odel-with-cubes-from-a-robotic-arm

March 29, 2024

1 Simulation of production line with defects

In this notebook we will train a Anomalib model using the Anomalib API and our own dataset. This notebook is also part of the Dobot series notebooks.

1.0.1 Use case

Using the Dobot Magician we could simulate a production line system. Imagine we have a cubes factory and they need to know when a defect piece appear in the process. We know very well what is the aspecto of the normal cubes. Defects are coming no often and we need to put those defect cubes out of the production line.

Class	Yellow cube	Red cube	Green cube	Inferencing using Anomalib
Normal				
Abnormal				

Using Anomalib we are expecting to see this result.

2 Installing Anomalib

To install anomalib with the required dependencies, please follow the steps under `Install from source` on GitHub.

2.1 Imports

[1]:
```
"""501a_training_a_model_with_cubes_from_a_robotic_arm.ipynb."""

from pathlib import Path

from anomalib.data.utils import read_image
from anomalib.deploy import OpenVINOInferencer
```

2.2 Download dataset and Robot API/Driver

We should prepare the folder to save the dataset and the Dobot API and drivers. To download the dataset and the Dobot API and drivers we will use anomalib's `download_and_extract` utility function.

[2]:
```
from anomalib.data.utils import DownloadInfo, download_and_extract

dataset_download_info = DownloadInfo(
    name="cubes.zip",
    url="https://github.com/openvinotoolkit/anomalib/releases/download/dobot/
 ↪cubes.zip",
    hashsum="182ce0a48dabf452bf9a6aeb83132466088e30ed7a5c35d7d3a10a9fc11daac4",
)
api_download_info = DownloadInfo(
    name="dobot_api.zip",
    url="https://github.com/openvinotoolkit/anomalib/releases/download/dobot/
 ↪dobot_api.zip",
    hashsum="eb79bb9c6346be1628a0fe5e1196420dcc4e122ab1aa0d5abbc82f63236f0527",
)
download_and_extract(root=Path.cwd(), info=dataset_download_info)
download_and_extract(root=Path.cwd(), info=api_download_info)
```

```
cubes.zip: 6.99MB [00:01, 5.86MB/s]
dobot_api.zip: 3.69MB [00:00, 5.43MB/s]
```

2.2.1 Dataset: Cubes

Prepare your own dataset for normal and defect pieces.

[3]:
```
from anomalib.data import Folder
from anomalib import TaskType

datamodule = Folder(
    name="cubes",
    root=Path.cwd() / "cubes",
    normal_dir="normal",
    abnormal_dir="abnormal",
    normal_split_ratio=0.2,
    image_size=(256, 256),
    train_batch_size=32,
    eval_batch_size=32,
    task=TaskType.CLASSIFICATION,
)
datamodule.setup()

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

```
dict_keys(['image_path', 'label', 'image'])
```

[4]:
```
# Check image size
print(data["image"].shape)
```

```
torch.Size([32, 3, 256, 256])
```

2.3 Model

`anomalib` supports a wide range of unsupervised anomaly detection models. The table in this link shows the list of models currently supported by `anomalib` library.

2.3.1 Prepare the Model

We will use Padim model for this use case, which could be imported from `anomalib.models`.

[5]:
```
from anomalib.models import Padim

model = Padim(
    backbone="resnet18",
    layers=["layer1", "layer2", "layer3"],
)
```

2.4 Training

Now that we set up the datamodule and model, we could now train the model.

The final component to train the model is `Engine` object, which handles train/test/predict/export pipeline. Let's create the engine object to train the model.

[6]:
```
from anomalib.engine import Engine
from anomalib.utils.normalization import NormalizationMethod

engine = Engine(
    normalization=NormalizationMethod.MIN_MAX,
    threshold="F1AdaptiveThreshold",
    task=TaskType.CLASSIFICATION,
    image_metrics=["AUROC"],
    accelerator="auto",
    check_val_every_n_epoch=1,
    devices=1,
    max_epochs=1,
    num_sanity_val_steps=0,
    val_check_interval=1.0,
)

engine.fit(model=model, datamodule=datamodule)
```

```
/home/djameln/miniconda3/envs/anomalibv1source/lib/python3.10/site-
packages/torchmetrics/utilities/prints.py:36: UserWarning: Metric
`PrecisionRecallCurve` will save all targets and predictions in buffer. For
large datasets this may lead to large memory footprint.
  warnings.warn(*args, **kwargs)
GPU available: True (cuda), used: True
```

