

101-btech

March 29, 2024

1 Use BTech Dataset via API

2 Installing Anomalib

The easiest way to install anomalib is to use pip. You can install it from the command line using the following command:

```
[ ]: %pip install anomalib
```

2.1 Setting up the Dataset Directory

This cell is to ensure we change the directory to have access to the datasets.

```
[1]: from pathlib import Path

# NOTE: Provide the path to the dataset root directory.
# If the datasets is not downloaded, it will be downloaded
# to this directory.
dataset_root = Path.cwd().parent / "datasets" / "BTech"
```

2.2 Use BeanTech Dataset via API

```
[2]: # flake8: noqa
import numpy as np
from PIL import Image
from torchvision.transforms.v2 import Resize
from torchvision.transforms.v2.functional import to_pil_image

from anomalib.data.image.btech import BTech, BTechDataset
from anomalib import TaskType
```

2.2.1 Torch Dataset

2.2.2 DataModule

Anomalib data modules are based on PyTorch Lightning (PL)'s `LightningDataModule` class. This class handles all the boilerplate code related to subset splitting, and creating the dataset and

dataloader instances. A datamodule instance can be directly passed to a PL Trainer which is responsible for carrying out Anomalib's training/testing/inference pipelines.

In the current example, we will show how an Anomalib data module can be created for the BeanTech Dataset, and how we can obtain training and testing dataloaders from it.

To create a datamodule, we simply pass the path to the root folder of the dataset on the file system, together with some basic parameters related to pre-processing and image loading:

```
[3]: btech_datamodule = BTech(  
    root=dataset_root,  
    category="01",  
    image_size=256,  
    train_batch_size=32,  
    eval_batch_size=32,  
    num_workers=0,  
    task=TaskType.SEGMENTATION,  
)
```

For the illustrative purposes of the current example, we need to manually call the `prepare_data` and `setup` methods. Normally it is not necessary to call these methods explicitly, as the PL Trainer would call these automatically under the hood.

`prepare_data` checks if the dataset files can be found at the specified file system location. If not, it will download the dataset and place it in the folder.

`setup` applies the subset splitting and prepares the PyTorch dataset objects for each of the train/val/test subsets.

```
[4]: btech_datamodule.prepare_data()  
     btech_datamodule.setup()
```

After the datamodule has been set up, we can use it to obtain the dataloaders of the different subsets.

```
[5]: # Train images  
i, data = next(enumerate(btech_datamodule.train_dataloader()))  
print(data.keys(), data["image"].shape)
```

```
dict_keys(['image_path', 'label', 'image', 'mask']) torch.Size([32, 3, 1600,  
1600])
```

```
[6]: # Test images  
i, data = next(enumerate(btech_datamodule.test_dataloader()))  
print(data.keys(), data["image"].shape, data["mask"].shape)
```

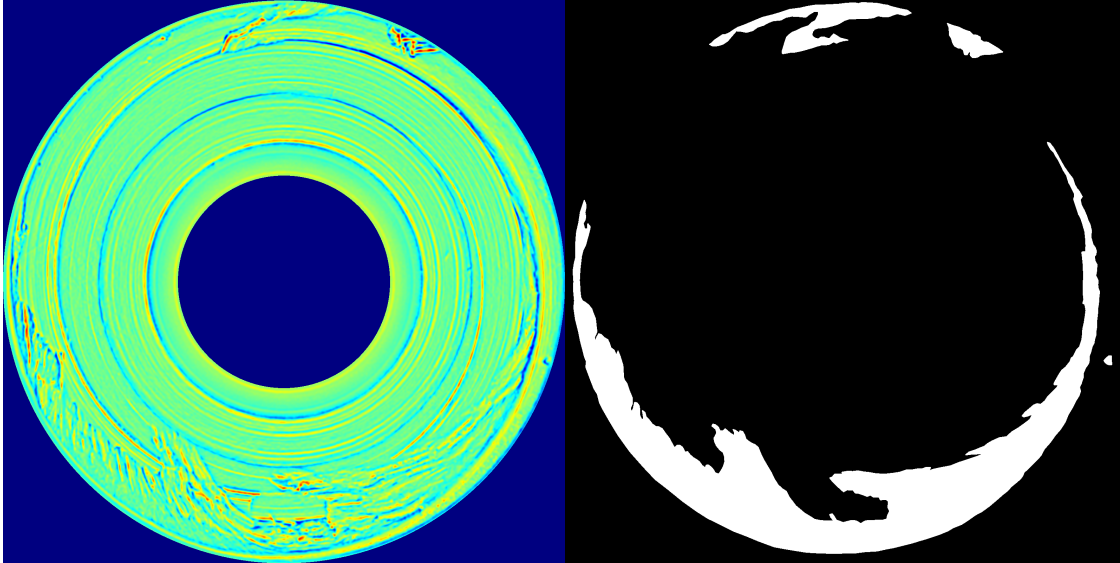
```
dict_keys(['image_path', 'label', 'image', 'mask']) torch.Size([32, 3, 1600,  
1600]) torch.Size([32, 1600, 1600])
```

As can be seen above, creating the dataloaders is pretty straightforward, which could be directly used for training/testing/inference. We could visualize samples from the dataloaders as well.

```
[7]: img = to_pil_image(data["image"][0].clone())
      msk = to_pil_image(data["mask"][0]).convert("RGB")

      Image.fromarray(np.hstack((np.array(img), np.array(msk))))
```

[7]:



2.2.3 Torch Dataset

In some cases it might be desirable to create a standalone PyTorch dataset without a PL data module. For example, this could be useful for training a PyTorch model outside Anomalib, so without the use of a PL Trainer instance. In such cases, the PyTorch Dataset instance can be instantiated directly.

```
[ ]: BTechDataset[??]
```

We can add some transforms that will be applied to the images using torchvision. Let's add a transform that resizes the input image to 256x256 pixels.

```
[8]: image_size = (256, 256)
      transform = Resize(image_size, antialias=True)
```

Classification Task

```
[9]: # BTechDataset Classification Train Set
      btech_dataset_classification_train = BTechDataset(
          root=dataset_root,
          category="01",
          transform=transform,
          split="train",
          task=TaskType.CLASSIFICATION,
```

```
)
btech_dataset_classification_train.samples.head()
```

```
[9]:
```

	path	split	label	\
0	/home/djameln/datasets/BTech/01	train	ok	
1	/home/djameln/datasets/BTech/01	train	ok	
2	/home/djameln/datasets/BTech/01	train	ok	
3	/home/djameln/datasets/BTech/01	train	ok	
4	/home/djameln/datasets/BTech/01	train	ok	

	image_path	\
0	/home/djameln/datasets/BTech/01/train/ok/0000.png	
1	/home/djameln/datasets/BTech/01/train/ok/0001.png	
2	/home/djameln/datasets/BTech/01/train/ok/0002.png	
3	/home/djameln/datasets/BTech/01/train/ok/0003.png	
4	/home/djameln/datasets/BTech/01/train/ok/0004.png	

	mask_path	label_index
0	/home/djameln/datasets/BTech/01/ground_truth/o...	0
1	/home/djameln/datasets/BTech/01/ground_truth/o...	0
2	/home/djameln/datasets/BTech/01/ground_truth/o...	0
3	/home/djameln/datasets/BTech/01/ground_truth/o...	0
4	/home/djameln/datasets/BTech/01/ground_truth/o...	0

```
[10]: sample = btech_dataset_classification_train[0]
print(sample.keys(), sample["image"].shape)
```

```
dict_keys(['image_path', 'label', 'image']) torch.Size([3, 256, 256])
```

As can be seen above, when we choose classification task and train split, the dataset only returns image. This is mainly because training only requires normal images and no labels. Now let's try test split for the classification task

```
[11]: # BTech Classification Test Set
btech_dataset_classification_test = BTechDataset(
    root=dataset_root,
    category="01",
    transform=transform,
    split="test",
    task=TaskType.CLASSIFICATION,
)
sample = btech_dataset_classification_test[0]
print(sample.keys(), sample["image"].shape, sample["image_path"],
      ↪sample["label"])
```

```
dict_keys(['image_path', 'label', 'image']) torch.Size([3, 256, 256])
/home/djameln/datasets/BTech/01/test/ko/0000.png 1
```

where a classification test sample returns image, image_path and label. image_path is used to

extract the filename when saving images.

Segmentation Task It is also possible to configure the BTech dataset for the segmentation task, where the dataset object returns image and ground-truth mask.

```
[12]: # BTech Segmentation Train Set
btech_dataset_segmentation_train = BTechDataset(
    root=dataset_root,
    category="01",
    transform=transform,
    split="train",
    task=TaskType.SEGMENTATION,
)
btech_dataset_segmentation_train.samples.head()
```

```
[12]:
```

	path	split	label	\
0	/home/djameln/datasets/BTech/01	train	ok	
1	/home/djameln/datasets/BTech/01	train	ok	
2	/home/djameln/datasets/BTech/01	train	ok	
3	/home/djameln/datasets/BTech/01	train	ok	
4	/home/djameln/datasets/BTech/01	train	ok	

	image_path	\
0	/home/djameln/datasets/BTech/01/train/ok/0000.png	
1	/home/djameln/datasets/BTech/01/train/ok/0001.png	
2	/home/djameln/datasets/BTech/01/train/ok/0002.png	
3	/home/djameln/datasets/BTech/01/train/ok/0003.png	
4	/home/djameln/datasets/BTech/01/train/ok/0004.png	

	mask_path	label_index
0	/home/djameln/datasets/BTech/01/ground_truth/o...	0
1	/home/djameln/datasets/BTech/01/ground_truth/o...	0
2	/home/djameln/datasets/BTech/01/ground_truth/o...	0
3	/home/djameln/datasets/BTech/01/ground_truth/o...	0
4	/home/djameln/datasets/BTech/01/ground_truth/o...	0

The above dataframe stores all the necessary information regarding the dataset. `__getitem__` method returns the corresponding information depending on the task type or train/test split.

```
[13]: # BTech Segmentation Test Set
btech_dataset_segmentation_test = BTechDataset(
    root=dataset_root,
    category="01",
    transform=transform,
    split="test",
    task=TaskType.SEGMENTATION,
)
sample = btech_dataset_segmentation_test[20]
```

```
print(sample.keys(), sample["image"].shape, sample["mask"].shape)
```

```
dict_keys(['image_path', 'label', 'image', 'mask']) torch.Size([3, 256, 256])  
torch.Size([256, 256])
```

Let's visualize the image and the mask...

```
[14]: # img = Image.open(sample["image_path"]).resize(image_size)  
img = to_pil_image(sample["image"].clone())  
msk = to_pil_image(sample["mask"]).convert("RGB")  
  
Image.fromarray(np.hstack((np.array(img), np.array(msk))))
```

[14]:

