- **anomalous_ratio** (*float*) – Fraction of source samples that will be converted into anomalous samples.

Return type
DataFrame

3.3.2 Models

Model Components   Learn more about components to design your own anomaly detection models.

Image Models   Learn more about image anomaly detection models.

Video Models   Learn more about video anomaly detection models.

Model Components

Feature Extractors   Learn more about anomalib feature extractors to extract features from backbones.

Dimensionality Reduction   Learn more about dimensionality reduction models.

Normalizing Flows   Learn more about `freia` normalizing flows model components.

Sampling Components   Learn more about various sampling components.

Filters   Learn more about filters for post-processing.

Classification   Learn more about classification model components.

Cluster   Learn more about cluster model components.

Statistical Components   Learn more about classification model components.

Feature Extractors

Feature extractors.

`class anomalib.models.components.feature_extractors.`**`BackboneParams`**(*class_path*, *init_args=<factory>*)

Bases: `object`

Used for serializing the backbone.

`class anomalib.models.components.feature_extractors.`**`TimmFeatureExtractor`**(*backbone*, *layers*, *pre_trained=True*, *requires_grad=False*)

Bases: `Module`

Extract features from a CNN.

Parameters

- **backbone** (*nn.Module*) – The backbone to which the feature extraction hooks are attached.
- **layers** (*Iterable[str]*) – List of layer names of the backbone to which the hooks are attached.

- **pre_trained** (*bool*) – Whether to use a pre-trained backbone. Defaults to True.
- **requires_grad** (*bool*) – Whether to require gradients for the backbone. Defaults to False. Models like `stfpm` use the feature extractor model as a trainable network. In such cases gradient computation is required.

Example

```
import torch
from anomalib.models.components.feature_extractors import TimmFeatureExtractor

model = TimmFeatureExtractor(model="resnet18", layers=['layer1', 'layer2', 'layer3
↪'])
input = torch.rand((32, 3, 256, 256))
features = model(input)

print([layer for layer in features.keys()])
# Output: ['layer1', 'layer2', 'layer3']

print([feature.shape for feature in features.values()]()
# Output: [torch.Size([32, 64, 64, 64]), torch.Size([32, 128, 32, 32]), torch.
↪Size([32, 256, 16, 16])]
```

forward(*inputs*)

Forward-pass input tensor into the CNN.

Parameters
inputs (*torch.Tensor*) – Input tensor

Return type
dict[str, Tensor]

Returns
Feature map extracted from the CNN

Example

```
model = TimmFeatureExtractor(model="resnet50", layers=['layer3'])
input = torch.rand((32, 3, 256, 256))
features = model.forward(input)
```

class anomalib.models.components.feature_extractors.**TorchFXFeatureExtractor**(*backbone*, *return_nodes*, *weights=None*, *requires_grad=False*, *tracer_kwargs=None*)

Bases: `Module`

Extract features from a CNN.

Parameters

- **backbone** (*str | BackboneParams | dict | nn.Module*) – The backbone to which the feature extraction hooks are attached. If the name is provided, the model is loaded from torchvision. Otherwise, the model class can be provided and it will try to load the weights from the provided weights file. Last, an instance of nn.Module can also be passed directly.

- **return_nodes** (*Iterable[str]*) – List of layer names of the backbone to which the hooks are attached. You can find the names of these nodes by using `get_graph_node_names` function.
- **weights** (*str | WeightsEnum | None*) – Weights enum to use for the model. Torchvision models require `WeightsEnum`. These enums are defined in `torchvision.models.<model>`. You can pass the weights path for custom models.
- **requires_grad** (*bool*) – Models like `stfpm` use the feature extractor for training. In such cases we should set `requires_grad` to `True`. Default is `False`.
- **tracer_kwargs** (*dict | None*) – a dictionary of keyword arguments for NodePathTracer (which passes them onto it's parent class torch.fx.Tracer). Can be used to allow not tracing through a list of problematic modules, by passing a list of *leaf_modules* as one of the *tracer_kwargs*.

Example

With torchvision models:

```
import torch
from anomalib.models.components.feature_extractors import TorchFXFeatureExtractor
from torchvision.models.efficientnet import EfficientNet_B5_Weights

feature_extractor = TorchFXFeatureExtractor(
    backbone="efficientnet_b5",
    return_nodes=["features.6.8"],
    weights=EfficientNet_B5_Weights.DEFAULT
)

input = torch.rand((32, 3, 256, 256))
features = feature_extractor(input)

print([layer for layer in features.keys()])
# Output: ["features.6.8"]

print([feature.shape for feature in features.values()])
# Output: [torch.Size([32, 304, 8, 8])]
```

With custom models:

```
import torch
from anomalib.models.components.feature_extractors import TorchFXFeatureExtractor

feature_extractor = TorchFXFeatureExtractor(
    "path.to.CustomModel", ["linear_relu_stack.3"], weights="path/to/weights.pth"
)

input = torch.randn(1, 1, 28, 28)
features = feature_extractor(input)
```

(continues on next page)

(continued from previous page)

```
print([layer for layer in features.keys()])
# Output: ["linear_relu_stack.3"]
```

with model instances:

```
import torch
from anomalib.models.components.feature_extractors import TorchFXFeatureExtractor
from timm import create_model

model = create_model("resnet18", pretrained=True)
feature_extractor = TorchFXFeatureExtractor(model, ["layer1"])

input = torch.rand((32, 3, 256, 256))
features = feature_extractor(input)

print([layer for layer in features.keys()])
# Output: ["layer1"]

print([feature.shape for feature in features.values()])
# Output: [torch.Size([32, 64, 64, 64])]
```

forward(*inputs*)

Extract features from the input.

Return type
`dict[str, Tensor]`

initialize_feature_extractor(*backbone*, *return_nodes*, *weights=None*, *requires_grad=False*, *tracer_kwargs=None*)

Extract features from a CNN.

Parameters

- **backbone** (`BackboneParams | nn.Module`) – The backbone to which the feature extraction hooks are attached. If the name is provided for BackboneParams, the model is loaded from torchvision. Otherwise, the model class can be provided and it will try to load the weights from the provided weights file. Last, an instance of the model can be provided as well, which will be used as-is.
- **return_nodes** (`Iterable[str]`) – List of layer names of the backbone to which the hooks are attached. You can find the names of these nodes by using `get_graph_node_names` function.
- **weights** (`str | WeightsEnum | None`) – Weights enum to use for the model. Torchvision models require `WeightsEnum`. These enums are defined in `torchvision.models.<model>`. You can pass the weights path for custom models.
- **requires_grad** (`bool`) – Models like `stfpm` use the feature extractor for training. In such cases we should set `requires_grad` to `True`. Default is `False`.
- **tracer_kwargs** (`dict | None`) – a dictionary of keyword arguments for NodePathTracer (which passes them onto it's parent class torch.fx.Tracer). Can be used to allow not tracing through a list of problematic modules, by passing a list of *leaf_modules* as one of the *tracer_kwargs*.

Return type
`GraphModule`

Returns
Feature Extractor based on TorchFX.

`anomalib.models.components.feature_extractors.`**`dryrun_find_featuremap_dims`**(*feature_extractor*, *input_size*, *layers*)

Dry run an empty image of *input_size* size to get the featuremap tensors' dimensions (num_features, resolution).

Returns

maping of *layer* -> *dimensions dict*
Each *dimension dict* has two keys: *num_features* (int) and `resolution`(tuple[int, int]).

Return type
tuple[int, int]

Dimensionality Reduction

Algorithms for decomposition and dimensionality reduction.

class `anomalib.models.components.dimensionality_reduction.`**`PCA`**(*n_components*)

Bases: `DynamicBufferMixin`

Principle Component Analysis (PCA).

Parameters
n_components (*float*) – Number of components. Can be either integer number of components or a ratio between 0-1.

Example

```
>>> import torch
>>> from anomalib.models.components import PCA
```

Create a PCA model with 2 components:

```
>>> pca = PCA(n_components=2)
```

Create a random embedding and fit a PCA model.

```
>>> embedding = torch.rand(1000, 5).cuda()
>>> pca = PCA(n_components=2)
>>> pca.fit(embedding)
```

Apply transformation:

```
>>> transformed = pca.transform(embedding)
>>> transformed.shape
torch.Size([1000, 2])
```

`fit`(*dataset*)

Fits the PCA model to the dataset.

Parameters
dataset (*torch.Tensor*) – Input dataset to fit the model.

Return type
`None`

Example

```
>>> pca.fit(embedding)
>>> pca.singular_vectors
tensor([9.6053, 9.2763], device='cuda:0')
```

```
>>> pca.mean
tensor([0.4859, 0.4959, 0.4906, 0.5010, 0.5042], device='cuda:0')
```

fit_transform(*dataset*)

Fit and transform PCA to dataset.

Parameters
`dataset` (*torch.Tensor*) – Dataset to which the PCA if fit and transformed

Return type
`Tensor`

Returns
Transformed dataset

Example

```
>>> pca.fit_transform(embedding)
>>> transformed_embedding = pca.fit_transform(embedding)
>>> transformed_embedding.shape
torch.Size([1000, 2])
```

forward(*features*)

Transform the features.

Parameters
`features` (*torch.Tensor*) – Input features

Return type
`Tensor`

Returns
Transformed features

Example

```
>>> pca(embedding).shape
torch.Size([1000, 2])
```

inverse_transform(*features*)

Inverses the transformed features.

Parameters
`features` (*torch.Tensor*) – Transformed features

Return type
`Tensor`

Returns
Inverse features

Example

```
>>> inverse_embedding = pca.inverse_transform(transformed_embedding)
>>> inverse_embedding.shape
torch.Size([1000, 5])
```

transform(*features*)

Transform the features based on singular vectors calculated earlier.

Parameters
features (*torch.Tensor*) – Input features

Return type
Tensor

Returns
Transformed features

Example

```
>>> pca.transform(embedding)
>>> transformed_embedding = pca.transform(embedding)
```

```
>>> embedding.shape
torch.Size([1000, 5])
#
>>> transformed_embedding.shape
torch.Size([1000, 2])
```

class anomalib.models.components.dimensionality_reduction.**SparseRandomProjection**(*eps=0.1*, *random_state=None*)

Bases: object

Sparse Random Projection using PyTorch operations.

Parameters

- **eps** (*float, optional*) – Minimum distortion rate parameter for calculating Johnson-Lindenstrauss minimum dimensions. Defaults to 0.1.
- **random_state** (*int | None, optional*) – Uses the seed to set the random state for sample_without_replacement function. Defaults to None.

Example

To fit and transform the embedding tensor, use the following code:

```
import torch
from anomalib.models.components import SparseRandomProjection

sparse_embedding = torch.rand(1000, 5).cuda()
model = SparseRandomProjection(eps=0.1)
```

Fit the model and transform the embedding tensor:

```
model.fit(sparse_embedding)
projected_embedding = model.transform(sparse_embedding)

print(projected_embedding.shape)
# Output: torch.Size([1000, 5920])
```

fit(*embedding*)

Generate sparse matrix from the embedding tensor.

Parameters
embedding (*torch.Tensor*) – embedding tensor for generating embedding

Returns

Return self to be used as

```
>>> model = SparseRandomProjection()
>>> model = model.fit()
```

Return type
(*SparseRandomProjection*)

transform(*embedding*)

Project the data by using matrix product with the random matrix.

Parameters
embedding (*torch.Tensor*) – Embedding of shape (n_samples, n_features) The input data to project into a smaller dimensional space

Returns

Sparse matrix of shape
(n_samples, n_components) Projected array.

Return type
projected_embedding (torch.Tensor)

Example

```
>>> projected_embedding = model.transform(embedding)
>>> projected_embedding.shape
torch.Size([1000, 5920])
```

Normalizing Flows

All In One Block Layer.

class anomalib.models.components.flow.**AllInOneBlock**(*dims_in*, *dims_c=None*, *subnet_constructor=None*, *affine_clamping=2.0*, *gin_block=False*, *global_affine_init=1.0*, *global_affine_type='SOFTPLUS'*, *permute_soft=False*, *learned_householder_permutation=0*, *reverse_permutation=False*)

Bases: `InvertibleModule`

Module combining the most common operations in a normalizing flow or similar model.

It combines affine coupling, permutation, and global affine transformation ('ActNorm'). It can also be used as GIN coupling block, perform learned householder permutations, and use an inverted pre-permutation. The affine transformation includes a soft clamping mechanism, first used in Real-NVP. The block as a whole performs the following computation:

$$y = VR\;\Psi(s_\mathrm{global}) \odot \mathrm{Coupling}\Big(R^{-1} V^{-1} x\Big) + t_\mathrm{global}$$

- The inverse pre-permutation of x (i.e. $R^{-1} V^{-1}$) is optional (see `reverse_permutation` below).
- The learned householder reflection matrix V is also optional all together (see `learned_householder_permutation` below).
- For the coupling, the input is split into x_1, x_2 along the channel dimension. Then the output of the coupling operation is the two halves $u = \mathrm{concat}(u_1, u_2)$.

$$\begin{aligned} u_1 &= x_1 \odot \exp \Big(\alpha \tanh\big(s(x_2) \big) \Big) + t(x_2) \\ u_2 &= x_2 \end{aligned}$$

Because $\tanh(s) \in [-1, 1]$, this clamping mechanism prevents exploding values in the exponential. The hyperparameter α can be adjusted.

Parameters

- **subnet_constructor** (`Callable | None`) – class or callable `f`, called as `f(channels_in, channels_out)` and should return a torch.nn.Module. Predicts coupling coefficients s, t.
- **affine_clamping** (`float`) – clamp the output of the multiplicative coefficients before exponentiation to +/- `affine_clamping` (see α above).
- **gin_block** (`bool`) – Turn the block into a GIN block from Sorrenson et al, 2019. Makes it so that the coupling operations as a whole is volume preserving.
- **global_affine_init** (`float`) – Initial value for the global affine scaling s_global.
- **global_affine_init** – `'SIGMOID'`, `'SOFTPLUS'`, or `'EXP'`. Defines the activation to be used on the beta for the global affine scaling (Ψ above).
- **permute_soft** (`bool`) – bool, whether to sample the permutation matrix R from $SO(N)$, or to use hard permutations instead. Note, `permute_soft=True` is very slow when working with >512 dimensions.
- **learned_householder_permutation** (`int`) – Int, if >0, turn on the matrix V above, that represents multiple learned householder reflections. Slow if large number. Dubious whether it actually helps network performance.
- **reverse_permutation** (`bool`) – Reverse the permutation before the block, as introduced by Putzky et al, 2019. Turns on the $R^{-1} V^{-1}$ pre-multiplication above.

forward(*x*, *c=None*, *rev=False*, *jac=True*)

See base class docstring.

Return type

`tuple[tuple[Tensor], Tensor]`

output_dims(*input_dims*)

Output dimensions of the layer.

Parameters
input_dims (*list[tuple[int]]*) – Input dimensions.

Returns
Output dimensions.

Return type
list[tuple[int]]

Sampling Components

Sampling methods.

class anomalib.models.components.sampling.**KCenterGreedy**(*embedding*, *sampling_ratio*)

Bases: `object`

Implements k-center-greedy method.

Parameters

- **embedding** (*torch.Tensor*) – Embedding vector extracted from a CNN
- **sampling_ratio** (*float*) – Ratio to choose coreset size from the embedding size.

Example

```
>>> embedding.shape
torch.Size([219520, 1536])
>>> sampler = KCenterGreedy(embedding=embedding)
>>> sampled_idxs = sampler.select_coreset_idxs()
>>> coreset = embedding[sampled_idxs]
>>> coreset.shape
torch.Size([219, 1536])
```

get_new_idx()

Get index value of a sample.

Based on minimum distance of the cluster

Returns
Sample index

Return type
int

reset_distances()

Reset minimum distances.

Return type
`None`

sample_coreset(*selected_idxs=None*)

Select coreset from the embedding.

Parameters
`selected_idxs` (list[int] | None) – index of samples already selected. Defaults to an empty set.

Returns
Output coreset

Return type
Tensor

Example

```
>>> embedding.shape
torch.Size([219520, 1536])
>>> sampler = KCenterGreedy(...)
>>> coreset = sampler.sample_coreset()
>>> coreset.shape
torch.Size([219, 1536])
```

select_coreset_idxs(*selected_idxs=None*)

Greedily form a coreset to minimize the maximum distance of a cluster.

Parameters
`selected_idxs` (list[int] | None) – index of samples already selected. Defaults to an empty set.

Return type
`list[int]`

Returns
indices of samples selected to minimize distance to cluster centers

update_distances(*cluster_centers*)

Update min distances given cluster centers.

Parameters
`cluster_centers` (*list[int]*) – indices of cluster centers

Return type
`None`

Filters

Implements filters used by models.

class anomalib.models.components.filters.**GaussianBlur2d**(*sigma, channels=1, kernel_size=None, normalize=True, border_type='reflect', padding='same'*)

Bases: `Module`

Compute GaussianBlur in 2d.

Makes use of kornia functions, but most notably the kernel is not computed during the forward pass, and does not depend on the input size. As a caveat, the number of channels that are expected have to be provided during initialization.

forward(*input_tensor*)

Blur the input with the computed Gaussian.

Parameters
`input_tensor` (*torch.Tensor*) – Input tensor to be blurred.

Returns
Blurred output tensor.

Return type
Tensor

Classification

Classification modules.

class anomalib.models.components.classification.**FeatureScalingMethod**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `str`, `Enum`

Determines how the feature embeddings are scaled.

class anomalib.models.components.classification.**KDEClassifier**(*n_pca_components=16*, *feature_scaling_method=FeatureScalingMethod.SCALE*, *max_training_points=40000*)

Bases: `Module`

Classification module for KDE-based anomaly detection.

Parameters

- **n_pca_components** (*int, optional*) – Number of PCA components. Defaults to 16.
- **feature_scaling_method** (FeatureScalingMethod, *optional*) – Scaling method applied to features before passing to KDE. Options are *norm* (normalize to unit vector length) and *scale* (scale to max length observed in training).
- **max_training_points** (*int, optional*) – Maximum number of training points to fit the KDE model. Defaults to 40000.

compute_kde_scores(*features*, *as_log_likelihood=False*)

Compute the KDE scores.

The scores calculated from the KDE model are converted to densities. If *as_log_likelihood* is set to true then
the log of the scores are calculated.

Parameters

- **features** (*torch.Tensor*) – Features to which the PCA model is fit.
- **as_log_likelihood** (*bool | None, optional*) – If true, gets log likelihood scores. Defaults to False.

Returns
Score

Return type
(torch.Tensor)

static compute_probabilities(*scores*)

Convert density scores to anomaly probabilities (see https://www.desmos.com/calculator/ifju7eesg7).

Parameters
scores (*torch.Tensor*) – density of an image.

Return type
`Tensor`

Returns
probability that image with {density} is anomalous

fit(*embeddings*)

Fit a kde model to embeddings.

Parameters
embeddings (*torch.Tensor*) – Input embeddings to fit the model.

Return type
`bool`

Returns
Boolean confirming whether the training is successful.

forward(*features*)

Make predictions on extracted features.

Return type
`Tensor`

pre_process(*feature_stack*, *max_length=None*)

Pre-process the CNN features.

Parameters

- **feature_stack** (*torch.Tensor*) – Features extracted from CNN
- **max_length** (*Tensor | None*) – Used to unit normalize the feature_stack vector. If `max_len` is not provided, the length is calculated from the `feature_stack`. Defaults to None.

Returns
Stacked features and length

Return type
(Tuple)

predict(*features*)

Predicts the probability that the features belong to the anomalous class.

Parameters
features (*torch.Tensor*) – Feature from which the output probabilities are detected.

Return type
`Tensor`

Returns
Detection probabilities

Cluster

Clustering algorithm implementations using PyTorch.

class anomalib.models.components.cluster.**GaussianMixture**(*n_components*, *n_iter=100*, *tol=0.001*)

Bases: DynamicBufferMixin

Gaussian Mixture Model.

Parameters

- **n_components** (*int*) – Number of components.
- **n_iter** (*int*) – Maximum number of iterations to perform. Defaults to 100.
- **tol** (*float*) – Convergence threshold. Defaults to 1e-3.

Example

The following examples shows how to fit a Gaussian Mixture Model to some data and get the cluster means and predicted labels and log-likelihood scores of the data.

```
>>> import torch
>>> from anomalib.models.components.cluster import GaussianMixture
>>> model = GaussianMixture(n_components=2)
>>> data = torch.tensor(
...     [
...             [2, 1], [2, 2], [2, 3],
...             [7, 5], [8, 5], [9, 5],
...     ]
... ).float()
>>> model.fit(data)
>>> model.means  # get the means of the gaussians
tensor([[8., 5.],
        [2., 2.]])
>>> model.predict(data)  # get the predicted cluster label of each sample
tensor([1, 1, 1, 0, 0, 0])
>>> model.score_samples(data)  # get the log-likelihood score of each sample
tensor([3.8295, 4.5795, 3.8295, 3.8295, 4.5795, 3.8295])
```

fit(*data*)

Fit the model to the data.

Parameters

data (*Tensor*) – Data to fit the model to. Tensor of shape (n_samples, n_features).

Return type

None

predict(*data*)

Predict the cluster labels of the data.

Parameters

data (*Tensor*) – Samples to assign to clusters. Tensor of shape (n_samples, n_features).

Returns

Tensor of shape (n_samples,) containing the predicted cluster label of each sample.

Return type
Tensor

`score_samples`(*data*)

Assign a likelihood score to each sample in the data.

Parameters
data (*Tensor*) – Samples to assign scores to. Tensor of shape (n_samples, n_features).

Returns
Tensor of shape (n_samples,) containing the log-likelihood score of each sample.

Return type
Tensor

class `anomalib.models.components.cluster.KMeans`(*n_clusters*, *max_iter=10*)

Bases: `object`

Initialize the KMeans object.

Parameters

- **n_clusters** (*int*) – The number of clusters to create.
- **max_iter** (*int, optional)*) – The maximum number of iterations to run the algorithm. Defaults to 10.

`fit`(*inputs*)

Fit the K-means algorithm to the input data.

Parameters
inputs (*torch.Tensor*) – Input data of shape (batch_size, n_features).

Returns
A tuple containing the labels of the input data with respect to the identified clusters and the cluster centers themselves. The labels have a shape of (batch_size,) and the cluster centers have a shape of (n_clusters, n_features).

Return type
tuple

Raises
ValueError – If the number of clusters is less than or equal to 0.

`predict`(*inputs*)

Predict the labels of input data based on the fitted model.

Parameters
inputs (*torch.Tensor*) – Input data of shape (batch_size, n_features).

Returns
The predicted labels of the input data with respect to the identified clusters.

Return type
torch.Tensor

Raises
AttributeError – If the KMeans object has not been fitted to input data.

Stats Components

Statistical functions.

class anomalib.models.components.stats.**GaussianKDE**(*dataset=None*)

Bases: `DynamicBufferMixin`

Gaussian Kernel Density Estimation.

Parameters
`dataset` (*Tensor | None, optional*) – Dataset on which to fit the KDE model. Defaults to None.

static cov(*tensor*)

Calculate the unbiased covariance matrix.

Parameters
`tensor` (*torch.Tensor*) – Input tensor from which covariance matrix is computed.

Return type
`Tensor`

Returns
Output covariance matrix.

fit(*dataset*)

Fit a KDE model to the input dataset.

Parameters
`dataset` (*torch.Tensor*) – Input dataset.

Return type
`None`

Returns
None

forward(*features*)

Get the KDE estimates from the feature map.

Parameters
`features` (*torch.Tensor*) – Feature map extracted from the CNN

Return type
`Tensor`

Returns: KDE Estimates

class anomalib.models.components.stats.**MultiVariateGaussian**

Bases: `DynamicBufferMixin`, `Module`

Multi Variate Gaussian Distribution.

fit(*embedding*)

Fit multi-variate gaussian distribution to the input embedding.

Parameters
`embedding` (*torch.Tensor*) – Embedding vector extracted from CNN.

Return type
`list[Tensor]`

Returns
Mean and the covariance of the embedding.

`forward`(*embedding*)

Calculate multivariate Gaussian distribution.

Parameters
embedding (*torch.Tensor*) – CNN features whose dimensionality is reduced via either random sampling or PCA.

Return type
`list[Tensor]`

Returns
mean and inverse covariance of the multi-variate gaussian distribution that fits the features.

Image Models

CFA Coupled-hypersphere-based Feature Adaptation for Target-Oriented Anomaly Localization

C-Flow Real-Time Unsupervised Anomaly Detection via Conditional Normalizing Flows

CS-Flow Fully Convolutional Cross-Scale-Flows for Image-based Defect Detection

DFKDE Deep Feature Kernel Density Estimation

DFM Probabilistic Modeling of Deep Features for Out-of-Distribution and Adversarial Detection

DRAEM DRÆM – A discriminatively trained reconstruction embedding for surface anomaly detection

DSR DSR – A Dual Subspace Re-Projection Network for Surface Anomaly Detection

Efficient AD EfficientAD: Accurate Visual Anomaly Detection at Millisecond-Level Latencies

FastFlow FastFlow: Unsupervised Anomaly Detection and Localization via 2D Normalizing Flows

GANomaly GANomaly: Semi-Supervised Anomaly Detection via Adversarial Training

PaDiM PaDiM: A Patch Distribution Modeling Framework for Anomaly Detection and Localization

Patchcore Towards Total Recall in Industrial Anomaly Detection

Reverse Distillation Anomaly Detection via Reverse Distillation from One-Class Embedding.

R-KDE Region-Based Kernel Density Estimation (RKDE)

STFPM Student-Teacher Feature Pyramid Matching for Unsupervised Anomaly Detection

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

WinCLIP WinCLIP: Zero-/Few-Shot Anomaly Classification and Segmentation

CFA

Lightning Implementatation of the CFA Model.

CFA: Coupled-hypersphere-based Feature Adaptation for Target-Oriented Anomaly Localization

Paper https://arxiv.org/abs/2206.04325

class anomalib.models.image.cfa.lightning_model.**Cfa**(*backbone='wide_resnet50_2'*, *gamma_c=1*, *gamma_d=1*, *num_nearest_neighbors=3*, *num_hard_negative_features=3*, *radius=1e-05*)

Bases: `AnomalyModule`

CFA: Coupled-hypersphere-based Feature Adaptation for Target-Oriented Anomaly Localization.

Parameters

- **backbone** (*str*) – Backbone CNN network Defaults to `"wide_resnet50_2"`.
- **gamma_c** (*int, optional*) – gamma_c value from the paper. Defaults to `1`.
- **gamma_d** (*int, optional*) – gamma_d value from the paper. Defaults to `1`.
- **num_nearest_neighbors** (*int*) – Number of nearest neighbors. Defaults to `3`.
- **num_hard_negative_features** (*int*) – Number of hard negative features. Defaults to `3`.
- **radius** (*float*) – Radius of the hypersphere to search the soft boundary. Defaults to `1e-5`.

backward(*loss*, **args*, ***kwargs*)

Perform backward-pass for the CFA model.

Parameters

- **loss** (*torch.Tensor*) – Loss value.
- ***args** – Arguments.
- ****kwargs** – Keyword arguments.

Return type

`None`

configure_optimizers()

Configure optimizers for the CFA Model.

Returns

Adam optimizer for each decoder

Return type

Optimizer

property learning_type: LearningType

Return the learning type of the model.

Returns

Learning type of the model.

Return type

LearningType

on_train_start()

Initialize the centroid for the memory bank computation.

Return type
None

property trainer_arguments: dict[str, Any]

CFA specific trainer arguments.

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

Perform the training step for the CFA model.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Batch input.
- ***args** – Arguments.
- ****kwargs** – Keyword arguments.

Returns
Loss value.

Return type
STEP_OUTPUT

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

Perform the validation step for the CFA model.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Input batch.
- ***args** – Arguments.
- ****kwargs** – Keyword arguments.

Returns
Anomaly map computed by the model.

Return type
dict

Torch Implementatation of the CFA Model.

CFA: Coupled-hypersphere-based Feature Adaptation for Target-Oriented Anomaly Localization

Paper https://arxiv.org/abs/2206.04325

class anomalib.models.image.cfa.torch_model.**CfaModel**(*backbone*, *gamma_c*, *gamma_d*, *num_nearest_neighbors*, *num_hard_negative_features*, *radius*)

Bases: DynamicBufferMixin

Torch implementation of the CFA Model.

Parameters

- **backbone** (*str*) – Backbone CNN network.
- **gamma_c** (*int*) – gamma_c parameter from the paper.
- **gamma_d** (*int*) – gamma_d parameter from the paper.
- **num_nearest_neighbors** (*int*) – Number of nearest neighbors.
- **num_hard_negative_features** (*int*) – Number of hard negative features.
- **radius** (*float*) – Radius of the hypersphere to search the soft boundary.

`compute_distance`(*target_oriented_features*)

Compute distance using target oriented features.

Parameters
`target_oriented_features` (*torch.Tensor*) – Target oriented features computed using the descriptor.

Returns
Distance tensor.

Return type
Tensor

`forward`(*input_tensor*)

Forward pass.

Parameters
`input_tensor` (*torch.Tensor*) – Input tensor.

Raises
`ValueError` – When the memory bank is not initialized.

Returns
Loss or anomaly map depending on the train/eval mode.

Return type
Tensor

`get_scale`(*input_size*)

Get the scale of the feature map.

Parameters
`input_size` (*tuple[int, int]*) – Input size of the image tensor.

Return type
`Size`

`initialize_centroid`(*data_loader*)

Initialize the Centroid of the Memory Bank.

Parameters
`data_loader` (*DataLoader*) – Train Dataloader.

Returns
Memory Bank.

Return type
Tensor

Loss function for the Cfa Model Implementation.

class `anomalib.models.image.cfa.loss.CfaLoss`(*num_nearest_neighbors*, *num_hard_negative_features*, *radius*)

Bases: `Module`

Cfa Loss.

Parameters

- `num_nearest_neighbors` (*int*) – Number of nearest neighbors.
- `num_hard_negative_features` (*int*) – Number of hard negative features.
- `radius` (*float*) – Radius of the hypersphere to search the soft boundary.

`forward`(*distance*)

Compute the CFA loss.

Parameters

`distance` (*torch.Tensor*) – Distance computed using target oriented features.

Returns

CFA loss.

Return type

Tensor

Anomaly Map Generator for the CFA model implementation.

`class anomalib.models.image.cfa.anomaly_map.AnomalyMapGenerator`(*num_nearest_neighbors*, *sigma=4*)

Bases: `Module`

Generate Anomaly Heatmap.

`compute_anomaly_map`(*score*, *image_size=None*)

Compute anomaly map based on the score.

Parameters

- **score** (*torch.Tensor*) – Score tensor.
- **image_size** (*tuple[int, int] | torch.Size | None, optional*) – Size of the input image.

Returns

Anomaly map.

Return type

Tensor

`compute_score`(*distance*, *scale*)

Compute score based on the distance.

Parameters

- **distance** (*torch.Tensor*) – Distance tensor computed using target oriented features.
- **scale** (*tuple[int, int]*) – Height and width of the largest feature map.

Returns

Score value.

Return type

Tensor

`forward`(***kwargs*)

Return anomaly map.

Raises

distance` and scale keys are not found –

Returns

Anomaly heatmap.

Return type

Tensor

C-Flow

Cflow.

Real-Time Unsupervised Anomaly Detection via Conditional Normalizing Flows.

For more details, see the paper: Real-Time Unsupervised Anomaly Detection via Conditional Normalizing Flows.

class anomalib.models.image.cflow.lightning_model.**Cflow**(*backbone='wide_resnet50_2'*, *layers=('layer2', 'layer3', 'layer4')*, *pre_trained=True*, *fiber_batch_size=64*, *decoder='freia-cflow'*, *condition_vector=128*, *coupling_blocks=8*, *clamp_alpha=1.9*, *permute_soft=False*, *lr=0.0001*)

Bases: `AnomalyModule`

PL Lightning Module for the CFLOW algorithm.

Parameters

- **backbone** (*str, optional*) – Backbone CNN architecture. Defaults to `"wide_resnet50_2"`.
- **layers** (*Sequence[str], optional*) – Layers to extract features from. Defaults to `("layer2", "layer3", "layer4")`.
- **pre_trained** (*bool, optional*) – Whether to use pre-trained weights. Defaults to `True`.
- **fiber_batch_size** (*int, optional*) – Fiber batch size. Defaults to `64`.
- **decoder** (*str, optional*) – Decoder architecture. Defaults to `"freia-cflow"`.
- **condition_vector** (*int, optional*) – Condition vector size. Defaults to `128`.
- **coupling_blocks** (*int, optional*) – Number of coupling blocks. Defaults to `8`.
- **clamp_alpha** (*float, optional*) – Clamping value for the alpha parameter. Defaults to `1.9`.
- **permute_soft** (*bool, optional*) – Whether to use soft permutation. Defaults to `False`.
- **lr** (*float, optional*) – Learning rate. Defaults to `0.0001`.

configure_optimizers()

Configure optimizers for each decoder.

Returns
Adam optimizer for each decoder

Return type
Optimizer

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]

C-FLOW specific trainer arguments.

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

Perform the training step of CFLOW.

For each batch, decoder layers are trained with a dynamic fiber batch size. Training step is performed manually as multiple training steps are involved

per batch of input images

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Input batch
- ***args** – Arguments.
- ****kwargs** – Keyword arguments.

Return type

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

Returns

Loss value for the batch

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

Perform the validation step of CFLOW.

Similar to the training step, encoder features are extracted from the CNN for each batch, and anomaly map is computed.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Input batch
- ***args** – Arguments.
- ****kwargs** – Keyword arguments.

Return type

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

Returns

Dictionary containing images, anomaly maps, true labels and masks. These are required in *validation_epoch_end* for feature concatenation.

PyTorch model for CFlow model implementation.

class anomalib.models.image.cflow.torch_model.**CflowModel**(*backbone*, *layers*, *pre_trained=True*, *fiber_batch_size=64*, *decoder='freia-cflow'*, *condition_vector=128*, *coupling_blocks=8*, *clamp_alpha=1.9*, *permute_soft=False*)

Bases: `Module`

CFLOW: Conditional Normalizing Flows.

Parameters

- **backbone** (*str*) – Backbone CNN architecture.

- **layers** (*Sequence[str]*) – Layers to extract features from.
- **pre_trained** (*bool*) – Whether to use pre-trained weights. Defaults to `True`.
- **fiber_batch_size** (*int*) – Fiber batch size. Defaults to `64`.
- **decoder** (*str*) – Decoder architecture. Defaults to `"freia-cflow"`.
- **condition_vector** (*int*) – Condition vector size. Defaults to `128`.
- **coupling_blocks** (*int*) – Number of coupling blocks. Defaults to `8`.
- **clamp_alpha** (*float*) – Clamping value for the alpha parameter. Defaults to `1.9`.
- **permute_soft** (*bool*) – Whether to use soft permutation. Defaults to `False`.

forward(*images*)

Forward-pass images into the network to extract encoder features and compute probability.

Parameters
images (`Tensor`) – Batch of images.

Return type
`Tensor`

Returns
Predicted anomaly maps.

Anomaly Map Generator for CFlow model implementation.

class `anomalib.models.image.cflow.anomaly_map.`**AnomalyMapGenerator**(*pool_layers*)

Bases: `Module`

Generate Anomaly Heatmap.

compute_anomaly_map(*distribution*, *height*, *width*, *image_size*)

Compute the layer map based on likelihood estimation.

Parameters

- **distribution** (*list[torch.Tensor]*) – List of likelihoods for each layer.
- **height** (*list[int]*) – List of heights of the feature maps.
- **width** (*list[int]*) – List of widths of the feature maps.
- **image_size** (*tuple[int, int] | torch.Size | None*) – Size of the input image.

Return type
`Tensor`

Returns
Final Anomaly Map

forward(***kwargs*)

Return anomaly_map.

Expects *distribution*, *height* and 'width' keywords to be passed explicitly

Example

```
>>> anomaly_map_generator = AnomalyMapGenerator(image_size=tuple(hparams.model.
↪input_size),
>>>         pool_layers=pool_layers)
>>> output = self.anomaly_map_generator(distribution=dist, height=height,␣
↪width=width)
```

Raises
`ValueError` – *distribution*, *height* and 'width' keys are not found

Returns
anomaly map

Return type
torch.Tensor

CS-Flow

Fully Convolutional Cross-Scale-Flows for Image-based Defect Detection.

https://arxiv.org/pdf/2110.02855.pdf

class `anomalib.models.image.csflow.lightning_model.`**`Csflow`**(*cross_conv_hidden_channels=1024*, *n_coupling_blocks=4*, *clamp=3*, *num_channels=3*)

Bases: `AnomalyModule`

Fully Convolutional Cross-Scale-Flows for Image-based Defect Detection.

Parameters

- **n_coupling_blocks** (*int*) – Number of coupling blocks in the model. Defaults to `4`.
- **cross_conv_hidden_channels** (*int*) – Number of hidden channels in the cross convolution. Defaults to `1024`.
- **clamp** (*int*) – Clamp value for glow layer. Defaults to 3.
- **num_channels** (*int*) – Number of channels in the model. Defaults to 3.

`configure_optimizers()`

Configure optimizers.

Returns
Adam optimizer

Return type
Optimizer

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]

CS-Flow-specific trainer arguments.

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

Perform the training step of CS-Flow.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Input batch
- **args** – Arguments.
- **kwargs** – Keyword arguments.

Return type

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

Returns

Loss value

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

Perform the validation step for CS Flow.

Parameters

- **batch** (*torch.Tensor*) – Input batch
- **args** – Arguments.
- **kwargs** – Keyword arguments.

Returns

Dictionary containing the anomaly map, scores, etc.

Return type

dict[str, torch.Tensor]

PyTorch model for CS-Flow implementation.

class anomalib.models.image.csflow.torch_model.**CsFlowModel**(*input_size*, *cross_conv_hidden_channels*, *n_coupling_blocks=4*, *clamp=3*, *num_channels=3*)

Bases: Module

CS Flow Module.

Parameters

- **input_size** (*tuple[int, int]*) – Input image size.
- **cross_conv_hidden_channels** (*int*) – Number of hidden channels in the cross convolution.
- **n_coupling_blocks** (*int*) – Number of coupling blocks. Defaults to 4.
- **clamp** (*float*) – Clamp value for the coupling blocks. Defaults to 3.
- **num_channels** (*int*) – Number of channels in the input image. Defaults to 3.

forward(*images*)

Forward method of the model.

Parameters

images (*torch.Tensor*) – Input images.

Returns

During training: tuple containing the z_distribution for three scales
and the sum of log determinant of the Jacobian. During evaluation: tuple containing anomaly maps and anomaly scores

Return type
tuple[torch.Tensor, torch.Tensor]

Loss function for the CS-Flow Model Implementation.

`class anomalib.models.image.csflow.loss.CsFlowLoss(*args, **kwargs)`

Bases: `Module`

Loss function for the CS-Flow Model Implementation.

`forward(z_dist, jacobians)`

Compute the loss CS-Flow.

Parameters

- **z_dist** (*torch.Tensor*) – Latent space image mappings from NF.
- **jacobians** (*torch.Tensor*) – Jacobians of the distribution

Return type
`Tensor`

Returns
Loss value

Anomaly Map Generator for CS-Flow model.

`class anomalib.models.image.csflow.anomaly_map.AnomalyMapGenerator(input_dims, mode=AnomalyMapMode.ALL)`

Bases: `Module`

Anomaly Map Generator for CS-Flow model.

Parameters

- **input_dims** (*tuple[int, int, int]*) – Input dimensions.
- **mode** (AnomalyMapMode) – Anomaly map mode. Defaults to `AnomalyMapMode.ALL`.

`forward(inputs)`

Get anomaly maps by taking mean of the z-distributions across channels.

By default it computes anomaly maps for all the scales as it gave better performance on initial tests. Use `AnomalyMapMode.MAX` for the largest scale as mentioned in the paper.

Parameters

- **inputs** (*torch.Tensor*) – z-distributions for the three scales.
- **mode** (AnomalyMapMode) – Anomaly map mode.

Returns
Anomaly maps.

Return type
Tensor

class anomalib.models.image.csflow.anomaly_map.**AnomalyMapMode**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `str`, `Enum`

Generate anomaly map from all the scales or the max.

DFKDE

DFKDE: Deep Feature Kernel Density Estimation.

class anomalib.models.image.dfkde.lightning_model.**Dfkde**(*backbone='resnet18'*, *layers=('layer4',)*, *pre_trained=True*, *n_pca_components=16*, *feature_scaling_method=FeatureScalingMethod.SCALE*, *max_training_points=40000*)

Bases: `MemoryBankMixin`, `AnomalyModule`

DFKDE: Deep Feature Kernel Density Estimation.

Parameters

- **backbone** (*str*) – Pre-trained model backbone. Defaults to `"resnet18"`.
- **layers** (*Sequence[str], optional*) – Layers to extract features from. Defaults to `("layer4",)`.
- **pre_trained** (*bool, optional*) – Boolean to check whether to use a pre_trained backbone. Defaults to `True`.
- **n_pca_components** (*int, optional*) – Number of PCA components. Defaults to `16`.
- **feature_scaling_method** (*FeatureScalingMethod, optional*) – Feature scaling method. Defaults to `FeatureScalingMethod.SCALE`.
- **max_training_points** (*int, optional*) – Number of training points to fit the KDE model. Defaults to `40000`.

static **configure_optimizers**()

DFKDE doesn't require optimization, therefore returns no optimizers.

Return type
`None`

fit()

Fit a KDE Model to the embedding collected from the training set.

Return type
`None`

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 DFKDE-specific trainer arguments.

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

Perform the training step of DFKDE. For each batch, features are extracted from the CNN.

Parameters

- **(batch** (*batch*) – dict[str, str | torch.Tensor]): Batch containing image filename, image, label and mask
- **args** – Arguments.
- **kwargs** – Keyword arguments.

Return type

`None`

Returns

Deep CNN features.

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

Perform the validation step of DFKDE.

Similar to the training step, features are extracted from the CNN for each batch.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Input batch
- **args** – Arguments.
- **kwargs** – Keyword arguments.

Return type

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

Returns

Dictionary containing probability, prediction and ground truth values.

Normality model of DFKDE.

class anomalib.models.image.dfkde.torch_model.**DfkdeModel**(*backbone*, *layers*, *pre_trained=True*, *n_pca_components=16*, *feature_scaling_method=FeatureScalingMethod.SCALE*, *max_training_points=40000*)

Bases: `Module`

Normality Model for the DFKDE algorithm.

Parameters

- **backbone** (*str*) – Pre-trained model backbone.
- **layers** (*Sequence[str]*) – Layers to extract features from.
- **pre_trained** (*bool, optional*) – Boolean to check whether to use a pre_trained backbone. Defaults to `True`.
- **n_pca_components** (*int, optional*) – Number of PCA components. Defaults to 16.
- **feature_scaling_method** (*FeatureScalingMethod, optional*) – Feature scaling method. Defaults to `FeatureScalingMethod.SCALE`.

- **max_training_points** (*int, optional*) – Number of training points to fit the KDE model. Defaults to 40000.

forward(*batch*)

Prediction by normality model.

Parameters
batch (*torch.Tensor*) – Input images.

Returns
Predictions

Return type
Tensor

get_features(*batch*)

Extract features from the pretrained network.

Parameters
batch (*torch.Tensor*) – Image batch.

Returns
torch.Tensor containing extracted features.

Return type
Tensor

DFM

DFM: Deep Feature Modeling.

https://arxiv.org/abs/1909.11786

class anomalib.models.image.dfm.lightning_model.**Dfm**(*backbone='resnet50'*, *layer='layer3'*, *pre_trained=True*, *pooling_kernel_size=4*, *pca_level=0.97*, *score_type='fre'*)

Bases: MemoryBankMixin, AnomalyModule

DFM: Deep Featured Kernel Density Estimation.

Parameters

- **backbone** (*str*) – Backbone CNN network Defaults to "resnet50".
- **layer** (*str*) – Layer to extract features from the backbone CNN Defaults to "layer3".
- **pre_trained** (*bool, optional*) – Boolean to check whether to use a pre_trained backbone. Defaults to True.
- **pooling_kernel_size** (*int, optional*) – Kernel size to pool features extracted from the CNN. Defaults to 4.
- **pca_level** (*float, optional*) – Ratio from which number of components for PCA are calculated. Defaults to 0.97.
- **score_type** (*str, optional*) – Scoring type. Options are *fre* and *nll*. Defaults to fre.

static configure_optimizers()

DFM doesn't require optimization, therefore returns no optimizers.

Return type
None

`fit()`

Fit a PCA transformation and a Gaussian model to dataset.

Return type
`None`

`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 DFM-specific trainer arguments.

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

Perform the training step of DFM.

For each batch, features are extracted from the CNN.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Input batch
- **args** – Arguments.
- **kwargs** – Keyword arguments.

Return type
`None`

Returns
Deep CNN features.

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

Perform the validation step of DFM.

Similar to the training step, features are extracted from the CNN for each batch.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Input batch
- **args** – Arguments.
- **kwargs** – Keyword arguments.

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

Returns
Dictionary containing FRE anomaly scores and anomaly maps.

PyTorch model for DFM model implementation.

`class anomalib.models.image.dfm.torch_model.DFMModel`(*backbone*, *layer*, *pre_trained=True*, *pooling_kernel_size=4*, *n_comps=0.97*, *score_type='fre'*)

Bases: `Module`

Model for the DFM algorithm.

Parameters

- **backbone** (*str*) – Pre-trained model backbone.
- **layer** (*str*) – Layer from which to extract features.
- **pre_trained** (*bool, optional*) – Boolean to check whether to use a pre_trained backbone. Defaults to `True`.
- **pooling_kernel_size** (*int, optional*) – Kernel size to pool features extracted from the CNN. Defaults to `4`.
- **n_comps** (*float, optional*) – Ratio from which number of components for PCA are calculated. Defaults to `0.97`.
- **score_type** (*str, optional*) – Scoring type. Options are *fre* and *nll*. Anomaly Defaults to `fre`. Segmentation is supported with *fre* only. If using *nll*, set *task* in config.yaml to classification Defaults to `classification`.

`fit`(*dataset*)

Fit a pca transformation and a Gaussian model to dataset.

Parameters
dataset (*torch.Tensor*) – Input dataset to fit the model.

Return type
`None`

`forward`(*batch*)

Compute score from input images.

Parameters
batch (*torch.Tensor*) – Input images

Returns
Scores

Return type
Tensor

`get_features`(*batch*)

Extract features from the pretrained network.

Parameters
batch (*torch.Tensor*) – Image batch.

Returns
torch.Tensor containing extracted features.

Return type
Tensor

`score`(*features*, *feature_shapes*)

Compute scores.

Scores are either PCA-based feature reconstruction error (FRE) scores or the Gaussian density-based NLL scores

Parameters

- **features** (*torch.Tensor*) – semantic features on which PCA and density modeling is performed.

- **feature_shapes** (*tuple*) – shape of *features* tensor. Used to generate anomaly map of correct shape.

Returns
numpy array of scores

Return type
score (torch.Tensor)

class anomalib.models.image.dfm.torch_model.**SingleClassGaussian**

Bases: DynamicBufferMixin

Model Gaussian distribution over a set of points.

fit(*dataset*)

Fit a Gaussian model to dataset X.

Covariance matrix is not calculated directly using: `C = X.X^T` Instead, it is represented in terms of the Singular Value Decomposition of X: `X = U.S.V^T` Hence, `C = U.S^2.U^T` This simplifies the calculation of the log-likelihood without requiring full matrix inversion.

Parameters
dataset (*torch.Tensor*) – Input dataset to fit the model.

Return type
None

forward(*dataset*)

Provide the same functionality as *fit*.

Transforms the input dataset based on singular values calculated earlier.

Parameters
dataset (*torch.Tensor*) – Input dataset

Return type
None

score_samples(*features*)

Compute the NLL (negative log likelihood) scores.

Parameters
features (*torch.Tensor*) – semantic features on which density modeling is performed.

Returns
Torch tensor of scores

Return type
nll (torch.Tensor)

DRAEM

DRÆM - A discriminatively trained reconstruction embedding for surface anomaly detection.

Paper https://arxiv.org/abs/2108.07610

class anomalib.models.image.draem.lightning_model.**Draem**(*enable_sspcab=False*, *sspcab_lambda=0.1*, *anomaly_source_path=None*, *beta=(0.1, 1.0)*)

Bases: AnomalyModule

DRÆM: A discriminatively trained reconstruction embedding for surface anomaly detection.

Parameters

- **enable_sspcab** (*bool*) – Enable SSPCAB training. Defaults to `False`.
- **sspcab_lambda** (*float*) – SSPCAB loss weight. Defaults to `0.1`.
- **anomaly_source_path** (*str | None*) – Path to folder that contains the anomaly source images. Random noise will be used if left empty. Defaults to `None`.

configure_optimizers()

Configure the Adam optimizer.

Return type
`Optimizer`

property learning_type: LearningType

Return the learning type of the model.

Returns
Learning type of the model.

Return type
LearningType

setup_sspcab()

Prepare the model for the SSPCAB training step by adding forward hooks for the SSPCAB layer activations.

Return type
`None`

property trainer_arguments: dict[str, Any]

Return DRÆM-specific trainer arguments.

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

Perform the training step of DRAEM.

Feeds the original image and the simulated anomaly image through the network and computes the training loss.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Batch containing image filename, image, label and mask
- **args** – Arguments.
- **kwargs** – Keyword arguments.

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

Returns
Loss dictionary

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

Perform the validation step of DRAEM. The Softmax predictions of the anomalous class are used as anomaly map.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Batch of input images
- **args** – Arguments.
- **kwargs** – Keyword arguments.

Return type
Union[Tensor, Mapping[str, Any], None]

Returns
Dictionary to which predicted anomaly maps have been added.

PyTorch model for the DRAEM model implementation.

class anomalib.models.image.draem.torch_model.**DraemModel**(*sspcab=False*)

Bases: Module

DRAEM PyTorch model consisting of the reconstructive and discriminative sub networks.

Parameters
sspcab (*bool*) – Enable SSPCAB training. Defaults to False.

forward(*batch*)

Compute the reconstruction and anomaly mask from an input image.

Parameters
batch (*torch.Tensor*) – batch of input images

Return type
Tensor | tuple[Tensor, Tensor]

Returns
Predicted confidence values of the anomaly mask. During training the reconstructed input images are returned as well.

Loss function for the DRAEM model implementation.

class anomalib.models.image.draem.loss.**DraemLoss**

Bases: Module

Overall loss function of the DRAEM model.

The total loss consists of the sum of the L2 loss and Focal loss between the reconstructed image and the input image, and the Structural Similarity loss between the predicted and GT anomaly masks.

forward(*input_image*, *reconstruction*, *anomaly_mask*, *prediction*)

Compute the loss over a batch for the DRAEM model.

Return type
Tensor

DSR

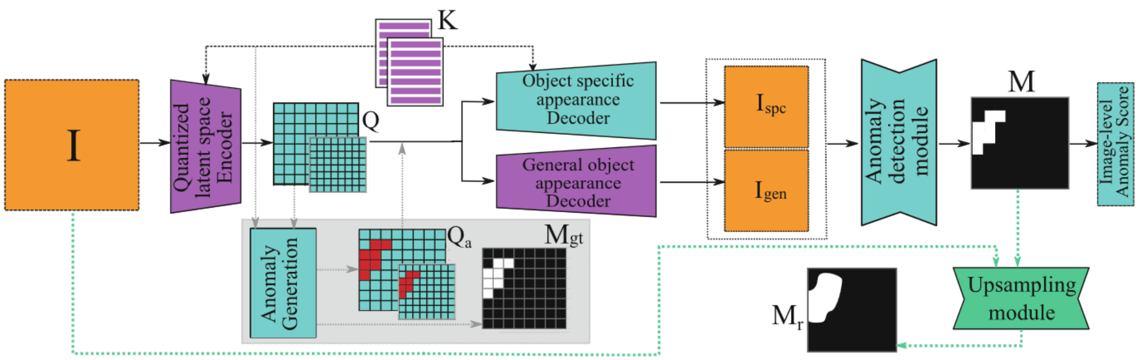
This is the implementation of the DSR paper.

Model Type: Segmentation

Description

DSR is a quantized-feature based algorithm that consists of an autoencoder with one encoder and two decoders, coupled with an anomaly detection module. DSR learns a codebook of quantized representations on ImageNet, which are then used to encode input images. These quantized representations also serve to sample near-in-distribution anomalies, since they do not rely on external datasets. Training takes place in three phases. The encoder and "general object decoder", as well as the codebook, are pretrained on ImageNet. Defects are then generated at the feature level using the codebook on the quantized representations, and are used to train the object-specific decoder as well as the anomaly detection module. In the final phase of training, the upsampling module is trained on simulated image-level smudges in order to output more robust anomaly maps.

Architecture



PyTorch model for the DSR model implementation.

class anomalib.models.image.dsr.torch_model.**AnomalyDetectionModule**(*in_channels*, *out_channels*, *base_width*)

Bases: `Module`

Anomaly detection module.

Module that detects the preseßnce of an anomaly by comparing two images reconstructed by the object specific decoder and the general object decoder.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **out_channels** (*int*) – Number of output channels.
- **base_width** (*int*) – Base dimensionality of the layers of the autoencoder.

forward(*batch_real*, *batch_anomaly*)

Computes the anomaly map over corresponding real and anomalous images.

Parameters

- **batch_real** (*torch.Tensor*) – Batch of real, non defective images.
- **batch_anomaly** (*torch.Tensor*) – Batch of potentially anomalous images.

Return type

`Tensor`

Returns
The anomaly segmentation map.

class anomalib.models.image.dsr.torch_model.**DecoderBot**(*in_channels*, *num_hiddens*, *num_residual_layers*, *num_residual_hiddens*)

Bases: `Module`

General appearance decoder module to reconstruct images while keeping possible anomalies.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **num_hiddens** (*int*) – Number of hidden channels.
- **num_residual_layers** (*int*) – Number of residual layers in residual stack.
- **num_residual_hiddens** (*int*) – Number of channels in residual layers.

forward(*inputs*)

Decode quantized feature maps into an image.

Parameters
inputs (*torch.Tensor*) – Quantized feature maps.

Return type
`Tensor`

Returns
Decoded image.

class anomalib.models.image.dsr.torch_model.**DiscreteLatentModel**(*num_hiddens*, *num_residual_layers*, *num_residual_hiddens*, *num_embeddings*, *embedding_dim*)

Bases: `Module`

Discrete Latent Model.

Autoencoder quantized model that encodes the input images into quantized feature maps and generates a reconstructed image using the general appearance decoder.

Parameters

- **num_hiddens** (*int*) – Number of hidden channels.
- **num_residual_layers** (*int*) – Number of residual layers in residual stacks.
- **num_residual_hiddens** (*int*) – Number of channels in residual layers.
- **num_embeddings** (*int*) – Size of embedding dictionary.
- **embedding_dim** (*int*) – Dimension of embeddings.

forward(*batch*, *anomaly_mask=None*, *anom_str_lo=None*, *anom_str_hi=None*)

Generate quantized feature maps.

Generates quantized feature maps of batch of input images as well as their reconstruction based on the general appearance decoder.

Parameters

- **batch** (*Tensor*) – Batch of input images.

- **anomaly_mask** (*Tensor | None*) – Anomaly mask to be used to generate anomalies on the quantized feature maps.
- **anom_str_lo** (*torch.Tensor | None*) – Strength of generated anomaly lo.
- **anom_str_hi** (*torch.Tensor | None*) – Strength of generated anomaly hi.

Returns

If generating an anomaly mask:

- General object decoder-decoded anomalous image
- Reshaped ground truth anomaly map
- Non defective quantized lo feature
- Non defective quantized hi feature
- Non quantized subspace encoded defective lo feature
- Non quantized subspace encoded defective hi feature

Else:

- General object decoder-decoded image
- Quantized lo feature
- Quantized hi feature

Return type
dict[str, torch.Tensor]

generate_fake_anomalies_joined(*features*, *embeddings*, *memory_torch_original*, *mask*, *strength*)

Generate quantized anomalies.

Parameters

- **features** (*torch.Tensor*) – Features on which the anomalies will be generated.
- **embeddings** (*torch.Tensor*) – Embeddings to use to generate the anomalies.
- **memory_torch_original** (*torch.Tensor*) – Weight of embeddings.
- **mask** (*torch.Tensor*) – Original anomaly mask.
- **strength** (*float*) – Strength of generated anomaly.

Returns
Anomalous embedding.

Return type
torch.Tensor

property vq_vae_bot: *VectorQuantizer*

Return `self._vq_vae_bot`.

property vq_vae_top: *VectorQuantizer*

Return `self._vq_vae_top`.

class anomalib.models.image.dsr.torch_model.**DsrModel**(*latent_anomaly_strength=0.2*, *embedding_dim=128*, *num_embeddings=4096*, *num_hiddens=128*, *num_residual_layers=2*, *num_residual_hiddens=64*)

Bases: `Module`

DSR PyTorch model.

Consists of the discrete latent model, image reconstruction network, subspace restriction modules, anomaly detection module and upsampling module.

Parameters

- **embedding_dim** (*int*) – Dimension of codebook embeddings.
- **num_embeddings** (*int*) – Number of embeddings.
- **latent_anomaly_strength** (*float*) – Strength of the generated anomalies in the latent space.
- **num_hiddens** (*int*) – Number of output channels in residual layers.
- **num_residual_layers** (*int*) – Number of residual layers.
- **num_residual_hiddens** (*int*) – Number of intermediate channels.

forward(*batch*, *anomaly_map_to_generate=None*)

Compute the anomaly mask from an input image.

Parameters

- **batch** (*torch.Tensor*) – Batch of input images.
- **anomaly_map_to_generate** (*torch.Tensor | None*) – anomaly map to use to generate quantized defects.
- **2** (*If not training phase*) –
- **None.** (*should be*) –

Returns

If testing:

- ”anomaly_map”: Upsampled anomaly map
- ”pred_score”: Image score

If training phase 2:

- ”recon_feat_hi”: Reconstructed non-quantized hi features of defect (F~_hi)
- ”recon_feat_lo”: Reconstructed non-quantized lo features of defect (F~_lo)
- ”embedding_bot”: Quantized features of non defective img (Q_hi)
- ”embedding_top”: Quantized features of non defective img (Q_lo)
- ”obj_spec_image”: Object-specific-decoded image (I_spc)
- ”anomaly_map”: Predicted segmentation mask (M)
- ”true_mask”: Resized ground-truth anomaly map (M_gt)

If training phase 3:

- ”anomaly_map”: Reconstructed anomaly map

Return type

dict[str, torch.Tensor]

`load_pretrained_discrete_model_weights`(*ckpt*)

Load pre-trained model weights.

Return type
None

class anomalib.models.image.dsr.torch_model.**EncoderBot**(*in_channels*, *num_hiddens*, *num_residual_layers*, *num_residual_hiddens*)

Bases: `Module`

Encoder module for bottom quantized feature maps.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **num_hiddens** (*int*) – Number of hidden channels.
- **num_residual_layers** (*int*) – Number of residual layers in residual stacks.
- **num_residual_hiddens** (*int*) – Number of channels in residual layers.

forward(*batch*)

Encode inputs to be quantized into the bottom feature map.

Parameters
batch (*torch.Tensor*) – Batch of input images.

Return type
Tensor

Returns
Encoded feature maps.

class anomalib.models.image.dsr.torch_model.**EncoderTop**(*in_channels*, *num_hiddens*, *num_residual_layers*, *num_residual_hiddens*)

Bases: `Module`

Encoder module for top quantized feature maps.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **num_hiddens** (*int*) – Number of hidden channels.
- **num_residual_layers** (*int*) – Number of residual layers in residual stacks.
- **num_residual_hiddens** (*int*) – Number of channels in residual layers.

forward(*batch*)

Encode inputs to be quantized into the top feature map.

Parameters
batch (*torch.Tensor*) – Batch of input images.

Return type
Tensor

Returns
Encoded feature maps.

class anomalib.models.image.dsr.torch_model.**FeatureDecoder**(*base_width*, *out_channels=1*)

Bases: `Module`

Feature decoder for the subspace restriction network.

Parameters

- **base_width** (*int*) – Base dimensionality of the layers of the autoencoder.
- **out_channels** (*int*) – Number of output channels.

forward(_, __, *b3*)

Decode a batch of latent features to a non-quantized representation.

Parameters

- **_** (*torch.Tensor*) – Top latent feature layer.
- **__** (*torch.Tensor*) – Middle latent feature layer.
- **b3** (*torch.Tensor*) – Bottom latent feature layer.

Return type

`Tensor`

Returns

Decoded non-quantized representation.

class anomalib.models.image.dsr.torch_model.**FeatureEncoder**(*in_channels*, *base_width*)

Bases: `Module`

Feature encoder for the subspace restriction network.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **base_width** (*int*) – Base dimensionality of the layers of the autoencoder.

forward(*batch*)

Encode a batch of input features to the latent space.

Parameters

batch (*torch.Tensor*) – Batch of input images.

Return type

`tuple[Tensor, Tensor, Tensor]`

Returns: Encoded feature maps.

class anomalib.models.image.dsr.torch_model.**ImageReconstructionNetwork**(*in_channels*, *num_hiddens*, *num_residual_layers*, *num_residual_hiddens*)

Bases: `Module`

Image Reconstruction Network.

Image reconstruction network that reconstructs the image from a quantized representation.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **num_hiddens** (*int*) – Number of output channels in residual layers.

- **num_residual_layers** (*int*) – Number of residual layers.
- **num_residual_hiddens** (*int*) – Number of intermediate channels.

forward(*inputs*)

Reconstructs an image from a quantized representation.

Parameters
inputs (*torch.Tensor*) – Quantized features.

Return type
Tensor

Returns
Reconstructed image.

class anomalib.models.image.dsr.torch_model.**Residual**(*in_channels*, *out_channels*, *num_residual_hiddens*)

Bases: Module

Residual layer.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **out_channels** (*int*) – Number of output channels.
- **num_residual_hiddens** (*int*) – Number of intermediate channels.

forward(*batch*)

Compute residual layer.

Parameters
batch (*torch.Tensor*) – Batch of input images.

Return type
Tensor

Returns
Computed feature maps.

class anomalib.models.image.dsr.torch_model.**ResidualStack**(*in_channels*, *num_hiddens*, *num_residual_layers*, *num_residual_hiddens*)

Bases: Module

Stack of residual layers.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **num_hiddens** (*int*) – Number of output channels in residual layers.
- **num_residual_layers** (*int*) – Number of residual layers.
- **num_residual_hiddens** (*int*) – Number of intermediate channels.

forward(*batch*)

Compute residual stack.

Parameters
batch (*torch.Tensor*) – Batch of input images.

Return type
Tensor

Returns
Computed feature maps.

class anomalib.models.image.dsr.torch_model.**SubspaceRestrictionModule**(*base_width*)

Bases: Module

Subspace Restriction Module.

Subspace restriction module that restricts the appearance subspace into configurations that agree with normal appearances and applies quantization.

Parameters
base_width (*int*) – Base dimensionality of the layers of the autoencoder.

forward(*batch*, *quantization*)

Generate the quantized anomaly-free representation of an anomalous image.

Parameters

- **batch** (*torch.Tensor*) – Batch of input images.
- **quantization** (*function | object*) – Quantization function.

Return type
tuple[Tensor, Tensor]

Returns
Reconstructed batch of non-quantized features and corresponding quantized features.

class anomalib.models.image.dsr.torch_model.**SubspaceRestrictionNetwork**(*in_channels=64*, *out_channels=64*, *base_width=64*)

Bases: Module

Subspace Restriction Network.

Subspace restriction network that reconstructs the input image into a non-quantized configuration that agrees with normal appearances.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **out_channels** (*int*) – Number of output channels.
- **base_width** (*int*) – Base dimensionality of the layers of the autoencoder.

forward(*batch*)

Reconstruct non-quantized representation from batch.

Generate non-quantized feature maps from potentially anomalous images, to be quantized into non-anomalous quantized representations.

Parameters
batch (*torch.Tensor*) – Batch of input images.

Return type
Tensor

Returns
Reconstructed non-quantized representation.

class anomalib.models.image.dsr.torch_model.**UnetDecoder**(*base_width*, *out_channels=1*)

Bases: Module

Decoder of the Unet network.

Parameters

- **base_width** (*int*) – Base dimensionality of the layers of the autoencoder.
- **out_channels** (*int*) – Number of output channels.

forward(*b1*, *b2*, *b3*, *b4*)

Decodes latent represnetations into an image.

Parameters

- **b1** (*torch.Tensor*) – First (top level) quantized feature map.
- **b2** (*torch.Tensor*) – Second quantized feature map.
- **b3** (*torch.Tensor*) – Third quantized feature map.
- **b4** (*torch.Tensor*) – Fourth (bottom level) quantized feature map.

Return type

Tensor

Returns

Reconstructed image.

class anomalib.models.image.dsr.torch_model.**UnetEncoder**(*in_channels*, *base_width*)

Bases: Module

Encoder of the Unet network.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **base_width** (*int*) – Base dimensionality of the layers of the autoencoder.

forward(*batch*)

Encodes batch of images into a latent representation.

Parameters

batch (*torch.Tensor*) – Quantized features.

Return type

tuple[Tensor, Tensor, Tensor, Tensor]

Returns

Latent representations of the input batch.

class anomalib.models.image.dsr.torch_model.**UnetModel**(*in_channels=64*, *out_channels=64*, *base_width=64*)

Bases: Module

Autoencoder model that reconstructs the input image.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **out_channels** (*int*) – Number of output channels.
- **base_width** (*int*) – Base dimensionality of the layers of the autoencoder.

forward(*batch*)

Reconstructs an input batch of images.

Parameters
`batch` (*torch.Tensor*) – Batch of input images.

Return type
`Tensor`

Returns
Reconstructed images.

class anomalib.models.image.dsr.torch_model.**UpsamplingModule**(*in_channels=8*, *out_channels=2*, *base_width=64*)

Bases: `Module`

Module that upsamples the generated anomaly mask to full resolution.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **out_channels** (*int*) – Number of output channels.
- **base_width** (*int*) – Base dimensionality of the layers of the autoencoder.

forward(*batch_real*, *batch_anomaly*, *batch_segmentation_map*)

Computes upsampled segmentation maps.

Parameters

- **batch_real** (*torch.Tensor*) – Batch of real, non defective images.
- **batch_anomaly** (*torch.Tensor*) – Batch of potentially anomalous images.
- **batch_segmentation_map** (*torch.Tensor*) – Batch of anomaly segmentation maps.

Return type
`Tensor`

Returns
Upsampled anomaly segmentation maps.

class anomalib.models.image.dsr.torch_model.**VectorQuantizer**(*num_embeddings*, *embedding_dim*)

Bases: `Module`

Module that quantizes a given feature map using learned quantization codebooks.

Parameters

- **num_embeddings** (*int*) – Size of embedding codebook.
- **embedding_dim** (*int*) – Dimension of embeddings.

property embedding: Tensor

Return embedding.

forward(*inputs*)

Calculates quantized feature map.

Parameters
`inputs` (*torch.Tensor*) – Non-quantized feature maps.

Return type
`Tensor`

Returns
Quantized feature maps.

DSR - A Dual Subspace Re-Projection Network for Surface Anomaly Detection.

Paper https://link.springer.com/chapter/10.1007/978-3-031-19821-2_31

class `anomalib.models.image.dsr.lightning_model.Dsr`(*latent_anomaly_strength=0.2*, *upsampling_train_ratio=0.7*)

Bases: `AnomalyModule`

DSR: A Dual Subspace Re-Projection Network for Surface Anomaly Detection.

Parameters

- **latent_anomaly_strength** (*float*) – Strength of the generated anomalies in the latent space. Defaults to 0.2
- **upsampling_train_ratio** (*float*) – Ratio of training steps for the upsampling module. Defaults to 0.7

`configure_optimizers()`

Configure the Adam optimizer for training phases 2 and 3.

Does not train the discrete model (phase 1)

Returns
Dictionary of optimizers

Return type
dict[str, torch.optim.Optimizer | torch.optim.lr_scheduler.LRScheduler]

property `learning_type`: **LearningType**

Return the learning type of the model.

Returns
Learning type of the model.

Return type
LearningType

`on_train_epoch_start()`

Display a message when starting to train the upsampling module.

Return type
`None`

`on_train_start()`

Load pretrained weights of the discrete model when starting training.

Return type
`None`

`prepare_pretrained_model()`

Download pre-trained models if they don't exist.

Return type
`Path`

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

Required trainer arguments.

training_step(*batch*)

Training Step of DSR.

Feeds the original image and the simulated anomaly mask during first phase. During second phase, feeds a generated anomalous image to train the upsampling module.

Parameters
batch (*dict[str, str | Tensor]*) – Batch containing image filename, image, label and mask

Returns
Loss dictionary

Return type
STEP_OUTPUT

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

Validation step of DSR.

The Softmax predictions of the anomalous class are used as anomaly map.

Parameters

- **batch** (*dict[str, str | Tensor]*) – Batch of input images
- ***args** – unused
- ****kwargs** – unused

Returns
Dictionary to which predicted anomaly maps have been added.

Return type
STEP_OUTPUT

Anomaly generator for the DSR model implementation.

class anomalib.models.image.dsr.anomaly_generator.**DsrAnomalyGenerator**(*p_anomalous=0.5*)

Bases: `Module`

Anomaly generator of the DSR model.

The anomaly is generated using a Perlin noise generator on the two quantized representations of an image. This generator is only used during the second phase of training! The third phase requires generating smudges over the input images.

Parameters
p_anomalous (*float, optional*) – Probability to generate an anomalous image.

augment_batch(*batch*)

Generate anomalous augmentations for a batch of input images.

Parameters
batch (*Tensor*) – Batch of input images

Returns
Ground truth masks corresponding to the anomalous perturbations.

Return type
Tensor

generate_anomaly(*height*, *width*)

Generate an anomalous mask.

Parameters

- **height** (*int*) – Height of generated mask.
- **width** (*int*) – Width of generated mask.

Returns
Generated mask.

Return type
Tensor

Loss function for the DSR model implementation.

class anomalib.models.image.dsr.loss.**DsrSecondStageLoss**

Bases: `Module`

Overall loss function of the second training phase of the DSR model.

The total loss consists of:

- MSE loss between non-anomalous quantized input image and anomalous subspace-reconstructed non-quantized input (hi and lo)
- MSE loss between input image and reconstructed image through object-specific decoder,
- Focal loss between computed segmentation mask and ground truth mask.

forward(*recon_nq_hi*, *recon_nq_lo*, *qu_hi*, *qu_lo*, *input_image*, *gen_img*, *seg*, *anomaly_mask*)

Compute the loss over a batch for the DSR model.

Parameters

- **recon_nq_hi** (*Tensor*) – Reconstructed non-quantized hi feature
- **recon_nq_lo** (*Tensor*) – Reconstructed non-quantized lo feature
- **qu_hi** (*Tensor*) – Non-defective quantized hi feature
- **qu_lo** (*Tensor*) – Non-defective quantized lo feature
- **input_image** (*Tensor*) – Original image
- **gen_img** (*Tensor*) – Object-specific decoded image
- **seg** (*Tensor*) – Computed anomaly map
- **anomaly_mask** (*Tensor*) – Ground truth anomaly map

Returns
Total loss

Return type
Tensor

class anomalib.models.image.dsr.loss.**DsrThirdStageLoss**

Bases: `Module`

Overall loss function of the third training phase of the DSR model.

The loss consists of a focal loss between the computed segmentation mask and the ground truth mask.

forward(*pred_mask*, *true_mask*)

Compute the loss over a batch for the DSR model.

Parameters

- **pred_mask** (*Tensor*) – Computed anomaly map
- **true_mask** (*Tensor*) – Ground truth anomaly map

Returns
Total loss

Return type
Tensor

Efficient AD

EfficientAd: Accurate Visual Anomaly Detection at Millisecond-Level Latencies.

https://arxiv.org/pdf/2303.14535.pdf.

class anomalib.models.image.efficient_ad.lightning_model.**EfficientAd**(*imagenet_dir='./datasets/imagenette'*, *teacher_out_channels=384*, *model_size=EfficientAdModelSize.S*, *lr=0.0001*, *weight_decay=1e-05*, *padding=False*, *pad_maps=True*, *batch_size=1*)

Bases: `AnomalyModule`

PL Lightning Module for the EfficientAd algorithm.

Parameters

- **imagenet_dir** (*Path | str*) – directory path for the Imagenet dataset Defaults to `./datasets/imagenette`.
- **teacher_out_channels** (*int*) – number of convolution output channels Defaults to `384`.
- **model_size** (*str*) – size of student and teacher model Defaults to `EfficientAdModelSize.S`.
- **lr** (*float*) – learning rate Defaults to `0.0001`.
- **weight_decay** (*float*) – optimizer weight decay Defaults to `0.00001`.
- **padding** (*bool*) – use padding in convoluional layers Defaults to `False`.
- **pad_maps** (*bool*) – relevant if padding is set to False. In this case, pad_maps = True pads the output anomaly maps so that their size matches the size in the padding = True case. Defaults to `True`.
- **batch_size** (*int*) – batch size for imagenet dataloader Defaults to 1.

configure_optimizers()

Configure optimizers.

Return type
`Optimizer`

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

map_norm_quantiles(*dataloader*)

Calculate 90% and 99.5% quantiles of the student(st) and autoencoder(ae).

Parameters
dataloader (*DataLoader*) – Dataloader of the respective dataset.

Returns
Dictionary of both the 90% and 99.5% quantiles of both the student and autoencoder feature maps.

Return type
dict[str, torch.Tensor]

on_train_start()

Called before the first training epoch.

First sets up the pretrained teacher model, then prepares the imagenette data, and finally calculates or loads the channel-wise mean and std of the training dataset and push to the model.

Return type
None

on_validation_start()

Calculate the feature map quantiles of the validation dataset and push to the model.

Return type
None

prepare_imagenette_data(*image_size*)

Prepare ImageNette dataset transformations.

Parameters
image_size (*tuple[int, int] | torch.Size*) – Image size.

Return type
None

prepare_pretrained_model()

Prepare the pretrained teacher model.

Return type
None

teacher_channel_mean_std(*dataloader*)

Calculate the mean and std of the teacher models activations.

Adapted from https://math.stackexchange.com/a/2148949

Parameters
dataloader (*DataLoader*) – Dataloader of the respective dataset.

Returns
Dictionary of channel-wise mean and std

Return type
dict[str, torch.Tensor]

property trainer_arguments: dict[str, Any]

Return EfficientAD trainer arguments.

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

Perform the training step for EfficientAd returns the student, autoencoder and combined loss.

Parameters

- **(batch** (*batch*) – dict[str, str | torch.Tensor]): Batch containing image filename, image, label and mask
- **args** – Additional arguments.
- **kwargs** – Additional keyword arguments.

Return type
dict[str, Tensor]

Returns
Loss.

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

Perform the validation step of EfficientAd returns anomaly maps for the input image batch.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Input batch
- **args** – Additional arguments.
- **kwargs** – Additional keyword arguments.

Return type
Union[Tensor, Mapping[str, Any], None]

Returns
Dictionary containing anomaly maps.

Torch model for student, teacher and autoencoder model in EfficientAd.

class anomalib.models.image.efficient_ad.torch_model.**EfficientAdModel**(*teacher_out_channels*, *model_size=EfficientAdModelSize.S*, *padding=False*, *pad_maps=True*)

Bases: Module

EfficientAd model.

Parameters

- **teacher_out_channels** (*int*) – number of convolution output channels of the pre-trained teacher model
- **model_size** (*str*) – size of student and teacher model
- **padding** (*bool*) – use padding in convoluional layers Defaults to False.
- **pad_maps** (*bool*) – relevant if padding is set to False. In this case, pad_maps = True pads the output anomaly maps so that their size matches the size in the padding = True case. Defaults to True.

choose_random_aug_image(*image*)

Choose a random augmentation function and apply it to the input image.

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

Returns
Augmented image.

Return type
Tensor

forward(*batch*, *batch_imagenet=None*, *normalize=True*)

Perform the forward-pass of the EfficientAd models.

Parameters

- **batch** (*torch.Tensor*) – Input images.
- **batch_imagenet** (*torch.Tensor*) – ImageNet batch. Defaults to None.
- **normalize** (*bool*) – Normalize anomaly maps or not

Returns
Predictions

Return type
Tensor

is_set(*p_dic*)

Check if any of the parameters in the parameter dictionary is set.

Parameters
p_dic (*nn.ParameterDict*) – Parameter dictionary.

Returns
Boolean indicating whether any of the parameters in the parameter dictionary is set.

Return type
bool

FastFlow

FastFlow Lightning Model Implementation.

https://arxiv.org/abs/2111.07677

class anomalib.models.image.fastflow.lightning_model.**Fastflow**(*backbone='resnet18'*, *pre_trained=True*, *flow_steps=8*, *conv3x3_only=False*, *hidden_ratio=1.0*)

Bases: `AnomalyModule`

PL Lightning Module for the FastFlow algorithm.

Parameters

- **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`.

configure_optimizers()

Configure optimizers for each decoder.

Returns
Adam optimizer for each decoder

Return type
Optimizer

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 FastFlow trainer arguments.

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

Perform the training step input and return the loss.

Parameters

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

Returns
Dictionary containing the loss value.

Return type
STEP_OUTPUT

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

Perform the validation step and return the anomaly map.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Input batch
- **args** – Additional arguments.
- **kwargs** – Additional keyword arguments.

Returns
batch dictionary containing anomaly-maps.

Return type
STEP_OUTPUT | None

FastFlow Torch Model Implementation.

class anomalib.models.image.fastflow.torch_model.**FastflowModel**(*input_size*, *backbone*, *pre_trained=True*, *flow_steps=8*, *conv3x3_only=False*, *hidden_ratio=1.0*)

Bases: `Module`

FastFlow.

Unsupervised Anomaly Detection and Localization via 2D Normalizing Flows.

Parameters

- **input_size** (*tuple[int, int]*) – Model input size.
- **backbone** (*str*) – Backbone CNN network
- **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`.

Raises

ValueError – When the backbone is not supported.

forward(*input_tensor*)

Forward-Pass the input to the FastFlow Model.

Parameters

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

Returns

During training, return

(hidden_variables, log-of-the-jacobian-determinants). During the validation/test, return the anomaly map.

Return type

Tensor | list[torch.Tensor] | tuple[list[torch.Tensor]]

Loss function for the FastFlow Model Implementation.

class anomalib.models.image.fastflow.loss.**FastflowLoss**(**args*, ***kwargs*)

Bases: `Module`

FastFlow Loss.

forward(*hidden_variables*, *jacobians*)

Calculate the Fastflow loss.

Parameters

- **hidden_variables** (*list[torch.Tensor]*) – Hidden variables from the fastflow model. f: X -> Z
- **jacobians** (*list[torch.Tensor]*) – Log of the jacobian determinants from the fastflow model.

Returns
Fastflow loss computed based on the hidden variables and the log of the Jacobians.

Return type
Tensor

FastFlow Anomaly Map Generator Implementation.

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

Bases: `Module`

Generate Anomaly Heatmap.

Parameters
input_size (*ListConfig | tuple*) – Input size.

`forward(hidden_variables)`

Generate Anomaly Heatmap.

This implementation generates the heatmap based on the flow maps computed from the normalizing flow (NF) FastFlow blocks. Each block yields a flow map, which overall is stacked and averaged to an anomaly map.

Parameters
hidden_variables (*list[torch.Tensor]*) – List of hidden variables from each NF FastFlow block.

Returns
Anomaly Map.

Return type
Tensor

GANomaly

GANomaly: Semi-Supervised Anomaly Detection via Adversarial Training.

https://arxiv.org/abs/1805.06725

`class anomalib.models.image.ganomaly.lightning_model.Ganomaly(batch_size=32, n_features=64, latent_vec_size=100, extra_layers=0, add_final_conv_layer=True, wadv=1, wcon=50, wenc=1, lr=0.0002, beta1=0.5, beta2=0.999)`

Bases: `AnomalyModule`

PL Lightning Module for the GANomaly Algorithm.

Parameters

- **batch_size** (*int*) – Batch size. Defaults to `32`.
- **n_features** (*int*) – Number of features layers in the CNNs. Defaults to `64`.
- **latent_vec_size** (*int*) – Size of autoencoder latent vector. Defaults to `100`.
- **extra_layers** (*int, optional*) – Number of extra layers for encoder/decoder. Defaults to `0`.

- **add_final_conv_layer** (*bool, optional*) – Add convolution layer at the end. Defaults to True.
- **wadv** (*int, optional*) – Weight for adversarial loss. Defaults to 1.
- **wcon** (*int, optional*) – Image regeneration weight. Defaults to 50.
- **wenc** (*int, optional*) – Latent vector encoder weight. Defaults to 1.
- **lr** (*float, optional*) – Learning rate. Defaults to 0.0002.
- **beta1** (*float, optional*) – Adam beta1. Defaults to 0.5.
- **beta2** (*float, optional*) – Adam beta2. Defaults to 0.999.

configure_optimizers()

Configure optimizers for each decoder.

Returns
Adam optimizer for each decoder

Return type
Optimizer

property learning_type: LearningType

Return the learning type of the model.

Returns
Learning type of the model.

Return type
LearningType

on_test_batch_end(*outputs*, *batch*, *batch_idx*, *dataloader_idx=0*)

Normalize outputs based on min/max values.

Return type
None

on_test_start()

Reset min max values before test batch starts.

Return type
None

on_validation_batch_end(*outputs*, *batch*, *batch_idx*, *dataloader_idx=0*)

Normalize outputs based on min/max values.

Return type
None

on_validation_start()

Reset min and max values for current validation epoch.

Return type
None

test_step(*batch*, *batch_idx*, **args*, ***kwargs*)

Update min and max scores from the current step.

Return type
Union[Tensor, Mapping[str, Any], None]

property trainer_arguments: dict[str, Any]

Return GANomaly trainer arguments.

training_step(*batch*, *batch_idx*)

Perform the training step.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Input batch containing images.
- **batch_idx** (*int*) – Batch index.
- **optimizer_idx** (*int*) – Optimizer which is being called for current training step.

Returns
Loss

Return type
STEP_OUTPUT

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

Update min and max scores from the current step.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Predicted difference between z and z_hat.
- **args** – Additional arguments.
- **kwargs** – Additional keyword arguments.

Returns
Output predictions.

Return type
(STEP_OUTPUT)

Torch models defining encoder, decoder, Generator and Discriminator.

Code adapted from https://github.com/samet-akcay/ganomaly.

class anomalib.models.image.ganomaly.torch_model.**GanomalyModel**(*input_size*, *num_input_channels*, *n_features*, *latent_vec_size*, *extra_layers=0*, *add_final_conv_layer=True*)

Bases: `Module`

Ganomaly Model.

Parameters

- **input_size** (*tuple[int, int]*) – Input dimension.
- **num_input_channels** (*int*) – Number of input channels.
- **n_features** (*int*) – Number of features layers in the CNNs.
- **latent_vec_size** (*int*) – Size of autoencoder latent vector.
- **extra_layers** (*int, optional*) – Number of extra layers for encoder/decoder. Defaults to `0`.
- **add_final_conv_layer** (*bool, optional*) – Add convolution layer at the end. Defaults to `True`.

forward(*batch*)

Get scores for batch.

Parameters
`batch` (*torch.Tensor*) – Images

Returns
Regeneration scores.

Return type
Tensor

static weights_init(*module*)

Initialize DCGAN weights.

Parameters
`module` (*nn.Module*) – [description]

Return type
`None`

Loss function for the GANomaly Model Implementation.

class `anomalib.models.image.ganomaly.loss.`**DiscriminatorLoss**

Bases: `Module`

Discriminator loss for the GANomaly model.

forward(*pred_real*, *pred_fake*)

Compute the loss for a predicted batch.

Parameters

- **pred_real** (*torch.Tensor*) – Discriminator predictions for the real image.
- **pred_fake** (*torch.Tensor*) – Discriminator predictions for the fake image.

Returns
The computed discriminator loss.

Return type
Tensor

class `anomalib.models.image.ganomaly.loss.`**GeneratorLoss**(*wadv=1*, *wcon=50*, *wenc=1*)

Bases: `Module`

Generator loss for the GANomaly model.

Parameters

- **wadv** (*int, optional*) – Weight for adversarial loss. Defaults to 1.
- **wcon** (*int, optional*) – Image regeneration weight. Defaults to `50`.
- **wenc** (*int, optional*) – Latent vector encoder weight. Defaults to 1.

forward(*latent_i*, *latent_o*, *images*, *fake*, *pred_real*, *pred_fake*)

Compute the loss for a batch.

Parameters

- **latent_i** (*torch.Tensor*) – Latent features of the first encoder.
- **latent_o** (*torch.Tensor*) – Latent features of the second encoder.
- **images** (*torch.Tensor*) – Real image that served as input of the generator.

- **fake** (*torch.Tensor*) – Generated image.
- **pred_real** (*torch.Tensor*) – Discriminator predictions for the real image.
- **pred_fake** (*torch.Tensor*) – Discriminator predictions for the fake image.

Returns
The computed generator loss.

Return type
Tensor

Padim

PaDiM: a Patch Distribution Modeling Framework for Anomaly Detection and Localization.

Paper https://arxiv.org/abs/2011.08785

`class anomalib.models.image.padim.lightning_model.Padim`(*backbone='resnet18'*, *layers=['layer1', 'layer2', 'layer3']*, *pre_trained=True*, *n_features=None*)

Bases: `MemoryBankMixin`, `AnomalyModule`

PaDiM: a Patch Distribution Modeling Framework for Anomaly Detection and Localization.

Parameters

- **backbone** (*str*) – Backbone CNN network Defaults to `resnet18`.
- **layers** (*list[str]*) – Layers to extract features from the backbone CNN Defaults to `["layer1", "layer2", "layer3"]`.
- **pre_trained** (*bool, optional*) – Boolean to check whether to use a pre_trained backbone. Defaults to `True`.
- **n_features** (*int, optional*) – Number of features to retain in the dimension reduction step. Default values from the paper are available for: resnet18 (100), wide_resnet50_2 (550). Defaults to `None`.

`static configure_optimizers()`

PADIM doesn't require optimization, therefore returns no optimizers.

Return type
`None`

`configure_transforms`(*image_size=None*)

Default transform for Padim.

Return type
`Transform`

`fit()`

Fit a Gaussian to the embedding collected from the training set.

Return type
`None`

`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, int | float]

Return PADIM trainer arguments.

Since the model does not require training, we limit the max_epochs to 1. Since we need to run training epoch before validation, we also set the sanity steps to 0

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

Perform the training step of PADIM. For each batch, hierarchical features are extracted from the CNN.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Batch containing image filename, image, label and mask
- **args** – Additional arguments.
- **kwargs** – Additional keyword arguments.

Return type
`None`

Returns
Hierarchical feature map

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

Perform a validation step of PADIM.

Similar to the training step, hierarchical features are extracted from the CNN for each batch.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Input batch
- **args** – Additional arguments.
- **kwargs** – Additional keyword arguments.

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

Returns
Dictionary containing images, features, true labels and masks. These are required in *validation_epoch_end* for feature concatenation.

PyTorch model for the PaDiM model implementation.

class anomalib.models.image.padim.torch_model.**PadimModel**(*layers*, *backbone='resnet18'*, *pre_trained=True*, *n_features=None*)

Bases: `Module`

Padim Module.

Parameters

- **layers** (*list[str]*) – Layers used for feature extraction
- **backbone** (*str, optional*) – Pre-trained model backbone. Defaults to "resnet18". Defaults to `resnet18`.
- **pre_trained** (*bool, optional*) – Boolean to check whether to use a pre_trained backbone. Defaults to `True`.

- **n_features** (*int, optional*) – Number of features to retain in the dimension reduction step. Default values from the paper are available for: resnet18 (100), wide_resnet50_2 (550). Defaults to None.

forward(*input_tensor*)

Forward-pass image-batch (N, C, H, W) into model to extract features.

Parameters

- **input_tensor** (Tensor) – Image-batch (N, C, H, W)
- **input_tensor** – torch.Tensor:

Return type
Tensor

Returns
Features from single/multiple layers.

Example

```
>>> x = torch.randn(32, 3, 224, 224)
>>> features = self.extract_features(input_tensor)
>>> features.keys()
dict_keys(['layer1', 'layer2', 'layer3'])
```

```
>>> [v.shape for v in features.values()]
[torch.Size([32, 64, 56, 56]),
torch.Size([32, 128, 28, 28]),
torch.Size([32, 256, 14, 14])]
```

generate_embedding(*features*)

Generate embedding from hierarchical feature map.

Parameters
features (*dict[str, torch.Tensor]*) – Hierarchical feature map from a CNN (ResNet18 or WideResnet)

Return type
Tensor

Returns
Embedding vector

PatchCore

Towards Total Recall in Industrial Anomaly Detection.

Paper https://arxiv.org/abs/2106.08265.

class anomalib.models.image.patchcore.lightning_model.**Patchcore**(*backbone='wide_resnet50_2'*, *layers=('layer2', 'layer3')*, *pre_trained=True*, *coreset_sampling_ratio=0.1*, *num_neighbors=9*)

Bases: `MemoryBankMixin`, `AnomalyModule`

PatchcoreLightning Module to train PatchCore algorithm.

Parameters

- **backbone** (*str*) – Backbone CNN network Defaults to `wide_resnet50_2`.
- **layers** (*list[str]*) – Layers to extract features from the backbone CNN Defaults to `["layer2", "layer3"]`.
- **pre_trained** (*bool, optional*) – Boolean to check whether to use a pre_trained backbone. Defaults to `True`.
- **coreset_sampling_ratio** (*float, optional*) – Coreset sampling ratio to subsample embedding. Defaults to `0.1`.
- **num_neighbors** (*int, optional*) – Number of nearest neighbors. Defaults to `9`.

configure_optimizers()

Configure optimizers.

Returns
Do not set optimizers by returning None.

Return type
None

configure_transforms(*image_size=None*)

Default transform for Padim.

Return type
`Transform`

fit()

Apply subsampling to the embedding collected from the training set.

Return type
`None`

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 Patchcore trainer arguments.

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

Generate feature embedding of the batch.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Batch containing image filename, image, label and mask
- **args** – Additional arguments.
- **kwargs** – Additional keyword arguments.

Returns
Embedding Vector

Return type
dict[str, np.ndarray]

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

Get batch of anomaly maps from input image batch.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Batch containing image filename, image, label and mask
- **args** – Additional arguments.
- **kwargs** – Additional keyword arguments.

Returns
Image filenames, test images, GT and predicted label/masks

Return type
dict[str, Any]

PyTorch model for the PatchCore model implementation.

class `anomalib.models.image.patchcore.torch_model.PatchcoreModel`(*layers*, *backbone='wide_resnet50_2'*, *pre_trained=True*, *num_neighbors=9*)

Bases: `DynamicBufferMixin`, `Module`

Patchcore Module.

Parameters

- **layers** (*list[str]*) – Layers used for feature extraction
- **backbone** (*str, optional*) – Pre-trained model backbone. Defaults to `resnet18`.
- **pre_trained** (*bool, optional*) – Boolean to check whether to use a pre_trained backbone. Defaults to `True`.
- **num_neighbors** (*int, optional*) – Number of nearest neighbors. Defaults to `9`.

`compute_anomaly_score`(*patch_scores*, *locations*, *embedding*)

Compute Image-Level Anomaly Score.

Parameters

- **patch_scores** (*torch.Tensor*) – Patch-level anomaly scores
- **locations** (`Tensor`) – Memory bank locations of the nearest neighbor for each patch location
- **embedding** (`Tensor`) – The feature embeddings that generated the patch scores

Returns
Image-level anomaly scores

Return type
Tensor

static euclidean_dist(*x*, *y*)

Calculate pair-wise distance between row vectors in x and those in y.

Replaces torch cdist with p=2, as cdist is not properly exported to onnx and openvino format. Resulting matrix is indexed by x vectors in rows and y vectors in columns.

Parameters

- **x** (Tensor) – input tensor 1
- **y** (Tensor) – input tensor 2

Return type
Tensor

Returns
Matrix of distances between row vectors in x and y.

forward(*input_tensor*)

Return Embedding during training, or a tuple of anomaly map and anomaly score during testing.

Steps performed: 1. Get features from a CNN. 2. Generate embedding based on the features. 3. Compute anomaly map in test mode.

Parameters
input_tensor (*torch.Tensor*) – Input tensor

Returns
Embedding for training, anomaly map and anomaly score for testing.

Return type
Tensor | dict[str, torch.Tensor]

generate_embedding(*features*)

Generate embedding from hierarchical feature map.

Parameters

- **features** (dict[str, Tensor]) – Hierarchical feature map from a CNN (ResNet18 or WideResnet)
- **features** – dict[str:Tensor]:

Return type
Tensor

Returns
Embedding vector

nearest_neighbors(*embedding*, *n_neighbors*)

Nearest Neighbours using brute force method and euclidean norm.

Parameters

- **embedding** (*torch.Tensor*) – Features to compare the distance with the memory bank.
- **n_neighbors** (*int*) – Number of neighbors to look at

Returns
Patch scores. Tensor: Locations of the nearest neighbor(s).

Return type
Tensor

`static reshape_embedding(embedding)`

Reshape Embedding.

Reshapes Embedding to the following format:

- [Batch, Embedding, Patch, Patch] to [Batch*Patch*Patch, Embedding]

Parameters
embedding (*torch.Tensor*) – Embedding tensor extracted from CNN features.

Returns
Reshaped embedding tensor.

Return type
Tensor

`subsample_embedding(embedding, sampling_ratio)`

Subsample embedding based on coreset sampling and store to memory.

Parameters

- **embedding** (*np.ndarray*) – Embedding tensor from the CNN
- **sampling_ratio** (*float*) – Coreset sampling ratio

Return type
None

Reverse Distillation

Anomaly Detection via Reverse Distillation from One-Class Embedding.

https://arxiv.org/abs/2201.10703v2

`class anomalib.models.image.reverse_distillation.lightning_model.ReverseDistillation(backbone='wide_resnet` *lay-ers=('layer1', 'layer2', 'layer3'), anomaly_map_mode=A pre_trained=True*)

Bases: `AnomalyModule`

PL Lightning Module for Reverse Distillation Algorithm.

Parameters

- **backbone** (*str*) – Backbone of CNN network Defaults to `wide_resnet50_2`.
- **layers** (*list[str]*) – Layers to extract features from the backbone CNN Defaults to `["layer1", "layer2", "layer3"]`.
- **anomaly_map_mode** (*AnomalyMapGenerationMode, optional*) – Mode to generate anomaly map. Defaults to `AnomalyMapGenerationMode.ADD`.
- **pre_trained** (*bool, optional*) – Boolean to check whether to use a pre_trained backbone. Defaults to `True`.

`configure_optimizers()`

Configure optimizers for decoder and bottleneck.

Returns
Adam optimizer for each decoder

Return type
Optimizer

`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 Reverse Distillation trainer arguments.

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

Perform a training step of Reverse Distillation Model.

Features are extracted from three layers of the Encoder model. These are passed to the bottleneck layer that are passed to the decoder network. The loss is then calculated based on the cosine similarity between the encoder and decoder features.

Parameters

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

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

Returns
Feature Map

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

Perform a validation step of Reverse Distillation Model.

Similar to the training step, encoder/decoder features are extracted from the CNN for each batch, and anomaly map is computed.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Input batch
- **args** – Additional arguments.
- **kwargs** – Additional keyword arguments.

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

Returns
Dictionary containing images, anomaly maps, true labels and masks. These are required in *validation_epoch_end* for feature concatenation.

PyTorch model for Reverse Distillation.

class anomalib.models.image.reverse_distillation.torch_model.**ReverseDistillationModel**(*backbone*, *input_size*, *layers*, *anomaly_map_mode*, *pre_trained=True*)

Bases: `Module`

Reverse Distillation Model.

To reproduce results in the paper, use torchvision model for the encoder:
self.encoder = torchvision.models.wide_resnet50_2(pretrained=True)

Parameters

- **backbone** (*str*) – Name of the backbone used for encoder and decoder.
- **input_size** (*tuple[int, int]*) – Size of input image.
- **layers** (*list[str]*) – Name of layers from which the features are extracted.
- **anomaly_map_mode** (*str*) – Mode used to generate anomaly map. Options are between `multiply` and `add`.
- **pre_trained** (*bool, optional*) – Boolean to check whether to use a pre_trained backbone. Defaults to `True`.

forward(*images*)

Forward-pass images to the network.

During the training mode the model extracts features from encoder and decoder networks. During evaluation mode, it returns the predicted anomaly map.

Parameters
images (*torch.Tensor*) – Batch of images

Returns

Encoder and decoder features
in training mode, else anomaly maps.

Return type
torch.Tensor | list[torch.Tensor] | tuple[list[torch.Tensor]]

Loss function for Reverse Distillation.

class anomalib.models.image.reverse_distillation.loss.**ReverseDistillationLoss**(**args*, ***kwargs*)

Bases: `Module`

Loss function for Reverse Distillation.

forward(*encoder_features*, *decoder_features*)

Compute cosine similarity loss based on features from encoder and decoder.

Based on the official code: https://github.com/hq-deng/RD4AD/blob/6554076872c65f8784f6ece8cfb39ce77e1aee12/main.py#L33C25-L33C25 Calculates loss from flattened arrays of features, see https://github.com/hq-deng/RD4AD/issues/22

Parameters

- **encoder_features** (*list[torch.Tensor]*) – List of features extracted from encoder
- **decoder_features** (*list[torch.Tensor]*) – List of features extracted from decoder

Returns
Cosine similarity loss

Return type
Tensor

Compute Anomaly map.

class anomalib.models.image.reverse_distillation.anomaly_map.**AnomalyMapGenerationMode**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `str`, `Enum`

Type of mode when generating anomaly imape.

class anomalib.models.image.reverse_distillation.anomaly_map.**AnomalyMapGenerator**(*image_size*, *sigma=4*, *mode=AnomalyMapGenerati*

Bases: `Module`

Generate Anomaly Heatmap.

Parameters

- **image_size** (*ListConfig, tuple*) – Size of original image used for upscaling the anomaly map.
- **sigma** (*int*) – Standard deviation of the gaussian kernel used to smooth anomaly map. Defaults to 4.
- **mode** (*AnomalyMapGenerationMode, optional*) – Operation used to generate anomaly map. Options are `AnomalyMapGenerationMode.ADD` and `AnomalyMapGenerationMode.MULTIPLY`. Defaults to `AnomalyMapGenerationMode.MULTIPLY`.

Raises
ValueError – In case modes other than multiply and add are passed.

forward(*student_features*, *teacher_features*)

Compute anomaly map given encoder and decoder features.

Parameters

- **student_features** (*list[torch.Tensor]*) – List of encoder features
- **teacher_features** (*list[torch.Tensor]*) – List of decoder features

Returns
Anomaly maps of length batch.

Return type
Tensor

R-KDE

Region Based Anomaly Detection With Real-Time Training and Analysis.

https://ieeexplore.ieee.org/abstract/document/8999287

class anomalib.models.image.rkde.lightning_model.**Rkde**(*roi_stage=RoiStage.RCNN*, *roi_score_threshold=0.001*, *min_box_size=25*, *iou_threshold=0.3*, *max_detections_per_image=100*, *n_pca_components=16*, *feature_scaling_method=FeatureScalingMethod.SCALE*, *max_training_points=40000*)

Bases: MemoryBankMixin, AnomalyModule

Region Based Anomaly Detection With Real-Time Training and Analysis.

Parameters

- **roi_stage** (RoiStage, *optional*) – Processing stage from which rois are extracted. Defaults to RoiStage.RCNN.
- **roi_score_threshold** (*float*, *optional*) – Mimumum confidence score for the region proposals. Defaults to 0.001.
- **min_size** (*int*, *optional*) – Minimum size in pixels for the region proposals. Defaults to 25.
- **iou_threshold** (*float*, *optional*) – Intersection-Over-Union threshold used during NMS. Defaults to 0.3.
- **max_detections_per_image** (*int*, *optional*) – Maximum number of region proposals per image. Defaults to 100.
- **n_pca_components** (*int*, *optional*) – Number of PCA components. Defaults to 16.
- **feature_scaling_method** (FeatureScalingMethod, *optional*) – Scaling method applied to features before passing to KDE. Options are *norm* (normalize to unit vector length) and *scale* (scale to max length observed in training). Defaults to FeatureScalingMethod.SCALE.
- **max_training_points** (*int*, *optional*) – Maximum number of training points to fit the KDE model. Defaults to 40000.

static configure_optimizers()

RKDE doesn't require optimization, therefore returns no optimizers.

Return type
None

fit()

Fit a KDE Model to the embedding collected from the training set.

Return type
None

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 R-KDE trainer arguments.

Returns
Arguments for the trainer.

Return type
dict[str, Any]

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

Perform a training Step of RKDE. For each batch, features are extracted from the CNN.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Batch containing image filename, image, label and mask
- **args** – Additional arguments.
- **kwargs** – Additional keyword arguments.

Return type
`None`

Returns
Deep CNN features.

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

Perform a validation Step of RKde.

Similar to the training step, features are extracted from the CNN for each batch.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Batch containing image filename, image, label and mask
- **args** – Additional arguments.
- **kwargs** – Additional keyword arguments.

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

Returns
Dictionary containing probability, prediction and ground truth values.

Torch model for region-based anomaly detection.

class anomalib.models.image.rkde.torch_model.**RkdeModel**(*roi_stage=RoiStage.RCNN*, *roi_score_threshold=0.001*, *min_box_size=25*, *iou_threshold=0.3*, *max_detections_per_image=100*, *n_pca_components=16*, *feature_scaling_method=FeatureScalingMethod.SCALE*, *max_training_points=40000*)

Bases: `Module`

Torch Model for the Region-based Anomaly Detection Model.

Parameters

- **roi_stage** (RoiStage, *optional*) – Processing stage from which rois are extracted. Defaults to `RoiStage.RCNN`.
- **roi_score_threshold** (*float, optional*) – Mimumum confidence score for the region proposals. Defaults to `0.001`.
- **min_size** (*int, optional*) – Minimum size in pixels for the region proposals. Defaults to `25`.
- **iou_threshold** (*float, optional*) – Intersection-Over-Union threshold used during NMS. Defaults to `0.3`.
- **max_detections_per_image** (*int, optional*) – Maximum number of region proposals per image. Defaults to `100`.
- **n_pca_components** (*int, optional*) – Number of PCA components. Defaults to `16`.
- **feature_scaling_method** (FeatureScalingMethod, *optional*) – Scaling method applied to features before passing to KDE. Options are *norm* (normalize to unit vector length) and *scale* (scale to max length observed in training). Defaults to `FeatureScalingMethod.SCALE`.
- **max_training_points** (*int, optional*) – Maximum number of training points to fit the KDE model. Defaults to `40000`.

`fit`(*embeddings*)

Fit the model using a set of collected embeddings.

Parameters
embeddings (*torch.Tensor*) – Input embeddings to fit the model.

Return type
`bool`

Returns
Boolean confirming whether the training is successful.

`forward`(*batch*)

Prediction by normality model.

Parameters
batch (*torch.Tensor*) – Input images.

Returns

The extracted features (when in training mode),
or the predicted rois and corresponding anomaly scores.

Return type
Tensor | tuple[torch.Tensor, torch.Tensor]

Region-based Anomaly Detection with Real Time Training and Analysis.

Feature Extractor.

class anomalib.models.image.rkde.feature_extractor.**FeatureExtractor**

Bases: Module

Feature Extractor module for Region-based anomaly detection.

forward(*batch*, *rois*)

Perform a forward pass of the feature extractor.

Parameters

- **batch** (*torch.Tensor*) – Batch of input images of shape [B, C, H, W].
- **rois** (*torch.Tensor*) – torch.Tensor of shape [N, 5] describing the regions-of-interest in the batch.

Returns

torch.Tensor containing a 4096-dimensional feature vector for every RoI location.

Return type

Tensor

Region-based Anomaly Detection with Real Time Training and Analysis.

Region Extractor.

class anomalib.models.image.rkde.region_extractor.**RegionExtractor**(*stage=RoiStage.RCNN*, *score_threshold=0.001*, *min_size=25*, *iou_threshold=0.3*, *max_detections_per_image=100*)

Bases: Module

Extracts regions from the image.

Parameters

- **stage** (RoiStage, *optional*) – Processing stage from which rois are extracted. Defaults to RoiStage.RCNN.
- **score_threshold** (*float*, *optional*) – Mimumum confidence score for the region proposals. Defaults to 0.001.
- **min_size** (*int*, *optional*) – Minimum size in pixels for the region proposals. Defaults to 25.
- **iou_threshold** (*float*, *optional*) – Intersection-Over-Union threshold used during NMS. Defaults to 0.3.
- **max_detections_per_image** (*int*, *optional*) – Maximum number of region proposals per image. Defaults to 100.

forward(*batch*)

Forward pass of the model.

Parameters

batch (*torch.Tensor*) – Batch of input images of shape [B, C, H, W].

Raises

ValueError – When stage is not one of rcnn or rpn.

Returns

Predicted regions, tensor of shape [N, 5] where N is the number of predicted regions in the batch,
and where each row describes the index of the image in the batch and the 4 bounding box coordinates.

Return type
Tensor

post_process_box_predictions(*pred_boxes*, *pred_scores*)

Post-processes the box predictions.

The post-processing consists of removing small boxes, applying nms, and keeping only the k boxes with the highest confidence score.

Parameters

- **pred_boxes** (*torch.Tensor*) – Box predictions of shape (N, 4).
- **pred_scores** (*torch.Tensor*) – torch.Tensor of shape () with a confidence score for each box prediction.

Returns
Post-processed box predictions of shape (N, 4).

Return type
list[torch.Tensor]

class anomalib.models.image.rkde.region_extractor.**RoiStage**(*value*, *names=None*, *, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `str`, `Enum`

Processing stage from which rois are extracted.

STFPM

STFPM: Student-Teacher Feature Pyramid Matching for Unsupervised Anomaly Detection.

https://arxiv.org/abs/2103.04257

class anomalib.models.image.stfpm.lightning_model.**Stfpm**(*backbone='resnet18'*, *layers=('layer1', 'layer2', 'layer3')*)

Bases: `AnomalyModule`

PL Lightning Module for the STFPM algorithm.

Parameters

- **backbone** (*str*) – Backbone CNN network Defaults to `resnet18`.
- **layers** (*list[str]*) – Layers to extract features from the backbone CNN Defaults to `[ "layer1", "layer2", "layer3"]`.

configure_optimizers()

Configure optimizers.

Returns
SGD optimizer

Return type
Optimizer

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]

Required trainer arguments.

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

Perform a training step of STFPM.

For each batch, teacher and student and teacher features are extracted from the CNN.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Input batch.
- **args** – Additional arguments.
- **kwargs** – Additional keyword arguments.

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

Returns
Loss value

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

Perform a validation Step of STFPM.

Similar to the training step, student/teacher features are extracted from the CNN for each batch, and anomaly map is computed.

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Input batch
- **args** – Additional arguments
- **kwargs** – Additional keyword arguments

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

Returns
Dictionary containing images, anomaly maps, true labels and masks. These are required in *validation_epoch_end* for feature concatenation.

PyTorch model for the STFPM model implementation.

class anomalib.models.image.stfpm.torch_model.**STFPMModel**(*layers*, *backbone='resnet18'*)

Bases: `Module`

STFPM: Student-Teacher Feature Pyramid Matching for Unsupervised Anomaly Detection.

Parameters

- **layers** (*list[str]*) – Layers used for feature extraction.
- **backbone** (*str, optional*) – Pre-trained model backbone. Defaults to `resnet18`.

forward(*images*)

Forward-pass images into the network.

During the training mode the model extracts the features from the teacher and student networks. During the evaluation mode, it returns the predicted anomaly map.

Parameters

images (*torch.Tensor*) – Batch of images.

Return type

Tensor | dict[str, Tensor] | tuple[dict[str, Tensor]]

Returns

Teacher and student features when in training mode, otherwise the predicted anomaly maps.

Loss function for the STFPM Model Implementation.

class anomalib.models.image.stfpm.loss.**STFPMLoss**

Bases: Module

Feature Pyramid Loss This class implmenents the feature pyramid loss function proposed in STFPM paper.

Example

```
>>> from anomalib.models.components.feature_extractors import TimmFeatureExtractor
>>> from anomalib.models.stfpm.loss import STFPMLoss
>>> from torchvision.models import resnet18
```

```
>>> layers = ['layer1', 'layer2', 'layer3']
>>> teacher_model = TimmFeatureExtractor(model=resnet18(pretrained=True),␣
↪layers=layers)
>>> student_model = TimmFeatureExtractor(model=resnet18(pretrained=False),␣
↪layers=layers)
>>> loss = Loss()
```

```
>>> inp = torch.rand((4, 3, 256, 256))
>>> teacher_features = teacher_model(inp)
>>> student_features = student_model(inp)
>>> loss(student_features, teacher_features)
    tensor(51.2015, grad_fn=<SumBackward0>)
```

compute_layer_loss(*teacher_feats*, *student_feats*)

Compute layer loss based on Equation (1) in Section 3.2 of the paper.

Parameters

- **teacher_feats** (*torch.Tensor*) – Teacher features
- **student_feats** (*torch.Tensor*) – Student features

Return type

Tensor

Returns

L2 distance between teacher and student features.

forward(*teacher_features*, *student_features*)

Compute the overall loss via the weighted average of the layer losses computed by the cosine similarity.

Parameters

- **teacher_features** (*dict[str, torch.Tensor]*) – Teacher features
- **student_features** (*dict[str, torch.Tensor]*) – Student features

Return type
Tensor

Returns
Total loss, which is the weighted average of the layer losses.

Anomaly Map Generator for the STFPM model implementation.

class anomalib.models.image.stfpm.anomaly_map.**AnomalyMapGenerator**

Bases: Module

Generate Anomaly Heatmap.

compute_anomaly_map(*teacher_features*, *student_features*, *image_size*)

Compute the overall anomaly map via element-wise production the interpolated anomaly maps.

Parameters

- **teacher_features** (*dict[str, torch.Tensor]*) – Teacher features
- **student_features** (*dict[str, torch.Tensor]*) – Student features
- **image_size** (*tuple[int, int]*) – Image size to which the anomaly map should be resized.

Return type
Tensor

Returns
Final anomaly map

compute_layer_map(*teacher_features*, *student_features*, *image_size*)

Compute the layer map based on cosine similarity.

Parameters

- **teacher_features** (*torch.Tensor*) – Teacher features
- **student_features** (*torch.Tensor*) – Student features
- **image_size** (*tuple[int, int]*) – Image size to which the anomaly map should be resized.

Return type
Tensor

Returns
Anomaly score based on cosine similarity.

forward(***kwargs*)

Return anomaly map.

Expects *teach_features* and *student_features* keywords to be passed explicitly.

Parameters
kwargs (*dict[str, torch.Tensor]*) – Keyword arguments

Example

```
>>> anomaly_map_generator = AnomalyMapGenerator(image_size=tuple(hparams.model.
→input_size))
>>> output = self.anomaly_map_generator(
        teacher_features=teacher_features,
        student_features=student_features
    )
```

Raises
`ValueError` – *teach_features* and *student_features* keys are not found

Returns
anomaly map

Return type
torch.Tensor

U-Flow

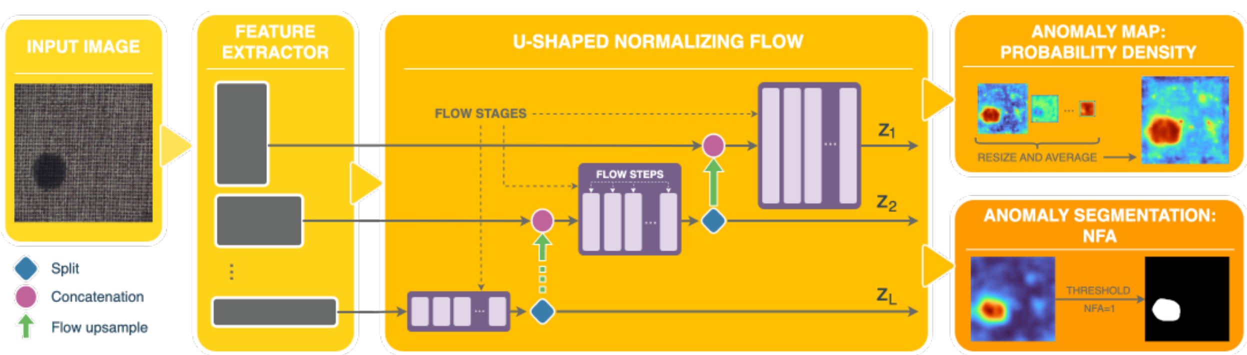
This is the implementation of the U-Flow paper.

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

WinCLIP

WinCLIP: Zero-/Few-Shot Anomaly Classification and Segmentation.

Paper https://arxiv.org/abs/2303.14814

class anomalib.models.image.winclip.lightning_model.**WinClip**(*class_name=None*, *k_shot=0*, *scales=(2, 3)*, *few_shot_source=None*)

Bases: AnomalyModule

WinCLIP Lightning model.

Parameters

- **class_name** (*str, optional*) – The name of the object class used in the prompt ensemble. Defaults to None.
- **k_shot** (*int*) – The number of reference images for few-shot inference. Defaults to 0.
- **scales** (*tuple[int], optional*) – The scales of the sliding windows used for multiscale anomaly detection. Defaults to (2, 3).
- **few_shot_source** (*str | Path, optional*) – Path to a folder of reference images used for few-shot inference. Defaults to None.

collect_reference_images(*dataloader*)

Collect reference images for few-shot inference.

The reference images are collected by iterating the training dataset until the required number of images are collected.

Returns
A tensor containing the reference images.

Return type
ref_images (Tensor)

static configure_optimizers()

WinCLIP doesn't require optimization, therefore returns no optimizers.

Return type
None

configure_transforms(*image_size=None*)

Configure the default transforms used by the model.

Return type
Transform

property learning_type: LearningType

The learning type of the model.

WinCLIP is a zero-/few-shot model, depending on the user configuration. Therefore, the learning type is set to LearningType.FEW_SHOT when k_shot is greater than zero and LearningType.ZERO_SHOT otherwise.

load_state_dict(*state_dict*, *strict=True*)

Load the state dict of the model.

Before loading the state dict, we restore the parameters of the frozen backbone to ensure that the model is loaded correctly. We also restore the auxiliary objects like threshold classes and normalization metrics.

Return type
`Any`

`state_dict()`

Return the state dict of the model.

Before returning the state dict, we remove the parameters of the frozen backbone to reduce the size of the checkpoint.

Return type
`OrderedDict[str, Any]`

property trainer_arguments: dict[str, int | float]

Set model-specific trainer arguments.

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

Validation Step of WinCLIP.

Return type
`dict`

PyTorch model for the WinCLIP implementation.

class anomalib.models.image.winclip.torch_model.**WinClipModel**(*class_name=None*, *reference_images=None*, *scales=(2, 3)*, *apply_transform=False*)

Bases: `DynamicBufferMixin`, `BufferListMixin`, `Module`

PyTorch module that implements the WinClip model for image anomaly detection.

Parameters

- **class_name** (*str, optional*) – The name of the object class used in the prompt ensemble. Defaults to `None`.
- **reference_images** (*torch.Tensor, optional*) – Tensor of shape `(K, C, H, W)` containing the reference images. Defaults to `None`.
- **scales** (*tuple[int], optional*) – The scales of the sliding windows used for multi-scale anomaly detection. Defaults to `(2, 3)`.
- **apply_transform** (*bool, optional*) – Whether to apply the default CLIP transform to the input images. Defaults to `False`.

`clip`

The CLIP model used for image and text encoding.

Type
CLIP

`grid_size`

The size of the feature map grid.

Type
tuple[int]

`k_shot`

The number of reference images used for few-shot anomaly detection.

Type
int

scales

The scales of the sliding windows used for multi-scale anomaly detection.

Type
tuple[int]

masks

The masks representing the sliding window locations.

Type
list[torch.Tensor] | None

_text_embeddings

The text embeddings for the compositional prompt ensemble.

Type
torch.Tensor | None

_visual_embeddings

The multi-scale embeddings for the reference images.

Type
list[torch.Tensor] | None

_patch_embeddings

The patch embeddings for the reference images.

Type
torch.Tensor | None

encode_image(*batch*)

Encode the batch of images to obtain image embeddings, window embeddings, and patch embeddings.

The image embeddings and patch embeddings are obtained by passing the batch of images through the model. The window embeddings are obtained by masking the feature map and passing it through the transformer. A forward hook is used to retrieve the intermediate feature map and share computation between the image and window embeddings.

Parameters
batch (*torch.Tensor*) – Batch of input images of shape `(N, C, H, W)`.

Returns
A tuple containing the image embeddings, window embeddings, and patch embeddings respectively.

Return type
Tuple[torch.Tensor, List[torch.Tensor], torch.Tensor]

Examples

```
>>> model = WinClipModel()
>>> model.prepare_masks()
>>> batch = torch.rand((1, 3, 240, 240))
>>> image_embeddings, window_embeddings, patch_embeddings = model.encode_
↪image(batch)
>>> image_embeddings.shape
torch.Size([1, 640])
>>> [embedding.shape for embedding in window_embeddings]
```

(continues on next page)

(continued from previous page)

```
[torch.Size([1, 196, 640]), torch.Size([1, 169, 640])]
>>> patch_embeddings.shape
torch.Size([1, 225, 896])
```

forward(*batch*)

Forward-pass through the model to obtain image and pixel scores.

Parameters

batch (*torch.Tensor*) – Batch of input images of shape (batch_size, C, H, W).

Returns

Tuple containing the image scores and pixel scores.

Return type

Tuple[torch.Tensor, torch.Tensor]

property patch_embeddings: Tensor

The patch embeddings used by the model.

setup(*class_name=None*, *reference_images=None*)

Setup WinCLIP.

WinCLIP's setup stage consists of collecting the text and visual embeddings used during inference. The following steps are performed, depending on the arguments passed to the model: - Collect text embeddings for zero-shot inference. - Collect reference images for few-shot inference. The k_shot attribute is updated based on the number of reference images.

The setup method is called internally by the constructor. However, it can also be called manually to update the text and visual embeddings after the model has been initialized.

Parameters

- **class_name** (*str*) – The name of the object class used in the prompt ensemble.
- **reference_images** (*torch.Tensor*) – Tensor of shape (batch_size, C, H, W) containing the reference images.

Return type

None

Examples

```
>>> model = WinClipModel()
>>> model.setup("transistor")
>>> model.text_embeddings.shape
torch.Size([2, 640])
```

```
>>> ref_images = torch.rand(2, 3, 240, 240)
>>> model = WinClipModel()
>>> model.setup("transistor", ref_images)
>>> model.k_shot
2
>>> model.visual_embeddings[0].shape
torch.Size([2, 196, 640])
```

```
>>> model = WinClipModel("transistor")
>>> model.k_shot
0
>>> model.setup(reference_images=ref_images)
>>> model.k_shot
2
```

```
>>> model = WinClipModel(class_name="transistor", reference_images=ref_images)
>>> model.text_embeddings.shape
torch.Size([2, 640])
>>> model.visual_embeddings[0].shape
torch.Size([2, 196, 640])
```

property text_embeddings: Tensor

The text embeddings used by the model.

property transform: Compose

The transform used by the model.

To obtain the transforms, we retrieve the transforms from the clip backbone. Since the original transforms are intended for PIL images, we prepend a ToPILImage transform to the list of transforms.

property visual_embeddings: list[Tensor]

The visual embeddings used by the model.

Video Models

AI VAD

AI VAD

Attribute-based Representations for Accurate and Interpretable Video Anomaly Detection.

Paper https://arxiv.org/pdf/2212.00789.pdf

class anomalib.models.video.ai_vad.lightning_model.**AiVad**(*box_score_thresh=0.7*, *persons_only=False*, *min_bbox_area=100*, *max_bbox_overlap=0.65*, *enable_foreground_detections=True*, *foreground_kernel_size=3*, *foreground_binary_threshold=18*, *n_velocity_bins=1*, *use_velocity_features=True*, *use_pose_features=True*, *use_deep_features=True*, *n_components_velocity=2*, *n_neighbors_pose=1*, *n_neighbors_deep=1*)

Bases: `MemoryBankMixin`, `AnomalyModule`

AI-VAD: Attribute-based Representations for Accurate and Interpretable Video Anomaly Detection.

Parameters

- **box_score_thresh** (*float*) – Confidence threshold for bounding box predictions. Defaults to 0.7.
- **persons_only** (*bool*) – When enabled, only regions labeled as person are included. Defaults to False.
- **min_bbox_area** (*int*) – Minimum bounding box area. Regions with a surface area lower than this value are excluded. Defaults to 100.
- **max_bbox_overlap** (*float*) – Maximum allowed overlap between bounding boxes. Defaults to 0.65.
- **enable_foreground_detections** (*bool*) – Add additional foreground detections based on pixel difference between consecutive frames. Defaults to True.
- **foreground_kernel_size** (*int*) – Gaussian kernel size used in foreground detection. Defaults to 3.
- **foreground_binary_threshold** (*int*) – Value between 0 and 255 which acts as binary threshold in foreground detection. Defaults to 18.
- **n_velocity_bins** (*int*) – Number of discrete bins used for velocity histogram features. Defaults to 1.
- **use_velocity_features** (*bool*) – Flag indicating if velocity features should be used. Defaults to True.
- **use_pose_features** (*bool*) – Flag indicating if pose features should be used. Defaults to True.
- **use_deep_features** (*bool*) – Flag indicating if deep features should be used. Defaults to True.
- **n_components_velocity** (*int*) – Number of components used by GMM density estimation for velocity features. Defaults to 2.
- **n_neighbors_pose** (*int*) – Number of neighbors used in KNN density estimation for pose features. Defaults to 1.
- **n_neighbors_deep** (*int*) – Number of neighbors used in KNN density estimation for deep features. Defaults to 1.

static **configure_optimizers**()

AI-VAD training does not involve fine-tuning of NN weights, no optimizers needed.

Return type
None

configure_transforms(*image_size=None*)

AI-VAD does not need a transform, as the region- and feature-extractors apply their own transforms.

Return type
Transform | None

fit()

Fit the density estimators to the extracted features from the training set.

Return type
None

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]

AI-VAD specific trainer arguments.

training_step(*batch*)

Training Step of AI-VAD.

Extract features from the batch of clips and update the density estimators.

Parameters
batch (*dict[str, str | torch.Tensor]*) – Batch containing image filename, image, label and mask

Return type
None

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

Perform the validation step of AI-VAD.

Extract boxes and box scores..

Parameters

- **batch** (*dict[str, str | torch.Tensor]*) – Input batch
- ***args** – Arguments.
- ****kwargs** – Keyword arguments.

Return type
Union[Tensor, Mapping[str, Any], None]

Returns
Batch dictionary with added boxes and box scores.

PyTorch model for AI-VAD model implementation.

Paper https://arxiv.org/pdf/2212.00789.pdf

class anomalib.models.video.ai_vad.torch_model.**AiVadModel**(*box_score_thresh=0.8*, *persons_only=False*, *min_bbox_area=100*, *max_bbox_overlap=0.65*, *enable_foreground_detections=True*, *foreground_kernel_size=3*, *foreground_binary_threshold=18*, *n_velocity_bins=8*, *use_velocity_features=True*, *use_pose_features=True*, *use_deep_features=True*, *n_components_velocity=5*, *n_neighbors_pose=1*, *n_neighbors_deep=1*)

Bases: Module

AI-VAD model.

Parameters

- **box_score_thresh** (*float*) – Confidence threshold for region extraction stage. Defaults to `0.8`.
- **persons_only** (*bool*) – When enabled, only regions labeled as person are included. Defaults to `False`.
- **min_bbox_area** (*int*) – Minimum bounding box area. Regions with a surface area lower than this value are excluded. Defaults to `100`.
- **max_bbox_overlap** (*float*) – Maximum allowed overlap between bounding boxes. Defaults to `0.65`.
- **enable_foreground_detections** (*bool*) – Add additional foreground detections based on pixel difference between consecutive frames. Defaults to `True`.
- **foreground_kernel_size** (*int*) – Gaussian kernel size used in foreground detection. Defaults to `3`.
- **foreground_binary_threshold** (*int*) – Value between 0 and 255 which acts as binary threshold in foreground detection. Defaults to `18`.
- **n_velocity_bins** (*int*) – Number of discrete bins used for velocity histogram features. Defaults to `8`.
- **use_velocity_features** (*bool*) – Flag indicating if velocity features should be used. Defaults to `True`.
- **use_pose_features** (*bool*) – Flag indicating if pose features should be used. Defaults to `True`.
- **use_deep_features** (*bool*) – Flag indicating if deep features should be used. Defaults to `True`.
- **n_components_velocity** (*int*) – Number of components used by GMM density estimation for velocity features. Defaults to `5`.
- **n_neighbors_pose** (*int*) – Number of neighbors used in KNN density estimation for pose features. Defaults to `1`.
- **n_neighbors_deep** (*int*) – Number of neighbors used in KNN density estimation for deep features. Defaults to `1`.

forward(*batch*)

Forward pass through AI-VAD model.

Parameters

batch (*torch.Tensor*) – Input image of shape (N, L, C, H, W)

Returns

List of bbox locations for each image. list[torch.Tensor]: List of per-bbox anomaly scores for each image. list[torch.Tensor]: List of per-image anomaly scores.

Return type

list[torch.Tensor]

Feature extraction module for AI-VAD model implementation.

class `anomalib.models.video.ai_vad.features.`**DeepExtractor**

Bases: `Module`

Deep feature extractor.

Extracts the deep (appearance) features from the input regions.

`forward`(*batch*, *boxes*, *batch_size*)

Extract deep features using CLIP encoder.

Parameters

- **batch** (*torch.Tensor*) – Batch of RGB input images of shape (N, 3, H, W)
- **boxes** (*torch.Tensor*) – Bounding box coordinates of shaspe (M, 5). First column indicates batch index of the bbox.
- **batch_size** (*int*) – Number of images in the batch.

Returns
Deep feature tensor of shape (M, 512)

Return type
Tensor

class `anomalib.models.video.ai_vad.features.FeatureExtractor`(*n_velocity_bins=8*, *use_velocity_features=True*, *use_pose_features=True*, *use_deep_features=True*)

Bases: `Module`

Feature extractor for AI-VAD.

Parameters

- **n_velocity_bins** (*int*) – Number of discrete bins used for velocity histogram features. Defaults to `8`.
- **use_velocity_features** (*bool*) – Flag indicating if velocity features should be used. Defaults to `True`.
- **use_pose_features** (*bool*) – Flag indicating if pose features should be used. Defaults to `True`.
- **use_deep_features** (*bool*) – Flag indicating if deep features should be used. Defaults to `True`.

`forward`(*rgb_batch*, *flow_batch*, *regions*)

Forward pass through the feature extractor.

Extract any combination of velocity, pose and deep features depending on configuration.

Parameters

- **rgb_batch** (*torch.Tensor*) – Batch of RGB images of shape (N, 3, H, W)
- **flow_batch** (*torch.Tensor*) – Batch of optical flow images of shape (N, 2, H, W)
- **regions** (*list[dict]*) – Region information per image in batch.

Returns
Feature dictionary per image in batch.

Return type
list[dict]

class `anomalib.models.video.ai_vad.features.FeatureType`(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `str`, `Enum`

Names of the different feature streams used in AI-VAD.

`class anomalib.models.video.ai_vad.features.`**`PoseExtractor`**`(*args, **kwargs)`

Bases: `Module`

Pose feature extractor.

Extracts pose features based on estimated body landmark keypoints.

`forward``(batch, boxes)`

Extract pose features using a human keypoint estimation model.

Parameters

- **batch** (*torch.Tensor*) – Batch of RGB input images of shape (N, 3, H, W)
- **boxes** (*torch.Tensor*) – Bounding box coordinates of shaspe (M, 5). First column indicates batch index of the bbox.

Returns

list of pose feature tensors for each image.

Return type

list[torch.Tensor]

`class anomalib.models.video.ai_vad.features.`**`VelocityExtractor`**`(n_bins=8)`

Bases: `Module`

Velocity feature extractor.

Extracts histograms of optical flow magnitude and direction.

Parameters

n_bins (*int*) – Number of direction bins used for the feature histograms.

`forward``(flows, boxes)`

Extract velocioty features by filling a histogram.

Parameters

- **flows** (*torch.Tensor*) – Batch of optical flow images of shape (N, 2, H, W)
- **boxes** (*torch.Tensor*) – Bounding box coordinates of shaspe (M, 5). First column indicates batch index of the bbox.

Returns

Velocity feature tensor of shape (M, n_bins)

Return type

Tensor

Regions extraction module of AI-VAD model implementation.

`class anomalib.models.video.ai_vad.regions.`**`RegionExtractor`**`(box_score_thresh=0.8, persons_only=False, min_bbox_area=100, max_bbox_overlap=0.65, enable_foreground_detections=True, foreground_kernel_size=3, foreground_binary_threshold=18)`

Bases: `Module`

Region extractor for AI-VAD.

Parameters

- **box_score_thresh** (*float*) – Confidence threshold for bounding box predictions. Defaults to `0.8`.
- **persons_only** (*bool*) – When enabled, only regions labeled as person are included. Defaults to `False`.
- **min_bbox_area** (*int*) – Minimum bounding box area. Regions with a surface area lower than this value are excluded. Defaults to `100`.
- **max_bbox_overlap** (*float*) – Maximum allowed overlap between bounding boxes. Defaults to `0.65`.
- **enable_foreground_detections** (*bool*) – Add additional foreground detections based on pixel difference between consecutive frames. Defaults to `True`.
- **foreground_kernel_size** (*int*) – Gaussian kernel size used in foreground detection. Defaults to `3`.
- **foreground_binary_threshold** (*int*) – Value between 0 and 255 which acts as binary threshold in foreground detection. Defaults to `18`.

`add_foreground_boxes`(*regions*, *first_frame*, *last_frame*, *kernel_size*, *binary_threshold*)

Add any foreground regions that were not detected by the region extractor.

This method adds regions that likely belong to the foreground of the video scene, but were not detected by the region extractor module. The foreground pixels are determined by taking the pixel difference between two consecutive video frames and applying a binary threshold. The final detections consist of all connected components in the foreground that do not fall in one of the bounding boxes predicted by the region extractor.

Parameters

- **regions** (*list[dict[str, torch.Tensor]]*) – Region detections for a batch of images, generated by the region extraction module.
- **first_frame** (*torch.Tensor*) – video frame at time t-1
- **last_frame** (*torch.Tensor*) – Video frame time t
- **kernel_size** (*int*) – Kernel size for Gaussian smoothing applied to input frames
- **binary_threshold** (*int*) – Binary threshold used in foreground detection, should be in range [0, 255]

Returns

region detections with foreground regions appended

Return type

list[dict[str, torch.Tensor]]

`forward`(*first_frame*, *last_frame*)

Perform forward-pass through region extractor.

Parameters

- **first_frame** (*torch.Tensor*) – Batch of input images of shape (N, C, H, W) forming the first frames in the clip.
- **last_frame** (*torch.Tensor*) – Batch of input images of shape (N, C, H, W) forming the last frame in the clip.

Returns

List of Mask RCNN predictions for each image in the batch.

Return type
list[dict]

`post_process_bbox_detections`(*regions*)

Post-process the region detections.

The region detections are filtered based on class label, bbox area and overlap with other regions.

Parameters
regions (*list[dict[str, torch.Tensor]]*) – Region detections for a batch of images, generated by the region extraction module.

Returns
Filtered regions

Return type
list[dict[str, torch.Tensor]]

`static subsample_regions`(*regions*, *indices*)

Subsample the items in a region dictionary based on a Tensor of indices.

Parameters

- **regions** (*dict[str, torch.Tensor]*) – Region detections for a single image in the batch.
- **indices** (*torch.Tensor*) – Indices of region detections that should be kept.

Returns
Subsampled region detections.

Return type
dict[str, torch.Tensor]

Optical Flow extraction module for AI-VAD implementation.

`class anomalib.models.video.ai_vad.flow.FlowExtractor`(**args*, ***kwargs*)

Bases: `Module`

Optical Flow extractor.

Computes the pixel displacement between 2 consecutive frames from a video clip.

`forward`(*first_frame*, *last_frame*)

Forward pass through the flow extractor.

Parameters

- **first_frame** (*torch.Tensor*) – Batch of starting frames of shape (N, 3, H, W).
- **last_frame** (*torch.Tensor*) – Batch of last frames of shape (N, 3, H, W).

Returns
Estimated optical flow map of shape (N, 2, H, W).

Return type
Tensor

`pre_process`(*first_frame*, *last_frame*)

Resize inputs to dimensions required by backbone.

Parameters

- **first_frame** (*torch.Tensor*) – Starting frame of optical flow computation.

- **last_frame** (*torch.Tensor*) – Last frame of optical flow computation.

Returns
Preprocessed first and last frame.

Return type
tuple[torch.Tensor, torch.Tensor]

Density estimation module for AI-VAD model implementation.

class anomalib.models.video.ai_vad.density.**BaseDensityEstimator**(**args*, ***kwargs*)

Bases: `Module`, `ABC`

Base density estimator.

abstract fit()

Compose model using collected features.

Return type
`None`

forward(*features*)

Update or predict depending on training status.

Return type
`Tensor` | `tuple[Tensor, Tensor]` | `None`

abstract predict(*features*)

Predict the density of a set of features.

Return type
`Tensor` | `tuple[Tensor, Tensor]`

abstract update(*features*, *group=None*)

Update the density model with a new set of features.

Return type
`None`

class anomalib.models.video.ai_vad.density.**CombinedDensityEstimator**(*use_pose_features=True*, *use_deep_features=True*, *use_velocity_features=False*, *n_neighbors_pose=1*, *n_neighbors_deep=1*, *n_components_velocity=5*)

Bases: `BaseDensityEstimator`

Density estimator for AI-VAD.

Combines density estimators for the different feature types included in the model.

Parameters

- **use_pose_features** (*bool*) – Flag indicating if pose features should be used. Defaults to `True`.
- **use_deep_features** (*bool*) – Flag indicating if deep features should be used. Defaults to `True`.
- **use_velocity_features** (*bool*) – Flag indicating if velocity features should be used. Defaults to `False`.

- **n_neighbors_pose** (*int*) – Number of neighbors used in KNN density estimation for pose features. Defaults to 1.
- **n_neighbors_deep** (*int*) – Number of neighbors used in KNN density estimation for deep features. Defaults to 1.
- **n_components_velocity** (*int*) – Number of components used by GMM density estimation for velocity features. Defaults to 5.

fit()

Fit the density estimation models on the collected features.

Return type
None

predict(*features*)

Predict the region- and image-level anomaly scores for an image based on a set of features.

Parameters
features (*dict[Tensor]*) – Dictionary containing extracted features for a single frame.

Returns
Region-level anomaly scores for all regions withing the frame. Tensor: Frame-level anomaly score for the frame.

Return type
Tensor

update(*features*, *group=None*)

Update the density estimators for the different feature types.

Parameters

- **features** (*dict[FeatureType, torch.Tensor]*) – Dictionary containing extracted features for a single frame.
- **group** (*str*) – Identifier of the video from which the frame was sampled. Used for grouped density estimation.

Return type
None

class anomalib.models.video.ai_vad.density.**GMMEstimator**(*n_components=2*)

Bases: *BaseDensityEstimator*

Density estimation based on Gaussian Mixture Model.

Parameters
n_components (*int*) – Number of components used in the GMM. Defaults to 2.

fit()

Fit the GMM and compute normalization statistics.

Return type
None

predict(*features*, *normalize=True*)

Predict the density of a set of feature vectors.

Parameters

- **features** (*torch.Tensor*) – Input feature vectors.

- **normalize** (*bool*) – Flag indicating if the density should be normalized to min-max stats of the feature bank. Defaults to `True`.

Returns
Density scores of the input feature vectors.

Return type
Tensor

`update`(*features*, *group=None*)

Update the feature bank.

Return type
`None`

`class anomalib.models.video.ai_vad.density.GroupedKNNEstimator`(*n_neighbors*)

Bases: `DynamicBufferMixin`, *BaseDensityEstimator*

Grouped KNN density estimator.

Keeps track of the group (e.g. video id) from which the features were sampled for normalization purposes.

Parameters
n_neighbors (*int*) – Number of neighbors used in KNN search.

`fit`()

Fit the KNN model by stacking the feature vectors and computing the normalization statistics.

Return type
`None`

`predict`(*features*, *group=None*, *n_neighbors=1*, *normalize=True*)

Predict the (normalized) density for a set of features.

Parameters

- **features** (*torch.Tensor*) – Input features that will be compared to the density model.
- **group** (*str, optional*) – Group (video id) from which the features originate. If passed, all features of the same group in the memory bank will be excluded from the density estimation. Defaults to `None`.
- **n_neighbors** (*int*) – Number of neighbors used in the KNN search. Defaults to `1`.
- **normalize** (*bool*) – Flag indicating if the density should be normalized to min-max stats of the feature bank. Defatuls to `True`.

Returns
Mean (normalized) distances of input feature vectors to k nearest neighbors in feature bank.

Return type
Tensor

`update`(*features*, *group=None*)

Update the internal feature bank while keeping track of the group.

Parameters

- **features** (*torch.Tensor*) – Feature vectors extracted from a video frame.
- **group** (*str*) – Identifier of the group (video) from which the frame was sampled.

Return type
`None`