and Pooling layers. 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 and Pooling by UpSampling. Then the last two layers of the decoder should be a BatchNormalization layer and Conv2DTranspose with 3 filters (in order to be able to compare output with input) activated by a sigmoid (which explains the BatchNormalization layer). The unknown number of Conv2D-Pooling blocks in the encoder (that determines the number of Conv2DTranspose-UpSampling in the decoder) has to be jointly connected with the number of Conv2D-Pooling layers of the network. See Model Architecture.

2.1.5 *Selected Model Architecture*

Some online resaerch and active trial and error with the network architecture gave us some clues about how to build

[2] The value all means that all classes are resampled to match the size of the majority class.
[3] For further information read the Keras image preprocessing API.

from scratch an effective, dataset-dependent model for image classification.

Obviously the network should start with a input layer, followed by blocks of Conv2D-Pooling[4] 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. Then this should be 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 BatchNormalization layer.

2.2 Results

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

2.3 Conclusions

Since the optimal epoch number varies with the size of training dataset, a possible third attempt to obtain it would have seen the best epoch number to use in the testing phase, when the model is re-trained on all training data, extrapolated from a (best-epoch, training-set-size) plot (given that stocasticity has not been relevant).

This has been practically impossible for us because of two main reasons: technical difficulty in combining Sci-kit learning curves with a Keras model necessarily trained with generators, and platform limitations. Those limitations prevented us from instantiating a single Pipeline object integrating all the elements (SMOTE, ImageDataGenerator, Model): this could have allowed us to perform a more extensive and reliable[5] grid search. Finally, as we have already mentioned, those computational limitations prevented us from implementing an effective convolutional autoencoder: if we used the full-sized images, the autoencoder may have been deeper and that could have plausibly yielded a better performance.

2.4 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).

2.5 Appendix

2.5.1 *Related Material*

- Explore the GitHub repository of the project.
- Read the code in the Jupyter notebook.
- Run the code in the Kaggle notebook.

2.5.2 *Platform Limitations*

Since the last unstable version of GPU-supported TensorFlow is required to run the code and we haven't been able to set the proper kernels up on our local machines, we have been constrained to rely on a publicly available cloud interactive environment like Kaggle, which 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 $\times$ 200 pixels.

[4] MaxPooling in our case.

[5] If coupled with cross validation instead of 80%-20% splitting.

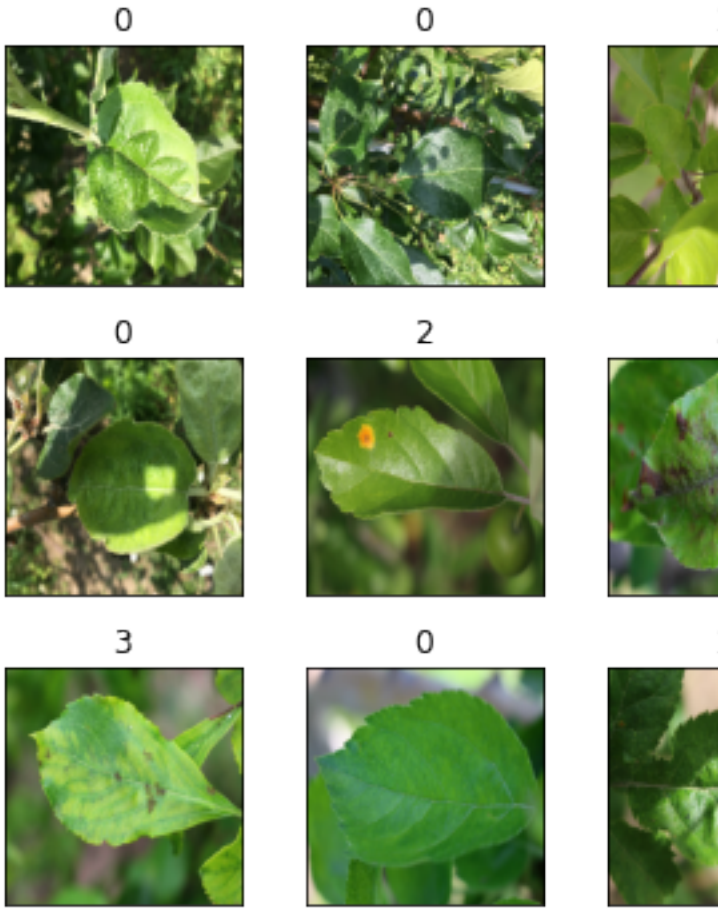


Figure 1: Sample of training images.