```
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
`Trainer(val_check_interval=1.0)` was configured so validation will run at the
end of the training epoch..
You are using a CUDA device ('NVIDIA GeForce RTX 3090') that has Tensor Cores.
To properly utilize them, you should set
`torch.set_float32_matmul_precision('medium' | 'high')` which will trade-off
precision for performance. For more details, read https://pytorch.org/docs/stabl
e/generated/torch.set_float32_matmul_precision.html#torch.set_float32_matmul_pre
cision
/home/djameln/miniconda3/envs/anomalibv1source/lib/python3.10/site-
packages/torchmetrics/utilities/prints.py:36: UserWarning: Metric `ROC` will
save all targets and predictions in buffer. For large datasets this may lead to
large memory footprint.
  warnings.warn(*args, **kwargs)
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]
/home/djameln/miniconda3/envs/anomalibv1source/lib/python3.10/site-
packages/lightning/pytorch/core/optimizer.py:180:
`LightningModule.configure_optimizers` returned `None`, this fit will run with
no optimizer

  | Name                  | Type                      | Params
-------------------------------------------------------------------
0 | model                 | PadimModel                | 2.8 M
1 | _transform            | Compose                   | 0
2 | normalization_metrics | MinMax                    | 0
3 | image_threshold       | F1AdaptiveThreshold       | 0
4 | pixel_threshold       | F1AdaptiveThreshold       | 0
5 | image_metrics         | AnomalibMetricCollection  | 0
6 | pixel_metrics         | AnomalibMetricCollection  | 0
-------------------------------------------------------------------
2.8 M     Trainable params
0         Non-trainable params
2.8 M     Total params
11.131    Total estimated model params size (MB)

Training: |          | 0/? [00:00<?, ?it/s]

/home/djameln/miniconda3/envs/anomalibv1source/lib/python3.10/site-
packages/lightning/pytorch/loops/optimization/automatic.py:129: `training_step`
returned `None`. If this was on purpose, ignore this warning…

Validation: |          | 0/? [00:00<?, ?it/s]

`Trainer.fit` stopped: `max_epochs=1` reached.
```

[7]:
```
# Validation
test_results = engine.test(model=model, datamodule=datamodule)
```

```
/home/djameln/miniconda3/envs/anomalibv1source/lib/python3.10/site-
packages/torchmetrics/utilities/prints.py:36: UserWarning: Metric
`PrecisionRecallCurve` will save all targets and predictions in buffer. For
large datasets this may lead to large memory footprint.
  warnings.warn(*args, **kwargs)
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]
Testing: |          | 0/? [00:00<?, ?it/s]
```

Test metric	DataLoader 0
image_AUROC	1.0

[8]:
```
from anomalib.deploy import ExportType

# Exporting model to OpenVINO
openvino_model_path = engine.export(
    model=model,
    export_type=ExportType.OPENVINO,
    export_root=str(Path.cwd()),
)
```

