Anomaly Detection Project - Subject n° 5

Advanced Machine Learning
Politecnico Di Torino
26-01-2023

Chloé TEMPO
s306595
s306595@studenti.polito.it

Maélis YONES
s306625
s306625@studenti.polito.it

Niccolo GIOVENALI
s316359
s316359@studenti.polito.it

Contents

1. Introduction

Our project is the **anomaly detection** one. The purpose is to build a model that is able to automatically detect anomalies (from subtle detail changes to large structural defects) in industrial products, despite having been trained on normal samples only. It focuses on the industrial setting and in particular the "cold-start" problem which refers to the difficulty of identifying normal patterns or behaviors in a new dataset or system that has not been previously seen or used. Here is the link of our github : https://github.com/chlotmpo/PathCore_anomaly_detection

1.1. PatchCore

PatchCore is an approach that has some specificities:
- A use of **mid-level features** in order to extract for each image a set of features, each one representing a specific image patch. Indeed, using deep level features may cause harm if the layer's choice is not carefully considered, since they become more and more biased towards the task on which the model has been trained.
- A **coreset-reduced patch-feature memory-bank** in order to reduce costs and time of computation execution by selecting a patch-features subset that maximize the coverage of the original set.

Then, there are 2 phases:
- First, the **anomaly detection** part where an image-level anomaly score should be defined for each test sample. This is done by finding the nearest nominal neighbor for each test sample patch-feature and then computing the maximum

distance between a test patch-feature and the corresponding nominal nearest neighbor. Using this strategy allows to find out which test patch-feature is responsible for guiding the decision about the normality of the test sample itself.
- Then, the **anomaly segmentation** to segment and localize the anomaly by computing the normality score for each test patch-feature and realigning them with their respective spatial location.

1.2. Scientific paper

To deploy this project, we used the provided paper: **"Towards Total Recall in Industrial Anomaly Detection"** which offers a new approach for the cold-start anomaly detection problem called **"PatchCore"** [1].

1.3. Dataset

For this project, we used the **MVTec Anomaly Detection** dataset, a specific dataset designed to benchmark industrial anomaly detection approaches.

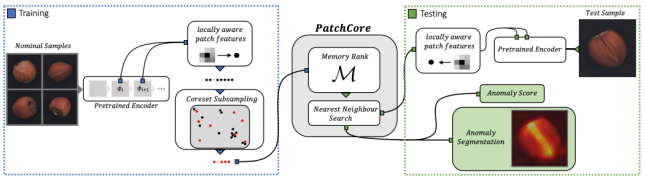
2. Related Work

As introduced previously, during this project, we tried to understand the PatchCore approach. PatchCore was created by 6 people, 5 workers at Amazon AWS and one student from University of Tübingen during its internship. To implement it, we based our searches and understanding on the article of the official version. This article is itself based on previous works for the industrial Anomaly Detection. The state-of-the-art shows that previous works based their detection performance using models pre-trained on large external natural image datasets (e.g ImageNet) without any adaptation to the data at hand. This has lead to methods reliant on better reuse of pre-trained features such as:
- **Sub-Image Anomaly Detection with Deep Pyramid Correspondences (SPADE)**
  - Apply the kNN method to sub-image level features.
  - Uses memory-banks.
- **PaDIM**
  - Uses a locally constrained bag-of-features approach.
  - Train model using patches with Manhalanobis distance measures.
  - Uses multivariate Gaussian distributions to describe the normality (constant-time inference).

PatchCore is mainly related to SPADE and PaDIM:
- **SPADE**: for the usage of a **memory-bank**. Instead of using nominal features extracted from a pre-trained backbone network with separate approaches for image and pixel-level anomaly detection, PatchCore uses the memory-bank with neighbourhood-aware patch level features critical to achieve higher performance (reduction of feature stacks to its coreset → fast linear-time inference).
- **PaDIM**: for the training of the model using **patches**. PaDim limits patch-level anomaly detection to Mahalanobis distance measures specific to each patch whereas PatchCore makes use of an efficient patch-feature memory-bank equally accessible to all patches evaluated at test time.

For our PatchCore implementation, we tried to look at a simpler implementation.

3. Methodological section



In order to execute the PatchCore algorithm, several methods are needed. Our work points at the ones we used and their functioning. To understand each part precisely, we carefully read the scientific paper related to this algorithm. Several parts are defined with all the needed details. So, we took the time to process each part allowing us to fully understand the process. We then gathered the parts in the paper with the ones of the official implementation of the code to have a complete understanding and then try to reimplement it.

