Maksym Maiboroda

Pet project: Crack wall detection by unsupervised learning.

Why Unsupervised Learning?

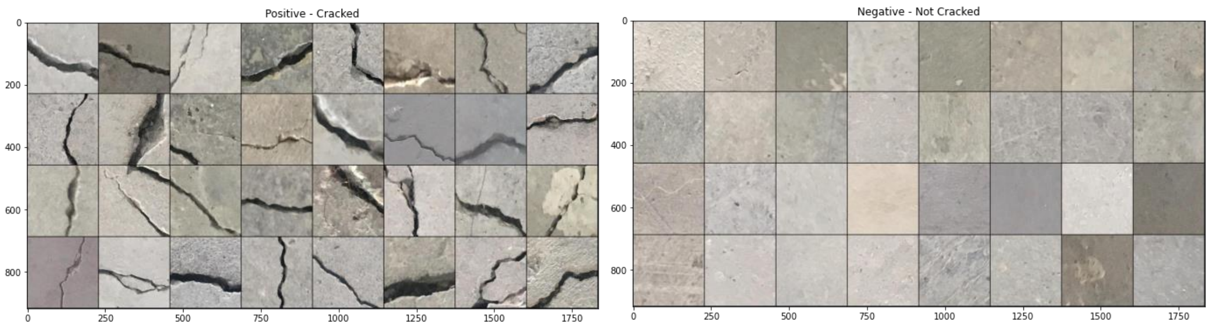
It is well known that in order to create a neural network with high performance, it is necessary to have a huge amount of data. The supervised learning approach requires high-quality data labeling, which significantly increases the cost of development. But more importantly, the supervised learning approach is poorly applicable to poorly known patterns of behavior. For example, in security systems, when it is necessary to detect fraud.

Formulation of the problem.

In this project, the dataset was chosen "Concrete Crack Images for Classification" and attempts have been made to classify images using unsupervised learning methods.

1. Exploratory data analysis.

The first step was to visualize the data. As we can see, the dataset consists of two classes. Positive class – with cracked and Negative – without cracks fig. 1.



(a) (b)

Fig. 1. Instances of dataset classes: a – Positive, b – Negative.

As visible to our eyes, the classes are clearly distinguishable among themselves. Which suggests to check the histograms of images and possible further classification by the histogram threshold value. We see that the histograms have a distribution close to Gaussian and offset from each other fig. 2.

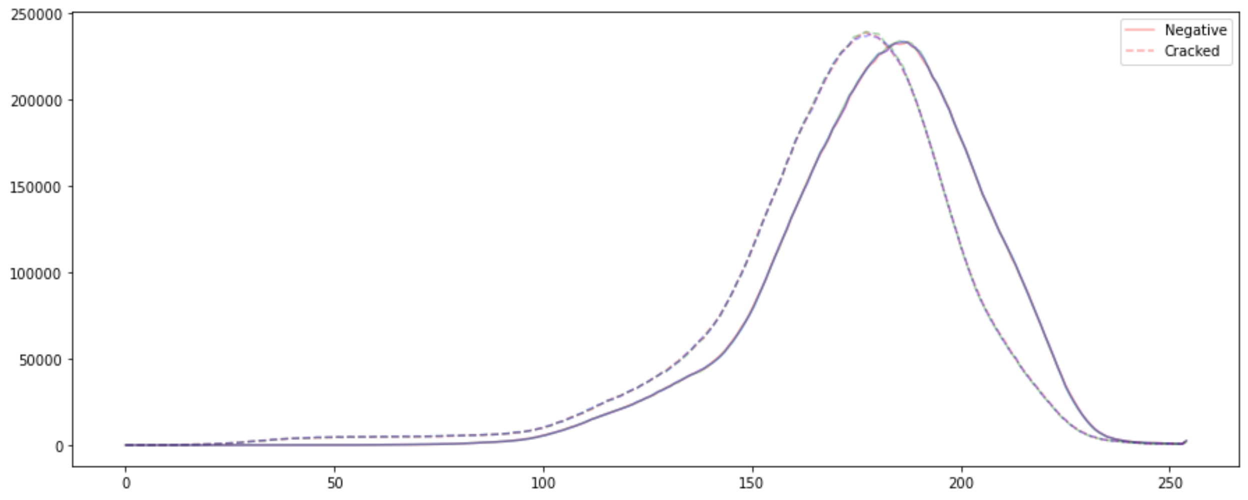
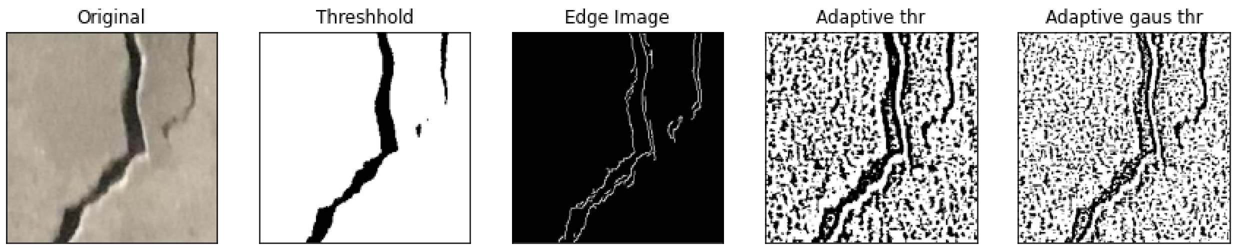
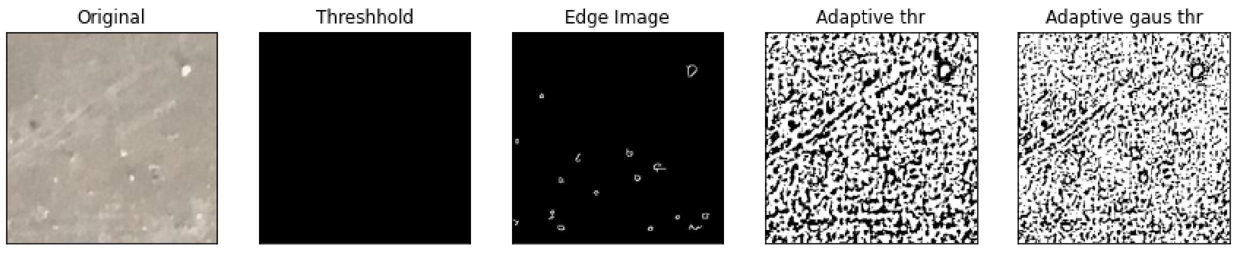


Fig. 2 Histogram distribution for each class on the entire dataset.

