Andrew Aquino

DS681 – Deep Learning for Computer Vision

Assignment 2

Introduction

Anomaly detection is used across many fields to detect defects, system failures and other potential issues. We must have a baseline norm and from there we can flat potential defects or anomalies in our system. One example is the use of computer vision to detect anomaly defects in the pharmaceutical sector. We can use computer vision to detect any incorrect shapes, cracks or missing imprints on the pills being manufactured. This can help reduce the rate of irregularities being shipped out.

For this assignment we were tasked to use MVTec Anomaly Detection dataset and Anomalib library to train a model that can evaluate for flat surfaces. From there we can extract those features and perform a similarity test. If the image being queried is similar then we can return 10 similar images, if not then its an anomaly and should be reported.

Resources

To accomplish this assignment we were to use the MVTec Anomaly Detection Dataset or MVTec AD. This dataset is used for evaluating unsupervised anomaly detection, with a focus on real-world scenarios. The dataset is divided into 15 categories which is mixture of different objects and textures, for example, it includes carpet and wood textures or bottle and metal nut objects. The data is split between training and test set that can be used to train the model. And the test images feature over 70 different types of defects, which can include scratches, dents or bumps.

We next had to use the open-source deep learning Anomalib library. This library focuses on anomaly detection algorithms. The library makes it super seamless and have ready to use models to accomplish this task. The Anomalib library allows us to use PatchCore which is powerful method that leverages feature extraction from pre-trained Convolutional Neural Networks (CNNs). PatchCore creates a memory bank of features from the defect images and stores them. Now that it has these features, during inference when an image is presented the model calculates the similarity between the test image and the feature closest to the stored feature int the memory bank. From there it can calculate if the image is an anomaly or not.

Finally, we use Postgres and its extension PGVECTOR. Simply, Postgres is an open-source database management system. With the PGVECTOR extension we can used the stored vectors from the PatchCore memory bank and query through it via Postgres. When we pass an image the system will perform a nearest neighbor vector search using euclidean or cosine distance. From here we can see if the image has similar images or is an anomaly.

Implementation

The Anomalib library makes it pretty unambiguous on how to use their resources and train their supplied models. First make sure that torch, torchvision, pytorch, torchmetrics and opencv-python is installed. From there we simply follow the Anomalib github installation process. Next the setup is pretty straightforward and I will walk through important module calls and they are needed to accomplish our task.

```
# this will load the MVTech module
mvt_module = MVTecAD(
    root="datasets/MVTecAD",
    category=category,
    train_batch_size=32,
    eval_batch_size=32,
    num_workers=4,
)

# preprossors point which is the image size and cropped size
preprocessor = Patchcore.configure_pre_processor(
    image_size=(256, 256),
    center_crop_size=(224, 224)
)

# this is the pretrained model
patchcore_model = Patchcore(
    backbone="wide_resnet50_2",
    layers=["layer2", "layer3"],
    coreset_sampling_ratio=0.1,
    pre_trained=True,
    pre_processor=preprocessor
)

# initialize the engine
engine = Engine()

# this will train the model
engine.fit(model= patchcore_model , datamodule= mvt_module )
```

1. MVTec AD Data Module (mvt_module)

- root – is the local directory where the MVTec AD dataset files are located
- category – are the sub-categories from the MVTec dataset, in our case I inputted a list of categories (tile, leather, and grid)
- train_batch_size and eval_batch_size – are the number of images processed together in one iteration during training and evaluation respectively.
- num_workers – helps speed up the process by loading the data in parallel during model training.

2. Preprocessor Configuration (preprocessor)

- This is simply the size of the image that is being inputted and the size of the image after being cropped. We sometimes need to change the size of the image so that it can be used in a pre-trained model. Some models only work for pictures of a certain size. Since I decided to use Wide ResNet I needed an image size of 224 x 224

3. PatchCore Model (patchcore_model)

- backbone - this selects a pre-trained CNN architecture used to extract features; in my case I used Wide ResNet
- layers - this specifies which layers I want the network to extract the patch features or more high-level image semantics.
- coreset_sampling_ratio – this is a crucial hyperparameter for PatchCore. This determines the percentage of training features that form the memory bank. We don't want a large number as this can cause a larger memory footprint (take up more memory) and can cause redundancy. So for my cause I stuck with 10%.
- pre_trained – just allows you to tell the system to use the pre-trained weights
- pre-processor – we past in the preprocessor configuration defined previously.

4. Engine

- We simply initialize the training loop object that will run on our GPU/CPU device
- engine.fit() - This starts the training process. Inputting the previously defined PatchCore model and data module.

