Anomaly detection based on Anomalib lib

Introduction

Anomaly detection plays a crucial role in industrial production, ensuring product quality and reducing defects. Detecting anomalies effectively can prevent faulty products from reaching customers, enhancing overall manufacturing efficiency. In recent years, deep learning-based methods have gained popularity for anomaly detection due to their ability to learn intricate patterns from data automatically.

In this tutorial, we explore two powerful anomaly detection models, PatchCore and EfficientAD, and demonstrate their application using the MVTec Anomaly Detection (MVTec-AD) dataset by anomalib. The MVTec-AD dataset is widely used for benchmarking anomaly detection methods, as it contains real-world industrial defect images across various product categories.

To keep this tutorial concise and practical, we focus on three representative categories from the MVTec-AD dataset: tile, leather, and grid. We will show how to utilize the Anomalib framework to apply these models, evaluate their performance using AUROC scores, and compare their effectiveness in detecting anomalies.

Anomalib

Anomalib is an open-source library designed to streamline anomaly detection using deep learning models.

Here is how to install it with conda

```
conda create -n anomalib_env python=3.10 -y
conda activate anomalib_env
pip install anomalib[full]
```

Hints: the python version is significant.

In the next sections, we will demonstrate how to use Anomalib to apply PatchCore and EfficientAD effectively.

PatchCore

PatchCore is a memory-efficient anomaly detection model that leverages patch-based feature extraction and core-set subsampling to identify defects effectively. The core idea

behind PatchCore is to capture normal feature distributions using a subset of training data and detect anomalies by measuring deviations from these learned patterns.

The steps are

1. **Feature Extraction:** PatchCore uses a pretrained convolutional neural network (CNN) to extract deep feature representations from image patches. These features capture the structural and texture details of normal samples.
2. **Core-Set Subsampling:** Instead of storing all extracted features, PatchCore applies core-set subsampling to retain **only the most representative features**. This step reduces memory usage and improves computational efficiency.
3. **Anomaly Detection:** During inference, features from test images are compared against the stored core-set features using distance-based metrics. Higher deviations indicate anomalies.
4. **Anomaly Scoring & Localization:** Anomaly scores are assigned based on the distance between test image features and the core-set. Additionally, an anomaly heatmap is generated to visualize defect locations.

We use anomalib to easier reproduce the model.

```
# 1 Import necessary libraries
from anomalib.data import MVTec
from anomalib.models import Patchcore
from anomalib.engine import Engine

# 2 Manually define categories
categories = ['grid', 'leather', 'tile']

# 3 Initialize PatchCore model
model = Patchcore(pre_trained=True)

# 4 Initialize results dictionary
results = {}

# 5 Loop through categories
for category in categories:
    print(f"Processing category: {category}")

    # Load dataset for the current category
    datamodule = MVTec(
        root="./MVTec/",
        category=category,
        train_batch_size=32,  # PatchCore allows batch size > 1
        eval_batch_size=32,
        num_workers=8,
    )

    print(f"Training and evaluating PatchCore on {category}")

    # Initialize training engine
```

```
    engine = Engine(max_epochs=10)

    # Train and test the model
    engine.fit(datamodule=datamodule, model=model)
    test_outputs = engine.test(datamodule=datamodule, model=model)

    # Store results
    results[category] = test_outputs[0]['image_AUROC']

# 6 Compute average AUROC across selected categories
if results:
    avg_auroc = sum(results.values()) / len(results)
    results["average"] = avg_auroc

# Print final results
print("PatchCore Results:", results)
```

The final AUROC score on the dataset is 0.9791, 0.8736 and 0.8770 for grid, leather and tile detection.

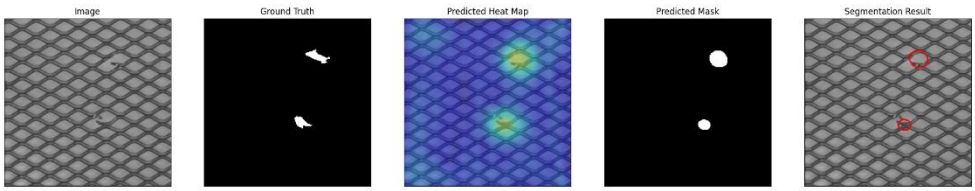

Figure 1 results of PatchCore

EfficientAD

EfficientAD is a deep learning-based anomaly detection model designed to achieve high detection performance with lower computational costs. It integrates an efficient feature extraction mechanism with a compact and scalable detection strategy, making it well-suited for industrial applications.

Similarly, the steps of EfficientAD are

1. **Feature Extraction:** EfficientAD employs a lightweight convolutional neural network (CNN) to extract meaningful representations from images, reducing the computational burden compared to heavier architectures.
2. **Anomaly Detection:** The extracted features are processed using a statistical or deep learning-based anomaly detection module that learns normal patterns from training data and identifies deviations during inference.

3. **Anomaly Scoring & Localization:** The model assigns an anomaly score to each test image based on the deviation from learned normal patterns. Additionally, heatmaps are generated to highlight anomalous regions in the image.