As possible features, we considered representation of the image in a black-and-white mask, that was found by the threshold, adaptive threshold, adaptive threshold with gaussian-weighted sum of the neighborhood values and image edges founded with Canny edge detector algorithm fig. 3.



(a)



(b)

Fig. 3 Examples of possible features for Positive (a) and Negative (b) data

Another widely used method for feature extraction is transfer learning. The idea of this approach is based on the use of a previously trained neural network model with a removed classifier part. A neural network with the architecture of

Resnet18 pretrained on the imagenet dataset was chosen since this model has a small size and pretty good performance.

At the output of Resnet18, we get a 512-dimensional vector. To visualize it we use the dimensionality reduction method called principal component analysis (PCA) fig. 4 - 5.

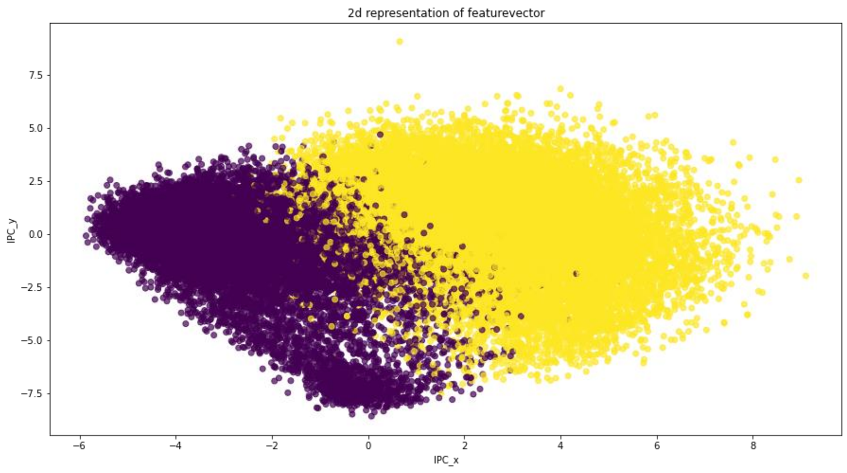


Fig. 4 Representation of the obtained 512 dimensional vectors for each image in the dataset on a 2D plane

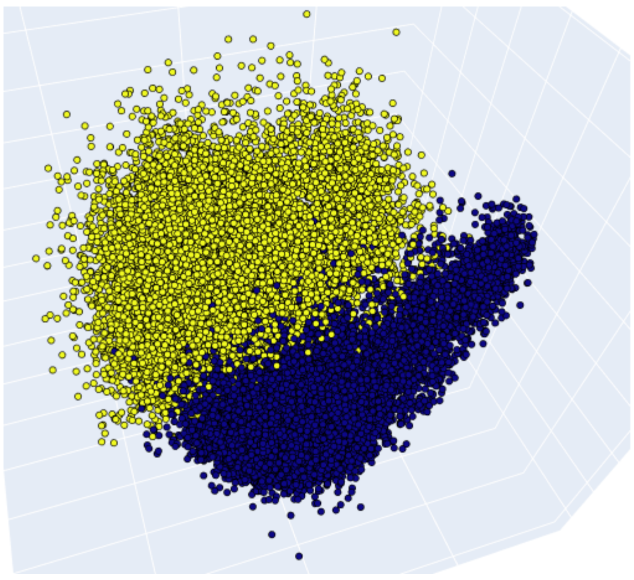


Fig. 5 Representation of the obtained 512 dimensional vectors for each image in the dataset on a 3D plane

After PCA 2 blobs are clearly visible so with a high chance of success, we can use the classical methods of statistical data processing such as clustering.

2. Base pipeline

2.1 K-means method.

In the first approach, the K-means method was used. K-means is a centroid-based clustering algorithm, where we calculate the distance between each data point and a centroid to assign it to a cluster. The result of this calculation is shown on fig. 6-7.

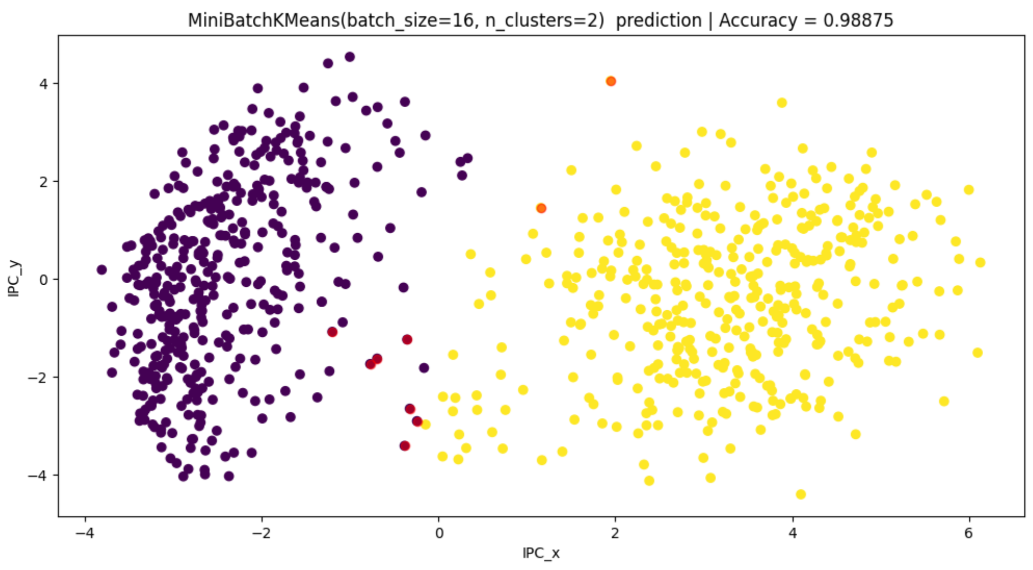


Fig. 6 Data clustering result. Green dots – centroids of the clusters, red dots – misclassified data (without input normalization). Accuracy = 98.875%

