Train a Model via API

This notebook demonstrates how to train, test and infer the FastFlow model via Anomalib API. Compared to the CLI entrypoints such as `tools/<train, test, inference>.py, the API offers more flexibility such as modifying the existing model or designing custom approaches.

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
```

Setting up the Dataset Directory

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

```
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" / "MVTec"
```

Imports

```
import numpy as np
from lightning.pytorch.callbacks import EarlyStopping, ModelCheckpoint
from matplotlib import pyplot as plt
from PIL import Image
from torch.utils.data import DataLoader

from anomalib.data import PredictDataset, MVTec
from anomalib.engine import Engine
from anomalib.models import Fastflow
from anomalib.utils.post_processing import superimpose_anomaly_map
from anomalib import TaskType
```

Data Module

To train the model end-to-end, we do need to have a dataset. In our previous notebooks, we demonstrate how to initialize benchmark- and custom datasets. In this tutorial, we will use MVTec AD DataModule. We assume that `datasets` directory is created in the `anomalib` root directory and `MVTec` dataset is located in `datasets` directory.

Before creating the dataset, let's define the task type that we will be working on. In this notebook, we will be working on a segmentation task. Therefore the `task` variable would be:

```
task = TaskType.SEGMENTATION

datamodule = MVTec(
    root=dataset_root,
    category="bottle",
    image_size=256,
    train_batch_size=32,
    eval_batch_size=32,
    num_workers=0,
    task=task,
)
```

FastFlow Model

Now that we have created the MVTec datamodule, we could create the FastFlow model. We could start with printing its docstring.

```
Fastflow??

Init signature:
Fastflow(
    backbone: str = 'resnet18',
    pre_trained: bool = True,
    flow_steps: int = 8,
    conv3x3_only: bool = False,
    hidden_ratio: float = 1.0,
) -> None
Source:
class Fastflow(AnomalyModule):
    """PL Lightning Module for the FastFlow algorithm.

    Args:
        input_size (tuple[int, int]): Model input size.
            Defaults to ``(256, 256)``.
        backbone (str): Backbone CNN network
            Defaults to ``resnet18``.
        pre_trained (bool, optional): Boolean to check whether to use
a pre_trained backbone.
            Defaults to ``True``.
        flow_steps (int, optional): Flow steps.
            Defaults to ``8``.
        conv3x3_only (bool, optinoal): Use only conv3x3 in fast_flow
model.
            Defaults to ``False``.
        hidden_ratio (float, optional): Ratio to calculate hidden var
channels.
```

```
            Defaults to ``1.0``.
    """

    def __init__(
        self,
        backbone: str = "resnet18",
        pre_trained: bool = True,
        flow_steps: int = 8,
        conv3x3_only: bool = False,
        hidden_ratio: float = 1.0,
    ) -> None:
        super().__init__()

        self.backbone = backbone
        self.pre_trained = pre_trained
        self.flow_steps = flow_steps
        self.conv3x3_only = conv3x3_only
        self.hidden_ratio = hidden_ratio

        self.loss = FastflowLoss()

        self.model: FastflowModel

    def _setup(self) -> None:
        assert self.input_size is not None, "Fastflow needs input size
to build torch model."
        self.model = FastflowModel(
            input_size=self.input_size,
            backbone=self.backbone,
            pre_trained=self.pre_trained,
            flow_steps=self.flow_steps,
            conv3x3_only=self.conv3x3_only,
            hidden_ratio=self.hidden_ratio,
        )

    def training_step(self, batch: dict[str, str | torch.Tensor],
*args, **kwargs) -> STEP_OUTPUT:
        """Perform the training step input and return the loss.

        Args:
            batch (batch: dict[str, str | torch.Tensor]): Input batch
            args: Additional arguments.
            kwargs: Additional keyword arguments.

        Returns:
            STEP_OUTPUT: Dictionary containing the loss value.
        """
        del args, kwargs  # These variables are not used.

        hidden_variables, jacobians = self.model(batch["image"])
```

```
        loss = self.loss(hidden_variables, jacobians)
        self.log("train_loss", loss.item(), on_epoch=True,
prog_bar=True, logger=True)
        return {"loss": loss}

    def validation_step(self, batch: dict[str, str | torch.Tensor],
*args, **kwargs) -> STEP_OUTPUT:
        """Perform the validation step and return the anomaly map.

        Args:
            batch (dict[str, str | torch.Tensor]): Input batch
            args: Additional arguments.
            kwargs: Additional keyword arguments.

        Returns:
            STEP_OUTPUT | None: batch dictionary containing anomaly-
maps.
        """
        del args, kwargs  # These variables are not used.

        anomaly_maps = self.model(batch["image"])
        batch["anomaly_maps"] = anomaly_maps
        return batch

    @property
    def trainer_arguments(self) -> dict[str, Any]:
        """Return FastFlow trainer arguments."""
        return {"gradient_clip_val": 0, "num_sanity_val_steps": 0}

    def configure_optimizers(self) -> torch.optim.Optimizer:
        """Configure optimizers for each decoder.

        Returns:
            Optimizer: Adam optimizer for each decoder
        """
        return optim.Adam(
            params=self.model.parameters(),
            lr=0.001,
            weight_decay=0.00001,
        )

    @property
    def learning_type(self) -> LearningType:
        """Return the learning type of the model.

        Returns:
            LearningType: Learning type of the model.
        """
        return LearningType.ONE_CLASS
File:
~/anomalib/src/anomalib/models/image/fastflow/lightning_model.py
```

```
Type:           ABCMeta
Subclasses:
```

```
model = Fastflow(backbone="resnet18", flow_steps=8)
```

Callbacks

To train the model properly, we will to add some other "non-essential" logic such as saving the weights, early-stopping, normalizing the anomaly scores and visualizing the input/output images. To achieve these we use `Callbacks`. Anomalib has its own callbacks and also supports PyTorch Lightning's native callbacks. So, let's create the list of callbacks we want to execute during the training.

```
callbacks = [
    ModelCheckpoint(
        mode="max",
        monitor="pixel_AUROC",
    ),
    EarlyStopping(
        monitor="pixel_AUROC",
        mode="max",
        patience=3,
    ),
]
```

Training

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

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

```
engine = Engine(
    callbacks=callbacks,
    pixel_metrics="AUROC",
    accelerator="auto",  # \<"cpu", "gpu", "tpu", "ipu", "hpu",
"auto">,
    devices=1,
    logger=False,
)
```

`Trainer` object has number of options that suit all specific needs. For more details, refer to Lightning Documentation to see how it could be tweaked to your needs.

Let's train the model now.

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

The training has finished after 12 epochs. This is because, we set the `EarlyStopping` criteria with a patience of 3, which terminated the training after `pixel_AUROC` stopped improving. If we increased the `patience`, the training would continue further.

Testing

Now that we trained the model, we could test the model to check the overall performance on the test set. We will also be writing the output of the test images to a file since we set `VisualizerCallback` in `callbacks`.

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

```
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0,1]
```

```
{"model_id":"4b40cd5a1e094248b521f07ef14291de","version_major":2,"vers
ion_minor":0}
```

Test metric	DataLoader 0
image_AUROC	1.0
image_F1Score	1.0
pixel_AUROC	0.9769068956375122

```
[{'pixel_AUROC': 0.9769068956375122, 'image_AUROC': 1.0,
'image_F1Score': 1.0}]
```

Inference

Since we have a trained model, we could infer the model on an individual image or folder of images. Anomalib has an `PredictDataset` to let you create an inference dataset. So let's try it.

```
inference_dataset = PredictDataset(path=dataset_root /
"bottle/test/broken_large/000.png")
inference_dataloader = DataLoader(dataset=inference_dataset)
```

We could utilize `Trainer`'s `predict` method to infer, and get the outputs to visualize

```
predictions = engine.predict(model=model,
dataloaders=inference_dataloader)[0]
```

`predictions` contain image, anomaly maps, predicted scores, labels and masks. These are all stored in a dictionary. We could check this by printing the `prediction` keys.