2.5.3 *Visualization*

PCA

Training Histories

Filters and Activation Maps

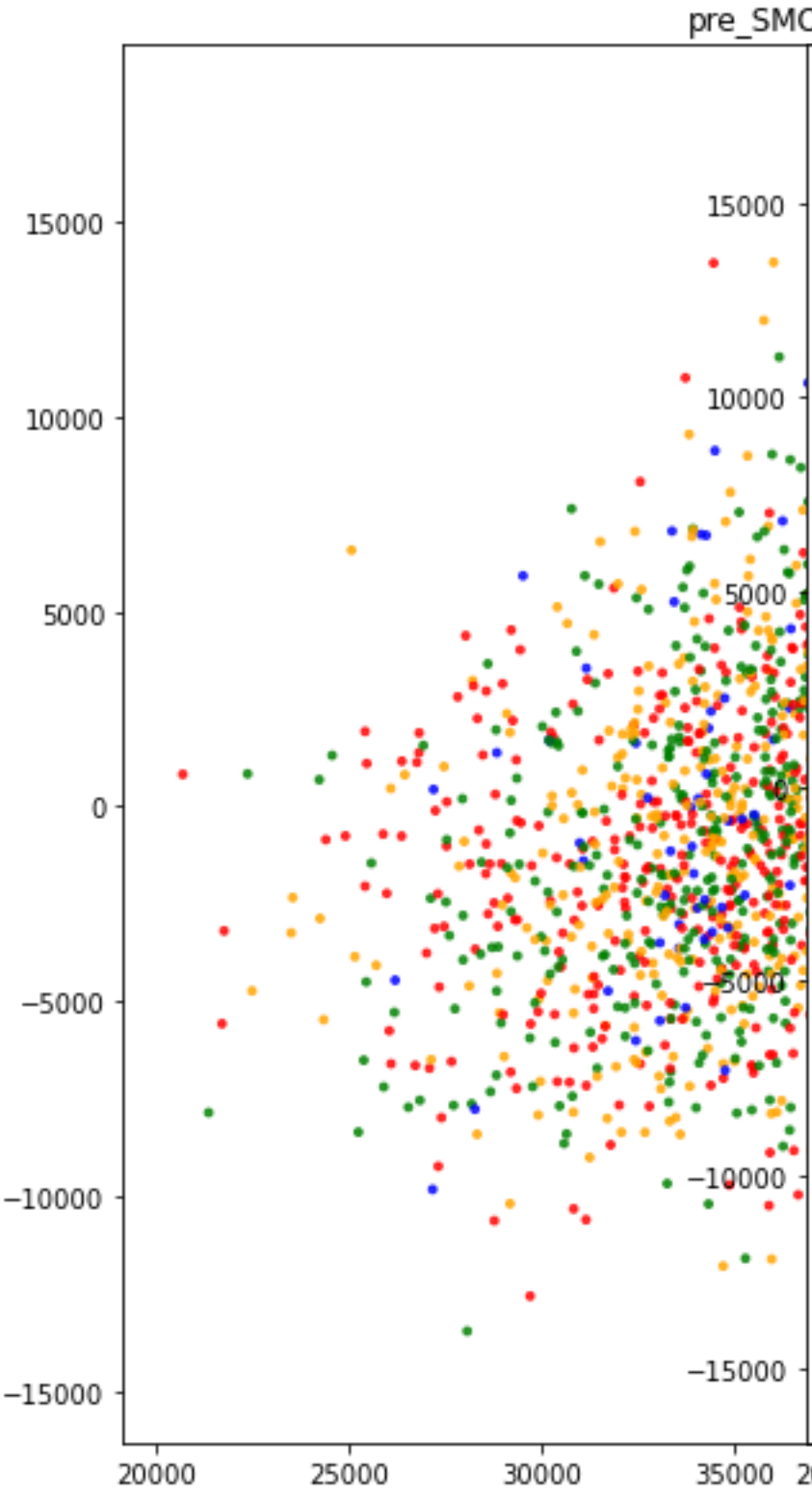


Figure 2: Pre-‘SMOTE‘ truncated SVD.