Postgres

As I mentioned before Postgres is great database management system and its extension PGVECTOR is a powerful vector database. The great thing of Patchcore is that its saves its data in PyTorch form for us. As we can simply see in these lines of code, in the main loop while the model is training we can save the Patchcore model memory bank data for each category.

```
memory_bank = patchcore_model.model.memory_bank.data
memory_bank_path = os.path.join(ASSET_DIR, f"{category}_memory_bank.pt")
torch.save(memory_bank, memory_bank_path)
print(f"Saved memory bank for {category} to {memory_bank_path}")
```

Since Postgres is better suited to run on local machines, that’s exactly what I did. But there was a small problem, since our class docker file is isolated from my main machine. I was having issue connecting to the local Postgres server. So solve this issue I had to change a few of the Postgres configuration files so that it can accept all local host address’s. I also had to modify the docker container docker-compose.yml file to internal host gateway as the image at the bottom showcases.

```
container_name: eng-ai-agents-dev
extra_hosts:      You, 4 hours ago • Uncom
  - "host.docker.internal:host-gateway"
```

From there I was finally able to connect to server and log in with the credentials I had setup previously. My code consisted of this flow and steps:

1. I first connected to the server and defined a few variables
    - This involved defining the location fo the Patchcore embeddings (data) location was stored
    - My database credentials
    - And defining the dimension of feature vector (in this case 1536)

2. I next connected to the Postgres server and executed a few SQL commands to setup a table
    - This included the name of the table
    - Unique ID for the table
    - The data type of the categories
    - The unique name of the embeddings

- And the goal to check if its anomalies or not (boolean)

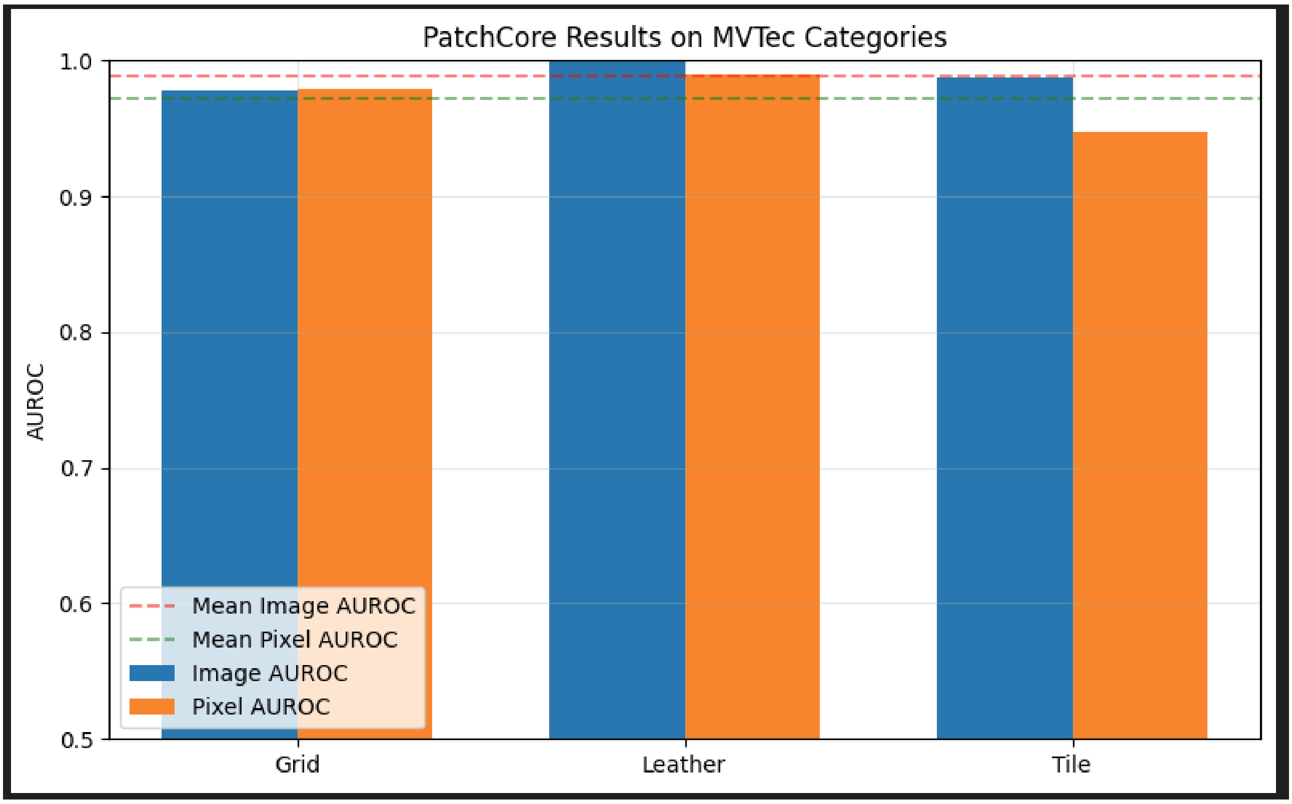
3. Next I inserted them embeddings which were the data from the Patchcore memory banks and make sure that to convert it so we can parse through it in the database.
   - tensor → numpy → Python list

4. I also created a function to insert “fake” anomalies, which the system just marks as anomalies and search for data similar to that anomaly.