3.1. First step : Model creation

To create the PatchCore model, we defined two classes in order to correctly initialize all the parameters and elements needed. We implemented a PatchCore class and his super class **KNNExtractor**. The KNNExtractor class allows defining the basics of the PatchCore class.

Several parameters are defining the structure of the class and the main ones are the following :
- *featExtract_model_name* which corresponds to the name of the neural network we want to use for the pre-trained part.
- *output_indices* which represents the output indices, layers, that we want to use as the feature extractor depth.
- *width_multiplier*, *num_classes* and *dropout* : some parameters that allow to define correctly the architecture of the pre-trained neural network model.

When an instance of this class is created, according to the model name passed in parameters, a pre-trained neural network is instantiated using the *torch.hub.load* module. This network enables the extraction of the features of the images that are trained and tested. The *output_indices* corresponds to the second and third layer in order to extract the mid-level features.

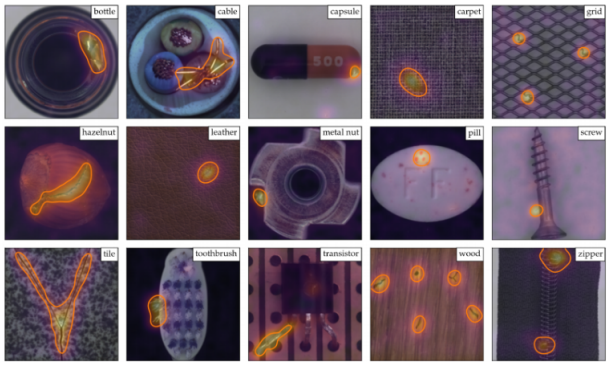
The PatchCore class's main parameters are :
- *f_coreset* : float number that corresponds to the percentage defined and it is used for the subsampling part (details are

given below in another section). It represents the fraction of the number of training samples that we want to keep from the memory-bank.
- *backbone_name* : this specifies the name of the neural network model used for the pre-trained part. This value can be overwritten over the one in the KNNExtractor class.
- *coreset_eps* : this parameter is used for the coreset subsampling part too. It is used for the selection of the subset of points.

When this class is instantiated, it gets the initial parameters from the KNNExtractor class and the specific one that were discussed just above. This class is characterized by some other methods such as : the **fit()** to train the algorithm and the **predict()** applied to the test images.

3.2. Second step : Datasets Loading

The model is applied on the challenging and widely used MVTec AD benchmark dataset. It contains over 5000 high-resolution images divided into fifteen different objects and texture categories. Each category comprises a set of defect-free training images and a test set of images with various kinds of defects as well as images without defects.



The dataset is structured as follow:

```
mvtec
|-- bottle
|-----|----- ground_truth
|-----|----- test
|-----|--------|------ good
|-----|--------|------ broken_large
|-----|--------|------ ...
|-----|----- train
|-----|--------|------ good
|-- cable
|-- ...
```

sorted by categories of objects, train, test and ground truth; and finally also divided by types of anomalies. In our implementation, we made a class **MVTecTestDataset** that contains the test and train dataset, based on the torch class **ImageFolder**, because of the nature and structure of the data. The transformation to be applied to the images is stored among its attributes, and it simply comprehends: resize to 256 px, center crop for 224 px, and normalization with mean and standard deviation of the ImageNet dataset. Those values come from the choices made by the authors to easily compare results with related previous publications. No data augmentation is applied, as this requires prior knowledge about class-retaining augmentation. The data should be present locally at the directory specified in the variable *DATASETS_PATH*. If not, it is downloaded from the MVTec website. The *__get_item__* method returns a list of 3 elements:
- The image, accessed by index, with the above mentioned transformations, and in the type of a **Tensor**.
- The corresponding ground truth mask, highlighting the anomaly.
- The label: **0 - 1** indicating, respectively, **good**, **defect**

The data loaders are then made by torch method **DataLoader**.