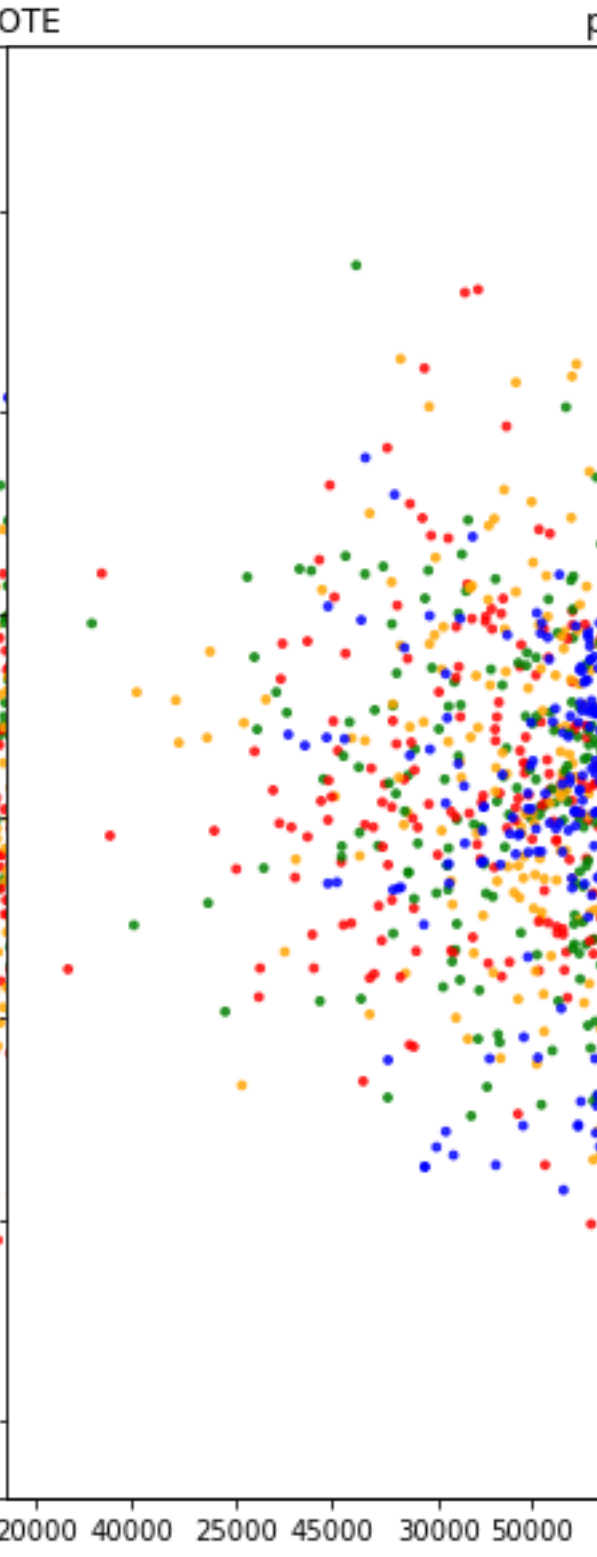
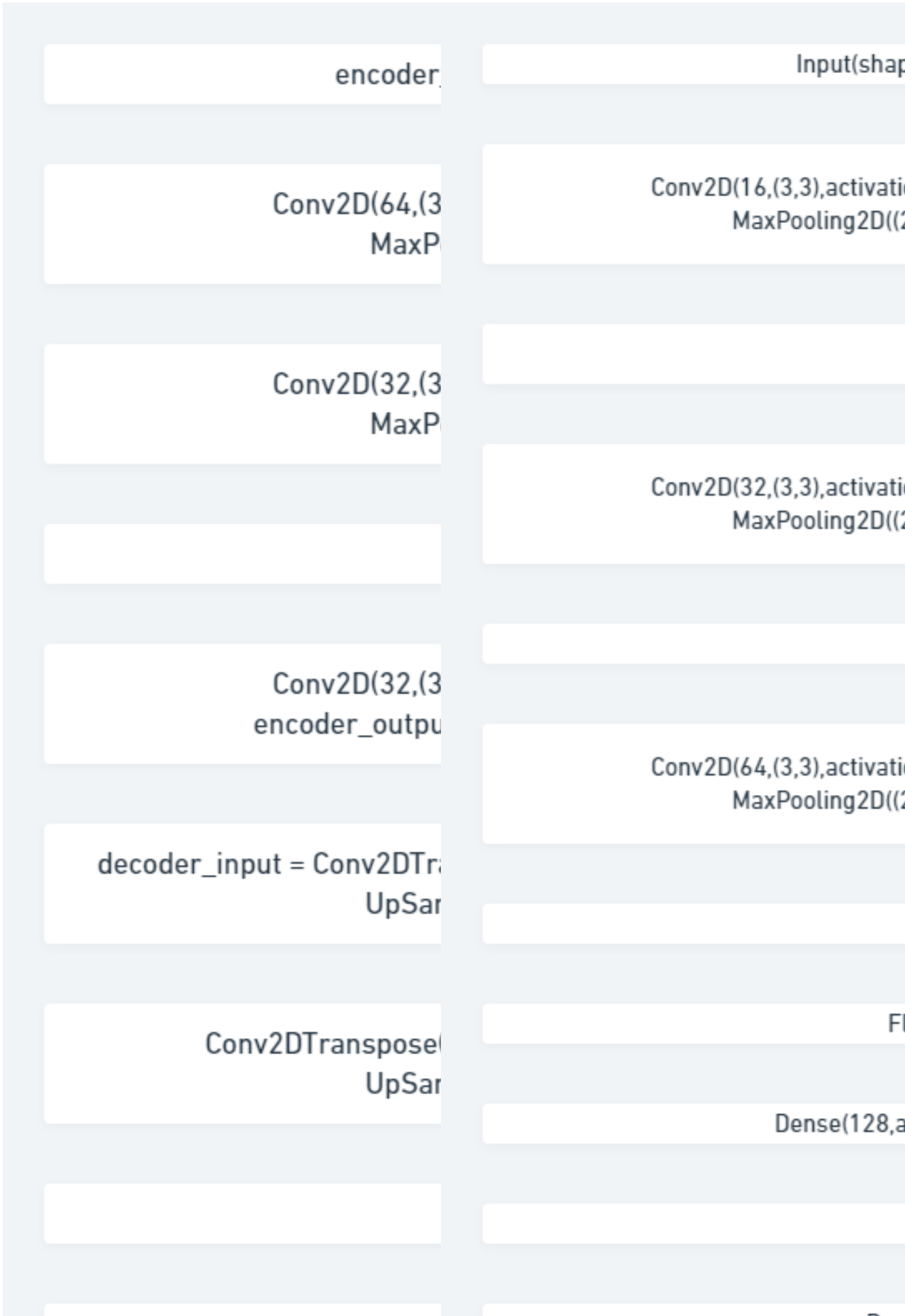