```
/home/djameln/miniconda3/envs/anomalibv1source/lib/python3.10/site-
packages/torch/onnx/_internal/jit_utils.py:307: UserWarning: Constant folding -
Only steps=1 can be constant folded for opset >= 10 onnx::Slice op. Constant
folding not applied. (Triggered internally at
../torch/csrc/jit/passes/onnx/constant_fold.cpp:179.)
  _C._jit_pass_onnx_node_shape_type_inference(node, params_dict, opset_version)
/home/djameln/miniconda3/envs/anomalibv1source/lib/python3.10/site-
packages/torch/onnx/utils.py:702: UserWarning: Constant folding - Only steps=1
can be constant folded for opset >= 10 onnx::Slice op. Constant folding not
applied. (Triggered internally at
../torch/csrc/jit/passes/onnx/constant_fold.cpp:179.)
  _C._jit_pass_onnx_graph_shape_type_inference(
/home/djameln/miniconda3/envs/anomalibv1source/lib/python3.10/site-
packages/torch/onnx/utils.py:1209: UserWarning: Constant folding - Only steps=1
can be constant folded for opset >= 10 onnx::Slice op. Constant folding not
applied. (Triggered internally at
../torch/csrc/jit/passes/onnx/constant_fold.cpp:179.)
  _C._jit_pass_onnx_graph_shape_type_inference(
```

2.5 OpenVINO Inference

Now that we trained and tested a model, we could check a single inference result using OpenVINO inferencer object. This will demonstrate how a trained model could be used for inference.

2.5.1 Load a Test Image

Let's read an image from the test set and perform inference using OpenVINO inferencer.

[9]:
```
from matplotlib import pyplot as plt

image_path = "./cubes/abnormal/input_20230210134059.jpg"
image = read_image(path="./cubes/abnormal/input_20230210134059.jpg")
plt.imshow(image)
```

[9]:
```
<matplotlib.image.AxesImage at 0x7f5648255420>
```

2.5.2 Load the OpenVINO Model

By default, the output files are saved into `results` directory. Let's check where the OpenVINO model is stored.

[10]:
```
metadata_path = openvino_model_path.parent / "metadata.json"
print(openvino_model_path.exists(), metadata_path.exists())
```

```
True True
```

[11]:
```
inferencer = OpenVINOInferencer(
    path=openvino_model_path,  # Path to the OpenVINO IR model.
    metadata=metadata_path,  # Path to the metadata file.
    device="CPU",  # We would like to run it on an Intel CPU.
)
```

2.5.3 Perform Inference

Predicting an image using OpenVINO inferencer is as simple as calling `predict` method.

[12]:
```
print(image.shape)
predictions = inferencer.predict(image=image)
```

```
(480, 640, 3)
```

where `predictions` contain any relevant information regarding the task type. For example, predictions for a segmentation model could contain image, anomaly maps, predicted scores, labels or masks.

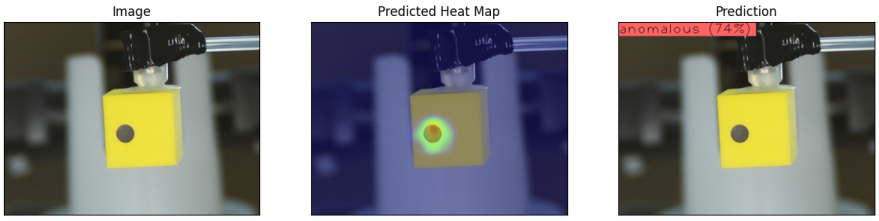
2.5.4 Visualizing Inference Results

`anomalib` provides a number of tools to visualize the inference results. Let's visualize the inference results using the `Visualizer` method.

[13]:
```
from anomalib.utils.visualization.image import ImageVisualizer,␣
↪VisualizationMode
from PIL import Image
```

```
visualizer = ImageVisualizer(mode=VisualizationMode.FULL, task=TaskType.
  ↪CLASSIFICATION)
output_image = visualizer.visualize_image(predictions)
Image.fromarray(output_image)
```

[13]:



Since `predictions` contain a number of information, we could specify which information we want to visualize. For example, if we want to visualize the predicted mask and the segmentation results, we could specify the task type as `TaskType.SEGMENTATION`, which would produce the following visualization.

[14]:
```
visualizer = ImageVisualizer(mode=VisualizationMode.FULL, task=TaskType.
  ↪SEGMENTATION)
output_image = visualizer.visualize_image(predictions)
Image.fromarray(output_image)
```

[14]: