U-Flow

Contents

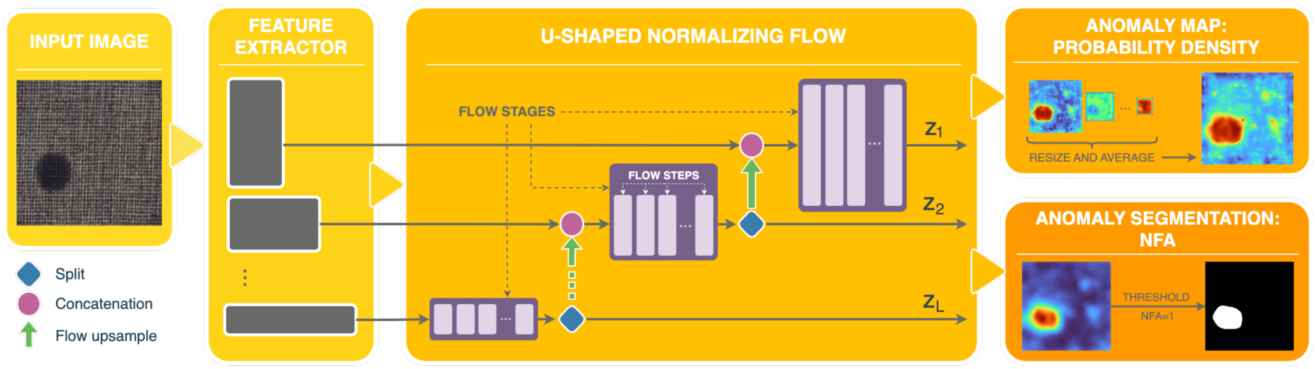
This is the implementation of the U-Flow paper.

Model Type: Segmentation

Description

U-Flow is a U-Shaped normalizing flow-based probability distribution estimator. The method consists of three phases. (1) Multi-scale feature extraction: a rich multi-scale representation is obtained with MSCaiT, by combining pre-trained image Transformers acting at different image scales. It can also be used any other feature extractor, such as ResNet. (2) U-shaped Normalizing Flow: by adapting the widely used U-like architecture to NFs, a fully invertible architecture is designed. This architecture is capable of merging the information from different scales while ensuring independence both intra- and inter-scales. To make it fully invertible, split and invertible up-sampling operations are used. (3) Anomaly score and segmentation computation: besides generating the anomaly map based on the likelihood of test data, we also propose to adapt the a contrario framework to obtain an automatic threshold by controlling the allowed number of false alarms.

Architecture



U-Flow torch model.

class **anomalib.models.image.uflow.torch_model.AffineCouplingSubnet**(*kernel_size*, *subnet_channels_ratio*)

Bases: `object`

Class for building the Affine Coupling subnet.

It is passed as an argument to the *AllInOneBlock* module.

Parameters:

- **kernel_size** (*int*) – Kernel size.
- **subnet_channels_ratio** (*float*) – Subnet channels ratio.

class **anomalib.models.image.uflow.torch_model.UflowModel**(*input_size=(448, 448)*, *flow_steps=4*, *backbone='mcait'*, *affine_clamp=2.0*, *affine_subnet_channels_ratio=1.0*, *permute_soft=False*)

Bases: `Module`

U-Flow model.

Parameters:

- **input_size** (*tuple[int, int]*) – Input image size.
- **flow_steps** (*int*) – Number of flow steps.
- **backbone** (*str*) – Backbone name.
- **affine_clamp** (*float*) – Affine clamp.
- **affine_subnet_channels_ratio** (*float*) – Affine subnet channels ratio.
- **permute_soft** (*bool*) – Whether to use soft permutation.

`build_flow(flow_steps)`

Build the flow model.

First we start with the input nodes, which have to match the feature extractor output. Then, we build the U-Shaped flow. Starting from the bottom (the coarsest scale), the flow is built as follows:

> 1. Pass the input through a Flow Stage (*build_flow_stage*).
> 2. Split the output of the flow stage into two parts, one that goes directly to the output,
> 3. and the other is up-sampled, and will be concatenated with the output of the next flow stage (next scale)
> 4. Repeat steps 1-3 for the next scale.

Finally, we build the Flow graph using the input nodes, the flow stages, and the output nodes.

Parameters:
flow_steps (*int*) – Number of flow steps.

Returns:
Flow model.

Return type:
ff.GraphINN

`build_flow_stage(in_node, flow_steps, condition_node=None)`

Build a flow stage, which is a sequence of flow steps.

Each flow stage is essentially a sequence of *flow_steps* Glow blocks (*AllInOneBlock*).

Parameters:

- **in_node** (*ff.Node*) – Input node.
- **flow_steps** (*int*) – Number of flow steps.
- **condition_node** (*ff.Node*) – Condition node.

Returns:

List of flow steps.

Return type:

List[ff.Node]

`encode(features)`

Return

Return type:

`tuple`[`Tensor`, `Tensor`]

`forward(image)`

Return anomaly map.

Return type:

`Tensor`

U-Flow: A U-shaped Normalizing Flow for Anomaly Detection with Unsupervised Threshold.

https://arxiv.org/pdf/2211.12353.pdf

`class anomalib.models.image.uflow.lightning_model.Uflow(backbone='mcait', flow_steps=4, affine_clamp=2.0, affine_subnet_channels_ratio=1.0, permute_soft=False)`

Bases: `AnomalyModule`

PL Lightning Module for the UFLOW algorithm.

`configure_optimizers()`

Return optimizer and scheduler.

Return type:

`tuple`[`list`[`LightningOptimizer`], `list`[`LRScheduler`]]

`configure_transforms(image_size=None)`

Default transform for Padim.

Return type:

`Transform`

property `learning_type`: *LearningType*

Return the learning type of the model.

Returns:

Learning type of the model.

Return type:

LearningType

property `trainer_arguments`: *dict[str, Any]*

Return EfficientAD trainer arguments.

`training_step(batch, *args, **kwargs)`

Training step.

Return type:

`Union`[`Tensor`, `Mapping`[`str`, `Any`], `None`]

`validation_step(batch, *args, **kwargs)`

Validation step.

Return type:

`Union`[`Tensor`, `Mapping`[`str`, `Any`], `None`]

UFlow Anomaly Map Generator Implementation.

class `anomalib.models.image.uflow.anomaly_map.AnomalyMapGenerator(input_size)`

Bases: `Module`

Generate Anomaly Heatmap and segmentation.

static `binomial_test(z, window_size, probability_thr, high_precision=False)`

The binomial test applied to validate or reject the null hypothesis that the pixel is normal.

The null hypothesis is that the pixel is normal, and the alternative hypothesis is that the pixel is anomalous. The binomial test is applied to a window around the pixel, and the number of pixels in the window that ares anomalous is compared to the number of pixels that are expected to be anomalous under the null hypothesis.

Parameters:

- **z** (`Tensor`) – Latent variable from the UFlow model. Tensor of shape (N, Cl, Hl, Wl), where N is the batch size, Cl is
- **channels** (*the number of*) –
- **variables** (*and Hl and Wl are the height and width of the latent*) –
- **respectively.** –
- **window_size** (*int*) – Window size for the binomial test.
- **probability_thr** (`float`) – Probability threshold for the binomial test.
- **high_precision** (`bool`) – Whether to use high precision for the binomial test.

Return type:

`Tensor`

Returns:

Log of the probability of the null hypothesis.

`compute_anomaly_map(latent_variables)`

Generate a likelihood-based anomaly map, from latent variables.

Parameters:

- **latent_variables** (`list`[`Tensor`]) – List of latent variables from the UFlow model. Each element is a tensor of shape
- **(N** –
- **Cl** –
- **Hl** –
- **Wl)** –
- **size** (*where N is the batch*) –
- **channels** (*Cl is the number of*) –
- **and** (*and Hl and Wl are the height*) –
- **variables** (*width of the latent*) –
- **respectively** –
- **l.** (*for each scale*) –

Return type:

`Tensor`

Returns:

Final Anomaly Map. Tensor of shape (N, 1, H, W), where N is the batch size, and H and W are the height and width of the input image, respectively.

`compute_anomaly_mask(z, window_size=7, binomial_probability_thr=0.5, high_precision=False)`

This method is not used in the basic functionality of training and testing.

It is a bit slow, so we decided to leave it as an option for the user. It is included as it is part of the U-Flow paper, and can be called separately if an unsupervised anomaly segmentation is needed.

Generate an anomaly mask, from latent variables. It is based on the NFA (Number of False Alarms) method, which is a statistical method to detect anomalies. The NFA is computed as the log of the probability of the null hypothesis, which is that all pixels are normal. First, we compute a list of candidate pixels, with suspiciously high values of z^2, by applying a binomial test to each pixel, looking at a window around it. Then, to compute the NFA values (actually the log-NFA), we evaluate how probable is that a pixel belongs to the normal distribution. The null-hypothesis is that under normality assumptions, all candidate pixels are uniformly distributed. Then, the detection is based on the concentration of candidate pixels.

Parameters:

- **z** (*list[torch.Tensor]*) – List of latent variables from the UFlow model. Each element is a tensor of shape (N, Cl, Hl, Wl), where N is the batch size, Cl is the number of channels, and Hl and Wl are the height and width of the latent variables, respectively, for each scale l.
- **window_size** (*int*) – Window size for the binomial test. Defaults to 7.
- **binomial_probability_thr** (*float*) – Probability threshold for the binomial test. Defaults to 0.5
- **high_precision** (*bool*) – Whether to use high precision for the binomial test. Defaults to False.

Return type:

`Tensor`

Returns:

Anomaly mask. Tensor of shape (N, 1, H, W), where N is the batch size, and H and W are the height and width of the input image, respectively.

`forward(latent_variables)`

Return anomaly map.

Return type:

`Tensor`

Previous
STFPM

Next
WinCLIP