Step-by-step Tutorial for AUROC Score Reporting on MVTec-AD Using PatchCore and EfficientAD

To start off, I cloned the anomalib repo:

```
!git clone https://github.com/openvinotoolkit/anomalib.git
```

Then, I made sure to install the anomalib library with all necessary packages:

```
!pip install -e .[full]
```

Now, I can run the anomalib API for PathCore and EfficientAD!

PatchCore

Documentation:

First, the API call:

```
from anomalib.data import MVTec
from anomalib.models import Patchcore
from anomalib.engine import Engine
```

We'll start with the "grid" category from the MVTec-AD dataset:

```
# Initialize components
datamodule = MVTec(category='grid')
model = Patchcore()
engine = Engine()
```

```
# Train the model
engine.fit(datamodule=datamodule, model=model)
```

Here is the output:

```
Selecting Coreset Indices.:   0%|          | 0/20697 [00:00<?, ?it/s]
Selecting Coreset Indices.:   0%|          | 1/20697 [00:00<46:47,  7.37it/s]
Selecting Coreset Indices.:   0%|          | 14/20697 [00:00<05:02, 68.33it/s]
Selecting Coreset Indices.:   0%|          | 30/20697 [00:00<03:16, 105.40it/s]
Selecting Coreset Indices.:   0%|          | 47/20697 [00:00<02:41, 127.68it/s]
Selecting Coreset Indices.:   0%|          | 63/20697 [00:00<02:32, 134.95it/s]
Selecting Coreset Indices.:   0%|          | 80/20697 [00:00<02:22, 144.90it/s]
Selecting Coreset Indices.:   0%|          | 97/20697 [00:00<02:15, 151.55it/s]
Selecting Coreset Indices.:   1%|          | 114/20697 [00:00<02:11, 156.00it/s]
Selecting Coreset Indices.:   1%|          | 132/20697 [00:00<02:08, 160.62it/s]
Selecting Coreset Indices.:   1%|          | 150/20697 [00:01<02:04, 164.46it/s]
Selecting Coreset Indices.:   1%|          | 168/20697 [00:01<02:02, 167.06it/s]
Selecting Coreset Indices.:   1%|          | 186/20697 [00:01<02:01, 168.52it/s]
Selecting Coreset Indices.:   1%|          | 203/20697 [00:01<02:01, 168.74it/s]
```

In effect, Pathcore uses these coresets to classify the training data. These coresets are a subset of patches selected from all those available in the dataset. This allows the model to perform a more efficient classifications over a smaller series of inputs during training. More specifically, Patchcore performs feature extraction and then applies k-nearest neighbor classification to the individual feature vectors extracted. This allows the model to group together inputs with similar feature vectors.
From the clusters, Patchcore then performs sampling to refine its training. This allows the model to identify outliers and be more robust overall to test input.

With the model trained, we are now ready to report the AUROC score for the “grid” category. By running the following code:

```
test_results = engine.test(model=model, datamodule=datamodule, ckpt_path=engine.trainer.checkpoint_callback.best_model_path)
```

We obtain not only the AUROC score but also the scores reported by all the metrics in the anomalib dataset.

Test metric	DataLoader 0
image_AUROC	0.9807853698730469
image_F1Score	0.9734513163566589
pixel_AUROC	0.9804317355155945
pixel_F1Score	0.38011202216148376

For the “grid” category, our image AUROC score is roughly 0.98.

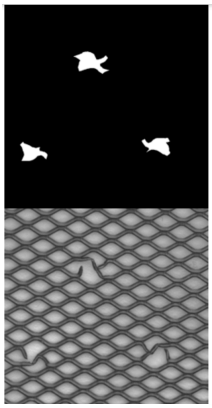
Running the same code on the “leather” and “tile” categories by simply changing the category parameter in the MVTec() call (reinserted below for ease):

```
# Initialize components
datamodule = MVTec(category='grid')
model = Patchcore()
engine = Engine()
```

We retrieve the following scores:
“Leather”: image AUROC = 1.0
“Tile”: image AUROC = 0.99

Average image AUROC across all three categories = (0.98 + 1 + 0.99) / 3 = 0.99

Visualization:



Using the “grid” category, we can compare the mask (above) with the anomalies in the test image (below). The mask shows exactly where the anomalies were observed in the test image.