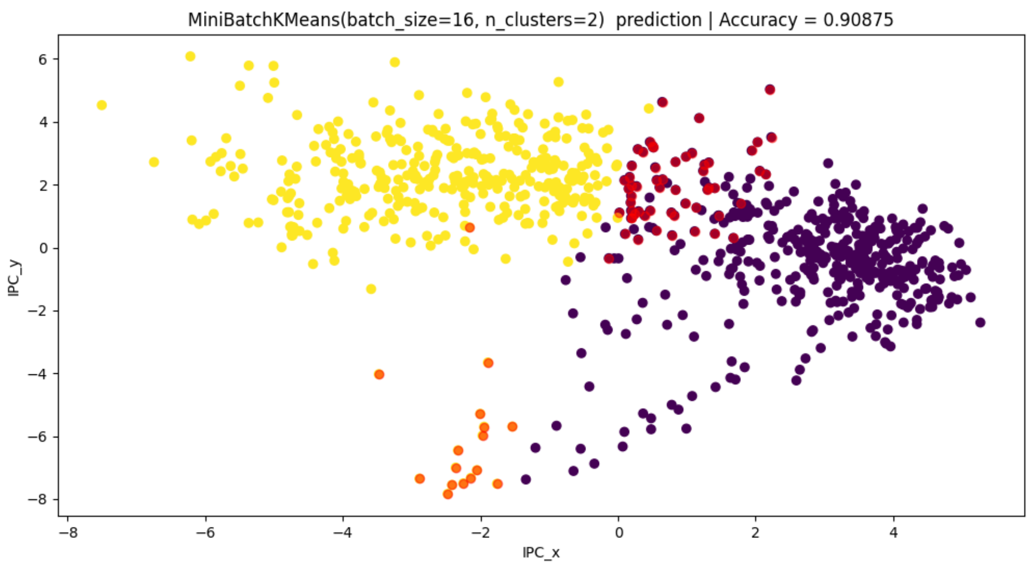


Fig. 7 Data clustering result. Red dots – misclassified data (with input normalization). Accuracy = 90.875%

2.2 BIRCH method

It is a clustering algorithm that can cluster large datasets by first generating a small and compact summary of the large dataset that retains as much information as possible. This smaller summary is then clustered instead of clustering the larger dataset. The result of test set clustering is shown in fig. 8 and fig. 9.

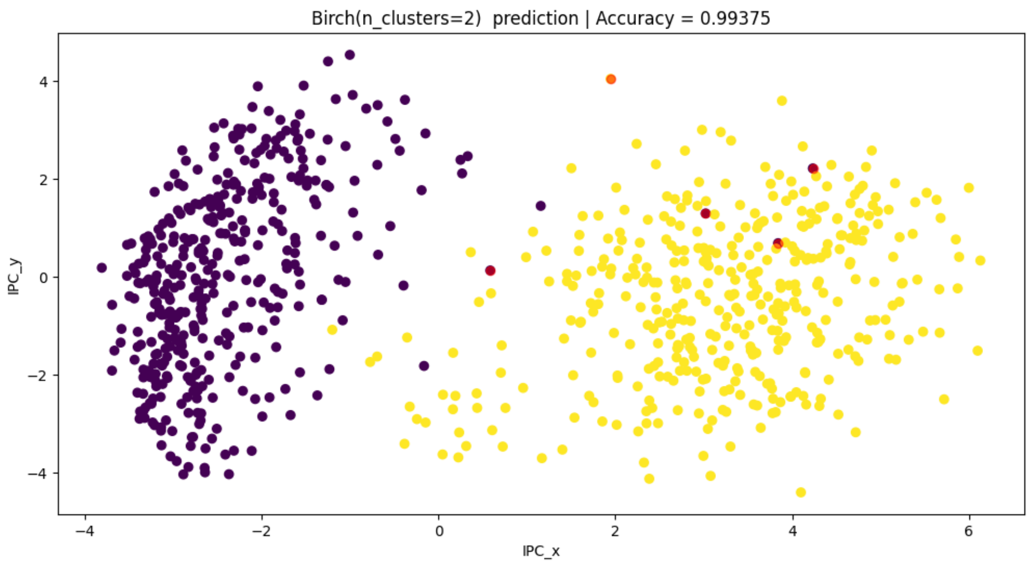


Fig. 8 Data clustering result. Red dots – misclassified data (without input normalization) Accuracy = 99.375%

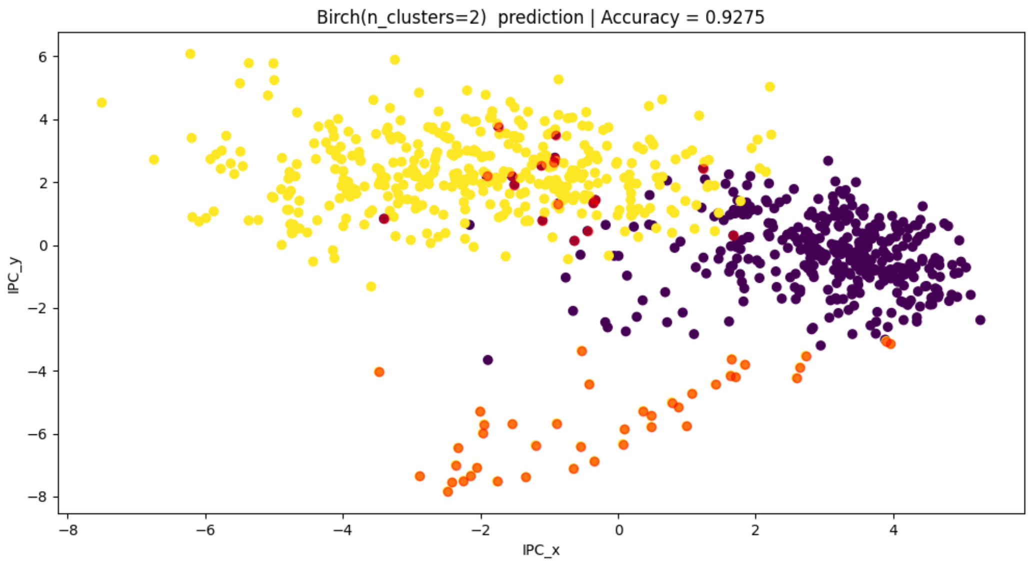


Fig. 9 Data clustering result. Red dots – misclassified data (with input normalization) Accuracy = 92.75%

2.3 Base pipeline conclusions.

After results analyze, as a baseline BIRCH was taken. In the case of balanced data, it showed a high accuracy equal 99.375%.

To analyze the algorithm's ability to work with unbalanced data, was made experiment with a different number of images in one of the classes fig. 10.

