Metrics

Contents

Custom anomaly evaluation metrics.

class* anomalib.metrics.AUPR(kwargs*)**

Bases: `PrecisionRecallCurve`

Area under the PR curve.

This metric computes the area under the precision-recall curve.

Parameters:

kwargs (`Any`) – Additional arguments to the TorchMetrics base class.

Examples

To compute the metric for a set of predictions and ground truth targets:

```
>>> true = torch.tensor([0, 1, 1, 1, 0, 0, 0, 0, 1, 1])
>>> pred = torch.tensor([0.59, 0.35, 0.72, 0.33, 0.73, 0.81, 0.30, 0.05, 0.04,
```

```
>>> metric = AUPR()
>>> metric(pred, true)
tensor(0.4899)
```

It is also possible to update the metric state incrementally within batches:

```
>>> for batch in dataloader:
...     # Compute prediction and target tensors
...     metric.update(pred, true)
>>> metric.compute()
```

Once the metric has been computed, we can plot the PR curve:

```
>>> figure, title = metric.generate_figure()
```

compute()

First compute PR curve, then compute area under the curve.

Return type:

`Tensor`

Returns:

Value of the AUPR metric

generate_figure()

Generate a figure containing the PR curve as well as the random baseline and the AUC.

Returns:

Tuple containing both the PR curve and the figure title to be used for logging

Return type:

tuple[Figure, str]

`update(preds, target)`

Update state with new values.

Need to flatten new values as PrecicionRecallCurve expects them in this format for binary classification.

Parameters:

- **preds** (*torch.Tensor*) – predictions of the model
- **target** (*torch.Tensor*) – ground truth targets

Return type:

`None`

class `anomalib.metrics.AUPRO(dist_sync_on_step=False, process_group=None, dist_sync_fn=None, fpr_limit=0.3, num_thresholds=None)`

Bases: `Metric`

Area under per region overlap (AUPRO) Metric.

Parameters:

- **dist_sync_on_step** (*bool*) – Synchronize metric state across processes at each `forward()` before returning the value at the step. Default: `False`
- **process_group** (*Optional[Any]*) – Specify the process group on which synchronization is called. Default: `None` (which selects the entire world)
- **dist_sync_fn** (*Optional[Callable]*) – Callback that performs the allgather operation on the metric state. When `None`, DDP will be used to perform the allgather. Default: `None`
- **fpr_limit** (*float*) – Limit for the false positive rate. Defaults to `0.3`.
- **num_thresholds** (*int*) – Number of thresholds to use for computing the roc curve. Defaults to `None`. If `None`, the roc curve is computed with the thresholds returned by `torchmetrics.functional.classification.thresholds`.

Examples

```
>>> import torch
>>> from anomalib.metrics import AUPRO
...
>>> labels = torch.randint(low=0, high=2, size=(1, 10, 5), dtype=torch.float32)
>>> preds = torch.rand_like(labels)
...
>>> aupro = AUPRO(fpr_limit=0.3)
>>> aupro(preds, labels)
tensor(0.4321)
```

Increasing the fpr_limit will increase the AUPRO value:

```
>>> aupro = AUPRO(fpr_limit=0.7)
>>> aupro(preds, labels)
tensor(0.5271)
```

`compute()`

Fist compute PRO curve, then compute and scale area under the curve.

Returns:
Value of the AUPRO metric

Return type:
Tensor

`compute_pro(`*`cca, target, preds`*`)`

Compute the pro/fpr value-pairs until the fpr specified by self.fpr_limit.

It leverages the fact that the overlap corresponds to the tpr, and thus computes the overall PRO curve by aggregating per-region tpr/fpr values produced by ROC-construction.

Returns:
tuple containing final fpr and tpr values.

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

`generate_figure()`

Generate a figure containing the PRO curve and the AUPRO.

Returns:

Tuple containing both the figure and the figure title to be used for logging

Return type:

tuple[Figure, str]

static `interp1d`(*old_x*, *old_y*, *new_x*)

Interpolate a 1D signal linearly to new sampling points.

Parameters:

- **old_x** (*torch.Tensor*) – original 1-D x values (same size as y)
- **old_y** (*torch.Tensor*) – original 1-D y values (same size as x)
- **new_x** (*torch.Tensor*) – x-values where y should be interpolated at

Returns:

y-values at corresponding new_x values.

