

---

# PyTorch-Metrics Documentation

*Release 0.10.0*

Lightning-AI et al.

Oct 04, 2022



# USER GUIDE

<b>1</b>	<b>Install TorchMetrics</b>	<b>3</b>
1.1	Quick Start	3
1.2	All TorchMetrics	6
1.3	Structure Overview	9
1.4	Implementing a Metric	19
1.5	TorchMetrics in PyTorch Lightning	26
1.6	Using Classification Metrics	30
1.7	Using Retrieval Metrics	32
1.8	Perceptual Evaluation of Speech Quality (PESQ)	33
1.9	Permutation Invariant Training (PIT)	35
1.10	Scale-Invariant Signal-to-Distortion Ratio (SI-SDR)	37
1.11	Scale-Invariant Signal-to-Noise Ratio (SI-SNR)	39
1.12	Short-Time Objective Intelligibility (STOI)	41
1.13	Signal to Distortion Ratio (SDR)	43
1.14	Signal-to-Noise Ratio (SNR)	46
1.15	Accuracy	48
1.16	AUC	64
1.17	AUROC	66
1.18	Average Precision	78
1.19	Binned Average Precision	90
1.20	Binned Precision Recall Curve	91
1.21	Binned Recall At Fixed Precision	93
1.22	Calibration Error	94
1.23	Cohen Kappa	101
1.24	Confusion Matrix	108
1.25	Coverage Error	119
1.26	Dice	123
1.27	Dice Score	127
1.28	Exact Match	128
1.29	F-1 Score	131
1.30	F-Beta Score	146
1.31	Hamming Distance	161
1.32	Hinge Loss	175
1.33	Jaccard Index	183
1.34	Label Ranking Average Precision	193
1.35	Label Ranking Loss	196
1.36	Matthews Correlation Coefficient	200
1.37	Precision	208
1.38	Precision Recall	224
1.39	Precision Recall Curve	226

1.40	Recall . . . . .	240
1.41	Recall At Fixed Precision . . . . .	255
1.42	ROC . . . . .	264
1.43	Specificity . . . . .	278
1.44	Stat Scores . . . . .	294
1.45	Error Relative Global Dim. Synthesis (ERGAS) . . . . .	310
1.46	Frechet Inception Distance (FID) . . . . .	312
1.47	Image Gradients . . . . .	314
1.48	Inception Score . . . . .	314
1.49	Kernel Inception Distance . . . . .	316
1.50	Learned Perceptual Image Patch Similarity (LPIPS) . . . . .	318
1.51	Multi-Scale SSIM . . . . .	319
1.52	Peak Signal-to-Noise Ratio (PSNR) . . . . .	322
1.53	Spectral Angle Mapper . . . . .	324
1.54	Spectral Distortion Index . . . . .	326
1.55	Structural Similarity Index Measure (SSIM) . . . . .	328
1.56	Universal Image Quality Index . . . . .	330
1.57	Mean-Average-Precision (mAP) . . . . .	332
1.58	Cosine Similarity . . . . .	336
1.59	Euclidean Distance . . . . .	337
1.60	Linear Similarity . . . . .	338
1.61	Manhattan Distance . . . . .	339
1.62	Concordance Corr. Coef. . . . .	340
1.63	Cosine Similarity . . . . .	341
1.64	Explained Variance . . . . .	343
1.65	KL Divergence . . . . .	345
1.66	Mean Absolute Error (MAE) . . . . .	347
1.67	Mean Absolute Percentage Error (MAPE) . . . . .	348
1.68	Mean Squared Error (MSE) . . . . .	350
1.69	Mean Squared Log Error (MSLE) . . . . .	351
1.70	Pearson Corr. Coef. . . . .	352
1.71	R2 Score . . . . .	354
1.72	Spearman Corr. Coef. . . . .	356
1.73	Symmetric Mean Absolute Percentage Error (SMAPE) . . . . .	358
1.74	Tweedie Deviance Score . . . . .	359
1.75	Weighted MAPE . . . . .	361
1.76	Retrieval Fall-Out . . . . .	363
1.77	Retrieval Hit Rate . . . . .	365
1.78	Retrieval Mean Average Precision (MAP) . . . . .	366
1.79	Retrieval Mean Reciprocal Rank (MRR) . . . . .	368
1.80	Retrieval Normalized DCG . . . . .	369
1.81	Retrieval Precision . . . . .	371
1.82	Precision Recall Curve . . . . .	373
1.83	Retrieval R-Precision . . . . .	375
1.84	Retrieval Recall . . . . .	377
1.85	BERT Score . . . . .	379
1.86	BLEU Score . . . . .	383
1.87	Char Error Rate . . . . .	384
1.88	ChrF Score . . . . .	386
1.89	Extended Edit Distance . . . . .	388
1.90	InfoLM . . . . .	390
1.91	Match Error Rate . . . . .	394
1.92	Perplexity . . . . .	396
1.93	ROUGE Score . . . . .	397

1.94	Sacre BLEU Score . . . . .	400
1.95	SQuAD . . . . .	402
1.96	Translation Edit Rate (TER) . . . . .	405
1.97	Word Error Rate . . . . .	407
1.98	Word Info. Lost . . . . .	408
1.99	Word Info. Preserved . . . . .	410
1.100	Concatenation . . . . .	411
1.101	Maximum . . . . .	412
1.102	Mean . . . . .	413
1.103	Minimum . . . . .	414
1.104	Sum . . . . .	414
1.105	Bootstrapper . . . . .	415
1.106	Classwise Wrapper . . . . .	416
1.107	Metric Tracker . . . . .	418
1.108	Min / Max . . . . .	420
1.109	Multi-output Wrapper . . . . .	421
1.110	torchmetrics.Metric . . . . .	423
1.111	torchmetrics.utilities.data . . . . .	427
1.112	TorchMetrics Governance . . . . .	429
1.113	Contributor Covenant Code of Conduct . . . . .	430
1.114	Contributing . . . . .	431
1.115	Changelog . . . . .	434
<b>2</b>	<b>Indices and tables</b>	<b>453</b>
	<b>Index</b>	<b>455</b>



TorchMetrics is a collection of 80+ PyTorch metrics implementations and an easy-to-use API to create custom metrics. It offers:

- A standardized interface to increase reproducibility
- Reduces Boilerplate
- Distributed-training compatible
- Rigorously tested
- Automatic accumulation over batches
- Automatic synchronization between multiple devices

You can use TorchMetrics in any PyTorch model, or within [PyTorch Lightning](#) to enjoy the following additional benefits:

- Your data will always be placed on the same device as your metrics
- You can log *Metric* objects directly in Lightning to reduce even more boilerplate





## INSTALL TORCHMETRICS

For pip users

```
pip install torchmetrics
```

Or directly from conda

```
conda install -c conda-forge torchmetrics
```

### 1.1 Quick Start

TorchMetrics is a collection of 80+ PyTorch metrics implementations and an easy-to-use API to create custom metrics. It offers:

- A standardized interface to increase reproducibility
- Reduces Boilerplate
- Distributed-training compatible
- Rigorously tested
- Automatic accumulation over batches
- Automatic synchronization between multiple devices

You can use TorchMetrics in any PyTorch model, or within [PyTorch Lightning](#) to enjoy additional features:

- This means that your data will always be placed on the same device as your metrics.
- Native support for logging metrics in Lightning to reduce even more boilerplate.

### 1.1.1 Install

You can install TorchMetrics using pip or conda:

```
# Python Package Index (PyPI)
pip install torchmetrics
# Conda
conda install -c conda-forge torchmetrics
```

Eventually if there is a missing PyTorch wheel for your OS or Python version you can simply compile [PyTorch](#) from source:

```
# Optional if you do not need compile GPU support
export USE_CUDA=0 # just to keep it simple
# you can install the latest state from master
pip install git+https://github.com/pytorch/pytorch.git
# OR set a particular PyTorch release
pip install git+https://github.com/pytorch/pytorch.git@<release-tag>
# and finalize with installing TorchMetrics
pip install torchmetrics
```

### 1.1.2 Using TorchMetrics

#### Functional metrics

Similar to `torch.nn`, most metrics have both a class-based and a functional version. The functional versions implement the basic operations required for computing each metric. They are simple python functions that as input take `torch.tensors` and return the corresponding metric as a `torch.tensor`. The code-snippet below shows a simple example for calculating the accuracy using the functional interface:

```
import torch
# import our library
import torchmetrics

# simulate a classification problem
preds = torch.randn(10, 5).softmax(dim=-1)
target = torch.randint(5, (10,))

acc = torchmetrics.functional.accuracy(preds, target)
```

#### Module metrics

Nearly all functional metrics have a corresponding class-based metric that calls it a functional counterpart underneath. The class-based metrics are characterized by having one or more internal metrics states (similar to the parameters of the PyTorch module) that allow them to offer additional functionalities:

- Accumulation of multiple batches
- Automatic synchronization between multiple devices
- Metric arithmetic

The code below shows how to use the class-based interface:

```
import torch
# import our library
import torchmetrics

# initialize metric
metric = torchmetrics.Accuracy()

n_batches = 10
for i in range(n_batches):
    # simulate a classification problem
    preds = torch.randn(10, 5).softmax(dim=-1)
    target = torch.randint(5, (10,))
    # metric on current batch
    acc = metric(preds, target)
    print(f"Accuracy on batch {i}: {acc}")

# metric on all batches using custom accumulation
acc = metric.compute()
print(f"Accuracy on all data: {acc}")

# Resetting internal state such that metric ready for new data
metric.reset()
```

### 1.1.3 Implementing your own metric

Implementing your own metric is as easy as subclassing a `torch.nn.Module`. Simply, subclass `Metric` and do the following:

1. Implement `__init__` where you call `self.add_state` for every internal state that is needed for the metrics computations
2. Implement update method, where all logic that is necessary for updating metric states go
3. Implement compute method, where the final metric computations happens

For practical examples and more info about implementing a metric, please see this [page](#).

## Development Environment

TorchMetrics provides a [Devcontainer](#) configuration for [Visual Studio Code](#) to use a [Docker container](#) as a pre-configured development environment. This avoids struggles setting up a development environment and makes them reproducible and consistent. Please follow the [installation instructions](#) and make yourself familiar with the [container tutorials](#) if you want to use them. In order to use GPUs, you can enable them within the `.devcontainer/devcontainer.json` file.

## 1.2 All TorchMetrics





## 1.3 Structure Overview

TorchMetrics is a Metrics API created for easy metric development and usage in PyTorch and PyTorch Lightning. It is rigorously tested for all edge cases and includes a growing list of common metric implementations.

The metrics API provides `update()`, `compute()`, `reset()` functions to the user. The metric *base class* inherits `torch.nn.Module` which allows us to call `metric(...)` directly. The `forward()` method of the base `Metric` class serves the dual purpose of calling `update()` on its input and simultaneously returning the value of the metric over the provided input.

These metrics work with DDP in PyTorch and PyTorch Lightning by default. When `.compute()` is called in distributed mode, the internal state of each metric is synced and reduced across each process, so that the logic present in `.compute()` is applied to state information from all processes.

This metrics API is independent of PyTorch Lightning. Metrics can directly be used in PyTorch as shown in the example:

```
from torchmetrics.classification import Accuracy

train_accuracy = Accuracy()
valid_accuracy = Accuracy()

for epoch in range(epochs):
    for x, y in train_data:
        y_hat = model(x)

        # training step accuracy
        batch_acc = train_accuracy(y_hat, y)
        print(f"Accuracy of batch{i} is {batch_acc}")

    for x, y in valid_data:
        y_hat = model(x)
        valid_accuracy.update(y_hat, y)

    # total accuracy over all training batches
    total_train_accuracy = train_accuracy.compute()

    # total accuracy over all validation batches
    total_valid_accuracy = valid_accuracy.compute()

    print(f"Training acc for epoch {epoch}: {total_train_accuracy}")
    print(f"Validation acc for epoch {epoch}: {total_valid_accuracy}")

    # Reset metric states after each epoch
    train_accuracy.reset()
    valid_accuracy.reset()
```

**Note:** Metrics contain internal states that keep track of the data seen so far. Do not mix metric states across training, validation and testing. It is highly recommended to re-initialize the metric per mode as shown in the examples above.

---

**Note:** Metric states are **not** added to the models `state_dict` by default. To change this, after initializing the metric, the method `.persistent(mode)` can be used to enable (`mode=True`) or disable (`mode=False`) this behaviour.

---

---

**Note:** Due to specialized logic around metric states, we in general do **not** recommend that metrics are initialized inside other metrics (nested metrics), as this can lead to weird behaviour. Instead consider subclassing a metric or use `torchmetrics.MetricCollection`.

---

### 1.3.1 Metrics and devices

Metrics are simple subclasses of `Module` and their metric states behave similar to buffers and parameters of modules. This means that metrics states should be moved to the same device as the input of the metric:

```
from torchmetrics import Accuracy

target = torch.tensor([1, 1, 0, 0], device=torch.device("cuda", 0))
preds = torch.tensor([0, 1, 0, 0], device=torch.device("cuda", 0))

# Metric states are always initialized on cpu, and needs to be moved to
# the correct device
confmat = Accuracy(num_classes=2).to(torch.device("cuda", 0))
out = confmat(preds, target)
print(out.device) # cuda:0
```

However, when **properly defined** inside a `Module` or `LightningModule` the metric will be automatically moved to the same device as the module when using `.to(device)`. Being **properly defined** means that the metric is correctly identified as a child module of the model (check `.children()` attribute of the model). Therefore, metrics cannot be placed in native python list and dict, as they will not be correctly identified as child modules. Instead of list use `ModuleList` and instead of dict use `ModuleDict`. Furthermore, when working with multiple metrics the native `MetricCollection` module can also be used to wrap multiple metrics.

```
from torchmetrics import Accuracy, MetricCollection

class MyModule(torch.nn.Module):
    def __init__(self):
        ...
        # valid ways metrics will be identified as child modules
        self.metric1 = Accuracy()
        self.metric2 = nn.ModuleList(Accuracy())
        self.metric3 = nn.ModuleDict({'accuracy': Accuracy()})
        self.metric4 = MetricCollection([Accuracy()]) # torchmetrics build-in collection_
↪class

    def forward(self, batch):
        data, target = batch
        preds = self(data)
        ...
        val1 = self.metric1(preds, target)
        val2 = self.metric2[0](preds, target)
        val3 = self.metric3['accuracy'](preds, target)
        val4 = self.metric4(preds, target)
```



You can always check which device the metric is located on using the `.device` property.

## Metrics in Dataparallel (DP) mode

When using metrics in [Dataparallel \(DP\)](#) mode, one should be aware DP will both create and clean-up replicas of Metric objects during a single forward pass. This has the consequence, that the metric state of the replicas will as default be destroyed before we can sync them. It is therefore recommended, when using metrics in DP mode, to initialize them with `dist_sync_on_step=True` such that metric states are synchronized between the main process and the replicas before they are destroyed.

Additionally, if metrics are used together with a *LightningModule* the metric update/logging should be done in the `<mode>_step_end` method (where `<mode>` is either `training`, `validation` or `test`), else it will lead to wrong accumulation. In practice do the following:

```
def training_step(self, batch, batch_idx):
    data, target = batch
    preds = self(data)
    ...
    return {'loss': loss, 'preds': preds, 'target': target}

def training_step_end(self, outputs):
    #update and log
    self.metric(outputs['preds'], outputs['target'])
    self.log('metric', self.metric)
```

## Metrics in Distributed Data Parallel (DDP) mode

When using metrics in [Distributed Data Parallel \(DDP\)](#) mode, one should be aware that DDP will add additional samples to your dataset if the size of your dataset is not equally divisible by `batch_size * num_processors`. The added samples will always be replicates of datapoints already in your dataset. This is done to secure an equal load for all processes. However, this has the consequence that the calculated metric value will be slightly biased towards those replicated samples, leading to a wrong result.

During training and/or validation this may not be important, however it is highly recommended when evaluating the test dataset to only run on a single gpu or use a `join` context in conjunction with DDP to prevent this behaviour.

### 1.3.2 Metrics and 16-bit precision

Most metrics in our collection can be used with 16-bit precision (`torch.half`) tensors. However, we have found the following limitations:

- In general pytorch had better support for 16-bit precision much earlier on GPU than CPU. Therefore, we recommend that anyone that want to use metrics with half precision on CPU, upgrade to atleast pytorch v1.6 where support for operations such as addition, subtraction, multiplication ect. was added.
- Some metrics does not work at all in half precision on CPU. We have explicitly stated this in their docstring, but they are also listed below:
  - *Peak Signal-to-Noise Ratio (PSNR)*
  - *Structural Similarity Index Measure (SSIM)*
  - *KL Divergence*

You can always check the precision/dtype of the metric by checking the `.dtype` property.

### 1.3.3 Metric Arithmetics

Metrics support most of python built-in operators for arithmetic, logic and bitwise operations.

For example for a metric that should return the sum of two different metrics, implementing a new metric is an overhead that is not necessary. It can now be done with:

```
first_metric = MyFirstMetric()
second_metric = MySecondMetric()

new_metric = first_metric + second_metric
```

`new_metric.update(*args, **kwargs)` now calls `update` of `first_metric` and `second_metric`. It forwards all positional arguments but forwards only the keyword arguments that are available in respective metric's update declaration. Similarly `new_metric.compute()` now calls `compute` of `first_metric` and `second_metric` and adds the results up. It is important to note that all implemented operations always returns a new metric object. This means that the line `first_metric == second_metric` will not return a bool indicating if `first_metric` and `second_metric` is the same metric, but will return a new metric that checks if the `first_metric.compute() == second_metric.compute()`.