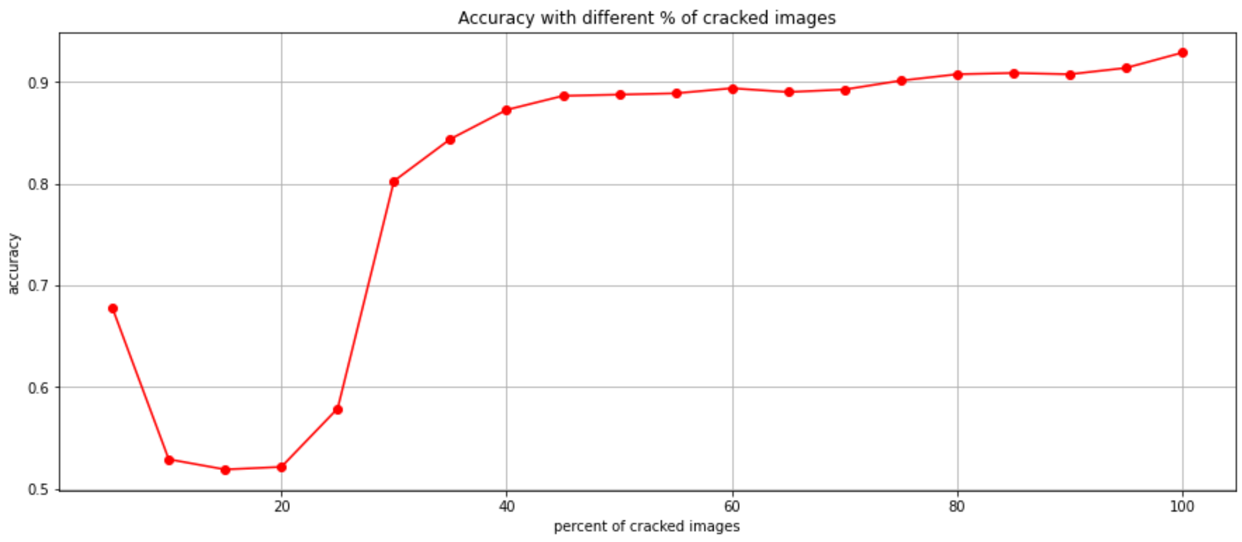


Fig. 10 Accuracy dependencies from number of images in class.

In this experiment, we showed the dependence of the accuracy of the model on the balance of classes. The number of images in the second class increased by 5% of images number in the first class. As we can see, if we need more than 90% of accuracy, the number of images of the second class should be more than 70% of the first class (In simple words, if we have 1000 instances in one class, then in the second one we need to have more than 700).

3. Unsupervised anomaly detection using autoencoder.

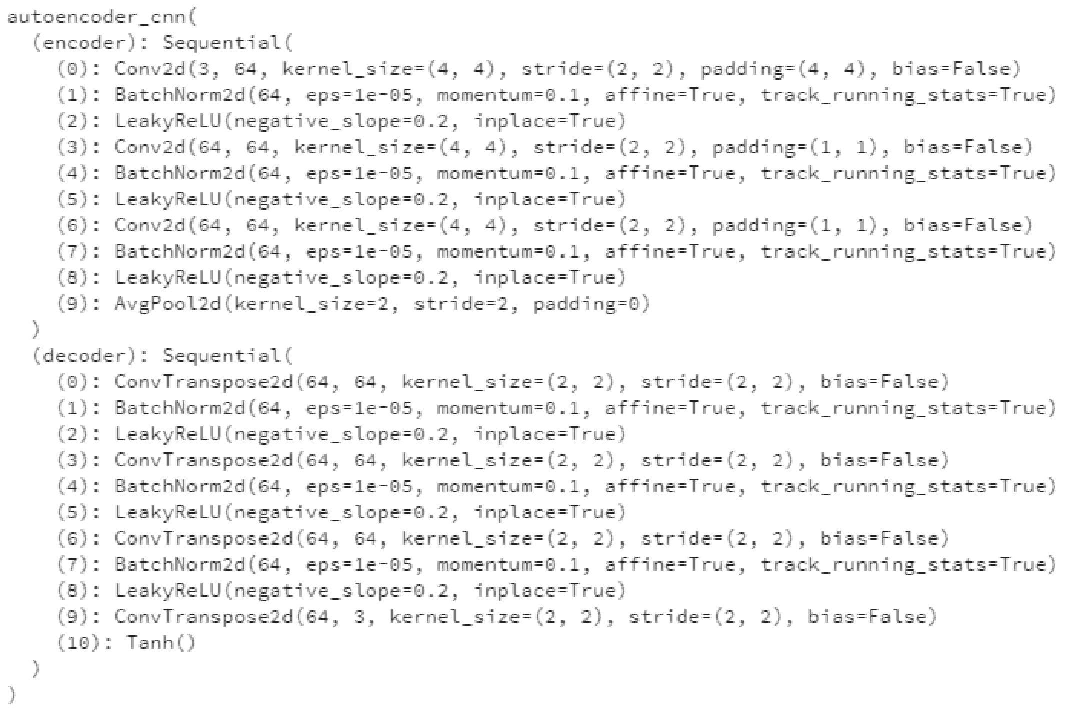
Previous subsection shows that clustering with statistical method is unsuitable for imbalanced data. And in order to fill this gap, a different approach was implemented. It is based on anomaly detection using an autoencoder.

The autoencoder principle is based on the fact that there is a bottleneck in the network architecture (latent space) that will contain general information about the input image. And the second part of the network will reconstruct the image according to the latent space. The idea is to train the autoencoder to reconstruct images. It is expected that with a strong dominance of one class, the autoencoder will learn to restore this class image with less loss. By setting the loss threshold, we can expect the separation of images with and without cracks in our case.