Interpretation:

The AUROC scores observed indicate almost perfect classification of the images as anomalous. In the case of the “leather” category the AUROC score of 1.0 indicates perfect classification using Patchcore. Thus, Patchcore is a reliable model for classifying anomalous images in the MVTecAD dataset.

EfficientAD:

Documentation:

The setup for the EfficientAD algorithm is similar to Patchcore, we simply need to import the right module and replace the model:

```
from anomalib.models import EfficientAd

datamodule = MVTec(num_workers=0)
datamodule.prepare_data()
datamodule.setup()

i, data = next(enumerate(datamodule.val_dataloader()))
data_dict = data.__dict__
print(data_dict.keys())

model = EfficientAd()
```

Here is what the output looks like:

```
Calculate Validation Dataset Quantiles: 100%|██████████| 5/5 [00:03<00:00,  1.50it/s]
Calculate Validation Dataset Quantiles: 100%|██████████| 5/5 [00:02<00:00,  1.68it/s]
Calculate Validation Dataset Quantiles: 100%|██████████| 5/5 [00:03<00:00,  1.50it/s]
Calculate Validation Dataset Quantiles: 100%|██████████| 5/5 [00:03<00:00,  1.57it/s]
Calculate Validation Dataset Quantiles: 100%|██████████| 5/5 [00:03<00:00,  1.66it/s]
```

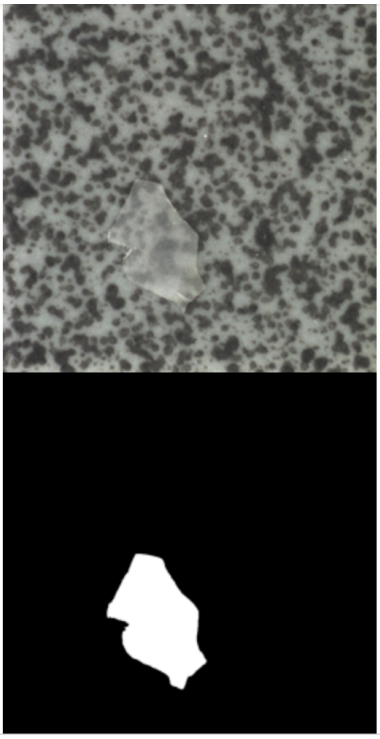
During EfficientAD's training, the model calculates validation quantiles to determine the spread of the anomaly scores across images. This allows the model to set a threshold to classify new images fed into it. For instance, values above this created threshold may be considered anomalous.
The EfficientAD algorithm uses a student-teacher architecture for anomaly detection. The teacher and students are both neural networks whereby the teacher is pretrained and is the reference for what "normal" images should look like. The student (also a neural network) then tries to replicate the feature extraction output by the teacher. The difference between what the student and teacher output then correlates to an anomaly score which allows the model to classify a given as anomalous or normal based on the threshold.

Using the same testing parameters as with Patchcore:

```
test_results = engine.test(model=model, datamodule=datamodule, ckpt_path=engine.trainer.checkpoint_callback.best_model_path)
```

We get the following AUROC scores:
"Grid": 1.0
"Leather": 0.9963
"Tile": 0.9874
Average AUROC: 0.9946

Visualization:



Again, the mask shows where the anomaly is almost perfectly.

Interpretation:

Like with Patchcore, the AUROC scores of the EfficientAD model observed indicate almost perfect classification of the images as anomalous. In the case of the “grid” category the AUROC score of 1.0 indicates perfect classification using EfficientAD. Thus, EfficientAD is a reliable model for classifying anomalous images in the MVTecAD dataset.