Figure 3: Post-‘SMOTE‘ truncated SVD.


encoder
Input(shap
Conv2D(64,(3
MaxP
Conv2D(16,(3,3),activati
MaxPooling2D((2
Conv2D(32,(3
MaxP
Conv2D(32,(3,3),activati
MaxPooling2D((2
Conv2D(32,(3
encoder_outpu
Conv2D(64,(3,3),activati
MaxPooling2D((2
decoder_input = Conv2DTr
UpSar
Fl
Conv2DTranspose(
UpSar
Dense(128,a

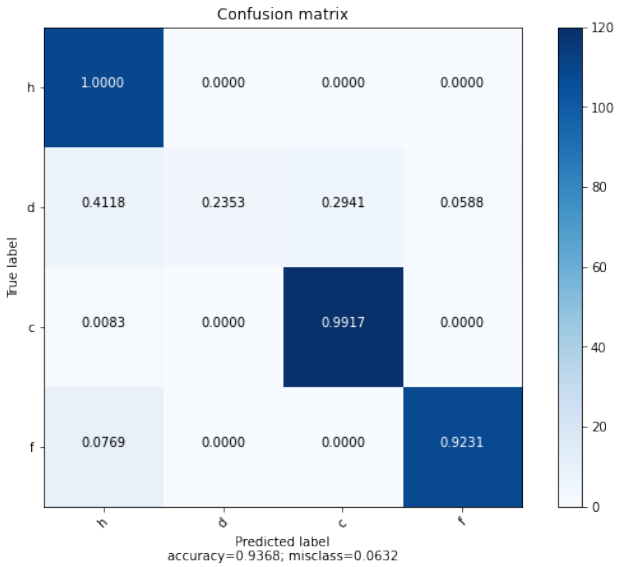


Figure 6: **FIGURE 6.** Confusion matrices of 'EKM' (left) and 'DenseNet121' (right).

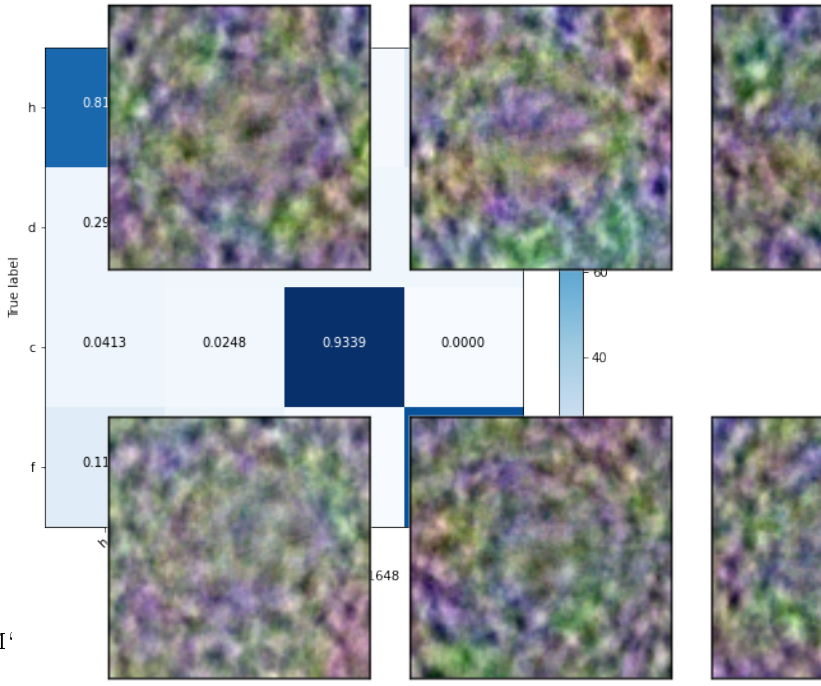


Figure 8: **FIGURE 8.** Directions 200-210 of the PCA.