This pattern is implemented for the following operators (with `a` being metrics and `b` being metrics, tensors, integer or floats):

- Addition (`a + b`)
- Bitwise AND (`a & b`)
- Equality (`a == b`)
- Floordivision (`a // b`)
- Greater Equal (`a >= b`)
- Greater (`a > b`)
- Less Equal (`a <= b`)
- Less (`a < b`)
- Matrix Multiplication (`a @ b`)
- Modulo (`a % b`)
- Multiplication (`a * b`)
- Inequality (`a != b`)
- Bitwise OR (`a | b`)
- Power (`a ** b`)
- Subtraction (`a - b`)
- True Division (`a / b`)
- Bitwise XOR (`a ^ b`)
- Absolute Value (`abs(a)`)
- Inversion (`~a`)
- Negative Value (`neg(a)`)
- Positive Value (`pos(a)`)
- Indexing (`a[0]`)

**Note:** Some of these operations are only fully supported from Pytorch v1.4 and onwards, explicitly we found: `add`, `mul`, `rmatmul`, `rsub`, `rmod`

### 1.3.4 MetricCollection

In many cases it is beneficial to evaluate the model output by multiple metrics. In this case the `MetricCollection` class may come in handy. It accepts a sequence of metrics and wraps these into a single callable metric class, with the same interface as any other metric.

Example:

```
from torchmetrics import MetricCollection, Accuracy, Precision, Recall
target = torch.tensor([0, 2, 0, 2, 0, 1, 0, 2])
preds = torch.tensor([2, 1, 2, 0, 1, 2, 2, 2])
metric_collection = MetricCollection([
    Accuracy(),
    Precision(num_classes=3, average='macro'),
    Recall(num_classes=3, average='macro')
])
print(metric_collection(preds, target))
```

```
{'Accuracy': tensor(0.1250),
 'Precision': tensor(0.0667),
 'Recall': tensor(0.1111)}
```

Similarly it can also reduce the amount of code required to log multiple metrics inside your `LightningModule`. In most cases we just have to replace `self.log` with `self.log_dict`.

```
from torchmetrics import Accuracy, MetricCollection, Precision, Recall

class MyModule(LightningModule):
    def __init__(self):
        metrics = MetricCollection([Accuracy(), Precision(), Recall()])
        self.train_metrics = metrics.clone(prefix='train_')
        self.valid_metrics = metrics.clone(prefix='val_')

    def training_step(self, batch, batch_idx):
        logits = self(x)
        # ...
        output = self.train_metrics(logits, y)
        # use log_dict instead of log
        # metrics are logged with keys: train_Accuracy, train_Precision and train_Recall
        self.log_dict(output)

    def validation_step(self, batch, batch_idx):
        logits = self(x)
        # ...
        self.valid_metrics.update(logits, y)

    def validation_epoch_end(self, outputs):
        # use log_dict instead of log
```

(continues on next page)

(continued from previous page)

```
# metrics are logged with keys: val_Accuracy, val_Precision and val_Recall
output = self.valid_metric.compute()
self.log_dict(output)
```

---

**Note:** *MetricCollection* as default assumes that all the metrics in the collection have the same call signature. If this is not the case, input that should be given to different metrics can given as keyword arguments to the collection.

---

An additional advantage of using the *MetricCollection* object is that it will automatically try to reduce the computations needed by finding groups of metrics that share the same underlying metric state. If such a group of metrics is found only one of them is actually updated and the updated state will be broadcasted to the rest of the metrics within the group. In the example above, this will lead to a 2x-3x lower computational cost compared to disabling this feature in the case of the validation metrics where only `update` is called (this feature does not work in combination with `forward`). However, this speedup comes with a fixed cost upfront, where the state-groups have to be determined after the first update. In case the groups are known beforehand, these can also be set manually to avoid this extra cost of the dynamic search. See the `compute_groups` argument in the class docs below for more information on this topic.

```
class torchmetrics.MetricCollection(metrics, *additional_metrics, prefix=None, postfix=None,
                                     compute_groups=True)
```

*MetricCollection* class can be used to chain metrics that have the same call pattern into one single class.

#### Parameters

- **metrics** (`Union[Metric, Sequence[Metric], Dict[str, Metric]]`) – One of the following
  - list or tuple (sequence): if metrics are passed in as a list or tuple, will use the metrics class name as key for output dict. Therefore, two metrics of the same class cannot be chained this way.
  - arguments: similar to passing in as a list, metrics passed in as arguments will use their metric class name as key for the output dict.
  - dict: if metrics are passed in as a dict, will use each key in the dict as key for output dict. Use this format if you want to chain together multiple of the same metric with different parameters. Note that the keys in the output dict will be sorted alphabetically.
- **prefix** (`Optional[str]`) – a string to append in front of the keys of the output dict
- **postfix** (`Optional[str]`) – a string to append after the keys of the output dict
- **compute\_groups** (`Union[bool, List[List[str]]]`) – By default the *MetricCollection* will try to reduce the computations needed for the metrics in the collection by checking if they belong to the same **compute group**. All metrics in a compute group share the same metric state and are therefore only different in their compute step e.g. accuracy, precision and recall can all be computed from the true positives/negatives and false positives/negatives. By default, this argument is `True` which enables this feature. Set this argument to `False` for disabling this behaviour. Can also be set to a list of lists of metrics for setting the compute groups yourself.

---

**Note:** The compute groups feature can significantly speedup the calculation of metrics under the right conditions. First, the feature is only available when calling the `update` method and not when calling `forward` method due to the internal logic of `forward` preventing this. Secondly, since we compute groups share metric states by reference, calling `.items()`, `.values()` etc. on the metric collection will break this reference and a copy of states are instead returned in this case (reference will be reestablished on the next call to `update`).

---

---

**Note:** Metric collections can be nested at initialization (see last example) but the output of the collection will still be a single flatten dictionary combining the prefix and postfix arguments from the nested collection.

---

### Raises

- **ValueError** – If one of the elements of `metrics` is not an instance of `pl.metrics.Metric`.
- **ValueError** – If two elements in `metrics` have the same name.
- **ValueError** – If `metrics` is not a list, tuple or a dict.
- **ValueError** – If `metrics` is dict and `additional_metrics` are passed in.
- **ValueError** – If `prefix` is set and it is not a string.
- **ValueError** – If `postfix` is set and it is not a string.

### Example (input as list):

```
>>> import torch
>>> from pprint import pprint
>>> from torchmetrics import MetricCollection, Accuracy, Precision, Recall, MeanSquaredError
>>> target = torch.tensor([0, 2, 0, 2, 0, 1, 0, 2])
>>> preds = torch.tensor([2, 1, 2, 0, 1, 2, 2, 2])
>>> metrics = MetricCollection([Accuracy(),
...                             Precision(num_classes=3, average='macro'),
...                             Recall(num_classes=3, average='macro')])
>>> metrics(preds, target)
{'Accuracy': tensor(0.1250), 'Precision': tensor(0.0667), 'Recall': tensor(0.1111)}
```

### Example (input as arguments):

```
>>> metrics = MetricCollection(Accuracy(), Precision(num_classes=3, average='macro'),
...                             Recall(num_classes=3, average='macro'))
>>> metrics(preds, target)
{'Accuracy': tensor(0.1250), 'Precision': tensor(0.0667), 'Recall': tensor(0.1111)}
```

### Example (input as dict):

```
>>> metrics = MetricCollection({'micro_recall': Recall(num_classes=3, average='micro'),
...                             'macro_recall': Recall(num_classes=3, average='macro')})
>>> same_metric = metrics.clone()
>>> pprint(metrics(preds, target))
{'macro_recall': tensor(0.1111), 'micro_recall': tensor(0.1250)}
>>> pprint(same_metric(preds, target))
{'macro_recall': tensor(0.1111), 'micro_recall': tensor(0.1250)}
```

### Example (specification of compute groups):

```

>>> metrics = MetricCollection(
...     Recall(num_classes=3, average='macro'),
...     Precision(num_classes=3, average='macro'),
...     MeanSquaredError(),
...     compute_groups=[['Recall', 'Precision'], ['MeanSquaredError']]
... )
>>> metrics.update(preds, target)
>>> pprint(metrics.compute())
{'MeanSquaredError': tensor(2.3750), 'Precision': tensor(0.0667), 'Recall':
tensor(0.1111)}
>>> pprint(metrics.compute_groups)
{0: ['Recall', 'Precision'], 1: ['MeanSquaredError']}

```

**Example (nested metric collections):**

```

>>> metrics = MetricCollection([
...     MetricCollection([
...         Accuracy(num_classes=3, average='macro'),
...         Precision(num_classes=3, average='macro')
...     ], postfix='_macro'),
...     MetricCollection([
...         Accuracy(num_classes=3, average='micro'),
...         Precision(num_classes=3, average='micro')
...     ], postfix='_micro'),
... ], prefix='valmetrics/')
>>> pprint(metrics(preds, target))
{'valmetrics/Accuracy_macro': tensor(0.1111),
'valmetrics/Accuracy_micro': tensor(0.1250),
'valmetrics/Precision_macro': tensor(0.0667),
'valmetrics/Precision_micro': tensor(0.1250)}

```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**add\_metrics**(*metrics*, *\*additional\_metrics*)

Add new metrics to Metric Collection.

**Return type** `None`

**clone**(*prefix=None*, *postfix=None*)

Make a copy of the metric collection :type `_sphinx_paramlinks_torchmetrics.MetricCollection.clone.prefix`: `Optional[str]` :param `_sphinx_paramlinks_torchmetrics.MetricCollection.clone.prefix`: a string to append in front of the metric keys :type `_sphinx_paramlinks_torchmetrics.MetricCollection.clone.postfix`: `Optional[str]` :param `_sphinx_paramlinks_torchmetrics.MetricCollection.clone.postfix`: a string to append after the keys of the output dict

**Return type** `MetricCollection`

**compute**()

Compute the result for each metric in the collection.

**Return type** `Dict[str, Any]`

**forward**(*\*args*, *\*\*kwargs*)

Iteratively call forward for each metric.

Positional arguments (args) will be passed to every metric in the collection, while keyword arguments (kwargs) will be filtered based on the signature of the individual metric.

**Return type** `Dict[str, Any]`

**items**(*keep\_base=False, copy\_state=True*)

Return an iterable of the ModuleDict key/value pairs.

**Parameters**

- **keep\_base** (`bool`) – Whether to add prefix/postfix on the collection.
- **copy\_state** (`bool`) – If metric states should be copied between metrics in the same compute group or just passed by reference

**Return type** `Iterable[Tuple[str, Module]]`

**keys**(*keep\_base=False*)

Return an iterable of the ModuleDict key. :type `_sphinx_paramlinks_torchmetrics.MetricCollection.keys.keep_base`: `bool` :param `_sphinx_paramlinks_torchmetrics.MetricCollection.keys.keep_base`: Whether to add prefix/postfix on the items collection.

**Return type** `Iterable[Hashable]`

**persistent**(*mode=True*)

Method for post-init to change if metric states should be saved to its `state_dict`.

**Return type** `None`

**reset**()

Iteratively call reset for each metric.

**Return type** `None`

**update**(\*args, \*\*kwargs)

Iteratively call update for each metric.

Positional arguments (args) will be passed to every metric in the collection, while keyword arguments (kwargs) will be filtered based on the signature of the individual metric.

**Return type** `None`

**values**(*copy\_state=True*)

Return an iterable of the ModuleDict values.

**Parameters** **copy\_state** (`bool`) – If metric states should be copied between metrics in the same compute group or just passed by reference

**Return type** `Iterable[Module]`

**property compute\_groups**: `Dict[int, List[str]]`

Return a dict with the current compute groups in the collection.

**Return type** `Dict[int, List[str]]`

### 1.3.5 Module vs Functional Metrics

The functional metrics follow the simple paradigm input in, output out. This means they don't provide any advanced mechanisms for syncing across DDP nodes or aggregation over batches. They simply compute the metric value based on the given inputs.

Also, the integration within other parts of PyTorch Lightning will never be as tight as with the Module-based interface. If you look for just computing the values, the functional metrics are the way to go. However, if you are looking for the best integration and user experience, please consider also using the Module interface.

### 1.3.6 Metrics and differentiability

Metrics support backpropagation, if all computations involved in the metric calculation are differentiable. All modular metric classes have the property `is_differentiable` that determines if a metric is differentiable or not.

However, note that the cached state is detached from the computational graph and cannot be back-propagated. Not doing this would mean storing the computational graph for each update call, which can lead to out-of-memory errors. In practise this means that:

```
MyMetric.is_differentiable  # returns True if metric is differentiable
metric = MyMetric()
val = metric(pred, target)  # this value can be back-propagated
val = metric.compute()      # this value cannot be back-propagated
```

A functional metric is differentiable if its corresponding modular metric is differentiable.

### 1.3.7 Metrics and hyperparameter optimization

If you want to directly optimize a metric it needs to support backpropagation (see section above). However, if you are just interested in using a metric for hyperparameter tuning and are not sure if the metric should be maximized or minimized, all modular metric classes have the `higher_is_better` property that can be used to determine this:

```
# returns True because accuracy is optimal when it is maximized
torchmetrics.Accuracy.higher_is_better

# returns False because the mean squared error is optimal when it is minimized
torchmetrics.MeanSquaredError.higher_is_better
```

### 1.3.8 Advanced metric settings

The following is a list of additional arguments that can be given to any metric class (in the `**kwargs` argument) that will alter how metric states are stored and synced.

If you are running metrics on GPU and are encountering that you are running out of GPU VRAM then the following argument can help:

- `compute_on_cpu` will automatically move the metric states to cpu after calling `update`, making sure that GPU memory is not filling up. The consequence will be that the `compute` method will be called on CPU instead of GPU. Only applies to metric states that are lists.

If you are running in a distributed environment, TorchMetrics will automatically take care of the distributed synchronization for you. However, the following three keyword arguments can be given to any metric class for further control over the distributed aggregation:



- `dist_sync_on_step`: This argument is `bool` that indicates if the metric should synchronize between different devices every time `forward` is called. Setting this to `True` is in general not recommended as synchronization is an expensive operation to do after each batch.
- `process_group`: By default we synchronize across the *world* i.e. all processes being computed on. You can provide an `torch._C._distributed_c10d.ProcessGroup` in this argument to specify exactly what devices should be synchronized over.
- `dist_sync_fn`: By default we use `torch.distributed.all_gather()` to perform the synchronization between devices. Provide another callable function for this argument to perform custom distributed synchronization.

## 1.4 Implementing a Metric

To implement your own custom metric, subclass the base `Metric` class and implement the following methods:

- `__init__()`: Each state variable should be called using `self.add_state(...)`.
- `update()`: Any code needed to update the state given any inputs to the metric.
- `compute()`: Computes a final value from the state of the metric.

We provide the remaining interface, such as `reset()` that will make sure to correctly reset all metric states that have been added using `add_state`. You should therefore not implement `reset()` yourself. Additionally, adding metric states with `add_state` will make sure that states are correctly synchronized in distributed settings (DDP). To see how metric states are synchronized across distributed processes, refer to `add_state()` docs from the base `Metric` class.

Example implementation:

```
from torchmetrics import Metric

class MyAccuracy(Metric):
    def __init__(self):
        super().__init__()
        self.add_state("correct", default=torch.tensor(0), dist_reduce_fx="sum")
        self.add_state("total", default=torch.tensor(0), dist_reduce_fx="sum")

    def update(self, preds: torch.Tensor, target: torch.Tensor):
        preds, target = self._input_format(preds, target)
        assert preds.shape == target.shape

        self.correct += torch.sum(preds == target)
        self.total += target.numel()

    def compute(self):
        return self.correct.float() / self.total
```

Additionally you may want to set the class properties: `is_differentiable`, `higher_is_better` and `full_state_update`. Note that none of them are strictly required for the metric to work.

```
from torchmetrics import Metric

class MyMetric(Metric):
    # Set to True if the metric is differentiable else set to False
    is_differentiable: Optional[bool] = None
```

(continues on next page)

(continued from previous page)

```
# Set to True if the metric reaches it optimal value when the metric is maximized.  
# Set to False if it when the metric is minimized.  
higher_is_better: Optional[bool] = True  
  
# Set to True if the metric during 'update' requires access to the global metric  
# state for its calculations. If not, setting this to False indicates that all  
# batch states are independent and we will optimize the runtime of 'forward'  
full_state_update: bool = True
```

### 1.4.1 Internal implementation details

This section briefly describes how metrics work internally. We encourage looking at the source code for more info. Internally, TorchMetrics wraps the user defined `update()` and `compute()` method. We do this to automatically synchronize and reduce metric states across multiple devices. More precisely, calling `update()` does the following internally:

1. Clears computed cache.
2. Calls user-defined `update()`.

Similarly, calling `compute()` does the following internally:

1. Syncs metric states between processes.
2. Reduce gathered metric states.
3. Calls the user defined `compute()` method on the gathered metric states.
4. Cache computed result.

From a user's standpoint this has one important side-effect: computed results are cached. This means that no matter how many times `compute` is called after one and another, it will continue to return the same result. The cache is first emptied on the next call to `update`.

`forward` serves the dual purpose of both returning the metric on the current data and updating the internal metric state for accumulating over multiple batches. The `forward()` method achieves this by combining calls to `update`, `compute` and `reset`. Depending on the class property `full_state_update`, `forward` can behave in two ways:

1. If `full_state_update` is `True` it indicates that the metric during `update` requires access to the full metric state and we therefore need to do two calls to `update` to secure that the metric is calculated correctly
  1. Calls `update()` to update the global metric state (for accumulation over multiple batches)
  2. Caches the global state.
  3. Calls `reset()` to clear global metric state.
  4. Calls `update()` to update local metric state.
  5. Calls `compute()` to calculate metric for current batch.
  6. Restores the global state.
2. If `full_state_update` is `False` (default) the metric state of one batch is completely independent of the state of other batches, which means that we only need to call `update` once.
  1. Caches the global state.
  2. Calls `reset` the metric to its default state

3. Calls `update` to update the state with local batch statistics
4. Calls `compute` to calculate the metric for the current batch
5. Reduce the global state and batch state into a single state that becomes the new global state

If implementing your own metric, we recommend trying out the metric with `full_state_update` class property set to both `True` and `False`. If the results are equal, then setting it to `False` will usually give the best performance.