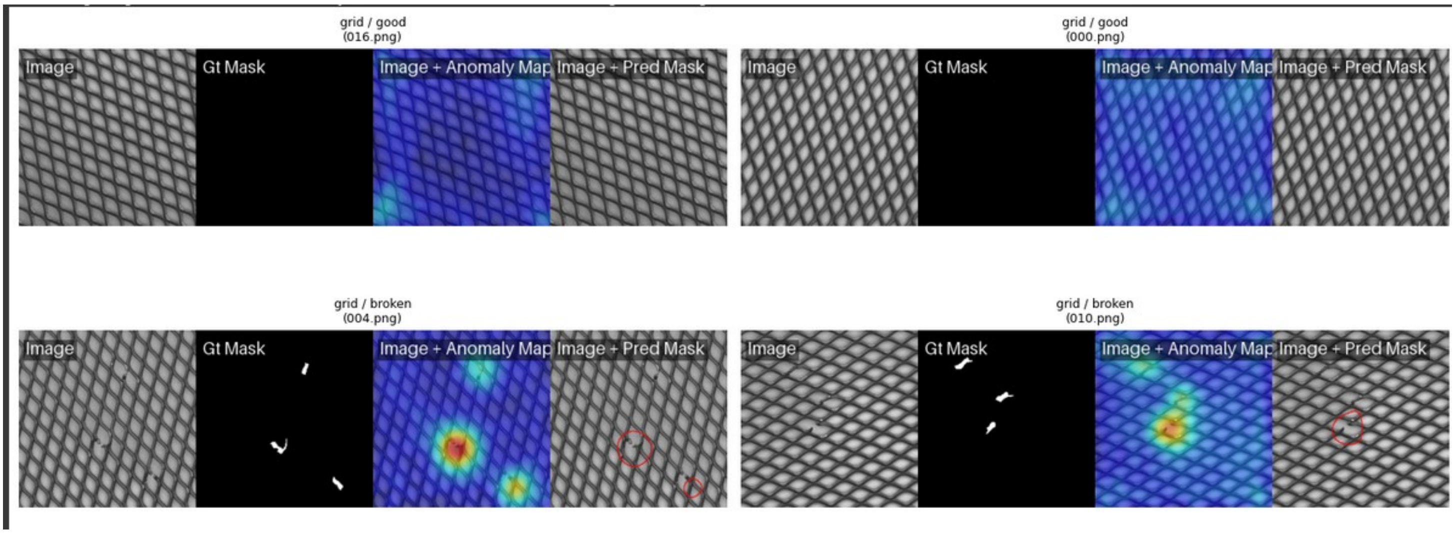
5. Lastly with these embedding the system those a few things:
   - First converts the embedding (vector) into a string so the server can read it
   - Then we run a SQL query to find similarities between vectors using **cosine distance operator** <=> from PGVECTOR
   - This converts the distance to similarity
   - I define some threshold (0.9) if the distance is above that threshold then the images are similar and it prints the top 10 similar images