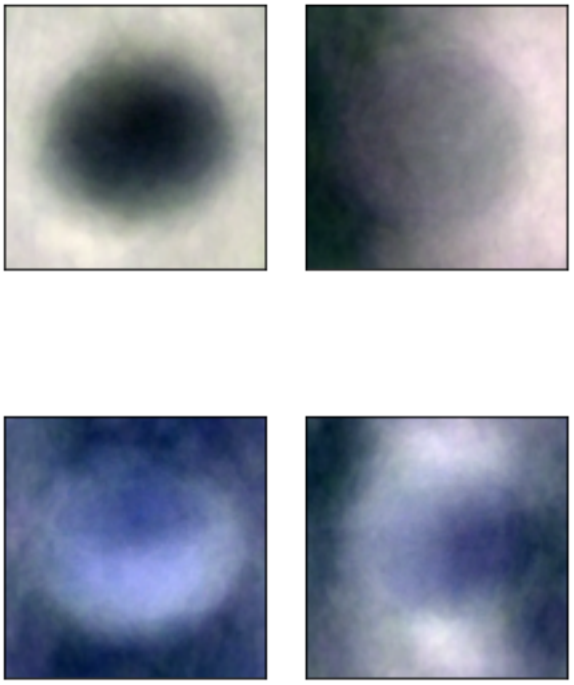


Figure 7: **FIGURE 7.** The first ten principal directions of the PCA.

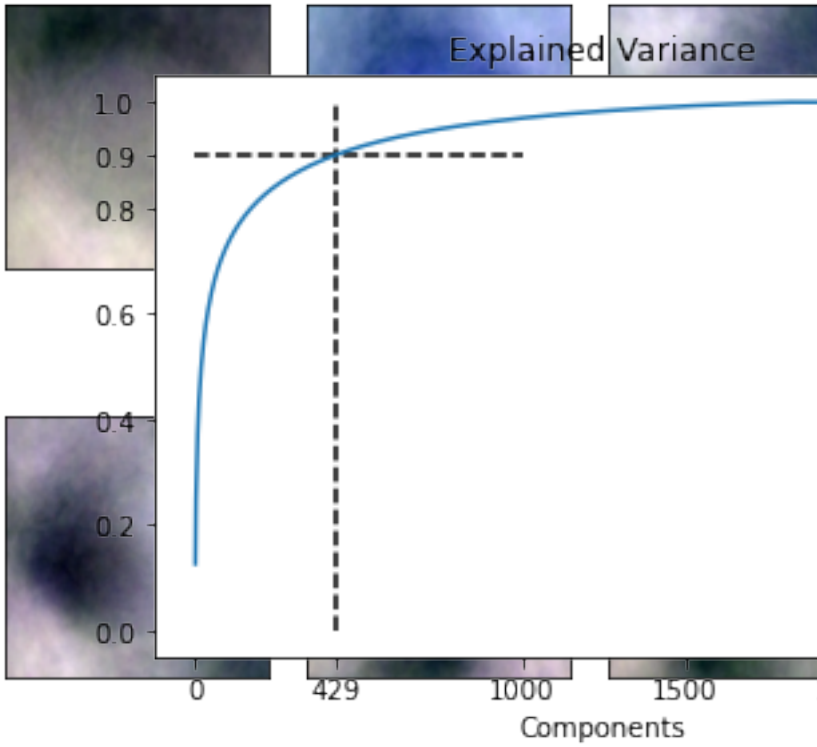
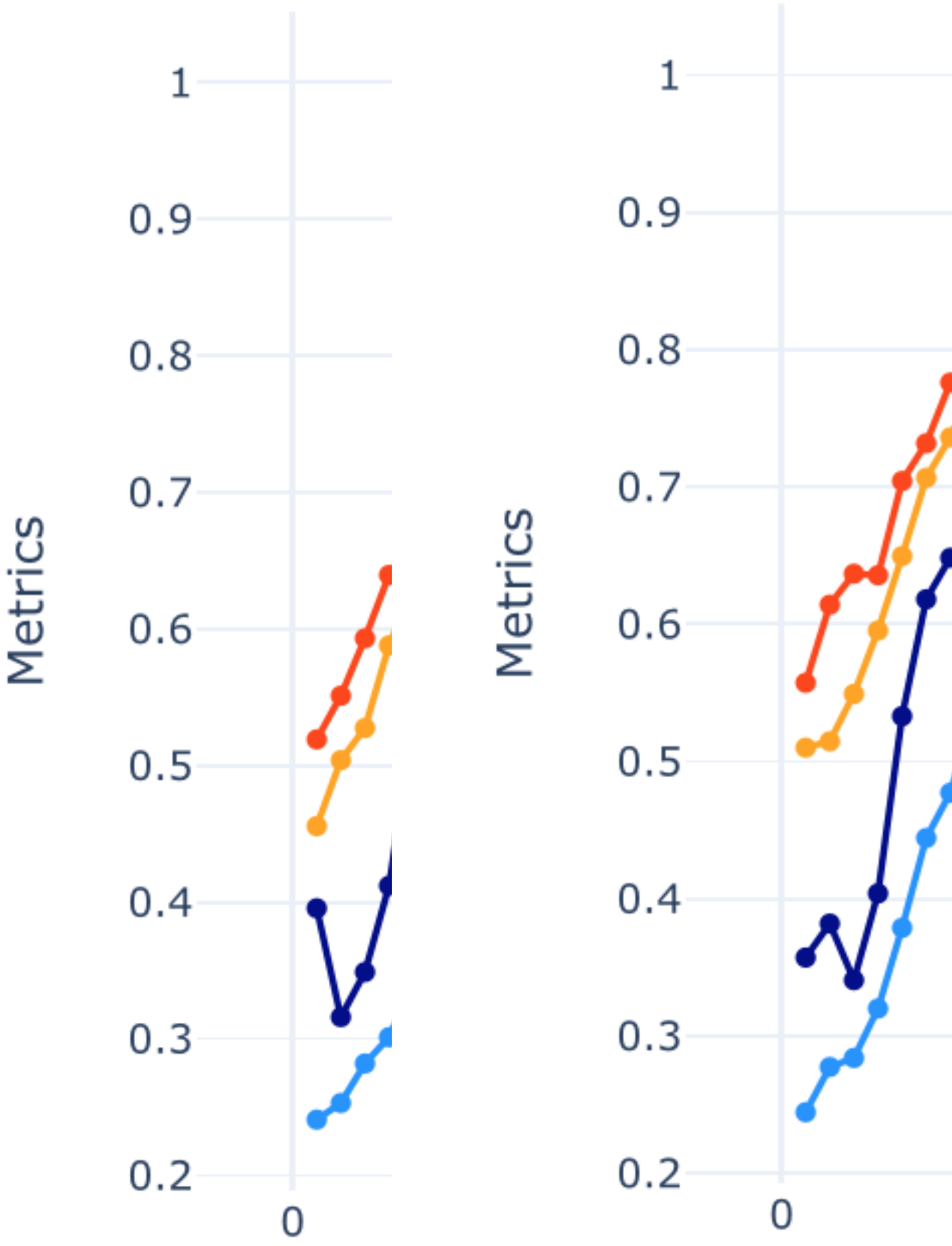