Similarity Search

Using the quadrant vector database, we can extract features from the models and run a similarity search.

We start by installing qdrant:

```
!pip install qdrant-client
```

Then we run quadrant vector db’s python integration code:

```
from qdrant_client import QdrantClient
from qdrant_client.http.models import VectorParams, Distance

collection_name = "mvtecad2"
embedding_dimension = 2048

qdrant_client.recreate_collection(
    collection_name=collection_name,
    vectors_config=VectorParams(size=embedding_dimension, distance=Distance.COSINE)
)
```

The embedding_dimension can be found by checking the feature vector's shape that we are using. In this case, I defined and extract_features function:

```
def extract_features(backbone, image_path):
    image = Image.open(image_path).convert("RGB")
    image = transform(image).unsqueeze(0)

    with torch.no_grad():
        features = backbone(image)

    return features.view(1, -1).cpu().numpy()
```

And my vector shape was (1, 2048). Now we run the search method from quadrant vector db:

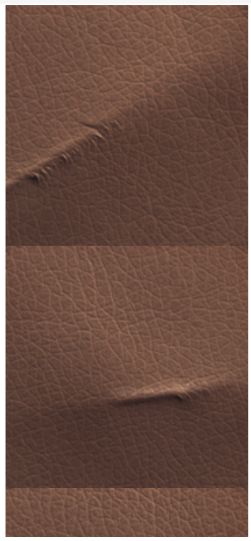
```
query_vector = query_features[0].tolist()

search_results = qdrant_client.search(
    collection_name=collection_name,
    query_vector=query_vector,
    limit=6
)

for result in search_results:
    print(f"ID: {result.id}, Score: {result.score}, Payload: {result.payload}")
```

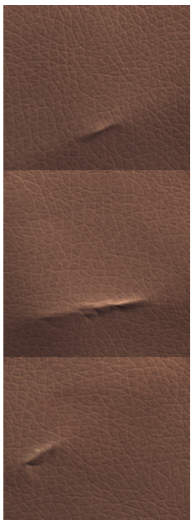
In my case I ran this on the following image path:
`/content/anomalib/datasets/MVTec/leather/test/fold/005.png`
And got the following top 5 results:

```
 {'image_path': '/content/anomalib/datasets/MVTec/leather/test/fold/015.png'}
{'image_path': '/content/anomalib/datasets/MVTec/leather/test/fold/001.png'}
 {'image_path': '/content/anomalib/datasets/MVTec/leather/test/fold/004.png'}
{'image_path': '/content/anomalib/datasets/MVTec/leather/test/fold/003.png'}
{'image_path': '/content/anomalib/datasets/MVTec/leather/test/fold/012.png'}
```

Note that the test image had a fold in it and the top 5 similar images also had a fold on them:

```
for result in search_results:
  show_image = Image.open(result.payload['image_path'])
  display(show_image.resize((178, 178)))
```





PDN Receptive Field Calculation

The figure shows how the Patch Description Network reduces the input patch from a height of 33x33 to an output feature vector with a height of 1. This means that each of these 384 output feature vectors describes one of the 3 33x33 patches. Computationally, at the level of a single channel, we can reverse the receptive field calculation.

Output = (Input + 2*padding - kernelsize) / stride + 1
becomes Input = (Output - 1)*stride + kernelsize - 2*padding

Note that the figure did not include any padding so we will not include it in our calculations. Additionally, we will assign a stride of 1 to convolutional steps and a stride of 2 to pooling steps.

(1-1)*1 + 4 = 4
(4-1)*1 + 3 = 6
(6-1)*2 + 2 = 12
(12-1)*1 + 4 = 15
(15-1)*2 + 2 = 30
(30-1)*1 + 4 = 33

So we retrieve the 33x33 input patch.