Next is the code of EfficientAD by anomalib

```python
# 1️⃣ Import necessary libraries
from anomalib.data import MVTec
from anomalib.models import EfficientAd
from anomalib.engine import Engine

# 2️⃣ Manually define categories
categories = ['grid', 'leather', 'tile']

# 3️⃣ Initialize EfficientAD model
model = EfficientAd()

# 4️⃣ Initialize results dictionary
results = {}

# 5️⃣ Loop through categories
for category in categories:
    print(f"Processing category: {category}")

    # Load dataset for the current category
    datamodule = MVTec(
        root="./MVTec/",
        category=category,
        train_batch_size=1,  # EfficientAD requires batch size = 1
        eval_batch_size=1,
        num_workers=8,
    )

    print(f"Training and evaluating EfficientAD on {category}")

    # Initialize training engine
    engine = Engine(max_epochs=10)

    # Train and test the model
    engine.fit(datamodule=datamodule, model=model)
    test_outputs = engine.test(datamodule=datamodule, model=model)

    # Store results
    results[category] = test_outputs[0]['image_AUROC']

# 6️⃣ Compute average AUROC across selected categories
if results:
    avg_auroc = sum(results.values()) / len(results)
    results["average"] = avg_auroc

# Print final results
print("EfficientAD Results:", results)
```

The AUROC score for each type is 0.9933, 0.6144 and 0.9084.

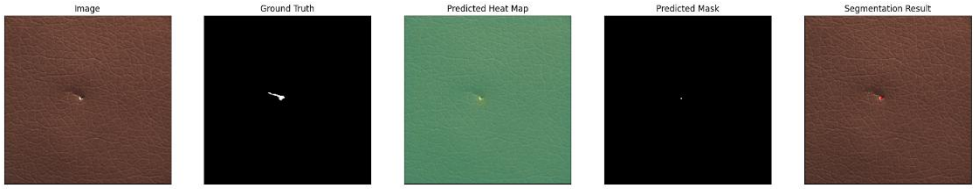

Figure 2 result of EfficientAD(leather)

Similarity Search

According to the features model extracted, we can perform similarity search. We send the model an anomaly image, then the model return 5 similar images.

```
import torch
import numpy as np
from anomalib.models import Patchcore
from anomalib.data.utils import read_image
from torchvision import transforms
from qdrant_client import QdrantClient
from qdrant_client.models import PointStruct, Distance, VectorParams

# Load pretrained PatchCore model
model = Patchcore(pre_trained=True)  # Initialize model structure
model.load_state_dict(torch.load("./patchcore_trained.ckpt"))  # Load weights
model.eval()  # Set to evaluation mode

# Connect to Qdrant (running locally)
qdrant = QdrantClient(":memory:")  # Use in-memory storage for testing

# Create a collection for storing features
qdrant.recreate_collection(
    collection_name="patchcore_features",
    vectors_config=VectorParams(size=model.hparams.embedding_size, distance=Distance.COSINE),
)

# Define preprocessing transform (same as PatchCore training)
transform = transforms.Compose([
    transforms.Resize((256, 256)),  # Resize if needed
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

def extract_features(image_path):
    """Extract PatchCore features from an image."""
    image = read_image(image_path)  # Load image
    image = transform(image).unsqueeze(0)  # Apply transforms and add batch dim

    with torch.no_grad():
        feature = model(image).squeeze().numpy()  # Extract features
```

```
    return feature

# Example: Store training images in Qdrant
import os
path = './MVTec/grid/test'
image_paths = []
for root, dirs, files in os.walk(path):
    for file in files:
        image_paths.append(os.path.join(root, file))

for i, img_path in enumerate(image_paths):
    feature = extract_features(img_path)
    qdrant.upload_points(
        collection_name="patchcore_features",
        points=[PointStruct(id=i, vector=feature, payload={"image_path": img_path})],
    )

def find_similar_images(query_image_path, top_k=5):
    """Finds top-K most similar images in Qdrant."""
    query_feature = extract_features(query_image_path)

    # Search for nearest neighbors
    results = qdrant.search(
        collection_name="patchcore_features",
        query_vector=query_feature,
        limit=top_k
    )

    # Retrieve the most similar images
    similar_images = [hit.payload["image_path"] for hit in results]
    return similar_images

# Example: Find 5 most similar images for a test image
query_image = "./MVTec/grid/ground_truth/broken/000_mask.png"
similar_images = find_similar_images(query_image)
print("Most similar images:", similar_images)
```

Patch Description Network (PDN) Receptive Field Calculation

The receptive field (RF) of a neuron in a CNN is the region of the input image that affects the neuron's activation.

The input is 33 x 33.

Step 1: First Layer

The first layer is 4x4 conv, the output is 30x30

Step 2: Second Layer

The second layer is 2x2 pool, the output is 15x15

Step 3: Third Layer

The third layer is 4x4 conv, the output is 12x12

Step 4: Fourth Layer

The fourth layer is 2x2 pool, the output is 6x6

Step 5: Fifth Layer

The fifth layer is 3x3 conv, the output is 4x4

Step 6: Sixth Layer

The sixth layer is 4x4 conv, the output is 1x1

Each 1x1 value includes original 33x33 information, thus each output feature vector describes a 33× 33 patch.

Reference:

Paul Bergmann, Kilian Batzner, Michael Fauser, David Sattlegger, Carsten Steger: **The MVTec Anomaly Detection Dataset: A Comprehensive Real-World Dataset for Unsupervised Anomaly Detection**; in: *International Journal of Computer Vision* 129(4):1038-1059, 2021, DOI: 10.1007/s11263-020-01400-4.

Paul Bergmann, Michael Fauser, David Sattlegger, Carsten Steger: **MVTec AD — A Comprehensive Real-World Dataset for Unsupervised Anomaly Detection**; in: *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR),* 9584-9592, 2019, DOI: 10.1109/CVPR.2019.00982.