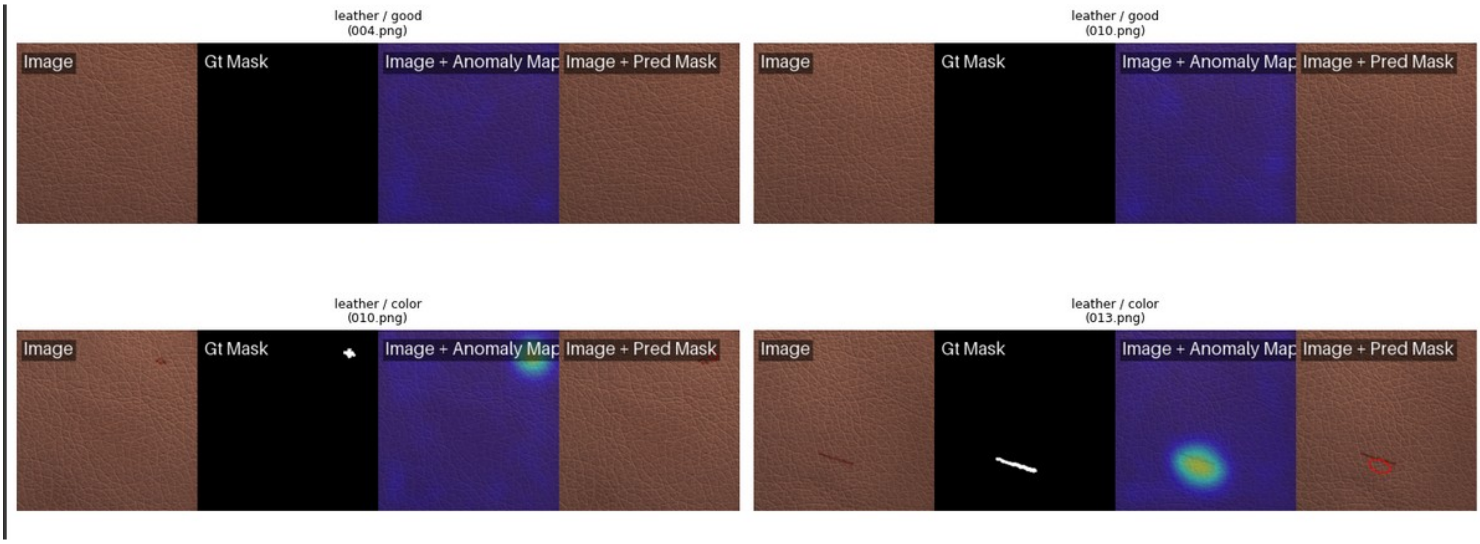
Results and Conclusion

This is the Area Under Receiver Operating Characteristic Curve (AUROC). This helps evaluate how well a model does at classification. In our case it was how well the model did at detecting anomaly detections. From the results we can see the model does extremely well at anomaly detection.



Here are some results from the model showing the difference between some anomalies and the good surfaces. The first pictures are grid between good (top) and broken (bottom). The next pictures is comparing leather between good (top) and color discoloration (bottom).





These are the results from using Postgres and finding similarities or anomaly images based on the inputted image.

```
(.venv) vscode → /workspaces/eng-ai-agents/assignments/assignment-2 (main) $ python Assingment2_Part2_Postgres.py

Query: leather_train_coreset_15609 (leather)

Normal image — showing top similar normals:
leather_train_coreset_15609 | leather    | Norm | Sim=1.0000
leather_train_coreset_4623 | leather    | Norm | Sim=0.9729
leather_train_coreset_5866 | leather    | Norm | Sim=0.9721
leather_train_coreset_14852 | leather    | Norm | Sim=0.9719
leather_train_coreset_7242 | leather    | Norm | Sim=0.9710
leather_train_coreset_3553 | leather    | Norm | Sim=0.9701
leather_train_coreset_14139 | leather    | Norm | Sim=0.9697
leather_train_coreset_10433 | leather    | Norm | Sim=0.9695
leather_train_coreset_13266 | leather    | Norm | Sim=0.9692
leather_train_coreset_4416 | leather    | Norm | Sim=0.9686
(.venv) vscode → /workspaces/eng-ai-agents/assignments/assignment-2 (main) $ python Assingment2_Part2_Postgres.py
Inserted 5 fake anomalies.

Query: tile_train_coreset_8861 (tile)

Normal image — showing top similar normals:
tile_train_coreset_8861   | tile       | Norm | Sim=1.0000
tile_train_coreset_6769   | tile       | Norm | Sim=0.9459
tile_train_coreset_14589  | tile       | Norm | Sim=0.9426
tile_train_coreset_15453  | tile       | Norm | Sim=0.9411
tile_train_coreset_2389   | tile       | Norm | Sim=0.9408
tile_train_coreset_14305  | tile       | Norm | Sim=0.9408
tile_train_coreset_12378  | tile       | Norm | Sim=0.9406
tile_train_coreset_14069  | tile       | Norm | Sim=0.9404
tile_train_coreset_6764   | tile       | Norm | Sim=0.9399
tile_train_coreset_10494  | tile       | Norm | Sim=0.9397
```

For this assignment we were tasked on deepening our understanding on how modern anomaly detection systems work. From using Patchcore and Anomalib for feature extraction. Using vector databases Postgres and PGVector for storing and parsing for similarity search. And of course evaluating the models performance. Overall, this assignment demonstrated how deep learning, feature embeddings, and vector databases can be combined into a practical and scalable anomaly detection system.