3.1 Dataset and network architecture preparation

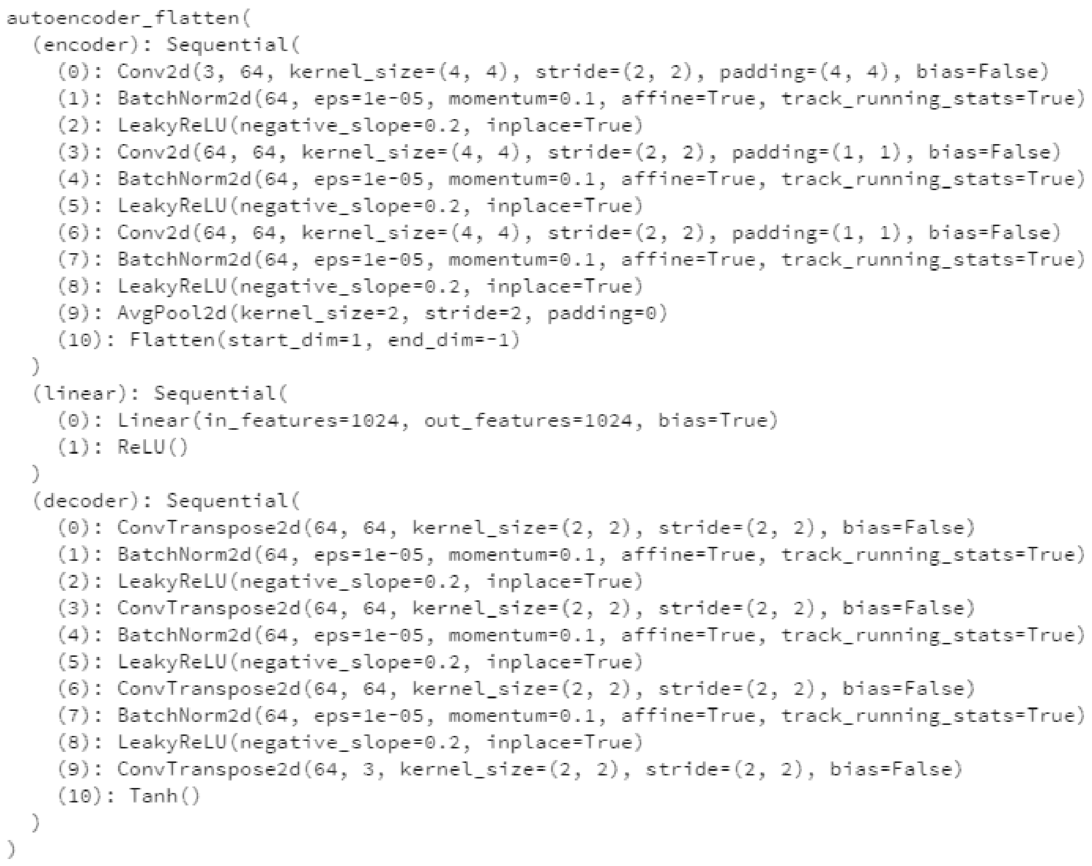
Due to hardware limitations, I had to resize the images to 64*64, and splitted to training, validation and test sets (60/20/20).

Two similar variants of the autoencoder were considered. The difference was that in one of them the latent space was a convolutional layer fig. 11, and in the second it was a fully connected layer fig. 12.



```
autoencoder_cnn(
  (encoder): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(4, 4), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace=True)
    (3): Conv2d(64, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): LeakyReLU(negative_slope=0.2, inplace=True)
    (6): Conv2d(64, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): LeakyReLU(negative_slope=0.2, inplace=True)
    (9): AvgPool2d(kernel_size=2, stride=2, padding=0)
  )
  (decoder): Sequential(
    (0): ConvTranspose2d(64, 64, kernel_size=(2, 2), stride=(2, 2), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace=True)
    (3): ConvTranspose2d(64, 64, kernel_size=(2, 2), stride=(2, 2), bias=False)
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): LeakyReLU(negative_slope=0.2, inplace=True)
    (6): ConvTranspose2d(64, 64, kernel_size=(2, 2), stride=(2, 2), bias=False)
    (7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): LeakyReLU(negative_slope=0.2, inplace=True)
    (9): ConvTranspose2d(64, 3, kernel_size=(2, 2), stride=(2, 2), bias=False)
    (10): Tanh()
  )
)
```

Fig. 11. Network architecture with a CNN latent vector



```
autoencoder_flatten(
  (encoder): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(4, 4), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace=True)
    (3): Conv2d(64, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): LeakyReLU(negative_slope=0.2, inplace=True)
    (6): Conv2d(64, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): LeakyReLU(negative_slope=0.2, inplace=True)
    (9): AvgPool2d(kernel_size=2, stride=2, padding=0)
    (10): Flatten(start_dim=1, end_dim=-1)
  )
  (linear): Sequential(
    (0): Linear(in_features=1024, out_features=1024, bias=True)
    (1): ReLU()
  )
  (decoder): Sequential(
    (0): ConvTranspose2d(64, 64, kernel_size=(2, 2), stride=(2, 2), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace=True)
    (3): ConvTranspose2d(64, 64, kernel_size=(2, 2), stride=(2, 2), bias=False)
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): LeakyReLU(negative_slope=0.2, inplace=True)
    (6): ConvTranspose2d(64, 64, kernel_size=(2, 2), stride=(2, 2), bias=False)
    (7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): LeakyReLU(negative_slope=0.2, inplace=True)
    (9): ConvTranspose2d(64, 3, kernel_size=(2, 2), stride=(2, 2), bias=False)
    (10): Tanh()
  )
)
```

Fig. 12. Network architecture with a fully connected latent vector

During training, the loss function was the root mean square error, the optimization algorithm was Adam. Training was for 100 epochs, with learning rate = 0.0001, which halved every 10 epochs.

Loss functions are shown on fig. 13-14

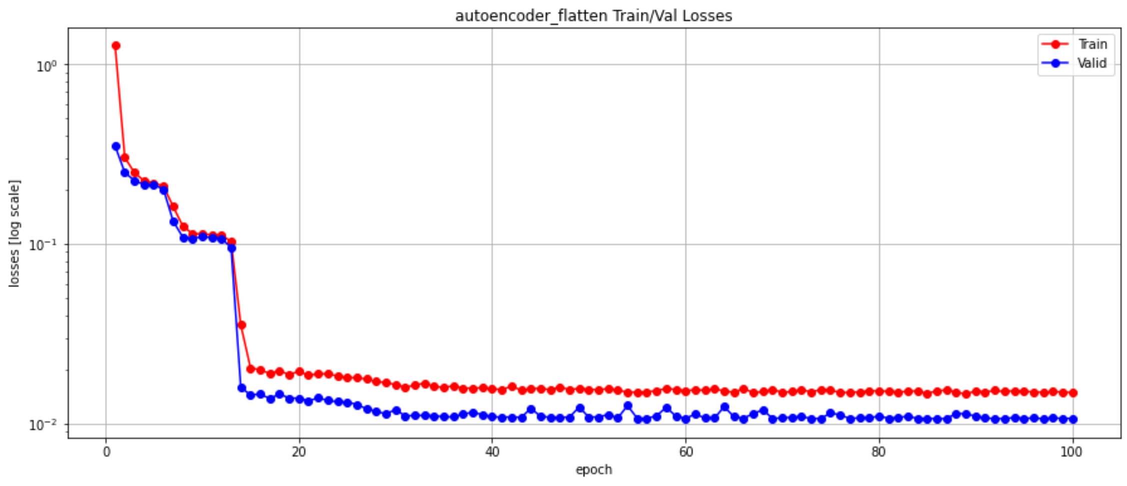


Fig. 13 Training and validation loss function dependency (AE with fully connected latent vector)

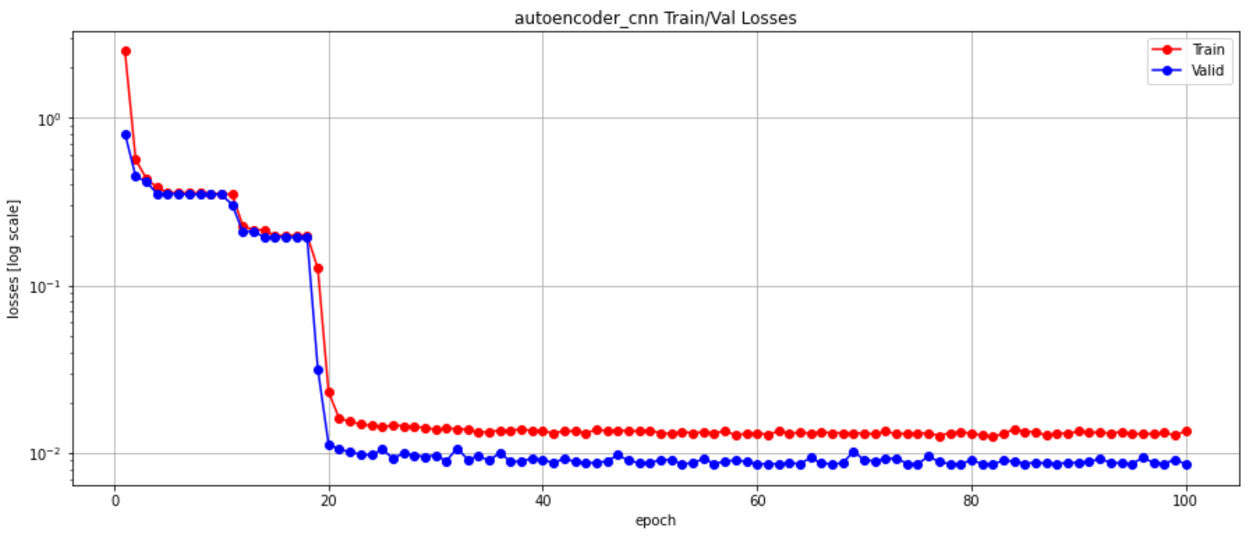


Fig. 14 Training and validation loss function dependency (AE with CNN latent vector)

To search for the detection threshold, a histogram of loss distributions for negative and positive images was constructed fig. 15-16. The threshold is determined automatically. Iteratively passing through the loss values and comparing accuracy at each step.

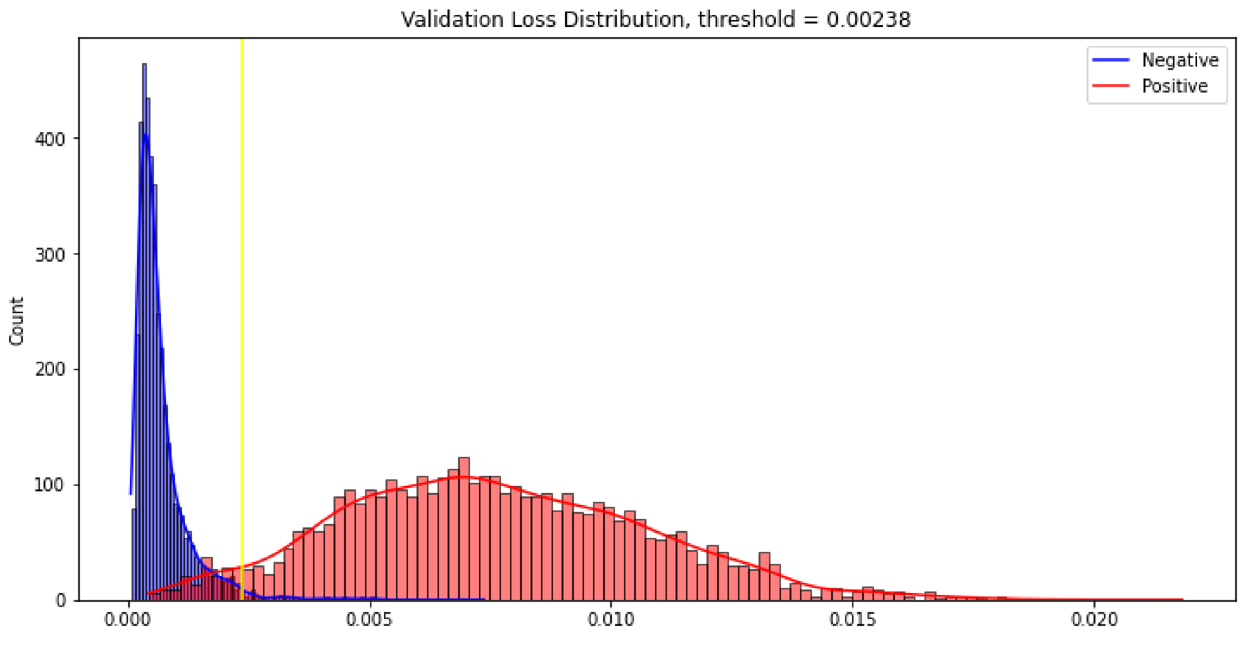


Fig. 15 Loss distribution for the validation dataset (FC NN). The yellow vertical line shows the optimal threshold.