Return type:

Tensor

`perform_cca`()

Perform the Connected Component Analysis on the self.target tensor.

Raises:

ValueError – ValueError is raised if self.target doesn’t conform with requirements imposed by kornia for connected component analysis.

Returns:

Components labeled from 0 to N.

Return type:

Tensor

`update`(*preds*, *target*)

Update state with new values.

Parameters:

- **preds** (*torch.Tensor*) – predictions of the model
- **target** (*torch.Tensor*) – ground truth targets

Return type:

`None`

class `anomalib.metrics.AUROC(`*thresholds=None*, *ignore_index=None*, *validate_args=True*, ***kwargs*`)`

Bases: `BinaryROC`

Area under the ROC curve.

Examples

```
>>> import torch
>>> from anomalib.metrics import AUROC
...
>>> preds = torch.tensor([0.13, 0.26, 0.08, 0.92, 0.03])
>>> target = torch.tensor([0, 0, 1, 1, 0])
...
>>> auroc = AUROC()
>>> auroc(preds, target)
tensor(0.6667)
```

It is possible to update the metric state incrementally:

```
>>> auroc.update(preds[:2], target[:2])
>>> auroc.update(preds[2:], target[2:])
>>> auroc.compute()
tensor(0.6667)
```

To plot the ROC curve, use the `generate_figure` method:

```
>>> fig, title = auroc.generate_figure()
```

`compute()`

First compute ROC curve, then compute area under the curve.

Returns:

Value of the AUROC metric

Return type:

Tensor

`generate_figure()`

Generate a figure containing the ROC curve, the baseline and the AUROC.

Returns:

Tuple containing both the figure and the figure title to be used for logging

Return type:

tuple[Figure, str]

`update(preds, target)`

Update state with new values.

Need to flatten new values as ROC expects them in this format for binary classification.

Parameters:

- **preds** (*torch.Tensor*) – predictions of the model
- **target** (*torch.Tensor*) – ground truth targets

Return type:

`None`

class `anomalib.metrics.AnomalyScoreDistribution(**kwargs)`

Bases: `Metric`

Mean and standard deviation of the anomaly scores of normal training data.

`compute()`

Compute stats.

Return type:

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

`update(*args, anomaly_scores=None, anomaly_maps=None, **kwargs)`

Update the precision-recall curve metric.

Return type:

`None`

`class anomalib.metrics.BinaryPrecisionRecallCurve(thresholds=None, ignore_index=None, validate_args=True, **kwargs)`

Bases: `BinaryPrecisionRecallCurve`

Binary precision-recall curve with without threshold prediction normalization.

`update(preds, target)`

Update metric state with new predictions and targets.

Unlike the base class, this accepts raw predictions and targets.

Parameters:

- **preds** (*Tensor*) – Predicted probabilities
- **target** (*Tensor*) – Ground truth labels

Return type:

`None`

`class anomalib.metrics.F1AdaptiveThreshold(default_value=0.5, **kwargs)`

Bases: `BinaryPrecisionRecallCurve`, `BaseThreshold`

Anomaly Score Threshold.

This class computes/stores the threshold that determines the anomalous label given anomaly scores. It initially computes the adaptive threshold to find the optimal f1_score and stores the computed adaptive threshold value.

Parameters:

default_value (`float`) – Default value of the threshold. Defaults to `0.5`.

Examples Print to PDF ▶

To find the best threshold that maximizes the F1 score, we could run the following:

```
>>> from anomalib.metrics import F1AdaptiveThreshold
>>> import torch
...
>>> labels = torch.tensor([0, 0, 0, 1, 1])
>>> preds = torch.tensor([2.3, 1.6, 2.6, 7.9, 3.3])
...
>>> adaptive_threshold = F1AdaptiveThreshold(default_value=0.5)
>>> threshold = adaptive_threshold(preds, labels)
>>> threshold
tensor(3.3000)
```

`compute()`

Compute the threshold that yields the optimal F1 score.

Compute the F1 scores while varying the threshold. Store the optimal threshold as attribute and return the maximum value of the F1 score.

Return type:

`Tensor`

Returns:

Value of the F1 score at the optimal threshold.

class `anomalib.metrics.F1Score(`*threshold=0.5*, *multidim_average='global'*, *ignore_index=None*, *validate_args=True*, *****kwargs***`)`

Bases: `BinaryF1Score`

This is a wrapper around torchmetrics' BinaryF1Score.

The idea behind this is to retain the current configuration otherwise the one from torchmetrics requires `task` as a parameter.

class anomalib.metrics.ManualThreshold(*default_value=0.5*, ***kwargs*)

Bases: `BaseThreshold`

Initialize Manual Threshold.

Parameters:

- **default_value** (*float, optional*) – Default threshold value. Defaults to `0.5`.
- **kwargs** – Any keyword arguments.

Examples

```
>>> from anomalib.metrics import ManualThreshold
>>> import torch
...
>>> manual_threshold = ManualThreshold(default_value=0.5)
...
>>> labels = torch.randint(low=0, high=2, size=(5,))
>>> preds = torch.rand(5)
...
>>> threshold = manual_threshold(preds, labels)
>>> threshold
tensor(0.5000, dtype=torch.float64)
```

As the threshold is manually set, the threshold value is the same as the `default_value`.

```
>>> labels = torch.randint(low=0, high=2, size=(5,))
>>> preds = torch.rand(5)
>>> threshold = manual_threshold(preds2, labels2)
>>> threshold
tensor(0.5000, dtype=torch.float64)
```

The threshold value remains the same even if the inputs change.

compute()

Compute the threshold.

In case of manual thresholding, the threshold is already set and does not need to be computed.

Returns:

Value of the optimal threshold.

Return type:

torch.Tensor

`update(*args, **kwargs)`

Do nothing.

Parameters:

- ***args** – Any positional arguments.
- ****kwargs** – Any keyword arguments.

Return type:

`None`

`class anomalib.metrics.MinMax(**kwargs)`

Bases: `Metric`

Track the min and max values of the observations in each batch.

Parameters:

- **full_state_update** (*bool, optional*) – Whether to update the state with the new values. Defaults to `True`.
- **kwargs** – Any keyword arguments.

Examples

```
>>> from anomalib.metrics import MinMax
>>> import torch
...
>>> predictions = torch.tensor([0.0807, 0.6329, 0.0559, 0.9860, 0.3595])
>>> minmax = MinMax()
>>> minmax(predictions)
(tensor(0.0559), tensor(0.9860))
```

It is possible to update the minmax values with a new tensor of predictions.

```
>>> new_predictions = torch.tensor([0.3251, 0.3169, 0.3072, 0.6247, 0.9999])
>>> minmax.update(new_predictions)
>>> minmax.compute()
(tensor(0.0559), tensor(0.9999))
```

`compute()`

Return min and max values.

Return type:

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

`update(predictions, *args, **kwargs)`

Update the min and max values.

Return type:

`None`

class `anomalib.metrics.PRO(threshold=0.5, **kwargs)`

Bases: `Metric`

Per-Region Overlap (PRO) Score.

This metric computes the macro average of the per-region overlap between the predicted anomaly masks and the ground truth masks.

Parameters:

- **threshold** (*float*) – Threshold used to binarize the predictions. Defaults to `0.5`.
- **kwargs** – Additional arguments to the TorchMetrics base class.

Example

Import the metric from the package:

```
>>> import torch
>>> from anomalib.metrics import PRO
```

Create random `preds` and `labels` tensors:

```
>>> labels = torch.randint(low=0, high=2, size=(1, 10, 5), dtype=torch.float32)
>>> preds = torch.rand_like(labels)
```

Compute the PRO score for labels and preds:

```
>>> pro = PRO(threshold=0.5)
>>> pro.update(preds, labels)
>>> pro.compute()
tensor(0.5433)
```

Note

Note that the example above shows random predictions and labels. Therefore, the PRO score above may not be reproducible.

`compute()`

Compute the macro average of the PRO score across all regions in all batches.

Return type:

`Tensor`

Example

To compute the metric based on the state accumulated from multiple batches, use the `compute` method:

```
>>> pro.compute()
tensor(0.5433)
```

`update(predictions, targets)`

Compute the PRO score for the current batch.

Parameters:

- **predictions** (*torch.Tensor*) – Predicted anomaly masks (Bx1xHxW)
- **targets** (*torch.Tensor*) – Ground truth anomaly masks (Bx1xHxW)

Return type:

`None`

Example

To update the metric state for the current batch, use the `update` method:

```
>>> pro.update(preds, labels)
```

Previous
Engine

Next
Loggers