Deployment

Contents

Functions for Inference and model deployment.

class **anomalib.deploy.ExportType**(***value***, ***names**=None*, ***, ***module**=None*, ***qualname**=None*, ***type**=None*, ***start**=1*, ***boundary**=None*)

Bases: `str`, `Enum`

Model export type.

Examples

```
>>> from anomalib.deploy import ExportType
>>> ExportType.ONNX
'onnx'
>>> ExportType.OPENVINO
'openvino'
>>> ExportType.TORCH
'torch'
```

class **anomalib.deploy.OpenVINOInferencer**(***path***, ***metadata**=None*, ***device**='AUTO'*, ***task**=None*, ***config**=None*)

Bases: `Inferencer`

OpenVINO implementation for the inference.

Parameters:

- **path** (*str* | *Path*) – Path to the openvino onnx, xml or bin file.
- **metadata** (*str* | *Path* | *dict, optional*) – Path to metadata file or a dict object defining the metadata. Defaults to `None`.
- **device** (*str* | *None, optional*) – Device to run the inference on (AUTO, CPU, GPU, NPU). Defaults to `AUTO`.
- **task** (*TaskType* | *None, optional*) – Task type. Defaults to `None`.
- **config** (*dict* | *None, optional*) – Configuration parameters for the inference Defaults to `None`.

Examples

Assume that we have an OpenVINO IR model and metadata files in the following structure:

```
$ tree weights
./weights
├── model.bin
├── model.xml
└── metadata.json
```

We could then create `OpenVINOInferencer` as follows:

```
>>> from anomalib.deploy.inferencers import OpenVINOInferencer
>>> inferencer = OpenVINOInferencer(
...     path="weights/model.xml",
...     metadata="weights/metadata.json",
...     device="CPU",
... )
```

This will ensure that the model is loaded on the `CPU` device and the metadata is loaded from the `metadata.json` file. To make a prediction, we can simply call the `predict` method:

```
>>> prediction = inferencer.predict(image="path/to/image.jpg")
```

Alternatively we can also pass the image as a PIL image or numpy array:

```
>>> from PIL import Image
>>> image = Image.open("path/to/image.jpg")
>>> prediction = inferencer.predict(image=image)
```

```
>>> import numpy as np
>>> image = np.random.rand(224, 224, 3)
>>> prediction = inferencer.predict(image=image)
```

`prediction` will be an `ImageResult` object containing the prediction results. For example, to visualize the heatmap, we can do the following:

```
>>> from matplotlib import pyplot as plt
>>> plt.imshow(result.heatmap)
```

It is also possible to visualize the true and predicted masks if the task is `TaskType.SEGMENTATION`:

```
>>> plt.imshow(result.gt_mask)
>>> plt.imshow(result.pred_mask)
```

`forward`(*image*)

Forward-Pass input tensor to the model.

Parameters:

image (*np.ndarray*) – Input tensor.

Returns:

Output predictions.

Return type:

np.ndarray

`load_model`(*path*)

Load the OpenVINO model.

Parameters:

path (*str* | *Path* | *tuple[bytes, bytes]*) – Path to the onnx or xml and bin files or tuple of .xml and .bin data as bytes.

Returns:

Input and Output blob names

together with the Executable network.

Return type:

[tuple[str, str, ExecutableNetwork]]

`post_process(predictions, metadata=None)`

Post process the output predictions.

Parameters:

- **predictions** (*np.ndarray*) – Raw output predicted by the model.
- **metadata** (*Dict, optional*) – Metadata. Post-processing step sometimes requires additional metadata such as image shape. This variable comprises such info. Defaults to None.

Returns:

Post processed prediction results.

Return type:

dict[str, Any]

`pre_process(image)`

Pre-process the input image by applying transformations.

Parameters:

image (*np.ndarray*) – Input image.

Returns:

pre-processed image.

Return type:

np.ndarray

`predict(image, metadata=None)`

Perform a prediction for a given input image.

The main workflow is (i) pre-processing, (ii) forward-pass, (iii) post-process.

Parameters:

- **image** (*Union[str, np.ndarray]*) – Input image whose output is to be predicted. It could be either a path to image or numpy array itself.
- **metadata** (`dict`[`str`, `Any`] | `None`) – Metadata information such as shape, threshold.

Returns:

Prediction results to be visualized.

Return type:

ImageResult

`class anomalib.deploy.TorchInferencer(path, device='auto')`

Bases: `Inferencer`

PyTorch implementation for the inference.

Parameters:

- **path** (*str* | *Path*) – Path to Torch model weights.
- **device** (*str*) – Device to use for inference. Options are `auto`, `cpu`, `cuda`. Defaults to `auto`.

Examples

Assume that we have a Torch `pt` model and metadata files in the following structure:

```
>>> from anomalib.deploy.inferencers import TorchInferencer
>>> inferencer = TorchInferencer(path="path/to/torch/model.pt", device="cpu")
```

This will ensure that the model is loaded on the `CPU` device. To make a prediction, we can simply call the `predict` method:

```
>>> from anomalib.data.utils import read_image
>>> image = read_image("path/to/image.jpg")
>>> result = inferencer.predict(image)
```

`result` will be an `ImageResult` object containing the prediction results. For example, to visualize the heatmap, we can do the following:

```
>>> from matplotlib import pyplot as plt
>>> plt.imshow(result.heatmap)
```

It is also possible to visualize the true and predicted masks if the task is `TaskType.SEGMENTATION`:

```
>>> plt.imshow(result.gt_mask)
>>> plt.imshow(result.pred_mask)
```

`forward`(*image*)

Forward-Pass input tensor to the model.

Parameters:

image (*torch.Tensor*) – Input tensor.

Returns:

Output predictions.

Return type:

Tensor

`load_model`(*path*)

Load the PyTorch model.

Parameters:

path (*str* | *Path*) – Path to the Torch model.

Returns:

Torch model.

Return type:

(nn.Module)

`post_process(predictions, metadata=None)`

Post process the output predictions.

Parameters:

- **predictions** (*Tensor | list[torch.Tensor] | dict[str, torch.Tensor]*) – Raw output predicted by the model.
- **metadata** (*dict, optional*) – Meta data. Post-processing step sometimes requires additional meta data such as image shape. This variable comprises such info. Defaults to None.

Returns:

Post processed prediction results.

Return type:

dict[str, str | float | np.ndarray]

`pre_process(image)`

Pre process the input image.

Parameters:

image (*np.ndarray*) – Input image

Returns:

pre-processed image.

Return type:

Tensor

`predict(image, metadata=None)`

Perform a prediction for a given input image.

The main workflow is (i) pre-processing, (ii) forward-pass, (iii) post-process.

Parameters:

- **image** (*Union[str, np.ndarray]*) – Input image whose output is to be predicted. It could be either a path to image or numpy array itself.
- **metadata** (`dict`[`str`, `Any`] | `None`) – Metadata information such as shape, threshold.

Returns:

Prediction results to be visualized.

Return type:

ImageResult

`anomalib.deploy.export_to_onnx(model, export_root, transform=None, task=None, export_type=ExportType.ONNX)`

Export model to onnx.

Parameters:

- **model** (*AnomalyModule*) – Model to export.
- **export_root** (*Path*) – Path to the root folder of the exported model.
- **transform** (*Transform, optional*) – Input transforms used for the model. If not provided, the transform is taken from the model. Defaults to `None`.
- **task** (*TaskType* | *None*) – Task type. Defaults to `None`.
- **export_type** (*ExportType*) – Mode to export the model. Since this method is used by OpenVINO export as well, we need to pass the export type so that the right export path is created. Defaults to `ExportType.ONNX`.

Returns:

Path to the exported onnx model.

Return type:

Path

Examples

Export the Lightning Model to ONNX:

```
>>> from anomalib.models import Patchcore
>>> from anomalib.data import Visa
>>> from anomalib.deploy import export_to_onnx
...
>>> datamodule = Visa()
>>> model = Patchcore()
...
>>> export_to_onnx(
...     model=model,
...     export_root="path/to/export",
...     transform=datamodule.test_data.transform,
...     task=datamodule.test_data.task
... )
```

Using Custom Transforms: This example shows how to use a custom `Compose` object for the `transform` argument.

```
>>> export_to_onnx(
...     model=model,
...     export_root="path/to/export",
...     task="segmentation",
... )
```

anomalib.deploy.export_to_openvino(*export_root*, *model*, *transform*=*None*, *ov_args*=*None*, *task*=*None*)

Convert onnx model to OpenVINO IR.

Parameters:

- **export_root** (*Path*) – Path to the export folder.
- **model** (*AnomalyModule*) – AnomalyModule to export.
- **transform** (*Transform, optional*) – Input transforms used for the model. If not provided, the transform is taken from the model. Defaults to `None`.
- **ov_args** (`dict`[`str`, `Any`] | `None`) – Model optimizer arguments for OpenVINO model conversion. Defaults to `None`.
- **task** (*TaskType* | *None*) – Task type. Defaults to `None`.

Returns:

Path to the exported onnx model.

Return type:

Path

Raises:

ModuleNotFoundError – If OpenVINO is not installed.

Returns:

Path to the exported OpenVINO IR.

Return type:

Path

Examples

Export the Lightning Model to OpenVINO IR: This example demonstrates how to export the Lightning Model to OpenVINO IR.

```
>>> from anomalib.models import Patchcore
>>> from anomalib.data import Visa
>>> from anomalib.deploy import export_to_openvino
...
>>> datamodule = Visa()
>>> model = Patchcore()
...
>>> export_to_openvino(
...     export_root="path/to/export",
...     model=model,
...     input_size=(224, 224),
...     transform=datamodule.test_data.transform,
...     task=datamodule.test_data.task
... )
```

Using Custom Transforms: This example shows how to use a custom `Transform` object for the `transform` argument.

```
>>> from torchvision.transforms.v2 import Resize
>>> transform = Resize(224, 224)
...
>>> export_to_openvino(
...     export_root="path/to/export",
...     model=model,
...     transform=transform,
...     task="segmentation",
... )
```

`anomalib.deploy.export_to_torch(model, export_root, transform=None,`

***task*=*None*)**

Export AnomalibModel to torch.

Parameters:

- **model** (*AnomalyModule*) – Model to export.
- **export_root** (*Path*) – Path to the output folder.
- **transform** (*Transform, optional*) – Input transforms used for the model. If not provided, the transform is taken from the model. Defaults to `None`.
- **task** (*TaskType* | *None*) – Task type. Defaults to `None`.

Returns:

Path to the exported pytorch model.

Return type:

Path

Examples

Assume that we have a model to train and we want to export it to torch format.

```
>>> from anomalib.data import Visa
>>> from anomalib.models import Patchcore
>>> from anomalib.engine import Engine
...
>>> datamodule = Visa()
>>> model = Patchcore()
>>> engine = Engine()
...
>>> engine.fit(model, datamodule)
```

Now that we have a model trained, we can export it to torch format.

```
>>> from anomalib.deploy import export_to_torch
...
>>> export_to_torch(
...     model=model,
...     export_root="path/to/export",
...     transform=datamodule.test_data.transform,
...     task=datamodule.test_data.task,
... )
```

Previous
CLI

Next
How to Guide