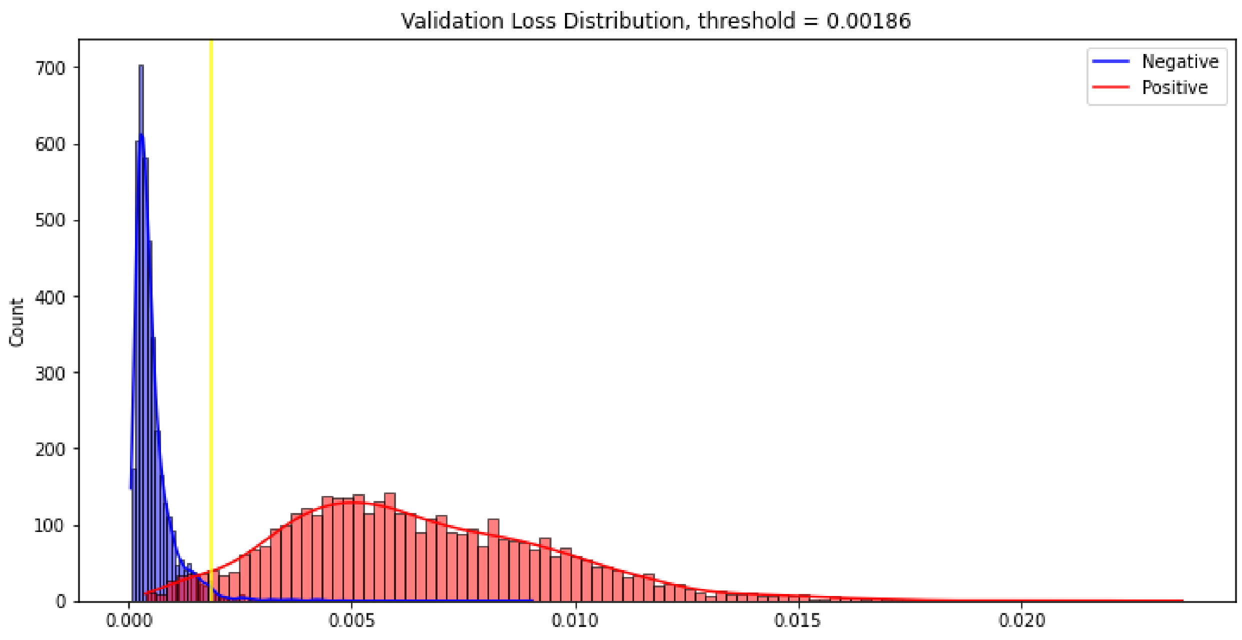


Fig. 16 Loss distribution for the validation dataset (CNN). The yellow vertical line shows the optimal threshold.

On a validation set with an optimal threshold value, it is possible to achieve accuracy 97.075 % and 97.1125 for FNN and CNN respectively.

Loss distribution for the test date set shown on fig. 17-18. On a test dataset with certain thresholds values, the accuracy was 96.825 % for FNN and same 96.825% for CNN.

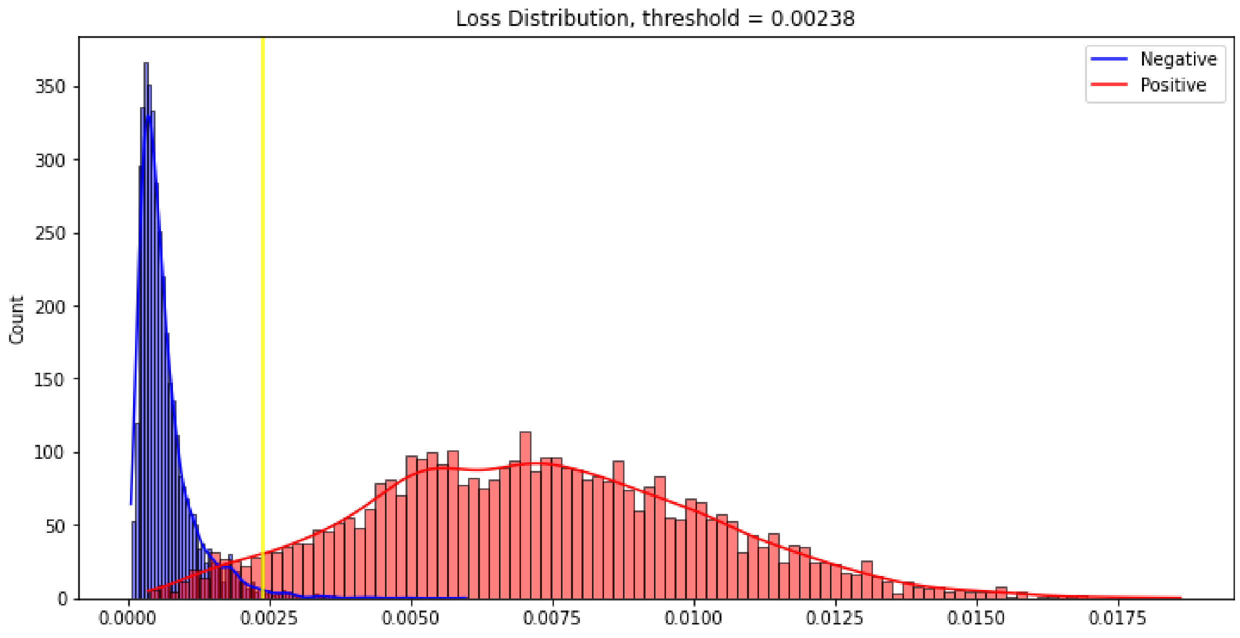


Fig. 17 Loss distribution for the Test dataset (FCNN)