3.3. Third step : Model Training

To train the model, the **fit()** method of the PatchCore class is invoked. This method is used to fit the model using the training data. This data is composed only of "good" images. This is the cold-start problem, train the model only with the correct samples and find the anomalous one during the test phase. The fit() method extracts the features from the sample of the training data, using the pre-trained neural network initiated in the creation of the Patch-Core model (in our case, the WideResNet network is chosen). Then, on the feature we perform adaptive average pooling. This is similar to local smoothing over each individual feature map. Eventually, some reshaping steps are applied and the resultant patches are stored in the memory-bank.

As describing in the specification of the project and in the scientific paper, the memory-bank needs to be reduced. Indeed, if the number of patches is too big, when the comparison with test patches is made, the computation time and resources needed are too expensive. The solution found by the authors is to apply a **greedy coreset subsampling**. It is better than a random subsampling that loses some significant information in the coverage of nominal features.

Conceptually, this subsampling method selects basic groups to find a subset $S \subset A$. So, the solution on A can be more closely but also more quickly approached by those calculated on S. The algorithm in the code performing this operation is called *get_coreset_idx()*. It takes in parameters a *tensor_list* and first performs a random projection on it to reduce the dimensionality of the input tensor list. A number **n** is given and corresponds to the number of tensors that are kept in this subsampling. It is calculated based on the percentage **f_coreset** present in the PatchCore class. The function iterates **n** times over the memory-bank. It aims at selecting a subset of the tensors based on the distances between them. The function iterates from **0** to **n** and for

each iteration, it updates the *last_item* and *min_distances* variables based on the distances between the tensors of the tensor_list. The tensor with the maximum value in min_distances is selected and its index is added to the list of indices.

The purpose of the algorithm is to find a subset of the *tensor_list* for the solutions on the original *tensor_list* to be more closely approximated by those calculated on the coreset. By selecting the tensor with the maximum distance, the algorithm aims at maximizing the diversity of the items in the coreset. Thus, it increases the chances of approximating the solutions of the original *tensor_list*. At the end of the loop, the function returns the list of selected indices.

3.4. Fourth step : Model Application and Prediction

The next step is to apply the fit model on the test data. This data is composed of both good and anomalous images. The **evaluate()** method of the KNNExtractor class is called for this step. First, it invokes the **predict()** method of the PatchCore class to apply the model on the test data and classify the image as a good one or as an anomalous one. For the process of this method, each sample has the same first treatment as in the training phase:
- The features are extracted using the neural network model pre-trained (WideResNet).
- The reshaping phase and patches are created.
- A **k nearest neighbors** search is performed for each patch in the memory-bank.
- Then, the distance is computed between the test image and its nearest neighbors in the memory-bank and the patch from the memory with the smallest distance is identified.
- At the end, an anomaly score is calculated based on the **equation number 7 in the paper**. This anomalous score allows us to determine if a test patch is considered as good or anomalous.

$$s = \left(1 - \frac{\exp \|m^{\text{test},*} - m^*\|_2}{\sum_{m \in \mathcal{N}_b(m^*)} \exp \|m^{\text{test},*} - m\|_2}\right) \cdot s^*$$

Figure 1. This is the formula of the anomaly score.

In this formula, **s** is the image-level anomaly score, **s*** is the maximum distance score between test patch-features in its patch collection to each respective nearest neighbor **m*** in the memory-bank. The equation in the parenthesis is a **re-weighting** factor, to account for the behaviour of neighbour patches: if memory-bank features closest to anomaly candidate **mtest***, **m***, are themselves far from neighbouring samples and thereby an already rare nominal occurrence, **we increase the anomaly score**.

Back in the **evaluate()** method, the predictions are retrieved and stored in a variable. A list of predictions is created to store all of them. A list containing the true image labels is also created.

Then, the **roc_auc()** method is applied on these two lists, to compare the prediction to the true labels but also to compute a score of performance. This method is used to compute the ROC metrics to evaluate the performance of the classification task. **ROC AUC** means **Receiver Operating Characteristics - Area Under the Curve**. It is commonly used to evaluate the performance of a binary classification model. This metric is settled by plotting the **True Positive Rate** (TPR) against the **False Positive Rate** (FPR) at various classification thresholds. The area under the curve is used as a summary of the model's performance. A perfect classifier is represented by an **AUC of 1**.

To describe the process, the first step is to create a list of pairs **(y_true, y_score)** and then sort it. By iterating over the **(y_true, y_score)** pairs, the true positive rate and false positive rate values are computed. After that, the AUC value is computed according to the classic formula and then returned. Finally, the **evaluate()** method returns the ROC metrics which is a good mean of evaluating the performance.