---

**class** `torchmetrics.Metric(**kwargs)`

Base class for all metrics present in the Metrics API.

Implements `add_state()`, `forward()`, `reset()` and a few other things to handle distributed synchronization and per-step metric computation.

Override `update()` and `compute()` functions to implement your own metric. Use `add_state()` to register metric state variables which keep track of state on each call of `update()` and are synchronized across processes when `compute()` is called.

---

**Note:** Metric state variables can either be `torch.Tensors` or an empty list which can be used to store `torch.Tensors`.

---



---

**Note:** Different metrics only override `update()` and not `forward()`. A call to `update()` is valid, but it won't return the metric value at the current step. A call to `forward()` automatically calls `update()` and also returns the metric value at the current step.

---

**Parameters** `kwargs` ([Any](#)) – additional keyword arguments, see [Advanced metric settings](#) for more info.

- **`compute_on_cpu`**: If metric state should be stored on CPU during computations. Only works for list states.
- **`dist_sync_on_step`**: If metric state should synchronize on `forward()`. Default is `False`
- **`process_group`**: The process group on which the synchronization is called. Default is the world.
- **`dist_sync_fn`**: function that performs the allgather option on the metric state. Default is an custom implementation that calls `torch.distributed.all_gather` internally.
- **`sync_on_compute`**: If metric state should synchronize when `compute` is called. Default is `True`.

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

**add\_state**(*name*, *default*, *dist\_reduce\_fx=None*, *persistent=False*)

Adds metric state variable. Only used by subclasses.

**Parameters**

- **`name`** (`str`) – The name of the state variable. The variable will then be accessible at `self.name`.
- **`default`** (`Union[list, Tensor]`) – Default value of the state; can either be a `torch.Tensor` or an empty list. The state will be reset to this value when `self.reset()` is called.

- **dist\_reduce\_fx** (*Optional*) – Function to reduce state across multiple processes in distributed mode. If value is "sum", "mean", "cat", "min" or "max" we will use `torch.sum`, `torch.mean`, `torch.cat`, `torch.min` and `torch.max` respectively, each with argument `dim=0`. Note that the "cat" reduction only makes sense if the state is a list, and not a tensor. The user can also pass a custom function in this parameter.
- **persistent** (*Optional*) – whether the state will be saved as part of the modules `state_dict`. Default is `False`.

---

**Note:** Setting `dist_reduce_fx` to `None` will return the metric state synchronized across different processes. However, there won't be any reduction function applied to the synchronized metric state.

The metric states would be synced as follows

- If the metric state is `torch.Tensor`, the synced value will be a stacked `torch.Tensor` across the process dimension if the metric state was a `torch.Tensor`. The original `torch.Tensor` metric state retains dimension and hence the synchronized output will be of shape `(num_process, ...)`.
- If the metric state is a `list`, the synced value will be a `list` containing the combined elements from all processes.

---

**Note:** When passing a custom function to `dist_reduce_fx`, expect the synchronized metric state to follow the format discussed in the above note.

---

#### Raises

- **ValueError** – If default is not a tensor or an empty `list`.
- **ValueError** – If `dist_reduce_fx` is not callable or one of "mean", "sum", "cat", `None`.

**Return type** `None`

#### `clone()`

Make a copy of the metric.

**Return type** `Metric`

#### `abstract compute()`

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Any`

#### `double()`

Method override default and prevent dtype casting.

Please use `metric.set_dtype(dtype)` instead.

**Return type** `Metric`

#### `float()`

Method override default and prevent dtype casting.

Please use `metric.set_dtype(dtype)` instead.

**Return type** `Metric`

**forward**(\*args, \*\*kwargs)

forward serves the dual purpose of both computing the metric on the current batch of inputs but also add the batch statistics to the overall accumululating metric state.

Input arguments are the exact same as corresponding update method. The returned output is the exact same as the output of compute.

**Return type** Any

**half**()

Method override default and prevent dtype casting.

Please use `metric.set_dtype(dtype)` instead.

**Return type** Metric

**persistent**(mode=False)

Method for post-init to change if metric states should be saved to its state\_dict.

**Return type** None

**reset**()

This method automatically resets the metric state variables to their default value.

**Return type** None

**set\_dtype**(dst\_type)

Special version of type for transferring all metric states to specific dtype :type  
`_sphinx_paramlinks_torchmetrics.Metric.set_dtype.dst_type:` Union[str, dtype] :param  
`_sphinx_paramlinks_torchmetrics.Metric.set_dtype.dst_type:` the desired type :type  
`_sphinx_paramlinks_torchmetrics.Metric.set_dtype.dst_type:` type or string

**Return type** Metric

**state\_dict**(destination=None, prefix="", keep\_vars=False)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to None are not included.

**Warning:** Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument `destination` as it is not designed for end-users.

### Parameters

- **destination** (*dict*, optional) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: None.
- **prefix** (*str*, optional) – a prefix added to parameter and buffer names to compose the keys in `state_dict`. Default: ''.
- **keep\_vars** (*bool*, optional) – by default the `Tensor`s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

**Returns** a dictionary containing a whole state of the module

**Return type** `dict`

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

**sync**(*dist\_sync\_fn=None, process\_group=None, should\_sync=True, distributed\_available=<function jit\_distributed\_available>*)

Sync function for manually controlling when metrics states should be synced across processes.

**Parameters**

- **dist\_sync\_fn** (`Optional[Callable]`) – Function to be used to perform states synchronization
- **process\_group** (`Optional[Any]`) – Specify the process group on which synchronization is called. default: *None* (which selects the entire world)
- **should\_sync** (`bool`) – Whether to apply to state synchronization. This will have an impact only when running in a distributed setting.
- **distributed\_available** (`Optional[Callable]`) – Function to determine if we are running inside a distributed setting

**Return type** `None`

**sync\_context**(*dist\_sync\_fn=None, process\_group=None, should\_sync=True, should\_unsync=True, distributed\_available=<function jit\_distributed\_available>*)

Context manager to synchronize the states between processes when running in a distributed setting and restore the local cache states after yielding.

**Parameters**

- **dist\_sync\_fn** (`Optional[Callable]`) – Function to be used to perform states synchronization
- **process\_group** (`Optional[Any]`) – Specify the process group on which synchronization is called. default: *None* (which selects the entire world)
- **should\_sync** (`bool`) – Whether to apply to state synchronization. This will have an impact only when running in a distributed setting.
- **should\_unsync** (`bool`) – Whether to restore the cache state so that the metrics can continue to be accumulated.
- **distributed\_available** (`Optional[Callable]`) – Function to determine if we are running inside a distributed setting

**Return type** `Generator`

**type**(*dst\_type*)

Method override default and prevent dtype casting.

Please use `metric.set_dtype(dtype)` instead.

**Return type** `Metric`

**unsync**(*should\_unsync=True*)

Unsync function for manually controlling when metrics states should be reverted back to their local states.

**Parameters** **should\_unsync** (*bool*) – Whether to perform unsync

**Return type** *None*

**abstract update**(*\*, \*\*\_\_*)

Override this method to update the state variables of your metric class.

**Return type** *None*

**property device:** *torch.device*

Return the device of the metric.

**Return type** *device*

## 1.4.2 Contributing your metric to TorchMetrics

Wanting to contribute the metric you have implemented? Great, we are always open to adding more metrics to `torchmetrics` as long as they serve a general purpose. However, to keep all our metrics consistent we request that the implementation and tests gets formatted in the following way:

1. Start by reading our [contribution guidelines](#).
2. First implement the functional backend. This takes cares of all the logic that goes into the metric. The code should be put into a single file placed under `torchmetrics/functional/"domain"/"new_metric".py` where `domain` is the type of metric (classification, regression, nlp etc) and `new_metric` is the name of the metric. In this file, there should be the following three functions:
  1. `_new_metric_update(...)`: everything that has to do with type/shape checking and all logic required before distributed syncing need to go here.
  2. `_new_metric_compute(...)`: all remaining logic.
  3. `new_metric(...)`: essentially wraps the `_update` and `_compute` private functions into one public function that makes up the functional interface for the metric.

---

**Note:** The [functional accuracy](#) metric is a great example of this division of logic.

---

3. In a corresponding file placed in `torchmetrics/"domain"/"new_metric".py` create the module interface:
  1. Create a new module metric by subclassing `torchmetrics.Metric`.
  2. In the `__init__` of the module call `self.add_state` for as many metric states are needed for the metric to proper accumulate metric statistics.
  3. The module interface should essentially call the private `_new_metric_update(...)` in its `update` method and similarly the `_new_metric_compute(...)` function in its `compute`. No logic should really be implemented in the module interface. We do this to not have duplicate code to maintain.

---

**Note:** The module [Accuracy](#) metric that corresponds to the above functional example show-cases these steps.

---

4. Remember to add binding to the different relevant `__init__` files.

5. Testing is key to keeping `torchmetrics` trustworthy. This is why we have a very rigid testing protocol. This means that we in most cases require the metric to be tested against some other common framework (`sklearn`, `scipy` etc).
  1. Create a testing file in `unittests/"domain"/test_"new_metric".py`. Only one file is needed as it is intended to test both the functional and module interface.
  2. In that file, start by defining a number of test inputs that your metric should be evaluated on.
  3. Create a testclass `class NewMetric(MetricTester)` that inherits from `tests.helpers.testers.MetricTester`. This testclass should essentially implement the `test_"new_metric"_class` and `test_"new_metric"_fn` methods that respectively tests the module interface and the functional interface.
  4. The testclass should be parameterized (using `@pytest.mark.parametrize`) by the different test inputs defined initially. Additionally, the `test_"new_metric"_class` method should also be parameterized with an `ddp` parameter such that it gets tested in a distributed setting. If your metric has additional parameters, then make sure to also parameterize these such that different combinations of inputs and parameters gets tested.
  5. (optional) If your metric raises any exception, please add tests that showcase this.

---

**Note:** The [test file for accuracy](#) metric shows how to implement such tests.

---

If you only can figure out part of the steps, do not fear to send a PR. We will much rather receive working metrics that are not formatted exactly like our codebase, than not receiving any. Formatting can always be applied. We will gladly guide and/or help implement the remaining :]

## 1.5 TorchMetrics in PyTorch Lightning

TorchMetrics was originally created as part of [PyTorch Lightning](#), a powerful deep learning research framework designed for scaling models without boilerplate.

---

**Note:** TorchMetrics always offers compatibility with the last 2 major PyTorch Lightning versions, but we recommend to always keep both frameworks up-to-date for the best experience.

---

While TorchMetrics was built to be used with native PyTorch, using TorchMetrics with Lightning offers additional benefits:

- Modular metrics are automatically placed on the correct device when properly defined inside a `LightningModule`. This means that your data will always be placed on the same device as your metrics. No need to call `.to(device)` anymore!
- Native support for logging metrics in Lightning using `self.log` inside your `LightningModule`.
- The `.reset()` method of the metric will automatically be called at the end of an epoch.

The example below shows how to use a metric in your [LightningModule](#):

```
class MyModel(LightningModule):  
  
    def __init__(self):  
        ...  
        self.accuracy = torchmetrics.Accuracy()
```

(continues on next page)



(continued from previous page)

```

def training_step(self, batch, batch_idx):
    x, y = batch
    preds = self(x)
    ...
    # log step metric
    self.accuracy(preds, y)
    self.log('train_acc_step', self.accuracy)
    ...

def training_epoch_end(self, outs):
    # log epoch metric
    self.log('train_acc_epoch', self.accuracy)

```

Metric logging in Lightning happens through the `self.log` or `self.log_dict` method. Both methods only support the logging of *scalar-tensors*. While the vast majority of metrics in `torchmetrics` returns a scalar tensor, some metrics such as `ConfusionMatrix`, `ROC`, `MeanAveragePrecision`, `ROUGEScore` return outputs that are non-scalar tensors (often dicts or list of tensors) and should therefore be dealt with separately. For info about the return type and shape please look at the documentation for the `compute` method for each metric you want to log.

### 1.5.1 Logging TorchMetrics

Logging metrics can be done in two ways: either logging the metric object directly or the computed metric values. When *Metric* objects, which return a scalar tensor are logged directly in Lightning using the `LightningModule` `self.log` method, Lightning will log the metric based on `on_step` and `on_epoch` flags present in `self.log(...)`. If `on_epoch` is `True`, the logger automatically logs the end of epoch metric value by calling `.compute()`.

**Note:** `sync_dist`, `sync_dist_op`, `sync_dist_group`, `reduce_fx` and `tbptt_reduce_fx` flags from `self.log(...)` don't affect the metric logging in any manner. The metric class contains its own distributed synchronization logic.

This however is only true for metrics that inherit the base class `Metric`, and thus the functional metric API provides no support for in-built distributed synchronization or reduction functions.

```

class MyModule(LightningModule):

    def __init__(self):
        ...
        self.train_acc = torchmetrics.Accuracy()
        self.valid_acc = torchmetrics.Accuracy()

    def training_step(self, batch, batch_idx):
        x, y = batch
        preds = self(x)
        ...
        self.train_acc(preds, y)
        self.log('train_acc', self.train_acc, on_step=True, on_epoch=False)

    def validation_step(self, batch, batch_idx):
        logits = self(x)

```

(continues on next page)

(continued from previous page)

```
...
self.valid_acc(logits, y)
self.log('valid_acc', self.valid_acc, on_step=True, on_epoch=True)
```

As an alternative to logging the metric object and letting Lightning take care of when to reset the metric etc. you can also manually log the output of the metrics.

```
class MyModule(LightningModule):

    def __init__(self):
        ...
        self.train_acc = torchmetrics.Accuracy()
        self.valid_acc = torchmetrics.Accuracy()

    def training_step(self, batch, batch_idx):
        x, y = batch
        preds = self(x)
        ...
        batch_value = self.train_acc(preds, y)
        self.log('train_acc_step', batch_value)

    def training_epoch_end(self, outputs):
        self.train_acc.reset()

    def validation_step(self, batch, batch_idx):
        logits = self(x)
        ...
        self.valid_acc.update(logits, y)

    def validation_epoch_end(self, outputs):
        self.log('valid_acc_epoch', self.valid_acc.compute())
        self.valid_acc.reset()
```

Note that logging metrics this way will require you to manually reset the metrics at the end of the epoch yourself. In general, we recommend logging the metric object to make sure that metrics are correctly computed and reset. Additionally, we highly recommend that the two ways of logging are not mixed as it can lead to wrong results.

**Note:** When using any Modular metric, calling `self.metric(...)` or `self.metric.forward(...)` serves the dual purpose of calling `self.metric.update()` on its input and simultaneously returning the metric value over the provided input. So if you are logging a metric *only* on epoch-level (as in the example above), it is recommended to call `self.metric.update()` directly to avoid the extra computation.

```
class MyModule(LightningModule):

    def __init__(self):
        ...
        self.valid_acc = torchmetrics.Accuracy()

    def validation_step(self, batch, batch_idx):
        logits = self(x)
        ...
        self.valid_acc.update(logits, y)
```

(continues on next page)

(continued from previous page)

```
self.log('valid_acc', self.valid_acc, on_step=True, on_epoch=True)
```

## 1.5.2 Common Pitfalls

The following contains a list of pitfalls to be aware of:

- If using metrics in data parallel mode (dp), the metric update/logging should be done in the `<mode>_step_end` method (where `<mode>` is either `training`, `validation` or `test`). This is because dp split the batches during the forward pass and metric states are destroyed after each forward pass, thus leading to wrong accumulation. In practice do the following:

```
class MyModule(LightningModule):

    def training_step(self, batch, batch_idx):
        data, target = batch
        preds = self(data)
        # ...
        return {'loss': loss, 'preds': preds, 'target': target}

    def training_step_end(self, outputs):
        # update and log
        self.metric(outputs['preds'], outputs['target'])
        self.log('metric', self.metric)
```

- Modular metrics contain internal states that should belong to only one DataLoader. In case you are using multiple DataLoaders, it is recommended to initialize a separate modular metric instances for each DataLoader and use them separately. The same holds for using separate metrics for training, validation and testing.

```
class MyModule(LightningModule):

    def __init__(self):
        ...
        self.val_acc = nn.ModuleList([torchmetrics.Accuracy() for _ in range(2)])

    def val_dataloader(self):
        return [DataLoader(...), DataLoader(...)]

    def validation_step(self, batch, batch_idx, dataloader_idx):
        x, y = batch
        preds = self(x)
        ...
        self.val_acc[dataloader_idx](preds, y)
        self.log('val_acc', self.val_acc[dataloader_idx])
```

- Mixing the two logging methods by calling `self.log("val", self.metric)` in `{training}/{val}/{test}_step` method and then calling `self.log("val", self.metric.compute())` in the corresponding `{training}/{val}/{test}_epoch_end` method. Because the object is logged in the first case, Lightning will reset the metric before calling the second line leading to errors or nonsense results.
- Calling `self.log("val", self.metric(preds, target))` with the intention of logging the metric object. Because `self.metric(preds, target)` corresponds to calling the forward method, this will return a tensor

and not the metric object. Such logging will be wrong in this case. Instead it is important to separate into separate lines:

```
def training_step(self, batch, batch_idx):
    x, y = batch
    preds = self(x)
    ...
    # log step metric
    self.accuracy(preds, y) # compute metrics
    self.log('train_acc_step', self.accuracy) # log metric object
```

## 1.6 Using Classification Metrics

### 1.6.1 Input types

For the purposes of classification metrics, inputs (predictions and targets) are split into these categories (N stands for the batch size and C for number of classes):

Table 1: \*dtype binary means integers that are either 0 or 1

Type	preds shape	preds dtype	target shape	target dtype
Binary	(N,)	float	(N,)	binary*
Multi-class	(N,)	int	(N,)	int
Multi-class with logits or probabilities	(N, C)	float	(N,)	int
Multi-label	(N, ...)	float	(N, ...)	binary*
Multi-dimensional multi-class	(N, ...)	int	(N, ...)	int
Multi-dimensional multi-class with logits or probabilities	(N, C, ...)	float	(N, ...)	int

**Note:** All dimensions of size 1 (except N) are “squeezed out” at the beginning, so that, for example, a tensor of shape (N, 1) is treated as (N, ).

When predictions or targets are integers, it is assumed that class labels start at 0, i.e. the possible class labels are 0, 1, 2, 3, etc. Below are some examples of different input types

```
# Binary inputs
binary_preds = torch.tensor([0.6, 0.1, 0.9])
binary_target = torch.tensor([1, 0, 2])

# Multi-class inputs
mc_preds = torch.tensor([0, 2, 1])
mc_target = torch.tensor([0, 1, 2])

# Multi-class inputs with probabilities
mc_preds_probs = torch.tensor([[0.8, 0.2, 0], [0.1, 0.2, 0.7], [0.3, 0.6, 0.1]])
mc_target_probs = torch.tensor([0, 1, 2])

# Multi-label inputs
```

(continues on next page)

(continued from previous page)

```
ml_preds = torch.tensor([[0.2, 0.8, 0.9], [0.5, 0.6, 0.1], [0.3, 0.1, 0.1]])
ml_target = torch.tensor([[0, 1, 1], [1, 0, 0], [0, 0, 0]])
```

### Using the multiclass parameter

In some cases, you might have inputs which appear to be (multi-dimensional) multi-class but are actually binary/multi-label - for example, if both predictions and targets are integer (binary) tensors. Or it could be the other way around, you want to treat binary/multi-label inputs as 2-class (multi-dimensional) multi-class inputs.

For these cases, the metrics where this distinction would make a difference, expose the `multiclass` argument. Let's see how this is used on the example of `StatScores` metric.

First, let's consider the case with label predictions with 2 classes, which we want to treat as binary.

```
from torchmetrics.functional import stat_scores

# These inputs are supposed to be binary, but appear as multi-class
preds = torch.tensor([0, 1, 0])
target = torch.tensor([1, 1, 0])
```

As you can see below, by default the inputs are treated as multi-class. We can set `multiclass=False` to treat the inputs as binary - which is the same as converting the predictions to float beforehand.

```
>>> stat_scores(preds, target, reduce='macro', num_classes=2)
tensor([[1, 1, 1, 0, 1],
        [1, 0, 1, 1, 2]])
>>> stat_scores(preds, target, reduce='macro', num_classes=1, multiclass=False)
tensor([[1, 0, 1, 1, 2]])
>>> stat_scores(preds.float(), target, reduce='macro', num_classes=1)
tensor([[1, 0, 1, 1, 2]])
```

Next, consider the opposite example: inputs are binary (as predictions are probabilities), but we would like to treat them as 2-class multi-class, to obtain the metric for both classes.

```
preds = torch.tensor([0.2, 0.7, 0.3])
target = torch.tensor([1, 1, 0])
```

In this case we can set `multiclass=True`, to treat the inputs as multi-class.

```
>>> stat_scores(preds, target, reduce='macro', num_classes=1)
tensor([[1, 0, 1, 1, 2]])
>>> stat_scores(preds, target, reduce='macro', num_classes=2, multiclass=True)
tensor([[1, 1, 1, 0, 1],
        [1, 0, 1, 1, 2]])
```

## 1.7 Using Retrieval Metrics

### 1.7.1 Input details

For the purposes of retrieval metrics, inputs (indexes, predictions and targets) must have the same size (N stands for the batch size) and the following types:

indexes shape	indexes dtype	preds shape	preds dtype	target shape	target dtype
(N,...)	long	(N,...)	float	(N,...)	long or bool

---

**Note:** All dimensions are flattened at the beginning, so that, for example, a tensor of shape (N, M) is treated as (N \* M, ).

---

In Information Retrieval you have a query that is compared with a variable number of documents. For each pair (Q<sub>i</sub>, D<sub>j</sub>), a score is computed that measures the relevance of document D w.r.t. query Q. Documents are then sorted by score and you hope that relevant documents are scored higher. `target` contains the labels for the documents (relevant or not).

Since a query may be compared with a variable number of documents, we use `indexes` to keep track of which scores belong to the set of pairs (Q<sub>i</sub>, D<sub>j</sub>) having the same query Q<sub>i</sub>.

---

**Note:** *Retrieval* metrics are only intended to be used globally. This means that the average of the metric over each batch can be quite different from the metric computed on the whole dataset. For this reason, we suggest to compute the metric only when all the examples has been provided to the metric. When using *Pytorch Lightning*, we suggest to use `on_step=False` and `on_epoch=True` in `self.log` or to place the metric calculation in `training_epoch_end`, `validation_epoch_end` or `test_epoch_end`.

---

```
>>> from torchmetrics import RetrievalMAP
>>> # functional version works on a single query at a time
>>> from torchmetrics.functional import retrieval_average_precision

>>> # the first query was compared with two documents, the second with three
>>> indexes = torch.tensor([0, 0, 1, 1, 1])
>>> preds = torch.tensor([0.8, -0.4, 1.0, 1.4, 0.0])
>>> target = torch.tensor([0, 1, 0, 1, 1])

>>> rmap = RetrievalMAP() # or some other retrieval metric
>>> rmap(preds, target, indexes=indexes)
tensor(0.6667)

>>> # the previous instruction is roughly equivalent to
>>> res = []
>>> # iterate over indexes of first and second query
>>> for indexes in ([0, 1], [2, 3, 4]):
...     res.append(retrieval_average_precision(preds[indexes], target[indexes]))
>>> torch.stack(res).mean()
tensor(0.6667)
```

## 1.8 Perceptual Evaluation of Speech Quality (PESQ)

### 1.8.1 Module Interface

**class** torchmetrics.audio.pesq.**PerceptualEvaluationSpeechQuality**(*fs, mode, n\_processes=1, \*\*kwargs*)

Perceptual Evaluation of Speech Quality (PESQ)

This is a wrapper for the pesq package [1]. Note that input will be moved to *cpu* to perform the metric calculation.

---

**Note:** using this metrics requires you to have `pesq` install. Either install as `pip install torchmetrics[audio]` or `pip install pesq`. Note that `pesq` will compile with your currently installed version of `numpy`, meaning that if you upgrade `numpy` at some point in the future you will most likely have to reinstall `pesq`.

---

Forward accepts

- `preds`: shape [...,time]
- `target`: shape [...,time]

#### Parameters

- **fs** (`int`) – sampling frequency, should be 16000 or 8000 (Hz)
- **mode** (`str`) – 'wb' (wide-band) or 'nb' (narrow-band)
- **keep\_same\_device** – whether to move the pesq value to the device of preds
- **n\_processes** (`int`) – integer specifying the number of processes to run in parallel for the metric calculation. Only applies to batches of data and if `multiprocessing` package is installed.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

- **ModuleNotFoundError** – If `peqs` package is not installed
- **ValueError** – If `fs` is not either 8000 or 16000
- **ValueError** – If `mode` is not either "wb" or "nb"

#### Example

```
>>> from torchmetrics.audio.pesq import PerceptualEvaluationSpeechQuality
>>> import torch
>>> g = torch.manual_seed(1)
>>> preds = torch.randn(8000)
>>> target = torch.randn(8000)
>>> nb_pesq = PerceptualEvaluationSpeechQuality(8000, 'nb')
>>> nb_pesq(preds, target)
tensor(2.2076)
>>> wb_pesq = PerceptualEvaluationSpeechQuality(16000, 'wb')
>>> wb_pesq(preds, target)
tensor(1.7359)
```

## References

[1] <https://github.com/ludlows/python-pesq>

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes average PESQ.

**Return type** `Tensor`

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

## 1.8.2 Functional Interface

`torchmetrics.functional.audio.pesq.perceptual_evaluation_speech_quality(preds, target, fs, mode, keep_same_device=False, n_processes=1)`

PESQ (Perceptual Evaluation of Speech Quality)

This is a wrapper for the `pesq` package [1]. Note that input will be moved to `cpu` to perform the metric calculation.

---

**Note:** using this metrics requires you to have `pesq` install. Either install as `pip install torchmetrics[audio]` or `pip install pesq`. Note that `pesq` will compile with your currently installed version of `numpy`, meaning that if you upgrade `numpy` at some point in the future you will most likely have to reinstall `pesq`.

---

**Parameters**

- **preds** (`Tensor`) – shape `[..., time]`
- **target** (`Tensor`) – shape `[..., time]`
- **fs** (`int`) – sampling frequency, should be 16000 or 8000 (Hz)
- **mode** (`str`) – 'wb' (wide-band) or 'nb' (narrow-band)
- **keep\_same\_device** (`bool`) – whether to move the `pesq` value to the device of `preds`
- **n\_processes** (`int`) – integer specifying the number of processes to run in parallel for the metric calculation. Only applies to batches of data and if `multiprocessing` package is installed.

**Return type** `Tensor`

**Returns** `pesq` value of shape `[...]`

**Raises**

- **ModuleNotFoundError** – If `peqs` package is not installed
- **ValueError** – If `fs` is not either 8000 or 16000



- **ValueError** – If mode is not either "wb" or "nb"

### Example

```
>>> from torchmetrics.functional.audio.pesq import perceptual_evaluation_speech_
    ↪quality
>>> import torch
>>> g = torch.manual_seed(1)
>>> preds = torch.randn(8000)
>>> target = torch.randn(8000)
>>> perceptual_evaluation_speech_quality(preds, target, 8000, 'nb')
tensor(2.2076)
>>> perceptual_evaluation_speech_quality(preds, target, 16000, 'wb')
tensor(1.7359)
```

### References

[1] <https://github.com/ludlows/python-pesq>

## 1.9 Permutation Invariant Training (PIT)

### 1.9.1 Module Interface

**class** torchmetrics.PermutationInvariantTraining(*metric\_func*, *eval\_func*='max', \*\*kwargs)

Permutation invariant training (PermutationInvariantTraining). The PermutationInvariantTraining implements the famous Permutation Invariant Training method.

[1] in speech separation field in order to calculate audio metrics in a permutation invariant way.

Forward accepts

- **preds**: shape [batch, spk, ...]
- **target**: shape [batch, spk, ...]

#### Parameters

- **metric\_func** (**Callable**) – a metric function accept a batch of target and estimate, i.e. `metric_func(preds[:, i, ...], target[:, j, ...])`, and returns a batch of metric tensors [batch]
- **eval\_func** (**str**) – the function to find the best permutation, can be 'min' or 'max', i.e. the smaller the better or the larger the better.
- **kwargs** (**Any**) – Additional keyword arguments for either the `metric_func` or distributed communication, see *Advanced metric settings* for more info.

**Returns** average PermutationInvariantTraining metric

### Example

```
>>> import torch
>>> from torchmetrics import PermutationInvariantTraining
>>> from torchmetrics.functional import scale_invariant_signal_noise_ratio
>>> _ = torch.manual_seed(42)
>>> preds = torch.randn(3, 2, 5) # [batch, spk, time]
>>> target = torch.randn(3, 2, 5) # [batch, spk, time]
>>> pit = PermutationInvariantTraining(scale_invariant_signal_noise_ratio, 'max')
>>> pit(preds, target)
tensor(-2.1065)
```

**Reference:** [1] D. Yu, M. Kolbaek, Z.-H. Tan, J. Jensen, Permutation invariant training of deep models for speaker-independent multi-talker speech separation, in: 2017 IEEE Int. Conf. Acoust. Speech Signal Process. ICASSP, IEEE, New Orleans, LA, 2017: pp. 241–245. <https://doi.org/10.1109/ICASSP.2017.7952154>.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### **compute()**

Computes average PermutationInvariantTraining metric.

**Return type** `Tensor`

#### **update(preds, target)**

Update state with predictions and targets.

#### **Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

## 1.9.2 Functional Interface

`torchmetrics.functional.permutation_invariant_training(preds, target, metric_func, eval_func='max', **kwargs)`

Permutation invariant training (PIT). The `permutation_invariant_training` implements the famous Permutation Invariant Training method.

[1] in speech separation field in order to calculate audio metrics in a permutation invariant way.

#### **Parameters**

- **preds** (`Tensor`) – shape [batch, spk, ...]
- **target** (`Tensor`) – shape [batch, spk, ...]
- **metric\_func** (`Callable`) – a metric function accept a batch of target and estimate, i.e. `metric_func(preds[:, i, ...], target[:, j, ...])`, and returns a batch of metric tensors [batch]
- **eval\_func** (`str`) – the function to find the best permutation, can be 'min' or 'max', i.e. the smaller the better or the larger the better.
- **kwargs** (`Any`) – Additional args for `metric_func`

**Return type** `Tuple[Tensor, Tensor]`

**Returns** best\_metric of shape [batch] best\_perm of shape [batch]

### Example

```
>>> from torchmetrics.functional.audio import scale_invariant_signal_distortion_
    ↪ratio
>>> # [batch, spk, time]
>>> preds = torch.tensor([[-0.0579,  0.3560, -0.9604], [-0.1719,  0.3205,  0.
    ↪2951]])
>>> target = torch.tensor([[-0.0579,  0.3560, -0.9604], [-0.1719,  0.3205,  0.
    ↪2951]])
>>> best_metric, best_perm = permutation_invariant_training(
...     preds, target, scale_invariant_signal_distortion_ratio, 'max')
>>> best_metric
tensor([-5.1091])
>>> best_perm
tensor([[0, 1]])
>>> pit_permutate(preds, best_perm)
tensor([[-0.0579,  0.3560, -0.9604],
        [-0.1719,  0.3205,  0.2951]])
```

**Reference:** [1] [Permutation Invariant Training of Deep Models](#)

## 1.10 Scale-Invariant Signal-to-Distortion Ratio (SI-SDR)

### 1.10.1 Module Interface

**class** `torchmetrics.ScaleInvariantSignalDistortionRatio`(*zero\_mean=False*, *\*\*kwargs*)

Scale-invariant signal-to-distortion ratio (SI-SDR). The SI-SDR value is in general considered an overall measure of how good a source sound.

Forward accepts

- `preds`: shape [...,time]
- `target`: shape [...,time]

#### Parameters

- **zero\_mean** (`bool`) – if to zero mean target and preds or not
- **kwargs** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises** `TypeError` – if target and preds have a different shape

**Returns** average si-sdr value

### Example

```
>>> import torch
>>> from torchmetrics import ScaleInvariantSignalDistortionRatio
>>> target = torch.tensor([3.0, -0.5, 2.0, 7.0])
>>> preds = torch.tensor([2.5, 0.0, 2.0, 8.0])
>>> si_sdr = ScaleInvariantSignalDistortionRatio()
>>> si_sdr(preds, target)
tensor(18.4030)
```

### References

[1] Le Roux, Jonathan, et al. “SDR half-baked or well done.” IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) 2019.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes average SI-SDR.

**Return type** `Tensor`

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

## 1.10.2 Functional Interface

`torchmetrics.functional.scale_invariant_signal_distortion_ratio(preds, target, zero_mean=False)`

Calculates Scale-invariant signal-to-distortion ratio (SI-SDR) metric. The SI-SDR value is in general considered an overall measure of how good a source sound.

**Parameters**

- **preds** (`Tensor`) – shape `[..., time]`
- **target** (`Tensor`) – shape `[..., time]`
- **zero\_mean** (`bool`) – If to zero mean target and preds or not

**Return type** `Tensor`

**Returns** si-sdr value of shape `[...]`

### Example

```
>>> from torchmetrics.functional.audio import scale_invariant_signal_distortion_
    ↪ratio
>>> target = torch.tensor([3.0, -0.5, 2.0, 7.0])
>>> preds = torch.tensor([2.5, 0.0, 2.0, 8.0])
>>> scale_invariant_signal_distortion_ratio(preds, target)
tensor(18.4030)
```

### References

[1] Le Roux, Jonathan, et al. “SDR half-baked or well done.” IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) 2019.

## 1.11 Scale-Invariant Signal-to-Noise Ratio (SI-SNR)

### 1.11.1 Module Interface

**class** torchmetrics.ScaleInvariantSignalNoiseRatio(\*\*kwargs)

Scale-invariant signal-to-noise ratio (SI-SNR).

Forward accepts

- preds: shape [...,time]
- target: shape [...,time]

**Parameters** **kwargs** (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

**Raises** **TypeError** – if target and preds have a different shape

**Returns** average si-snr value

### Example

```
>>> import torch
>>> from torchmetrics import ScaleInvariantSignalNoiseRatio
>>> target = torch.tensor([3.0, -0.5, 2.0, 7.0])
>>> preds = torch.tensor([2.5, 0.0, 2.0, 8.0])
>>> si_snr = ScaleInvariantSignalNoiseRatio()
>>> si_snr(preds, target)
tensor(15.0918)
```

## References

[1] Y. Luo and N. Mesgarani, “TaSNet: Time-Domain Audio Separation Network for Real-Time, Single-Channel Speech Separation,” 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2018, pp. 696-700, doi: 10.1109/ICASSP.2018.8462116.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes average SI-SNR.

**Return type** `Tensor`

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

### 1.11.2 Functional Interface

`torchmetrics.functional.scale_invariant_signal_noise_ratio(preds, target)`

Scale-invariant signal-to-noise ratio (SI-SNR).

**Parameters**

- **preds** (`Tensor`) – shape [..., time]
- **target** (`Tensor`) – shape [..., time]

**Return type** `Tensor`

**Returns** si-snr value of shape [...]

### Example

```
>>> import torch
>>> from torchmetrics.functional.audio import scale_invariant_signal_noise_ratio
>>> target = torch.tensor([3.0, -0.5, 2.0, 7.0])
>>> preds = torch.tensor([2.5, 0.0, 2.0, 8.0])
>>> scale_invariant_signal_noise_ratio(preds, target)
tensor(15.0918)
```

## References

[1] Y. Luo and N. Mesgarani, “TaSNet: Time-Domain Audio Separation Network for Real-Time, Single-Channel Speech Separation,” 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2018, pp. 696-700, doi: 10.1109/ICASSP.2018.8462116.

## 1.12 Short-Time Objective Intelligibility (STOI)

### 1.12.1 Module Interface

**class** `torchmetrics.audio.stoi.ShortTimeObjectiveIntelligibility`(*fs*, *extended=False*, *\*\*kwargs*)

STOI (Short-Time Objective Intelligibility, see [2,3]), a wrapper for the pystoi package [1]. Note that input will be moved to *cpu* to perform the metric calculation.

Intelligibility measure which is highly correlated with the intelligibility of degraded speech signals, e.g., due to additive noise, single-/multi-channel noise reduction, binary masking and vocoded speech as in CI simulations. The STOI-measure is intrusive, i.e., a function of the clean and degraded speech signals. STOI may be a good alternative to the speech intelligibility index (SII) or the speech transmission index (STI), when you are interested in the effect of nonlinear processing to noisy speech, e.g., noise reduction, binary masking algorithms, on speech intelligibility. Description taken from [Cees Taal's website](#).

---

**Note:** using this metrics requires you to have `pystoi` install. Either install as `pip install torchmetrics[audio]` or `pip install pystoi`

---

Forward accepts

- `preds`: shape `[...,time]`
- `target`: shape `[...,time]`

#### Parameters

- **`fs`** (`int`) – sampling frequency (Hz)
- **`extended`** (`bool`) – whether to use the extended STOI described in [4]
- **`kwargs`** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Returns** average STOI value

**Raises** `ModuleNotFoundError` – If `pystoi` package is not installed

### Example

```
>>> from torchmetrics.audio.stoi import ShortTimeObjectiveIntelligibility
>>> import torch
>>> g = torch.manual_seed(1)
>>> preds = torch.randn(8000)
>>> target = torch.randn(8000)
>>> stoi = ShortTimeObjectiveIntelligibility(8000, False)
>>> stoi(preds, target)
tensor(-0.0100)
```

## References

[1] <https://github.com/mpariente/pystoi>

[2] C.H.Taal, R.C.Hendriks, R.Heusdens, J.Jensen ‘A Short-Time Objective Intelligibility Measure for Time-Frequency Weighted Noisy Speech’, ICASSP 2010, Texas, Dallas.

[3] C.H.Taal, R.C.Hendriks, R.Heusdens, J.Jensen ‘An Algorithm for Intelligibility Prediction of Time-Frequency Weighted Noisy Speech’, IEEE Transactions on Audio, Speech, and Language Processing, 2011.

[4] J. Jensen and C. H. Taal, ‘An Algorithm for Predicting the Intelligibility of Speech Masked by Modulated Noise Maskers’, IEEE Transactions on Audio, Speech and Language Processing, 2016.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes average STOI.

**Return type** `Tensor`

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

### 1.12.2 Functional Interface

`torchmetrics.functional.audio.stoi.short_time_objective_intelligibility(preds, target, fs, extended=False, keep_same_device=False)`

STOI (Short-Time Objective Intelligibility, see [2,3]), a wrapper for the pystoi package [1]. Note that input will be moved to *cpu* to perform the metric calculation.

Intelligibility measure which is highly correlated with the intelligibility of degraded speech signals, e.g., due to additive noise, single/multi-channel noise reduction, binary masking and vocoded speech as in CI simulations. The STOI-measure is intrusive, i.e., a function of the clean and degraded speech signals. STOI may be a good alternative to the speech intelligibility index (SII) or the speech transmission index (STI), when you are interested in the effect of nonlinear processing to noisy speech, e.g., noise reduction, binary masking algorithms, on speech intelligibility. Description taken from [Cees Taal's website](#).

---

**Note:** using this metrics requires you to have pystoi install. Either install as `pip install torchmetrics[audio]` or `pip install pystoi`

---

**Parameters**

- **preds** (`Tensor`) – shape `[..., time]`
- **target** (`Tensor`) – shape `[..., time]`
- **fs** (`int`) – sampling frequency (Hz)
- **extended** (`bool`) – whether to use the extended STOI described in [4]



- **keep\_same\_device** (`bool`) – whether to move the stoi value to the device of preds

**Return type** `Tensor`

**Returns** stoi value of shape [...]

**Raises** `ModuleNotFoundError` – If pystoi package is not installed

### Example

```
>>> from torchmetrics.functional.audio.stoi import short_time_objective_
    ↪intelligibility
>>> import torch
>>> g = torch.manual_seed(1)
>>> preds = torch.randn(8000)
>>> target = torch.randn(8000)
>>> short_time_objective_intelligibility(preds, target, 8000).float()
tensor(-0.0100)
```

### References

[1] <https://github.com/mpariente/pystoi>

[2] C.H.Taal, R.C.Hendriks, R.Heusdens, J.Jensen ‘A Short-Time Objective Intelligibility Measure for Time-Frequency Weighted Noisy Speech’, ICASSP 2010, Texas, Dallas.

[3] C.H.Taal, R.C.Hendriks, R.Heusdens, J.Jensen ‘An Algorithm for Intelligibility Prediction of Time-Frequency Weighted Noisy Speech’, IEEE Transactions on Audio, Speech, and Language Processing, 2011.

[4] J. Jensen and C. H. Taal, ‘An Algorithm for Predicting the Intelligibility of Speech Masked by Modulated Noise Maskers’, IEEE Transactions on Audio, Speech and Language Processing, 2016.

## 1.13 Signal to Distortion Ratio (SDR)

### 1.13.1 Module Interface

```
class torchmetrics.SignalDistortionRatio(use_cg_iter=None, filter_length=512, zero_mean=False,
                                         load_diag=None, **kwargs)
```

Signal to Distortion Ratio (SDR) [1,2]

Forward accepts

- preds: shape [..., time]
- target: shape [..., time]

#### Parameters

- **use\_cg\_iter** (`Optional[int]`) – If provided, conjugate gradient descent is used to solve for the distortion filter coefficients instead of direct Gaussian elimination, which requires that `fast-bss-eval` is installed and pytorch version  $\geq 1.8$ . This can speed up the computation of the metrics in case the filters are long. Using a value of 10 here has been shown to provide good accuracy in most cases and is sufficient when using this loss to train neural separation networks.

- **filter\_length** (`int`) – The length of the distortion filter allowed
- **zero\_mean** (`bool`) – When set to True, the mean of all signals is subtracted prior to computation of the metrics
- **load\_diag** (`Optional[float]`) – If provided, this small value is added to the diagonal coefficients of the system metrics when solving for the filter coefficients. This can help stabilize the metric in the case where some reference signals may sometimes be zero
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torchmetrics.audio import SignalDistortionRatio
>>> import torch
>>> g = torch.manual_seed(1)
>>> preds = torch.randn(8000)
>>> target = torch.randn(8000)
>>> sdr = SignalDistortionRatio()
>>> sdr(preds, target)
tensor(-12.0589)
>>> # use with pit
>>> from torchmetrics.audio import PermutationInvariantTraining
>>> from torchmetrics.functional.audio import signal_distortion_ratio
>>> preds = torch.randn(4, 2, 8000) # [batch, spk, time]
>>> target = torch.randn(4, 2, 8000)
>>> pit = PermutationInvariantTraining(signal_distortion_ratio, 'max')
>>> pit(preds, target)
tensor(-11.6051)
```

### References

[1] Vincent, E., Gribonval, R., & Fevotte, C. (2006). Performance measurement in blind audio source separation. *IEEE Transactions on Audio, Speech and Language Processing*, 14(4), 1462–1469.

[2] Scheibler, R. (2021). SDR – Medium Rare with Fast Computations.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes average SDR.

**Return type** `Tensor`

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

### 1.13.2 Functional Interface

`torchmetrics.functional.signal_distortion_ratio(preds, target, use_cg_iter=None, filter_length=512, zero_mean=False, load_diag=None)`

Signal to Distortion Ratio (SDR) [1,2]

#### Parameters

- **preds** (`Tensor`) – shape `[..., time]`
- **target** (`Tensor`) – shape `[..., time]`
- **use\_cg\_iter** (`Optional[int]`) – If provided, conjugate gradient descent is used to solve for the distortion filter coefficients instead of direct Gaussian elimination, which requires that `fast-bss-eval` is installed and `pytorch` version `>= 1.8`. This can speed up the computation of the metrics in case the filters are long. Using a value of 10 here has been shown to provide good accuracy in most cases and is sufficient when using this loss to train neural separation networks.
- **filter\_length** (`int`) – The length of the distortion filter allowed
- **zero\_mean** (`bool`) – When set to `True`, the mean of all signals is subtracted prior to computation of the metrics
- **load\_diag** (`Optional[float]`) – If provided, this small value is added to the diagonal coefficients of the system metrics when solving for the filter coefficients. This can help stabilize the metric in the case where some reference signals may sometimes be zero

**Return type** `Tensor`

**Returns** sdr value of shape `[...]`

#### Example

```
>>> from torchmetrics.functional.audio import signal_distortion_ratio
>>> import torch
>>> g = torch.manual_seed(1)
>>> preds = torch.randn(8000)
>>> target = torch.randn(8000)
>>> signal_distortion_ratio(preds, target)
tensor(-12.0589)
>>> # use with permutation_invariant_training
>>> from torchmetrics.functional.audio import permutation_invariant_training
>>> preds = torch.randn(4, 2, 8000) # [batch, spk, time]
>>> target = torch.randn(4, 2, 8000)
>>> best_metric, best_perm = permutation_invariant_training(preds, target, signal_
↳ distortion_ratio, 'max')
>>> best_metric
tensor([-11.6375, -11.4358, -11.7148, -11.6325])
>>> best_perm
tensor([[1, 0],
        [0, 1],
        [1, 0],
        [0, 1]])
```

## References

- [1] Vincent, E., Gribonval, R., & Fevotte, C. (2006). Performance measurement in blind audio source separation. *IEEE Transactions on Audio, Speech and Language Processing*, 14(4), 1462–1469.
- [2] Scheibler, R. (2021). SDR – Medium Rare with Fast Computations.

## 1.14 Signal-to-Noise Ratio (SNR)

### 1.14.1 Module Interface

**class** `torchmetrics.SignalNoiseRatio`(*zero\_mean=False*, *\*\*kwargs*)

Signal-to-noise ratio (SNR):

$$\text{SNR} = \frac{P_{\text{signal}}}{P_{\text{noise}}}$$

where  $P$  denotes the power of each signal. The SNR metric compares the level of the desired signal to the level of background noise. Therefore, a high value of SNR means that the audio is clear.

Forward accepts

- `preds`: shape [..., time]
- `target`: shape [..., time]

#### Parameters

- **`zero_mean`** (`bool`) – if to zero mean target and preds or not
- **`kwargs`** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises** `TypeError` – if target and preds have a different shape

**Returns** average snr value

### Example

```
>>> import torch
>>> from torchmetrics import SignalNoiseRatio
>>> target = torch.tensor([3.0, -0.5, 2.0, 7.0])
>>> preds = torch.tensor([2.5, 0.0, 2.0, 8.0])
>>> snr = SignalNoiseRatio()
>>> snr(preds, target)
tensor(16.1805)
```

## References

[1] Le Roux, Jonathan, et al. “SDR half-baked or well done.” IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) 2019.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes average SNR.

**Return type** `Tensor`

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

### 1.14.2 Functional Interface

`torchmetrics.functional.signal_noise_ratio(preds, target, zero_mean=False)`

Signal-to-noise ratio (SNR):

$$\text{SNR} = \frac{P_{\text{signal}}}{P_{\text{noise}}}$$

where  $P$  denotes the power of each signal. The SNR metric compares the level of the desired signal to the level of background noise. Therefore, a high value of SNR means that the audio is clear.

**Parameters**

- **preds** (`Tensor`) – shape `[..., time]`
- **target** (`Tensor`) – shape `[..., time]`
- **zero\_mean** (`bool`) – if to zero mean target and preds or not

**Return type** `Tensor`

**Returns** snr value of shape `[...]`

#### Example

```
>>> from torchmetrics.functional.audio import signal_noise_ratio
>>> target = torch.tensor([3.0, -0.5, 2.0, 7.0])
>>> preds = torch.tensor([2.5, 0.0, 2.0, 8.0])
>>> signal_noise_ratio(preds, target)
tensor(16.1805)
```

## References

[1] Le Roux, Jonathan, et al. “SDR half-baked or well done.” IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) 2019.

## 1.15 Accuracy

### 1.15.1 Module Interface

```
class torchmetrics.Accuracy(threshold=0.5, num_classes=None, average='micro', mdmc_average=None,
                             ignore_index=None, top_k=None, multiclass=None, subset_accuracy=False,
                             task=None, num_labels=None, multidim_average='global', validate_args=True,
                             **kwargs)
```

---

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

---

Computes *Accuracy*:

$$\text{Accuracy} = \frac{1}{N} \sum_i^N 1(y_i = \hat{y}_i)$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

For multi-class and multi-dimensional multi-class data with probability or logits predictions, the parameter `top_k` generalizes this metric to a Top-K accuracy metric: for each sample the top-K highest probability or logit score items are considered to find the correct label.

For multi-label and multi-dimensional multi-class inputs, this metric computes the “global” accuracy by default, which counts all labels or sub-samples separately. This can be changed to subset accuracy (which requires all labels or sub-samples in the sample to be correctly predicted) by setting `subset_accuracy=True`.

Accepts all input types listed in *Input types*.

#### Parameters

- **num\_classes** (`Optional[int]`) – Number of classes. Necessary for 'macro', 'weighted' and None average methods.
- **threshold** (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied. Should be one of the following:
  - 'micro' [default]: Calculate the metric globally, across all samples and classes.
  - 'macro': Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
  - 'weighted': Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (`tp + fn`).

- 'none' or None: Calculate the metric for each class separately, and return the metric for every class.
- 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

---

**Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_average`.

---



---

**Note:** If 'none' and a given class doesn't occur in the `preds` or `target`, the value for the class will be nan.

---

- **mdmc\_average** (Optional[str]) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
  - None [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - 'samplewise': In this case, the statistics are computed separately for each sample on the N axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... (see [Input types](#)) as the N dimension within the sample, and computing the metric for the sample based on that.
  - 'global': In this case the N and ... dimensions of the inputs (see [Input types](#)) are flattened into a new N\_X sample axis, i.e. the inputs are treated as if they were (N\_X, C). From here on the `average` parameter applies as usual.
- **ignore\_index** (Optional[int]) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or 'none', the score for the ignored class will be returned as nan.
- **top\_k** (Optional[int]) – Number of the highest probability or logit score predictions considered finding the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (None) will be interpreted as 1 for these inputs.  
Should be left at default (None) for all other types of inputs.
- **multiclass** (Optional[bool]) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's [documentation section](#) for a more detailed explanation and examples.
- **subset\_accuracy** (bool) – Whether to compute subset accuracy for multi-label and multi-dimensional multi-class inputs (has no effect for other input types).
  - For multi-label inputs, if the parameter is set to True, then all labels for each sample must be correctly predicted for the sample to count as correct. If it is set to False, then all labels are counted separately - this is equivalent to flattening inputs beforehand (i.e. `preds = preds.flatten()` and same for `target`).
  - For multi-dimensional multi-class inputs, if the parameter is set to True, then all sub-sample (on the extra axis) must be correct for the sample to be counted as correct. If it is set to False, then all sub-samples are counted separately - this is equivalent, in the case of label predictions, to flattening the inputs beforehand (i.e. `preds = preds.flatten()` and same for `target`). Note that the `top_k` parameter still applies in both cases, if set.
- **kwargs** (Any) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Raises

- **ValueError** – If `top_k` is not an integer larger than 0.
- **ValueError** – If `average` is none of "micro", "macro", "weighted", "samples", "none", None.
- **ValueError** – If two different input modes are provided, eg. using multi-label with multi-class.
- **ValueError** – If `top_k` parameter is set for multi-label inputs.

### Example

```
>>> import torch
>>> from torchmetrics import Accuracy
>>> target = torch.tensor([0, 1, 2, 3])
>>> preds = torch.tensor([0, 2, 1, 3])
>>> accuracy = Accuracy()
>>> accuracy(preds, target)
tensor(0.5000)
```

```
>>> target = torch.tensor([0, 1, 2])
>>> preds = torch.tensor([[0.1, 0.9, 0], [0.3, 0.1, 0.6], [0.2, 0.5, 0.3]])
>>> accuracy = Accuracy(top_k=2)
>>> accuracy(preds, target)
tensor(0.6667)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes accuracy based on inputs passed in to `update` previously.

**Return type** `Tensor`

**update(preds, target)**

Update state with predictions and targets. See [Input types](#) for more information on input types.

**Parameters**

- **preds** (`Tensor`) – Predictions from model (logits, probabilities, or labels)
- **target** (`Tensor`) – Ground truth labels

**Return type** `None`

### BinaryAccuracy

```
class torchmetrics.classification.BinaryAccuracy(threshold=0.5, multidim_average='global',
                                                  ignore_index=None, validate_args=True,
                                                  **kwargs)
```

Computes *Accuracy* for binary tasks:

$$\text{Accuracy} = \frac{1}{N} \sum_i^N 1(y_i = \hat{y}_i)$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

Accepts the following input tensors:



- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

#### Parameters

- **threshold** (*float*) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (*Literal*['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - *global*: Additional dimensions are flattened along the batch dimension
  - *samplewise*: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (*Optional*[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (*bool*) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

**Returns** If *multidim\_average* is set to *global*, the metric returns a scalar value. If *multidim\_average* is set to *samplewise*, the metric returns (N,) vector consisting of a scalar value per sample.

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import BinaryAccuracy
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0, 0, 1, 1, 0, 1])
>>> metric = BinaryAccuracy()
>>> metric(preds, target)
tensor(0.6667)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import BinaryAccuracy
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> metric = BinaryAccuracy()
>>> metric(preds, target)
tensor(0.6667)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import BinaryAccuracy
>>> target = torch.tensor([[[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> metric = BinaryAccuracy(multidim_average='samplewise')
```

(continues on next page)

(continued from previous page)

```
>>> metric(preds, target)
tensor([0.3333, 0.1667])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes the final statistics.

**Return type** `Tensor`

**Returns**

The metric returns a tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to `[tp, fp, tn, fn, sup]` (sup stands for support and equals `tp + fn`). The shape depends on the `multidim_average` parameter:

- If `multidim_average` is set to `global`, the shape will be  $(5,)$
- If `multidim_average` is set to `samplewise`, the shape will be  $(N, 5)$

## MulticlassAccuracy

```
class torchmetrics.classification.MulticlassAccuracy(num_classes, top_k=1, average='macro',
                                                    multidim_average='global',
                                                    ignore_index=None, validate_args=True,
                                                    **kwargs)
```

Computes *Accuracy* for multiclass tasks:

$$\text{Accuracy} = \frac{1}{N} \sum_i^N 1(y_i = \hat{y}_i)$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

Accepts the following input tensors:

- `preds`:  $(N, \dots)$  (int tensor) or  $(N, C, \dots)$  (float tensor). If `preds` is a floating point we apply `torch.argmax` along the `C` dimension to automatically convert probabilities/logits into an int tensor.
- `target` (int tensor):  $(N, \dots)$

The influence of the additional dimension  $\dots$  (if present) will be determined by the `multidim_average` argument.

**Parameters**

- **num\_classes** (`int`) – Integer specifying the number of classes
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: Calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: Calculates statistic for each label and applies no reduction
- **top\_k** (`int`) – Number of highest probability or logit score predictions considered to find the correct label. Only works when `preds` contain probabilities/logits.

- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the `N` axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

#### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be `(C,)`
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

**Return type** The returned shape depends on the `average` and `multidim_average` arguments

#### Example (preds is int tensor):

```
>>> from torchmetrics.classification import MulticlassAccuracy
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> metric = MulticlassAccuracy(num_classes=3)
>>> metric(preds, target)
tensor(0.8333)
>>> metric = MulticlassAccuracy(num_classes=3, average=None)
>>> metric(preds, target)
tensor([0.5000, 1.0000, 1.0000])
```

#### Example (preds is float tensor):

```
>>> from torchmetrics.classification import MulticlassAccuracy
>>> target = target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> metric = MulticlassAccuracy(num_classes=3)
>>> metric(preds, target)
tensor(0.8333)
>>> metric = MulticlassAccuracy(num_classes=3, average=None)
>>> metric(preds, target)
tensor([0.5000, 1.0000, 1.0000])
```

#### Example (multidim tensors):

```
>>> from torchmetrics.classification import MulticlassAccuracy
>>> target = torch.tensor([[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]])
>>> preds = torch.tensor([[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]])
>>> metric = MulticlassAccuracy(num_classes=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.5000, 0.2778])
>>> metric = MulticlassAccuracy(num_classes=3, multidim_average='samplewise',
    ↪ average=None)
>>> metric(preds, target)
tensor([[1.0000, 0.0000, 0.5000],
        [0.0000, 0.3333, 0.5000]])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes the final statistics.

**Return type** `Tensor`

**Returns**

The metric returns a tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to  $[tp, fp, tn, fn, sup]$  ( $sup$  stands for support and equals  $tp + fn$ ). The shape depends on `average` and `multidim_average` parameters:

- If `multidim_average` is set to `global`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(5,)$
- If `average=None/'none'`, the shape will be  $(C, 5)$
- If `multidim_average` is set to `samplewise`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(N, 5)$
- If `average=None/'none'`, the shape will be  $(N, C, 5)$

## MultilabelAccuracy

```
class torchmetrics.classification.MultilabelAccuracy(num_labels, threshold=0.5, average='macro',
    multidim_average='global',
    ignore_index=None, validate_args=True,
    **kwargs)
```

Computes *Accuracy* for multilabel tasks:

$$\text{Accuracy} = \frac{1}{N} \sum_i^N 1(y_i = \hat{y}_i)$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

Accepts the following input tensors:

- **preds** (int or float tensor):  $(N, C, \dots)$ . If `preds` is a floating point tensor with values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **target** (int tensor):  $(N, C, \dots)$

The influence of the additional dimension  $\dots$  (if present) will be determined by the `multidim_average` argument.

### Parameters

- **num\_labels** (`int`) – Integer specifying the number of labels
- **threshold** (`float`) – Threshold for transforming probability to binary (0,1) predictions
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: Calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: Calculates statistic for each label and applies no reduction
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the `N` axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be `(C,)`
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

**Return type** The returned shape depends on the `average` and `multidim_average` arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import MultilabelAccuracy
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelAccuracy(num_labels=3)
>>> metric(preds, target)
tensor(0.6667)
>>> metric = MultilabelAccuracy(num_labels=3, average=None)
>>> metric(preds, target)
tensor([1.0000, 0.5000, 0.5000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import MultilabelAccuracy
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelAccuracy(num_labels=3)
>>> metric(preds, target)
tensor(0.6667)
>>> metric = MultilabelAccuracy(num_labels=3, average=None)
>>> metric(preds, target)
tensor([1.0000, 0.5000, 0.5000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import MultilabelAccuracy
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> metric = MultilabelAccuracy(num_labels=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.3333, 0.1667])
>>> metric = MultilabelAccuracy(num_labels=3, multidim_average='samplewise',
...                             average=None)
>>> metric(preds, target)
tensor([[0.5000, 0.5000, 0.0000],
        [0.0000, 0.0000, 0.5000]])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes the final statistics.

**Return type** `Tensor`

**Returns**

The metric returns a tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to  $[tp, fp, tn, fn, sup]$  ( $sup$  stands for support and equals  $tp + fn$ ). The shape depends on `average` and `multidim_average` parameters:

- If `multidim_average` is set to `global`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(5,)$
- If `average=None/'none'`, the shape will be  $(C, 5)$
- If `multidim_average` is set to `samplewise`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(N, 5)$
- If `average=None/'none'`, the shape will be  $(N, C, 5)$

## 1.15.2 Functional Interface

```
torchmetrics.functional.classification.accuracy(preds, target, average='micro',
                                                mdmc_average='global', threshold=0.5, top_k=None,
                                                subset_accuracy=False, num_classes=None,
                                                multiclass=None, ignore_index=None, task=None,
                                                num_labels=None, multidim_average='global',
                                                validate_args=True)
```

---

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

---

Computes *Accuracy*

$$\text{Accuracy} = \frac{1}{N} \sum_i^N 1(y_i = \hat{y}_i)$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

For multi-class and multi-dimensional multi-class data with probability or logits predictions, the parameter `top_k` generalizes this metric to a Top-K accuracy metric: for each sample the top-K highest probability or logits items are considered to find the correct label.

For multi-label and multi-dimensional multi-class inputs, this metric computes the “global” accuracy by default, which counts all labels or sub-samples separately. This can be changed to subset accuracy (which requires all labels or sub-samples in the sample to be correctly predicted) by setting `subset_accuracy=True`.

Accepts all input types listed in *Input types*.

### Parameters

- **preds** (*Tensor*) – Predictions from model (probabilities, logits or labels)
- **target** (*Tensor*) – Ground truth labels
- **average** (*Optional[Literal['micro', 'macro', 'weighted', 'none']]*) – Defines the reduction that is applied. Should be one of the following:
  - `'micro'` [default]: Calculate the metric globally, across all samples and classes.
  - `'macro'`: Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
  - `'weighted'`: Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (`tp + fn`).
  - `'none'` or `None`: Calculate the metric for each class separately, and return the metric for every class.
  - `'samples'`: Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

---

**Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_average`.

---

---

**Note:** If 'none' and a given class doesn't occur in the `preds` or `target`, the value for the class will be nan.

---

- **mdmc\_average** (`Optional[str]`) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
  - `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - `'samplewise'`: In this case, the statistics are computed separately for each sample on the `N` axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... (see [Input types](#)) as the `N` dimension within the sample, and computing the metric for the sample based on that.
  - `'global'`: In this case the `N` and ... dimensions of the inputs (see [Input types](#)) are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were `(N_X, C)`. From here on the `average` parameter applies as usual.
- **num\_classes** (`Optional[int]`) – Number of classes. Necessary for 'macro', 'weighted' and `None` average methods.
- **threshold** (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
- **top\_k** (`Optional[int]`) – Number of the highest probability or logit score predictions considered finding the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (`None`) will be interpreted as 1 for these inputs.  
Should be left at default (`None`) for all other types of inputs.
- **multiclass** (`Optional[bool]`) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's [documentation section](#) for a more detailed explanation and examples.
- **ignore\_index** (`Optional[int]`) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or `'none'`, the score for the ignored class will be returned as nan.
- **subset\_accuracy** (`bool`) – Whether to compute subset accuracy for multi-label and multi-dimensional multi-class inputs (has no effect for other input types).
  - For multi-label inputs, if the parameter is set to `True`, then all labels for each sample must be correctly predicted for the sample to count as correct. If it is set to `False`, then all labels are counted separately - this is equivalent to flattening inputs beforehand (i.e. `preds = preds.flatten()` and same for `target`).
  - For multi-dimensional multi-class inputs, if the parameter is set to `True`, then all sub-sample (on the extra axis) must be correct for the sample to be counted as correct. If it is set to `False`, then all sub-samples are counted separately - this is equivalent, in the case of label predictions, to flattening the inputs beforehand (i.e. `preds = preds.flatten()` and same for `target`). Note that the `top_k` parameter still applies in both cases, if set.

#### Raises

- **ValueError** – If `top_k` parameter is set for multi-label inputs.
- **ValueError** – If `average` is none of "micro", "macro", "weighted", "samples", "none", `None`.



- **ValueError** – If `mdmc_average` is not one of `None`, `"samplewise"`, `"global"`.
- **ValueError** – If `average` is set but `num_classes` is not provided.
- **ValueError** – If `num_classes` is set and `ignore_index` is not in the range `[0, num_classes)`.
- **ValueError** – If `top_k` is not an integer larger than 0.

### Example

```
>>> import torch
>>> from torchmetrics.functional import accuracy
>>> target = torch.tensor([0, 1, 2, 3])
>>> preds = torch.tensor([0, 2, 1, 3])
>>> accuracy(preds, target)
tensor(0.5000)
```

```
>>> target = torch.tensor([0, 1, 2])
>>> preds = torch.tensor([[0.1, 0.9, 0], [0.3, 0.1, 0.6], [0.2, 0.5, 0.3]])
>>> accuracy(preds, target, top_k=2)
tensor(0.6667)
```

Return type `Tensor`

### binary\_accuracy

`torchmetrics.functional.classification.binary_accuracy(preds, target, threshold=0.5, multidim_average='global', ignore_index=None, validate_args=True)`

Computes *Accuracy* for binary tasks:

$$\text{Accuracy} = \frac{1}{N} \sum_i^N 1(y_i = \hat{y}_i)$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If `preds` is a floating point tensor with values outside `[0,1]` range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the `multidim_average` argument.

#### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **threshold** (`float`) – Threshold for transforming probability to binary `{0,1}` predictions
- **multidim\_average** (`Literal``[``'global'``,``'samplewise'``]`) – Defines how additionally dimensions ... should be handled. Should be one of the following:

- **global**: Additional dimensions are flattened along the batch dimension
- **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Return type** `Tensor`

**Returns** If `multidim_average` is set to `global`, the metric returns a scalar value. If `multidim_average` is set to `samplewise`, the metric returns (N,) vector consisting of a scalar value per sample.

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import binary_accuracy
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0, 0, 1, 1, 0, 1])
>>> binary_accuracy(preds, target)
tensor(0.6667)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import binary_accuracy
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> binary_accuracy(preds, target)
tensor(0.6667)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import binary_accuracy
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> binary_accuracy(preds, target, multidim_average='samplewise')
tensor([0.3333, 0.1667])
```

## `multiclass_accuracy`

`torchmetrics.functional.classification.multiclass_accuracy`(*preds, target, num\_classes, average='macro', top\_k=1, multidim\_average='global', ignore\_index=None, validate\_args=True*)

Computes *Accuracy* for multiclass tasks:

$$\text{Accuracy} = \frac{1}{N} \sum_i^N 1(y_i = \hat{y}_i)$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

#### Parameters

- **preds** (**Tensor**) – Tensor with predictions
- **target** (**Tensor**) – Tensor with true labels
- **num\_classes** (**int**) – Integer specifying the number of classes
- **average** (**Optional**[**Literal**['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: Calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: Calculates statistic for each label and applies no reduction
- **top\_k** (**int**) – Number of highest probability or logit score predictions considered to find the correct label. Only works when preds contain probabilities/logits.
- **multidim\_average** (**Literal**['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (**Optional**[**int**]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (**bool**) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

#### Returns

- If **multidim\_average** is set to **global**:
  - If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor
  - If **average**=None/'none', the shape will be (C,)
- If **multidim\_average** is set to **samplewise**:
  - If **average**='micro'/'macro'/'weighted', the shape will be (N,)
  - If **average**=None/'none', the shape will be (N, C)

**Return type** The returned shape depends on the `average` and `multidim_average` arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_accuracy
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> multiclass_accuracy(preds, target, num_classes=3)
tensor(0.8333)
>>> multiclass_accuracy(preds, target, num_classes=3, average=None)
tensor([0.5000, 1.0000, 1.0000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_accuracy
>>> target = target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> multiclass_accuracy(preds, target, num_classes=3)
tensor(0.8333)
>>> multiclass_accuracy(preds, target, num_classes=3, average=None)
tensor([0.5000, 1.0000, 1.0000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import multiclass_accuracy
>>> target = torch.tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = torch.tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> multiclass_accuracy(preds, target, num_classes=3, multidim_average=
↳ 'samplewise')
tensor([0.5000, 0.2778])
>>> multiclass_accuracy(preds, target, num_classes=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[1.0000, 0.0000, 0.5000],
        [0.0000, 0.3333, 0.5000]])
```

## multilabel\_accuracy

`torchmetrics.functional.classification.multilabel_accuracy(preds, target, num_labels, threshold=0.5, average='macro', multidim_average='global', ignore_index=None, validate_args=True)`

Computes *Accuracy* for multilabel tasks:

$$\text{Accuracy} = \frac{1}{N} \sum_i^N 1(y_i = \hat{y}_i)$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

#### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: Calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: Calculates statistic for each label and applies no reduction
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

#### Returns

- If **multidim\_average** is set to **global**:
  - If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor
  - If **average**=None/'none', the shape will be (C,)
- If **multidim\_average** is set to **samplewise**:
  - If **average**='micro'/'macro'/'weighted', the shape will be (N,)
  - If **average**=None/'none', the shape will be (N, C)

**Return type** The returned shape depends on the **average** and **multidim\_average** arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_accuracy
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_accuracy(preds, target, num_labels=3)
tensor(0.6667)
>>> multilabel_accuracy(preds, target, num_labels=3, average=None)
tensor([1.0000, 0.5000, 0.5000])
```

Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import multilabel_accuracy
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_accuracy(preds, target, num_labels=3)
tensor(0.6667)
>>> multilabel_accuracy(preds, target, num_labels=3, average=None)
tensor([1.0000, 0.5000, 0.5000])
```

Example (multidim tensors):

```
>>> from torchmetrics.functional.classification import multilabel_accuracy
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> multilabel_accuracy(preds, target, num_labels=3, multidim_average=
↳ 'samplewise')
tensor([0.3333, 0.1667])
>>> multilabel_accuracy(preds, target, num_labels=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[0.5000, 0.5000, 0.0000],
        [0.0000, 0.0000, 0.5000]])
```

## 1.16 AUC

### 1.16.1 Module Interface

**class** torchmetrics.AUC(*reorder=False*, *\*\*kwargs*)

Computes Area Under the Curve (AUC) using the trapezoidal rule

Forward accepts two input tensors that should be 1D and have the same number of elements

---

**Note:** This metric has been deprecated in v0.10 and will be removed in v0.11.

---

#### Parameters

- **reorder** (*bool*) – AUC expects its first input to be sorted. If this is not the case, setting this argument to True will use a stable sorting algorithm to sort the input in descending order

- **kwargs** (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes AUC based on inputs passed in to update previously.

**Return type** *Tensor*

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (*Tensor*) – Predictions from model (probabilities, or labels)
- **target** (*Tensor*) – Ground truth labels

**Return type** *None*

## 1.16.2 Functional Interface

`torchmetrics.functional.auc(x, y, reorder=False)`

Computes Area Under the Curve (AUC) using the trapezoidal rule.

---

**Note:** This metric have been moved to `torchmetrics.utilities.compute` in v0.10 this version will be removed in v0.11.

---

**Parameters**

- **x** (*Tensor*) – x-coordinates, must be either increasing or decreasing
- **y** (*Tensor*) – y-coordinates
- **reorder** (*bool*) – if True, will reorder the arrays to make it either increasing or decreasing

**Return type** *Tensor*

**Returns** Tensor containing AUC score

**Raises**

- **ValueError** – If both x and y tensors are not 1d.
- **ValueError** – If both x and y don't have the same numnber of elements.
- **ValueError** – If x tesnsor is neither increasing nor decreasing.

### Example

```
>>> from torchmetrics.functional import auc
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> auc(x, y)
tensor(4.)
>>> auc(x, y, reorder=True)
tensor(4.)
```

## 1.17 AUROC

### 1.17.1 Module Interface

```
class torchmetrics.AUROC(num_classes=None, pos_label=None, average='macro', max_fpr=None,
                        task=None, thresholds=None, num_labels=None, ignore_index=None,
                        validate_args=True, **kwargs)
```

---

**Note:** From v0.10 an `'binary_*`, `'multiclass_*`, `'multilabel_*` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

---

Compute Area Under the Receiver Operating Characteristic Curve (**ROC AUC**). Works for both binary, multilabel and multiclass problems. In the case of multiclass, the values will be calculated based on a one-vs-the-rest approach.

Forward accepts

- **preds** (float tensor): (N, ...) (binary) or (N, C, ...) (multiclass) tensor with probabilities, where C is the number of classes.
- **target** (long tensor): (N, ...) or (N, C, ...) with integer labels

For non-binary input, if the **preds** and **target** tensor have the same size the input will be interpreted as multilabel and if **preds** have one dimension more than the **target** tensor the input will be interpreted as multiclass.

---

**Note:** If either the positive class or negative class is completely missing in the target tensor, the auroc score is meaningless in this case and a score of 0 will be returned together with an warning.

---

#### Parameters

- **num\_classes** (`Optional[int]`) – integer with number of classes for multi-label and multi-class problems.  
Should be set to `None` for binary problems
- **pos\_label** (`Optional[int]`) – integer determining the positive class. Default is `None` which for binary problem is translated to 1. For multiclass problems this argument should not be set as we iteratively change it in the range `[0, num_classes-1]`
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) –
  - `'micro'` computes metric globally. Only works for multilabel problems
  - `'macro'` computes metric for each class and uniformly averages them
  - `'weighted'` computes metric for each class and does a weighted-average, where each class is weighted by their support (accounts for class imbalance)
  - `None` computes and returns the metric per class
- **max\_fpr** (`Optional[float]`) – If not `None`, calculates standardized partial AUC over the range `[0, max_fpr]`. Should be a float between 0 and 1.



- **kwargs** (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

#### Raises

- **ValueError** – If average is none of None, "macro" or "weighted".
- **ValueError** – If max\_fpr is not a float in the range (0, 1].
- **RuntimeError** – If PyTorch version is below 1.6 since max\_fpr requires torch.bucketize which is not available below 1.6.
- **ValueError** – If the mode of data (binary, multi-label, multi-class) changes between batches.

#### Example (binary case):

```
>>> from torchmetrics import AUROC
>>> preds = torch.tensor([0.13, 0.26, 0.08, 0.19, 0.34])
>>> target = torch.tensor([0, 0, 1, 1, 1])
>>> auroc = AUROC(pos_label=1)
>>> auroc(preds, target)
tensor(0.5000)
```

#### Example (multiclass case):

```
>>> preds = torch.tensor([[0.90, 0.05, 0.05],
...                       [0.05, 0.90, 0.05],
...                       [0.05, 0.05, 0.90],
...                       [0.85, 0.05, 0.10],
...                       [0.10, 0.10, 0.80]])
>>> target = torch.tensor([0, 1, 1, 2, 2])
>>> auroc = AUROC(num_classes=3)
>>> auroc(preds, target)
tensor(0.7778)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### compute()

Computes AUROC based on inputs passed in to update previously.

**Return type** *Tensor*

#### update(preds, target)

Update state with predictions and targets.

#### Parameters

- **preds** (*Tensor*) – Predictions from model (probabilities, or labels)
- **target** (*Tensor*) – Ground truth labels

**Return type** *None*

## BinaryAUROC

```
class torchmetrics.classification.BinaryAUROC(max_fpr=None, thresholds=None, ignore_index=None,
                                              validate_args=True, **kwargs)
```

Compute Area Under the Receiver Operating Characteristic Curve ([ROC AUC](#)) for binary tasks. The AUROC score summarizes the ROC curve into an single number that describes the performance of a model for multiple thresholds at the same time. Notably, an AUROC score of 1 is a perfect score and an AUROC score of 0.5 corresponds to random guessing.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

### Parameters

- **max\_fpr** ([Optional\[float\]](#)) – If not *None*, calculates standardized partial AUC over the range [0, max\_fpr].
- **thresholds** ([Union\[int, List\[float\], Tensor, None\]](#)) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.
- **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Returns** A single scalar with the auroc score

### Example

```
>>> from torchmetrics.classification import BinaryAUROC
>>> preds = torch.tensor([0, 0.5, 0.7, 0.8])
>>> target = torch.tensor([0, 1, 1, 0])
>>> metric = BinaryAUROC(thresholds=None)
>>> metric(preds, target)
tensor(0.5000)
>>> metric = BinaryAUROC(thresholds=5)
>>> metric(preds, target)
tensor(0.5000)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### compute()

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tensor`

### MulticlassAUROC

```
class torchmetrics.classification.MulticlassAUROC(num_classes, average='macro', thresholds=None,
                                                  ignore_index=None, validate_args=True,
                                                  **kwargs)
```

Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) for multiclass tasks. The AUROC score summarizes the ROC curve into an single number that describes the performance of a model for multiple thresholds at the same time. Notably, an AUROC score of 1 is a perfect score and an AUROC score of 0.5 corresponds to random guessing.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the [0, n\_classes-1] range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

#### Parameters

- **num\_classes** (`int`) – Integer specifying the number of classes
- **average** (`Optional[Literal['macro', 'weighted', 'none']]`) – Defines the reduction that is applied over classes. Should be one of the following:
  - **macro**: Calculate score for each class and average them
  - **weighted**: Calculates score for each class and computes weighted average using their support

- "none" or None: Calculates score for each class and applies no reduction
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Returns** If *average=None*|"none" then a 1d tensor of shape (n\_classes,) will be returned with auroc score per class. If *average="macro"*|"weighted" then a single scalar is returned.

### Example

```
>>> from torchmetrics.classification import MulticlassAUROC
>>> preds = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                       [0.05, 0.75, 0.05, 0.05, 0.05],
...                       [0.05, 0.05, 0.75, 0.05, 0.05],
...                       [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> metric = MulticlassAUROC(num_classes=5, average="macro", thresholds=None)
>>> metric(preds, target)
tensor(0.5333)
>>> metric = MulticlassAUROC(num_classes=5, average=None, thresholds=None)
>>> metric(preds, target)
tensor([1.0000, 1.0000, 0.3333, 0.3333, 0.0000])
>>> metric = MulticlassAUROC(num_classes=5, average="macro", thresholds=5)
>>> metric(preds, target)
tensor(0.5333)
>>> metric = MulticlassAUROC(num_classes=5, average=None, thresholds=5)
>>> metric(preds, target)
tensor([1.0000, 1.0000, 0.3333, 0.3333, 0.0000])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

### compute()

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tensor`

## MultilabelAUROC

```
class torchmetrics.classification.MultilabelAUROC(num_labels, average='macro', thresholds=None,
                                                    ignore_index=None, validate_args=True,
                                                    **kwargs)
```

Compute Area Under the Receiver Operating Characteristic Curve (**ROC AUC**) for multilabel tasks. The AUROC score summarizes the ROC curve into an single number that describes the performance of a model for multiple thresholds at the same time. Notably, an AUROC score of 1 is a perfect score and an AUROC score of 0.5 corresponds to random guessing.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

### Parameters

- **num\_labels** (**int**) – Integer specifying the number of labels
- **average** (**Optional**[**Literal**['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum score over all labels
  - **macro**: Calculate score for each label and average them
  - **weighted**: Calculates score for each label and computes weighted average using their support
  - **"none"** or **None**: Calculates score for each label and applies no reduction
- **thresholds** (**Union**[**int**, **List**[**float**], **Tensor**, **None**]) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (**bool**) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.
- **kwargs** (**Any**) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Returns** If *average=**None**|**"none"* then a 1d tensor of shape (n\_classes,) will be returned with auroc score per class. If *average=**"micro|macro"**|**"weighted"* then a single scalar is returned.

### Example

```
>>> from torchmetrics.classification import MultilabelAUROC
>>> preds = torch.tensor([[0.75, 0.05, 0.35],
...                       [0.45, 0.75, 0.05],
...                       [0.05, 0.55, 0.75],
...                       [0.05, 0.65, 0.05]])
>>> target = torch.tensor([[1, 0, 1],
...                        [0, 0, 0],
...                        [0, 1, 1],
...                        [1, 1, 1]])
>>> metric = MultilabelAUROC(num_labels=3, average="macro", thresholds=None)
>>> metric(preds, target)
tensor(0.6528)
>>> metric = MultilabelAUROC(num_labels=3, average=None, thresholds=None)
>>> metric(preds, target)
tensor([0.6250, 0.5000, 0.8333])
>>> metric = MultilabelAUROC(num_labels=3, average="macro", thresholds=5)
>>> metric(preds, target)
tensor(0.6528)
>>> metric = MultilabelAUROC(num_labels=3, average=None, thresholds=5)
>>> metric(preds, target)
tensor([0.6250, 0.5000, 0.8333])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### `compute()`

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tensor`

## 1.17.2 Functional Interface

`torchmetrics.functional.auroc`(*preds*, *target*, *num\_classes=None*, *pos\_label=None*, *average='macro'*, *max\_fpr=None*, *sample\_weights=None*, *task=None*, *thresholds=None*, *num\_labels=None*, *ignore\_index=None*, *validate\_args=True*)

---

**Note:** From v0.10 an `'binary_*`, `'multiclass_*`, `'multilabel_*` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

---

Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC)

For non-binary input, if the `preds` and `target` tensor have the same size the input will be interpreted as multilabel and if `preds` have one dimension more than the `target` tensor the input will be interpreted as multiclass.

---

**Note:** If either the positive class or negative class is completely missing in the target tensor, the auroc score is

meaningless in this case and a score of 0 will be returned together with a warning.

### Parameters

- **preds** (`Tensor`) – predictions from model (logits or probabilities)
- **target** (`Tensor`) – Ground truth labels
- **num\_classes** (`Optional[int]`) – integer with number of classes for multi-label and multi-class problems. Should be set to `None` for binary problems
- **pos\_label** (`Optional[int]`) – integer determining the positive class. Default is `None` which for binary problem is translate to 1. For multiclass problems this argument should not be set as we iteratively change it in the range `[0,num_classes-1]`
- **average** (`Optional[Literal['macro', 'weighted', 'none']]`) –
  - 'macro' computes metric for each class and uniformly averages them
  - 'weighted' computes metric for each class and does a weighted-average, where each class is weighted by their support (accounts for class imbalance)
  - `None` computes and returns the metric per class
- **max\_fpr** (`Optional[float]`) – If not `None`, calculates standardized partial AUC over the range `[0, max_fpr]`. Should be a float between 0 and 1.
- **sample\_weights** (`Optional[Sequence]`) – sample weights for each data point

### Raises

- **ValueError** – If `max_fpr` is not a float in the range `(0, 1]`.
- **RuntimeError** – If PyTorch version is below 1.6 since `max_fpr` requires `torch.bucketize` which is not available below 1.6.
- **ValueError** – If `max_fpr` is not set to `None` and the mode is not `binary` since partial AUC computation is not available in `multilabel/multiclass`.
- **ValueError** – If `average` is none of `None`, "macro" or "weighted".

### Example (binary case):

```
>>> from torchmetrics.functional import auroc
>>> preds = torch.tensor([0.13, 0.26, 0.08, 0.19, 0.34])
>>> target = torch.tensor([0, 0, 1, 1, 1])
>>> auroc(preds, target, pos_label=1)
tensor(0.5000)
```

### Example (multiclass case):

```
>>> preds = torch.tensor([[0.90, 0.05, 0.05],
...                       [0.05, 0.90, 0.05],
...                       [0.05, 0.05, 0.90],
...                       [0.85, 0.05, 0.10],
...                       [0.10, 0.10, 0.80]])
>>> target = torch.tensor([0, 1, 1, 2, 2])
>>> auroc(preds, target, num_classes=3)
tensor(0.7778)
```

**Return type** `Union[Tensor, Tuple[Tensor, Tensor, Tensor], Tuple[List[Tensor], List[Tensor], List[Tensor]]]`

## binary\_auroc

`torchmetrics.functional.classification.binary_auroc(preds, target, max_fpr=None, thresholds=None, ignore_index=None, validate_args=True)`

Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) for binary tasks. The AUROC score summarizes the ROC curve into a single number that describes the performance of a model for multiple thresholds at the same time. Notably, an AUROC score of 1 is a perfect score and an AUROC score of 0.5 corresponds to random guessing.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **max\_fpr** (`Optional[float]`) – If not *None*, calculates standardized partial AUC over the range [0, max\_fpr].
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

**Return type** `Tuple[Tensor, Tensor, Tensor]`

**Returns** A single scalar with the auroc score



### Example

```
>>> from torchmetrics.functional.classification import binary_auroc
>>> preds = torch.tensor([0, 0.5, 0.7, 0.8])
>>> target = torch.tensor([0, 1, 1, 0])
>>> binary_auroc(preds, target, thresholds=None)
tensor(0.5000)
>>> binary_auroc(preds, target, thresholds=5)
tensor(0.5000)
```

### multiclass\_auroc

```
torchmetrics.functional.classification.multiclass_auroc(preds, target, num_classes,
                                                         average='macro', thresholds=None,
                                                         ignore_index=None, validate_args=True)
```

Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) for multiclass tasks. The AUROC score summarizes the ROC curve into a single number that describes the performance of a model for multiple thresholds at the same time. Notably, an AUROC score of 1 is a perfect score and an AUROC score of 0.5 corresponds to random guessing.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the [0, n\_classes-1] range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

#### Parameters

- **preds** (Tensor) – Tensor with predictions
- **target** (Tensor) – Tensor with true labels
- **num\_classes** (int) – Integer specifying the number of classes
- **average** (Optional[Literal['macro', 'weighted', 'none']]) – Defines the reduction that is applied over classes. Should be one of the following:
  - **macro**: Calculate score for each class and average them
  - **weighted**: Calculates score for each class and computes weighted average using their support
  - **"none"** or **None**: Calculates score for each class and applies no reduction
- **thresholds** (Union[int, List[float], Tensor, None]) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.

- If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
- If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
- If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (*bool*) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

**Return type** *Tensor*

**Returns** If *average=None*|"none" then a 1d tensor of shape (n\_classes,) will be returned with auroc score per class. If *average="macro"*|"weighted" then a single scalar is returned.

### Example

```
>>> from torchmetrics.functional.classification import multiclass_auroc
>>> preds = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                       [0.05, 0.75, 0.05, 0.05, 0.05],
...                       [0.05, 0.05, 0.75, 0.05, 0.05],
...                       [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> multiclass_auroc(preds, target, num_classes=5, average="macro", thresholds=None)
tensor(0.5333)
>>> multiclass_auroc(preds, target, num_classes=5, average=None, thresholds=None)
tensor([1.0000, 1.0000, 0.3333, 0.3333, 0.0000])
>>> multiclass_auroc(preds, target, num_classes=5, average="macro", thresholds=5)
tensor(0.5333)
>>> multiclass_auroc(preds, target, num_classes=5, average=None, thresholds=5)
tensor([1.0000, 1.0000, 0.3333, 0.3333, 0.0000])
```

### multilabel\_auroc

`torchmetrics.functional.classification.multilabel_auroc(preds, target, num_labels, average='macro', thresholds=None, ignore_index=None, validate_args=True)`

Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) for multilabel tasks. The AUROC score summarizes the ROC curve into a single number that describes the performance of a model for multiple thresholds at the same time. Notably, an AUROC score of 1 is a perfect score and an AUROC score of 0.5 corresponds to random guessing.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the

non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **num\_labels** (`int`) – Integer specifying the number of labels
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum score over all labels
  - `macro`: Calculate score for each label and average them
  - `weighted`: Calculates score for each label and computes weighted average using their support
  - `"none"` or `None`: Calculates score for each label and applies no reduction
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an `int` (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an `list` of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d `tensor` of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Return type** `Union[Tuple[Tensor, Tensor, Tensor], Tuple[List[Tensor], List[Tensor], List[Tensor]]]`

**Returns** If *average*=`None`|*"none"* then a 1d tensor of shape (n\_classes,) will be returned with auroc score per class. If *average*=*"micro"*|*"macro"*|*"weighted"* then a single scalar is returned.

### Example

```
>>> from torchmetrics.functional.classification import multilabel_auroc
>>> preds = torch.tensor([[0.75, 0.05, 0.35],
...                       [0.45, 0.75, 0.05],
...                       [0.05, 0.55, 0.75],
...                       [0.05, 0.65, 0.05]])
>>> target = torch.tensor([[1, 0, 1],
...                         [0, 0, 0],
...                         [0, 1, 1],
...                         [1, 1, 1]])
>>> multilabel_auroc(preds, target, num_labels=3, average="macro", thresholds=None)
tensor(0.6528)
>>> multilabel_auroc(preds, target, num_labels=3, average=None, thresholds=None)
```

(continues on next page)

(continued from previous page)

```

tensor([0.6250, 0.5000, 0.8333])
>>> multilabel_auroc(preds, target, num_labels=3, average="macro", thresholds=5)
tensor(0.6528)
>>> multilabel_auroc(preds, target, num_labels=3, average=None, thresholds=5)
tensor([0.6250, 0.5000, 0.8333])

```

## 1.18 Average Precision

### 1.18.1 Module Interface

```

class torchmetrics.AveragePrecision(num_classes=None, pos_label=None, average='macro', task=None,
                                     thresholds=None, num_labels=None, ignore_index=None,
                                     validate_args=True, **kwargs)

```

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entypoint to calling the three specialized versions.

Computes the average precision score, which summarises the precision recall curve into one number. Works for both binary and multiclass problems. In the case of multiclass, the values will be calculated based on a one-vs-the-rest approach.

Forward accepts

- **preds** (float tensor): (N, ...) (binary) or (N, C, ...) (multiclass) tensor with probabilities, where C is the number of classes.
- **target** (long tensor): (N, ...) with integer labels

#### Parameters

- **num\_classes** (`Optional[int]`) – integer with number of classes. Not necessary to provide for binary problems.
- **pos\_label** (`Optional[int]`) – integer determining the positive class. Default is `None` which for binary problem is translated to 1. For multiclass problems this argument should not be set as we iteratively change it in the range `[0, num_classes-1]`
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – defines the reduction that is applied in the case of multiclass and multilabel input. Should be one of the following:
  - `'macro'` [default]: Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
  - `'micro'`: Calculate the metric globally, across all samples and classes. Cannot be used with multiclass input.
  - `'weighted'`: Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support.

- 'none' or None: Calculate the metric for each class separately, and return the metric for every class.
- **kwargs** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (binary case):**

```
>>> from torchmetrics import AveragePrecision
>>> pred = torch.tensor([0, 0.1, 0.8, 0.4])
>>> target = torch.tensor([0, 1, 1, 1])
>>> average_precision = AveragePrecision(pos_label=1)
>>> average_precision(pred, target)
tensor(1.)
```

**Example (multiclass case):**

```
>>> pred = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                      [0.05, 0.75, 0.05, 0.05, 0.05],
...                      [0.05, 0.05, 0.75, 0.05, 0.05],
...                      [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> average_precision = AveragePrecision(num_classes=5, average=None)
>>> average_precision(pred, target)
[tensor(1.), tensor(1.), tensor(0.2500), tensor(0.2500), tensor(nan)]
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Compute the average precision score.