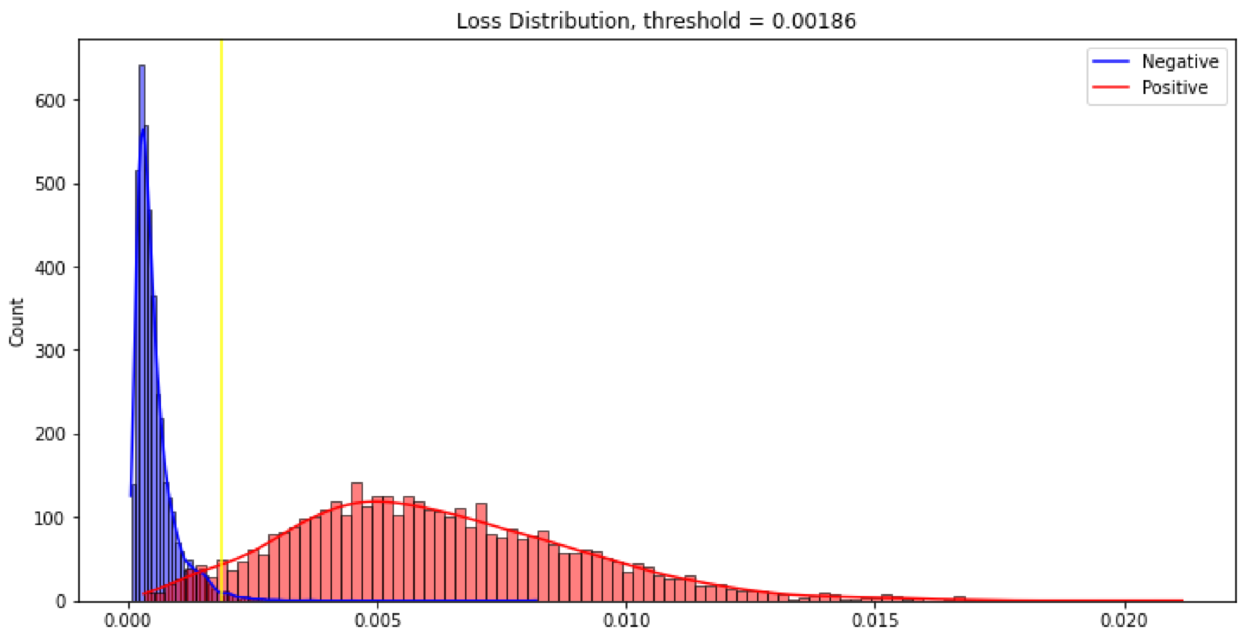


Fig. 18 Loss distribution for the Test dataset (CNN)

The following is a set of restored images and their heatmap.

The heatmap are normalized to the maximum pixel loss. Reconstructions results are shown fig. 19.

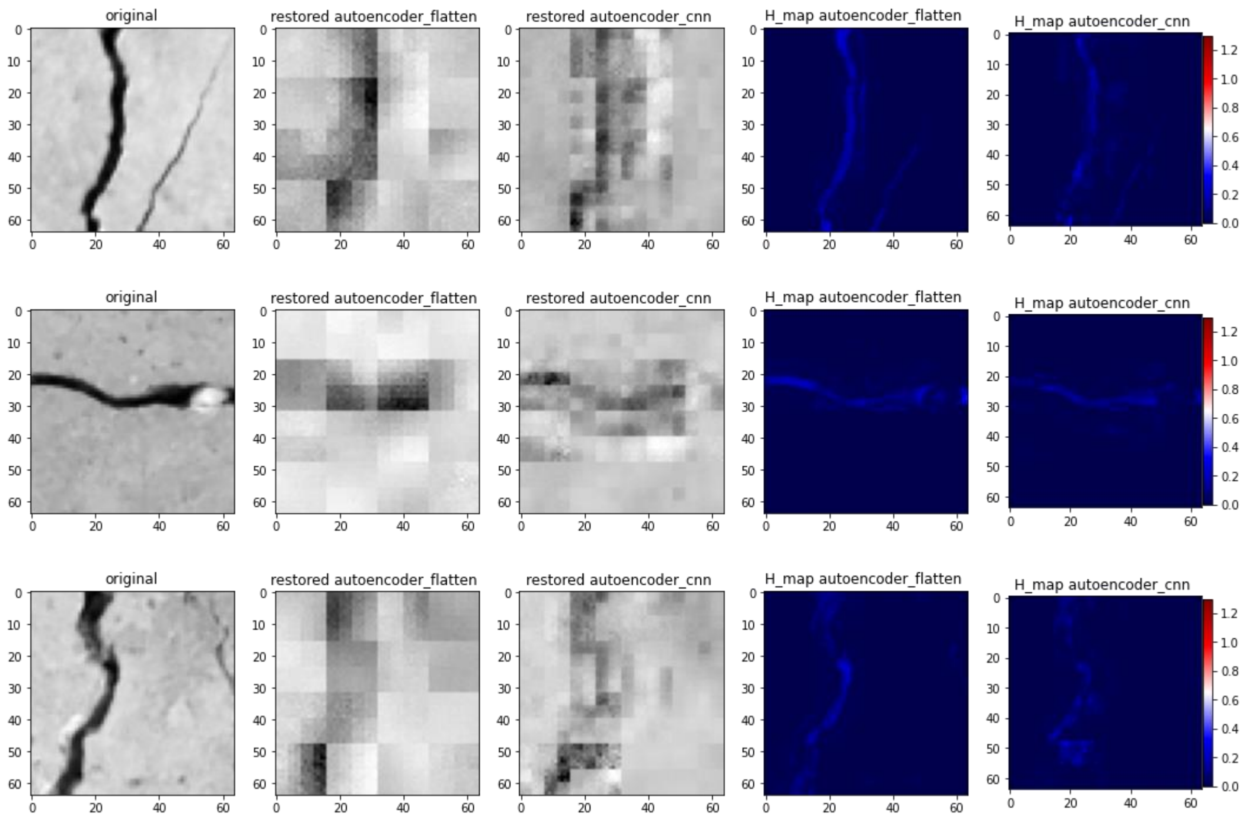


Fig. 19 Restored images using autoencoder with FNN and CNN and heatmaps

All heatmaps are normalized to the highest common value in order to be able to visually evaluate the performance of networks relative to each other.

Conclusions

This PET project aimed to expanding knowledge in unsupervised machine learning and deep learning. As the amount of produced data increases every year, unsupervised learning methods are likely to increase their usage.

In this work, experiments were carried out using statistical methods and deep learning methods. These experiments have shown that there is no one correct approach. In certain cases, any of these methods may be used. In our case, the dataset consisted of fairly easily visually distinguishable class examples. And for unsupervised images classification with high accuracy, it was enough to use the well-known method of K-means or BIRCH, which processed the features obtained from ResNet18 feature extractor that was pre-trained on the Alexnet dataset.

It turned out to be interesting that, in different sources, it is advised to normalize the input data similar to the one for the dataset on which the feature extractor was trained. However, the experiment showed that clustering using non-normalized data had greater accuracy.

These approaches have shown good quality, but they are completely unsuitable for highly unbalanced data or for tasks that are based on the search for anomalies. For such tasks, the use of an autoencoder is theoretically well suited.

For our data, the autoencoder approach also showed good results both with an unbalanced dataset in the anomaly search mode and with a balanced one. But this is more likely due to the strong visual difference between normal and anomalous data. With more similar data, it would be necessary to implement a more complex autoencoder architecture. Actually, determination of the minimum sufficient complexity of the network architecture for the available data is one of the main goals of the deep learning.

In our case, the formulation of the problem was in the definition of an anomaly, without its exact localization. Therefore, it was sufficient to use a small autoencoder with only a few layers in the encoder and decoder. Simplicity of input data did not require accurate visual reconstruction