3.5. Experiments and final results

As one first experiment, we tried to implement the whole architecture of the WideResNet neural network and the ResNet. We cannot use it though in the algorithm because in order to train a PatchCore model, we need to use a pre-trained version of the neural network used to extract features. In facts, it demands too much runtime on our PC. The code created to implement those 2 neural networks is available in the **neural_networks.py** file.

We tried to implement from scratch some **Pytorch** modules to have a better understanding of the whole program and how the process exactly works. The methods we tried to reimplement are:
- The **torch.cat** method, used to concatenate some tensors and create the patches.
- **torch.cdist** to compute the distances between 2 tensors.
- **torch.topk** to perform the top k neighbors research in the memory-bank.
- The **roc_auc** computation.

Finally, these methods were achieving less effective results as the ones from torch and significantly affected the overall performance of the program. Therefore, we kept the torch modules in the implementation to achieve the best results as possible.

To evaluate the performance of the PatchCore classification model, we implemented the **roc_auc** metric score on each dataset tested. This was done in order to compare the performance between them and with the official results obtained by the authors.

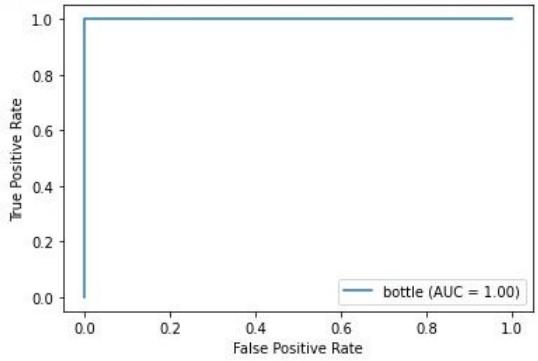


Figure 2. This is the ROC AUC Scores for the bottle dataset

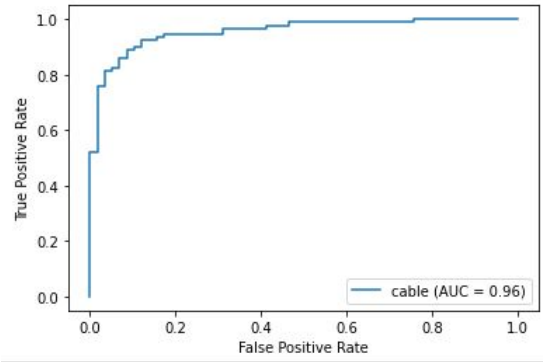


Figure 3. This is the ROC AUC Scores for the cable dataset

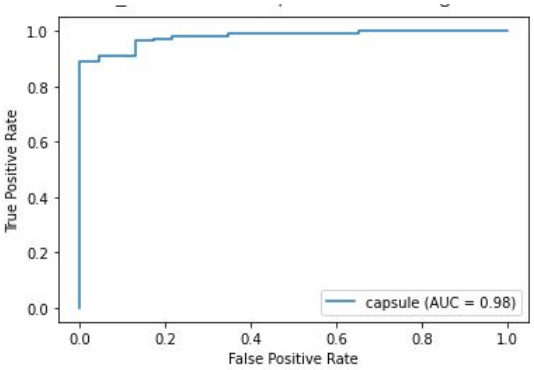


Figure 4. This is the ROC AUC Scores for the capsule dataset

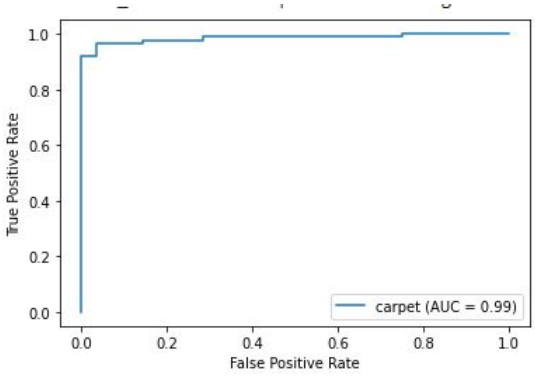


Figure 5. This is the ROC AUC Scores for the carpet dataset

In Figures 2,3,4,5, you can see the ROC curves computed on some classes, showing the good results. Further details on the overall ROC scores will follow later in the article.