Figure 9: **FIGURE 9.** Explained Variance.


Metrics
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
Metrics
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0

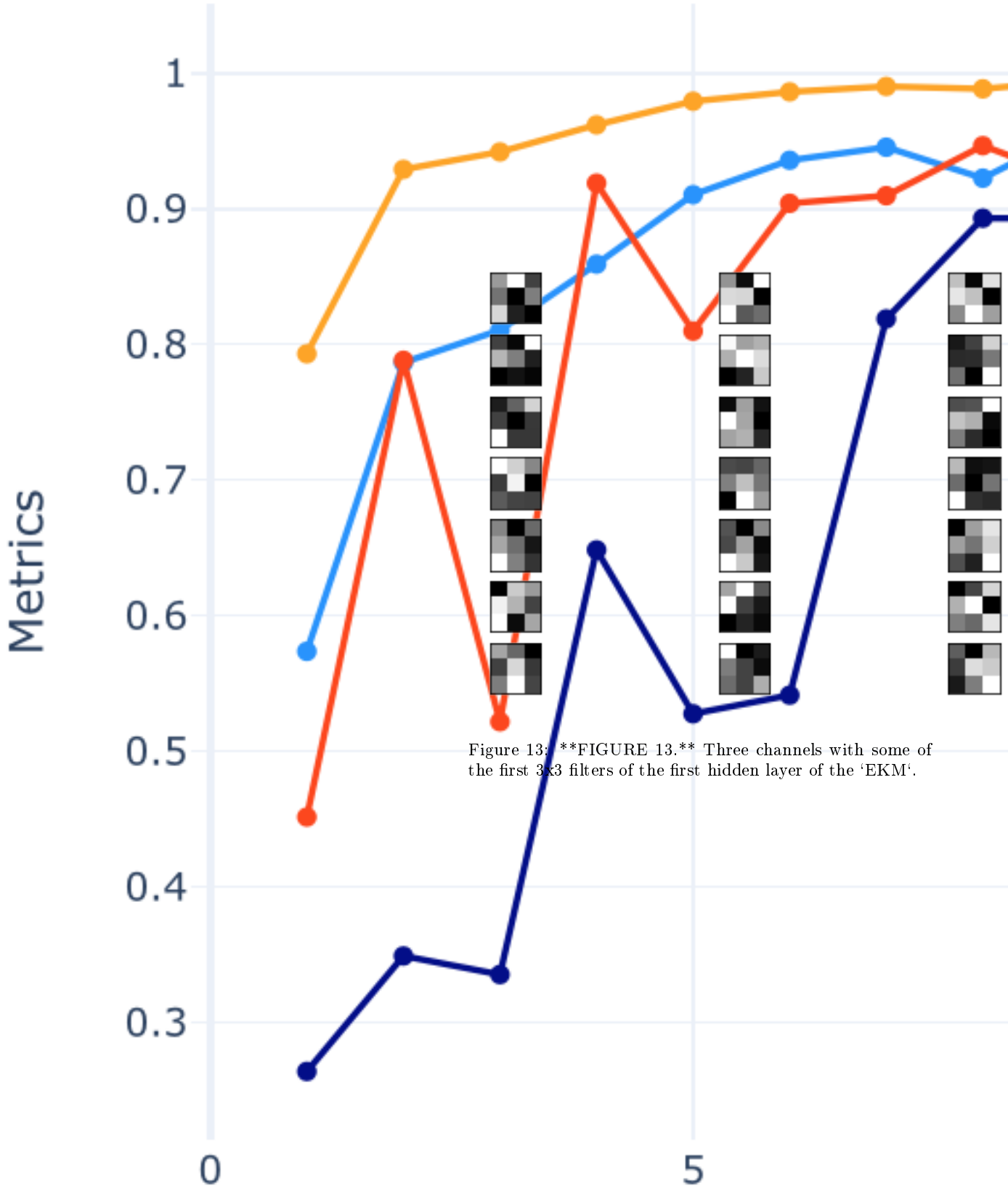


Figure 13: **FIGURE 13.** Three channels with some of the first 3x3 filters of the first hidden layer of the 'EKM'.