```
print(
    f'Image Shape: {predictions["image"].shape},\n'
    'Anomaly Map Shape: {predictions["anomaly_maps"].shape}, \n'
```

```
    'Predicted Mask Shape: {predictions["pred_masks"].shape}',
)
```

```
Image Shape: torch.Size([1, 3, 256, 256]),
Anomaly Map Shape: {predictions["anomaly_maps"].shape},
Predicted Mask Shape: {predictions["pred_masks"].shape}
```

Visualization

To properly visualize the predictions, we will need to perform some post-processing operations.

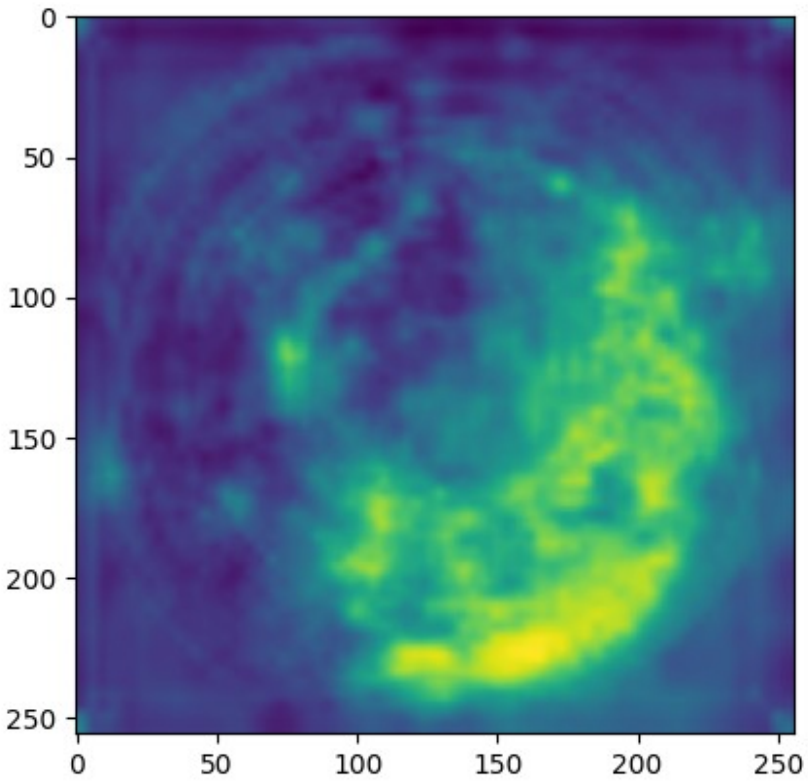
Let's first show the input image. To do so, we will use `image_path` key from the `predictions` dictionary, and read the image from path. Note that `predictions` dictionary already contains `image`. However, this is the normalized image with pixel values between 0 and 1. We will use the original image to visualize the input image.

```
image_path = predictions["image_path"][0]
image_size = predictions["image"].shape[-2:]
image = np.array(Image.open(image_path).resize(image_size))
```

The first output of the predictions is the anomaly map. As can be seen above, it's also a torch tensor and of size `torch.Size([1, 1, 256, 256])`. We therefore need to convert it to numpy and squeeze the dimensions to make it `256x256` output to visualize.

```
anomaly_map = predictions["anomaly_maps"][0]
anomaly_map = anomaly_map.cpu().numpy().squeeze()
plt.imshow(anomaly_map)
```

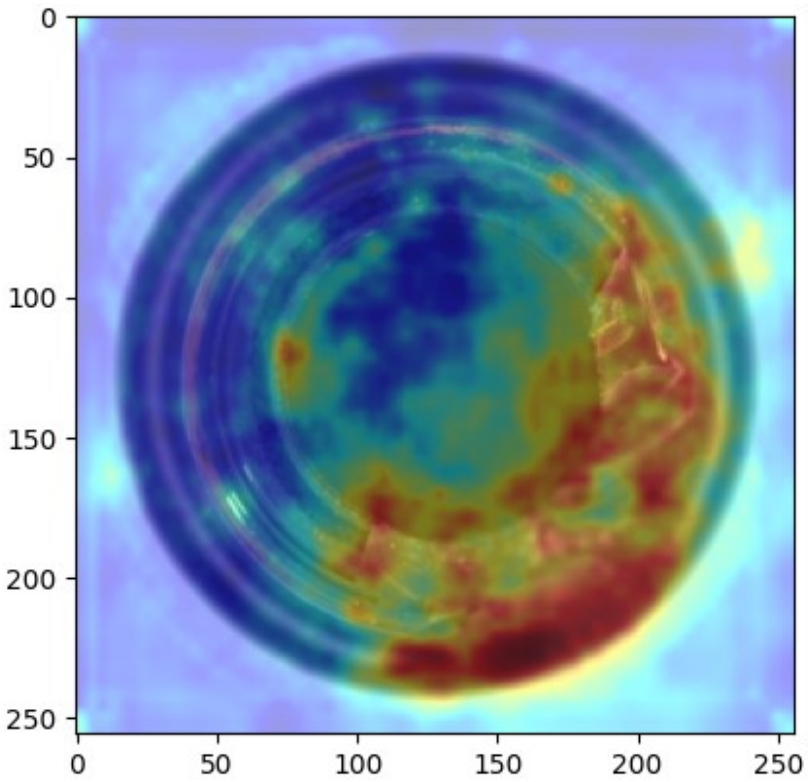
```
<matplotlib.image.AxesImage at 0x7fa298ad6fb0>
```



We could superimpose (overlay) the anomaly map on top of the original image to get a heat map. Anomalib has a built-in function to achieve this. Let's try it.

```
heat_map = superimpose_anomaly_map(anomaly_map=anomaly_map,
image=image, normalize=True)
plt.imshow(heat_map)
```

```
<matplotlib.image.AxesImage at 0x7fa298a71f00>
```



`predictions` also contains prediction scores and labels.

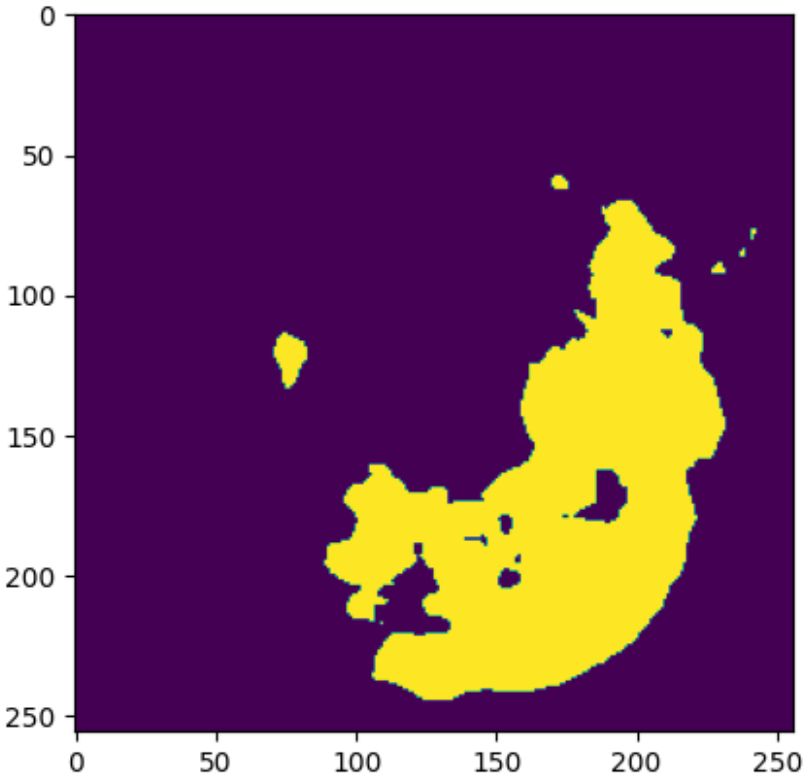
```
pred_score = predictions["pred_scores"][0]
pred_labels = predictions["pred_labels"][0]
print(pred_score, pred_labels)
```

```
tensor(0.6486) tensor(True)
```

The last part of the predictions is the mask that is predicted by the model. This is a boolean mask containing True/False for the abnormal/normal pixels, respectively.

```
pred_masks = predictions["pred_masks"][0].squeeze().cpu().numpy()
plt.imshow(pred_masks)
```

```
<matplotlib.image.AxesImage at 0x7fa298a016f0>
```



That wraps it! In this notebook, we show how we could train, test and finally infer a FastFlow model using Anomalib API.