As an experiment, we executed the model on each dataset while changing the value of the f_**coreset** variable. This determined the percentage of tensor to keep in the memory-bank during the subsampling phase in the training part. To choose the value, we took a look in the paper and tried the ones with which the authors obtained best results. We also performed the computation with the value of **f_coreset as 0.1 (representing 10% of the memory-bank) and a second execution with the value of f_coreset as 0.25 (representing 25% of the memory-bank)**.

All the results obtained are displayed in the following table comparing the results from the 2 different executions on all the datasets separately.

datasets	coreset=0.1	coreset=0.25
bottle dataset	1.0	1.0
cable dataset	0.9565	0.9636
capsule dataset	0.9793	0.9749
carpet dataset	0.9852	0.9856
grid dataset	0.9599	0.9574
leather dataset	1.0	1.0
metal_nut dataset	0.9829	0.9853
pill dataset	0.9400	0.9225
screw dataset	0.9205	0.9020
tile dataset	0.9986	0.9989
toothbrush dataset	0.9222	0.9222
transistor dataset	0.9954	0.9937
wood dataset	0.9885	0.9842
zipper dataset	0.9635	0.9664

The average roc_auc score obtained with the **f_coreset value as 0.1 is 0.971** and the average with **f_coreset = 0.25 is 0.968**. We obtained better results with f_coreset = 0.1 than with f_coreset = 0.25. However, the difference is not huge. Our results are a bit under the one from the author's version. They obtained an average roc_auc score equals to 0.991. This difference can be explained by the fact that our code is based on a simplified version of the official implementation. Some treatments are also simplified especially in the creation of the patches. Indeed, these small differences may explain the resulting score gap. To have a better insight of the differences between each dataset and each execution, we provide some visualization based on the obtained results.

The first two bar plots represent the different roc_auc scores obtained on each dataset, separately by execution and f_coreset value. We can see that the range in the score obtained is not very large and is between 0.9 - 1. These are very good results.

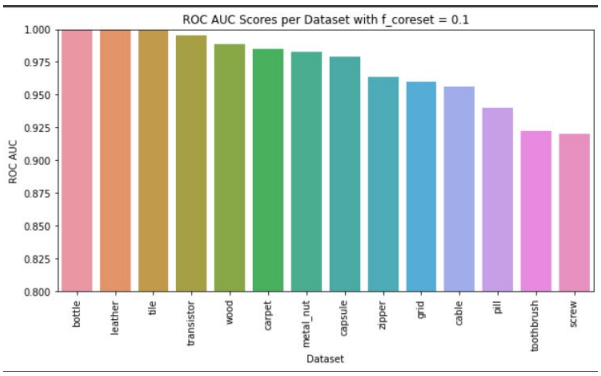


Figure 6. This is the ROC AUC Scores per dataset with f_corset=0.1

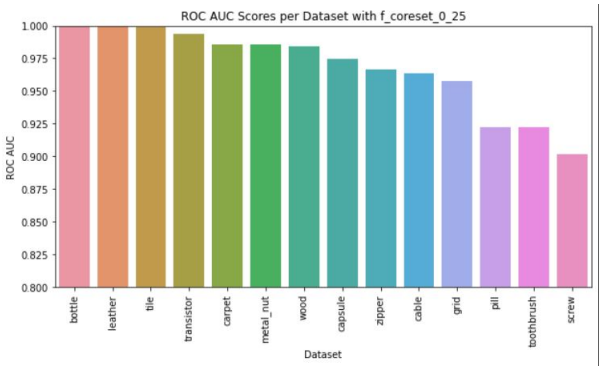


Figure 7. This is the ROC AUC Scores per dataset with f_corset=0.25.

In a second time, we provide a line plot that shows us the different roc_auc scores obtained to make a further comparison and analyses. The plot below shows the evolution in roc_auc score with the different f_coreset values, and the two lines are superposed to get a better view of the difference. We can see that there is only a slight difference between the two curves.

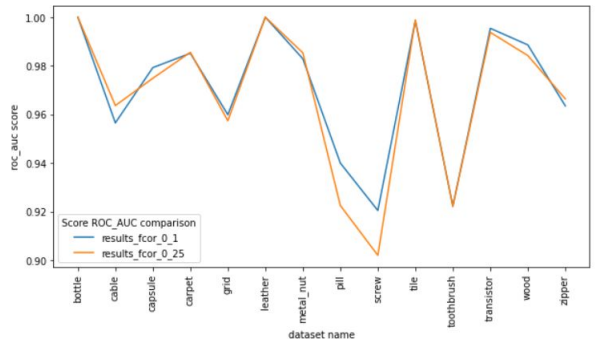


Figure 8. Comparison of the two previous graph.

Extension : Use of a different pre-training

ImageNet classification pre-training has been a standard solution to obtain generalizable features to leverage for downstream tasks during a long time. Some alternatives arised in the last few years and obtained even stronger and more generalizable representations. One of these alternatives is represented by **CLIP (Constrastive Language-Image Pre-training)** that is a model developed by **OpenAI**. It is trained to understand the relationship between languages and images. This unsupervised pre-training method is developed based on the similarity comparison between images and texts.

In order to use a different pre-training to run the PatchCore model, we can use the CLIP Image encoder. This is a part that is responsible for encoding images into a feature representation that can be used for various computer vision tasks. It is useful on image classification tasks and is represented by a deep convolutional neural network trained on a large dataset. Such as the **WideResNet** model, CLIP image encoder architecture is based on the ResNet architecture. To implement it in the PatchCore program, the library clip allows to load it and use it as a feature extractor.

In order to experiment this new implementation with this model as a new pre-trained method, we tried to change the lines in the **__init__** and the **__call__** methods of the KNNExtractor class. But it needs several modifications because changing only these lines does not work with our data loading configuration. We tried to develop our own method to forward and change the way our data is loaded because the CLIP model needs to apply preprocess steps on images and not on tensors. As a result of facts, we didn't achieve to make this part work correctly. We faced too many errors and were running out of time. That is why we removed these lines, kept the ImageNet pre-trained part. However, we gained new knowledge about the existence and use of CLIP image encoder.

4. Discussion

After understanding the full functionality of the PatchCore algorithm, we managed to reimplement it. This way, we have significantly improved our understanding of the anomaly detection task, especially, the "cold-start" problem. Because of small occurrence of defective pieces in the data set (defective pieces that can be described as outliers), it is important to initially fit the model with only "normal" pieces (that do not have any anomaly). PatchCore uses a pre-trained CNN on ImageNet to extract features from its images. In an extension part of the project, we tried to change the pre-training part to implement the CLIP image encoder model. We have not been able to generate results to compare performance. But, according to the official doc-

umentation of this new alternative, the results obtained on other tasks are very efficient and sometimes better than ImageNet. Regarding our experiments, we think that the results could be slightly improved with this change in pre-training.

The main contributions of PatchCore are the use of :
- **Mid-level features** allowing to extract for each image a set of features each one representing a specific image patch.
- The **corset-reduced patch-feature memory-bank** exploiting an iterative greedy algorithm to perform a selection of patch-features maximizing the coverage of the original set.

Understanding these main contributions was hard at the beginning. We now know how each part process and why it is important to the author's goal of reaching the total recall in industrial anomaly detection.

5. Conclusion

This PatchCore implementation exploits some of the topics viewed in this course, mainly how to smartly use various techniques of image classification, well tuned on the task to be performed. Transfer learning from the most efficient deep convolutional neural network (Wide ResNet 50), has been exploited to detect the features of images coming from a different subset (industrial manufactures versus natural images). Understanding that the anomalies to be identified are mostly due to manufacturing accidents, such as broken glass, scratches, burns or structural changes, driven the choice to only extract the features of the middle convolutional layers of the network, that coincide more with the abstract features required in an industrial environment.

From recent papers, the idea of using a memory-bank of patches trained on flawless objects has been used, to address the cold-start problem. The Coreset reduction can be seen as some kind of data preprocessing, where the main purpose is to obtain good overall results, with as little data as possible, by choosing a subset which contains as many significant information as possible. Here, the subsampling focuses on reducing the runtime.

With those elements set up, the prediction is smartly seen as the computation of an anomaly score, based on the distance between the sample patches and the memory-bank ones. Finally, the performance of the classifier is evaluated by computing the area under the curve ROC. Regarding the anomaly detection, and wisely picking existing algorithms, the PatchCore method outperforms the previous PaDIM and SPADE.

References

[1] Karsten Roth, Latha Pemula, Joaquin Zepeda, Bernhard Schölkopf, Thomas Brox, and Peter Gehler. Towards total recall in industrial anomaly detection. *CVPR*, pages 14318–14328, 2022. 2