Figure 12: **FIGURE 12.** Training history of the

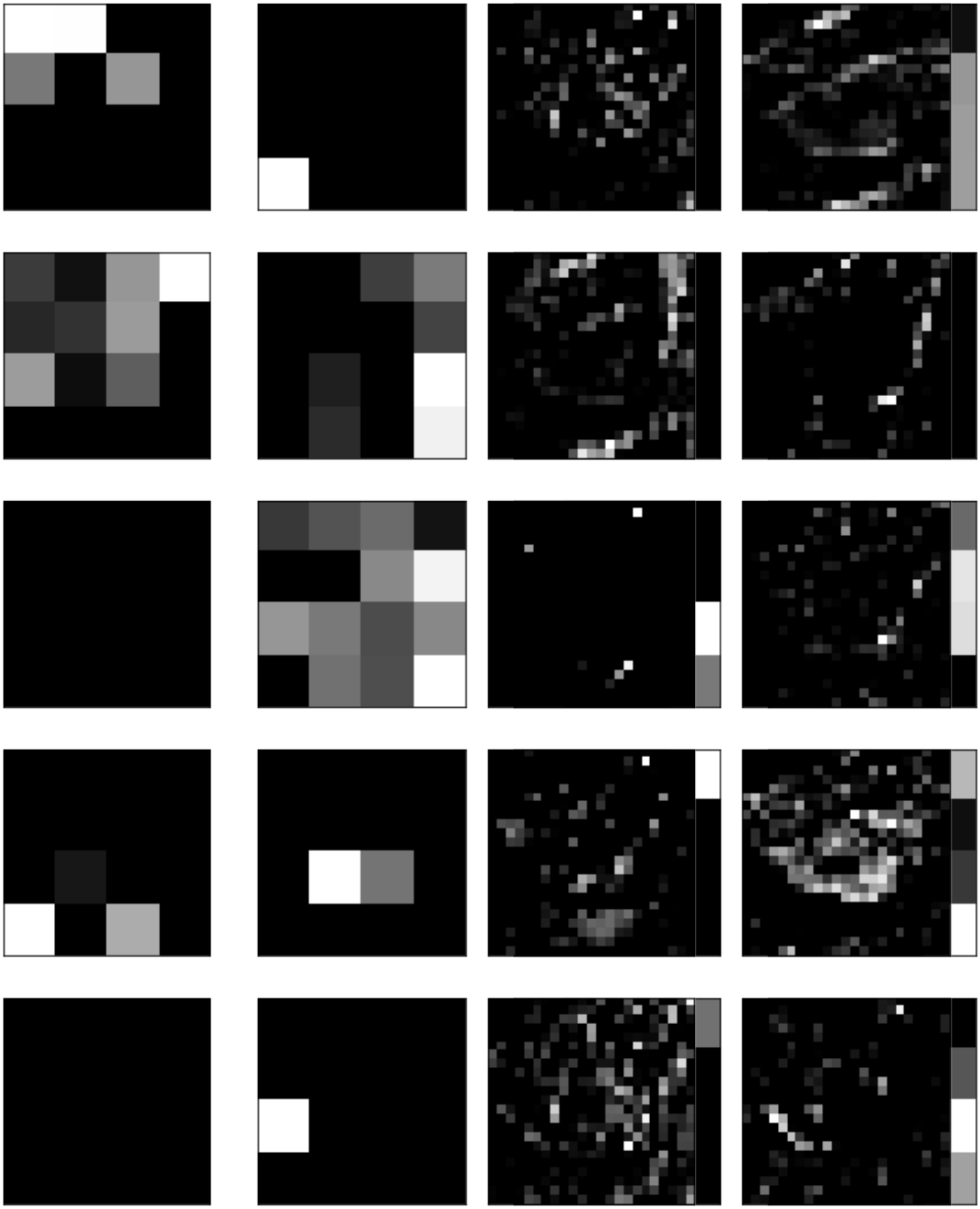


Figure 14: **FIGURE 14.** Activation maps of the second hidden layer of 'EKM'.

Figure 15: **FIGURE 15.** Activation maps of the sixth hidden layer of 'EKM'.

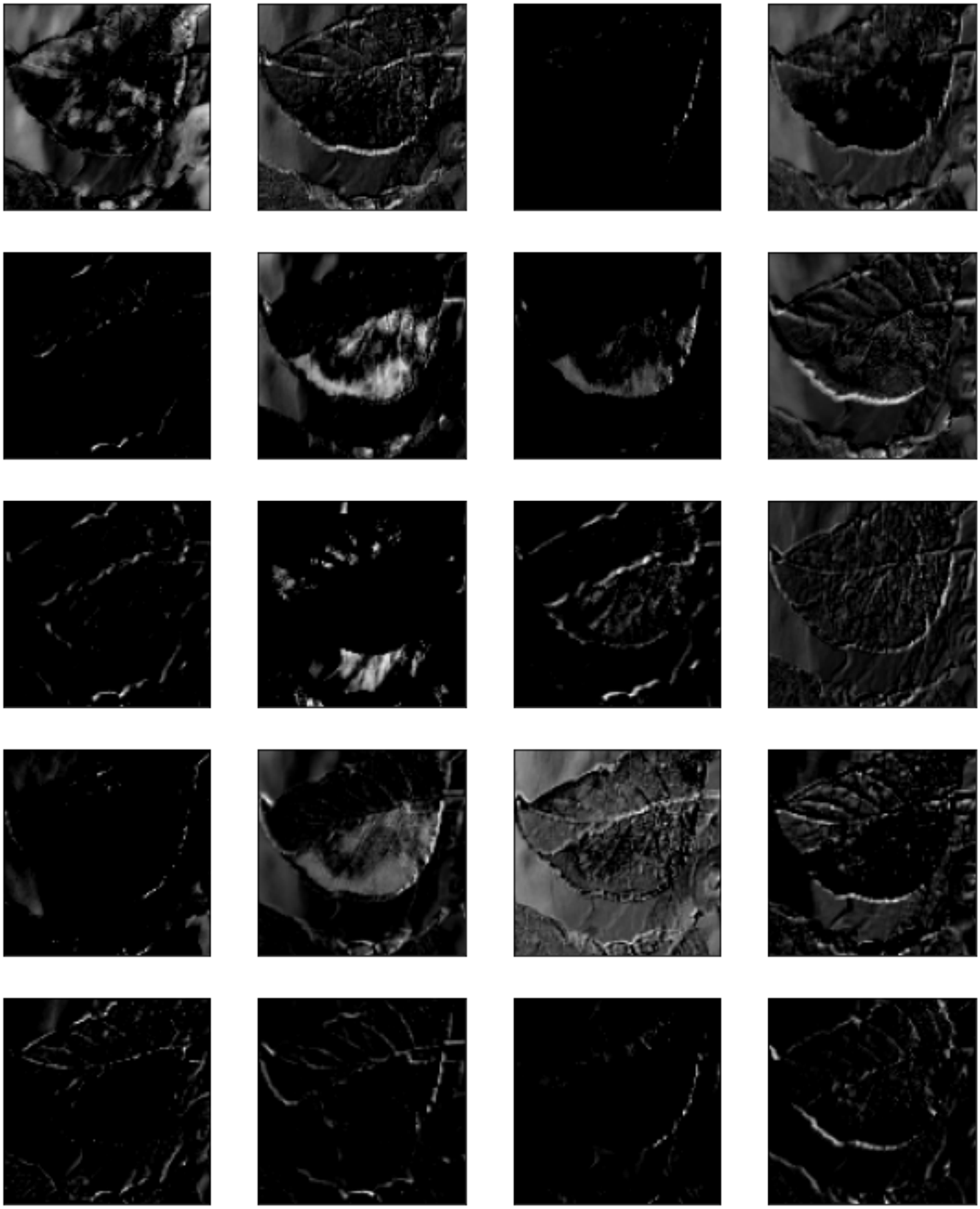


Figure 16: **FIGURE 13.** Activation maps of the ninth hidden layer of 'EKM'.