**Return type** `Union[Tensor, List[Tensor]]`

**Returns** tensor with average precision. If multiclass return list of such tensors, one for each class

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

## BinaryAveragePrecision

**class** `torchmetrics.classification.BinaryAveragePrecision`(*thresholds=None, ignore\_index=None, validate\_args=True, \*\*kwargs*)

Computes the average precision (AP) score for binary tasks. The AP score summarizes a precision-recall curve as an weighted mean of precisions at each threshold, with the difference in recall from the previous threshold as weight:

$$AP = \sum n(R_n - R_{n-1})P_n$$

where  $P_n, R_n$  is the respective precision and recall at threshold index  $n$ . This value is equivalent to the area under the precision-recall curve (AUPRC).

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

#### Parameters

- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.
- **kwargs** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Returns** A single scalar with the average precision score

#### Example

```
>>> from torchmetrics.classification import BinaryAveragePrecision
>>> preds = torch.tensor([0, 0.5, 0.7, 0.8])
>>> target = torch.tensor([0, 1, 1, 0])
>>> metric = BinaryAveragePrecision(thresholds=None)
>>> metric(preds, target)
tensor(0.5833)
>>> metric = BinaryAveragePrecision(thresholds=5)
>>> metric(preds, target)
tensor(0.6667)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### compute()

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tensor`

## MulticlassAveragePrecision

```
class torchmetrics.classification.MulticlassAveragePrecision(num_classes, average='macro',
                                                            thresholds=None,
                                                            ignore_index=None,
                                                            validate_args=True, **kwargs)
```

Computes the average precision (AP) score for binary tasks. The AP score summarizes a precision-recall curve as an weighted mean of precisions at each threshold, with the difference in recall from the previous threshold as weight:

$$AP = \sum n(R_n - R_{n-1})P_n$$

where  $P_n, R_n$  is the respective precision and recall at threshold index  $n$ . This value is equivalent to the area under the precision-recall curve (AUPRC).

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the [0, n\_classes-1] range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

### Parameters

- **num\_classes** (`int`) – Integer specifying the number of classes
- **average** (`Optional[Literal['macro', 'weighted', 'none']]`) – Defines the reduction that is applied over classes. Should be one of the following:
  - *macro*: Calculate score for each class and average them
  - *weighted*: Calculates score for each class and computes weighted average using their support
  - *"none"* or *None*: Calculates score for each class and applies no reduction
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

- **kwargs** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Returns** If *average=None|"none"* then a 1d tensor of shape (n\_classes, ) will be returned with AP score per class. If *average="macro"|"weighted"* then a single scalar is returned.

### Example

```
>>> from torchmetrics.classification import MulticlassAveragePrecision
>>> preds = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                       [0.05, 0.75, 0.05, 0.05, 0.05],
...                       [0.05, 0.05, 0.75, 0.05, 0.05],
...                       [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> metric = MulticlassAveragePrecision(num_classes=5, average="macro",
↳ thresholds=None)
>>> metric(preds, target)
tensor(0.6250)
>>> metric = MulticlassAveragePrecision(num_classes=5, average=None,
↳ thresholds=None)
>>> metric(preds, target)
tensor([1.0000, 1.0000, 0.2500, 0.2500, nan])
>>> metric = MulticlassAveragePrecision(num_classes=5, average="macro",
↳ thresholds=5)
>>> metric(preds, target)
tensor(0.5000)
>>> metric = MulticlassAveragePrecision(num_classes=5, average=None, thresholds=5)
>>> metric(preds, target)
tensor([1.0000, 1.0000, 0.2500, 0.2500, -0.0000])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### compute()

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tensor`

### MultilabelAveragePrecision

```
class torchmetrics.classification.MultilabelAveragePrecision(num_labels, average='macro',
                                                             thresholds=None,
                                                             ignore_index=None,
                                                             validate_args=True, **kwargs)
```

Computes the average precision (AP) score for binary tasks. The AP score summarizes a precision-recall curve as an weighted mean of precisions at each threshold, with the difference in recall from the previous threshold as weight:

$$AP = \sum n(R_n - R_{n-1})P_n$$

where  $P_n, R_n$  is the respective precision and recall at threshold index  $n$ . This value is equivalent to the area under the precision-recall curve (AUPRC).

Accepts the following input tensors:



- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

#### Parameters

- **num\_labels** (`int`) – Integer specifying the number of labels
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum score over all labels
  - `macro`: Calculate score for each label and average them
  - `weighted`: Calculates score for each label and computes weighted average using their support
  - `"none"` or `None`: Calculates score for each label and applies no reduction
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Returns** If *average=*`None`*|*`"none"` then a 1d tensor of shape (n\_classes, ) will be returned with AP score per class. If *average=*`"micro|macro"`*|*`"weighted"` then a single scalar is returned.

### Example

```

>>> from torchmetrics.classification import MultilabelAveragePrecision
>>> preds = torch.tensor([[0.75, 0.05, 0.35],
...                       [0.45, 0.75, 0.05],
...                       [0.05, 0.55, 0.75],
...                       [0.05, 0.65, 0.05]])
>>> target = torch.tensor([[1, 0, 1],
...                        [0, 0, 0],
...                        [0, 1, 1],
...                        [1, 1, 1]])
>>> metric = MultilabelAveragePrecision(num_labels=3, average="macro",
↳ thresholds=None)
>>> metric(preds, target)
tensor(0.7500)
>>> metric = MultilabelAveragePrecision(num_labels=3, average=None, thresholds=None)
>>> metric(preds, target)
tensor([0.7500, 0.5833, 0.9167])
>>> metric = MultilabelAveragePrecision(num_labels=3, average="macro", thresholds=5)
>>> metric(preds, target)
tensor(0.7778)
>>> metric = MultilabelAveragePrecision(num_labels=3, average=None, thresholds=5)
>>> metric(preds, target)
tensor([0.7500, 0.6667, 0.9167])

```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### compute()

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tensor`

## 1.18.2 Functional Interface

`torchmetrics.functional.average_precision(preds, target, num_classes=None, pos_label=None, average='macro', task=None, thresholds=None, num_labels=None, ignore_index=None, validate_args=True)`

**Note:** From v0.10 an `'binary_*`, `'multiclass_*`, `'multilabel_*` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

Computes the average precision score.

#### Parameters

- **preds** (`Tensor`) – predictions from model (logs or probabilities)
- **target** (`Tensor`) – ground truth values

- **num\_classes** (`Optional[int]`) – integer with number of classes. Not necessary to provide for binary problems.
- **pos\_label** (`Optional[int]`) – integer determining the positive class. Default is `None` which for binary problem is translated to 1. For multiclass problems this argument should not be set as we iteratively change it in the range `[0, num_classes-1]`
- **average** (`Optional[Literal['macro', 'weighted', 'none']]`) – defines the reduction that is applied in the case of multiclass and multilabel input. Should be one of the following:
  - 'macro' [default]: Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
  - 'weighted': Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support.
  - 'none' or `None`: Calculate the metric for each class separately, and return the metric for every class.

**Return type** `Union[List[Tensor], Tensor]`

**Returns** tensor with average precision. If multiclass will return list of such tensors, one for each class

**Example (binary case):**

```
>>> from torchmetrics.functional import average_precision
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 1, 1])
>>> average_precision(pred, target, pos_label=1)
tensor(1.)
```

**Example (multiclass case):**

```
>>> pred = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                      [0.05, 0.75, 0.05, 0.05, 0.05],
...                      [0.05, 0.05, 0.75, 0.05, 0.05],
...                      [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> average_precision(pred, target, num_classes=5, average=None)
[tensor(1.), tensor(1.), tensor(0.2500), tensor(0.2500), tensor(nan)]
```

## binary\_average\_precision

`torchmetrics.functional.classification.binary_average_precision(preds, target, thresholds=None, ignore_index=None, validate_args=True)`

Computes the average precision (AP) score for binary tasks. The AP score summarizes a precision-recall curve as an weighted mean of precisions at each threshold, with the difference in recall from the previous threshold as weight:

$$AP = \sum n(R_n - R_{n-1})P_n$$

where  $P_n, R_n$  is the respective precision and recall at threshold index  $n$ . This value is equivalent to the area under the precision-recall curve (AUPRC).

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

#### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **thresholds** ([Union\[int, List\[float\], Tensor, None\]](#)) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

**Return type** [Tensor](#)

**Returns** A single scalar with the average precision score

#### Example

```
>>> from torchmetrics.functional.classification import binary_average_precision
>>> preds = torch.tensor([0, 0.5, 0.7, 0.8])
>>> target = torch.tensor([0, 1, 1, 0])
>>> binary_average_precision(preds, target, thresholds=None)
tensor(0.5833)
>>> binary_average_precision(preds, target, thresholds=5)
tensor(0.6667)
```

## multiclass\_average\_precision

```
torchmetrics.functional.classification.multiclass_average_precision(preds, target, num_classes,
                                                                    average='macro',
                                                                    thresholds=None,
                                                                    ignore_index=None,
                                                                    validate_args=True)
```

Computes the average precision (AP) score for binary tasks. The AP score summarizes a precision-recall curve as an weighted mean of precisions at each threshold, with the difference in recall from the previous threshold as weight:

$$AP = \sum n(R_n - R_{n-1})P_n$$

where  $P_n, R_n$  is the respective precision and recall at threshold index  $n$ . This value is equivalent to the area under the precision-recall curve (AUPRC).

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the [0, n\_classes-1] range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **average** ([Optional\[Literal\['macro', 'weighted', 'none'\]\]](#)) – Defines the reduction that is applied over classes. Should be one of the following:
  - **macro**: Calculate score for each class and average them
  - **weighted**: Calculates score for each class and computes weighted average using their support
  - **"none"** or **None**: Calculates score for each class and applies no reduction
- **thresholds** ([Union\[int, List\[float\], Tensor, None\]](#)) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation

- If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (*bool*) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

**Return type** *Tensor*

**Returns** If *average=None|"none"* then a 1d tensor of shape (n\_classes, ) will be returned with AP score per class. If *average="macro"|"weighted"* then a single scalar is returned.

### Example

```
>>> from torchmetrics.functional.classification import multiclass_average_precision
>>> preds = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                       [0.05, 0.75, 0.05, 0.05, 0.05],
...                       [0.05, 0.05, 0.75, 0.05, 0.05],
...                       [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> multiclass_average_precision(preds, target, num_classes=5, average="macro",
    ↳ thresholds=None)
tensor(0.6250)
>>> multiclass_average_precision(preds, target, num_classes=5, average=None,
    ↳ thresholds=None)
tensor([1.0000, 1.0000, 0.2500, 0.2500, nan])
>>> multiclass_average_precision(preds, target, num_classes=5, average="macro",
    ↳ thresholds=5)
tensor(0.5000)
>>> multiclass_average_precision(preds, target, num_classes=5, average=None,
    ↳ thresholds=5)
tensor([1.0000, 1.0000, 0.2500, 0.2500, -0.0000])
```

### multilabel\_average\_precision

```
torchmetrics.functional.classification.multilabel_average_precision(preds, target, num_labels,
                                                                    average='macro',
                                                                    thresholds=None,
                                                                    ignore_index=None,
                                                                    validate_args=True)
```

Computes the average precision (AP) score for binary tasks. The AP score summarizes a precision-recall curve as an weighted mean of precisions at each threshold, with the difference in recall from the previous threshold as weight:

$$AP = \sum n(R_n - R_{n-1})P_n$$

where  $P_n, R_n$  is the respective precision and recall at threshold index  $n$ . This value is equivalent to the area under the precision-recall curve (AUPRC).

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.

- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

#### Parameters

- **preds** (Tensor) – Tensor with predictions
- **target** (Tensor) – Tensor with true labels
- **num\_labels** (int) – Integer specifying the number of labels
- **average** (Optional[Literal['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - *micro*: Sum score over all labels
  - *macro*: Calculate score for each label and average them
  - *weighted*: Calculates score for each label and computes weighted average using their support
  - *"none"* or *None*: Calculates score for each label and applies no reduction
- **thresholds** (Union[int, List[float], Tensor, None]) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (bool) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

#### Return type Tensor

**Returns** If *average=**None*|*"none"* then a 1d tensor of shape (n\_classes, ) will be returned with AP score per class. If *average=**"micro|macro"*|*"weighted"* then a single scalar is returned.

### Example

```

>>> from torchmetrics.functional.classification import multilabel_average_precision
>>> preds = torch.tensor([[0.75, 0.05, 0.35],
...                       [0.45, 0.75, 0.05],
...                       [0.05, 0.55, 0.75],
...                       [0.05, 0.65, 0.05]])
>>> target = torch.tensor([[1, 0, 1],
...                        [0, 0, 0],
...                        [0, 1, 1],
...                        [1, 1, 1]])
>>> multilabel_average_precision(preds, target, num_labels=3, average="macro",
↳ thresholds=None)
tensor(0.7500)
>>> multilabel_average_precision(preds, target, num_labels=3, average=None,
↳ thresholds=None)
tensor([0.7500, 0.5833, 0.9167])
>>> multilabel_average_precision(preds, target, num_labels=3, average="macro",
↳ thresholds=5)
tensor(0.7778)
>>> multilabel_average_precision(preds, target, num_labels=3, average=None,
↳ thresholds=5)
tensor([0.7500, 0.6667, 0.9167])

```

## 1.19 Binned Average Precision

### 1.19.1 Module Interface

**class** torchmetrics.**BinnedAveragePrecision**(num\_classes, thresholds=100, \*\*kwargs)

Computes the average precision score, which summarises the precision recall curve into one number. Works for both binary and multiclass problems. In the case of multiclass, the values will be calculated based on a one-vs-the-rest approach.

Computation is performed in constant-memory by computing precision and recall for thresholds buckets/thresholds (evenly distributed between 0 and 1).

Forward accepts

- preds (float tensor): (N, ...) (binary) or (N, C, ...) (multiclass) tensor with probabilities, where C is the number of classes.
- target (long tensor): (N, ...) with integer labels

#### Parameters

- num\_classes (int) – integer with number of classes. Not necessary to provide for binary problems.
- thresholds (Union[int, Tensor, List[float]]) – list or tensor with specific thresholds or a number of bins from linear sampling. It is used for computation will lead to more detailed curve and accurate estimates, but will be slower and consume more memory
- kwargs (Any) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises** **ValueError** – If thresholds is not a list or tensor



Example (binary case):

```
>>> from torchmetrics import BinnedAveragePrecision
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 1, 1])
>>> average_precision = BinnedAveragePrecision(num_classes=1, thresholds=10)
>>> average_precision(pred, target)
tensor(1.0000)
```

Example (multiclass case):

```
>>> pred = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                      [0.05, 0.75, 0.05, 0.05, 0.05],
...                      [0.05, 0.05, 0.75, 0.05, 0.05],
...                      [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> average_precision = BinnedAveragePrecision(num_classes=5, thresholds=10)
>>> average_precision(pred, target)
[tensor(1.0000), tensor(1.0000), tensor(0.2500), tensor(0.2500), tensor(-0.)]
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Returns float tensor of size `n_classes`.

**Return type** `Union[List[Tensor], Tensor]`

## 1.20 Binned Precision Recall Curve

### 1.20.1 Module Interface

**class** `torchmetrics.BinnedPrecisionRecallCurve`(*num\_classes*, *thresholds=100*, *\*\*kwargs*)

Computes precision-recall pairs for different thresholds. Works for both binary and multiclass problems. In the case of multiclass, the values will be calculated based on a one-vs-the-rest approach.

Computation is performed in constant-memory by computing precision and recall for `thresholds` buckets/thresholds (evenly distributed between 0 and 1).

Forward accepts

- `preds` (float tensor): (N, ...) (binary) or (N, C, ...) (multiclass) tensor with probabilities, where C is the number of classes.
- `target` (long tensor): (N, ...) or (N, C, ...) with integer labels

**Parameters**

- **num\_classes** (`int`) – integer with number of classes. For binary, set to 1.
- **thresholds** (`Union[int, Tensor, List[float]]`) – list or tensor with specific thresholds or a number of bins from linear sampling. It is used for computation will lead to more detailed curve and accurate estimates, but will be slower and consume more memory.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises** `ValueError` – If `thresholds` is not a `int`, `list` or `tensor`

Example (binary case):

```
>>> from torchmetrics import BinnedPrecisionRecallCurve
>>> pred = torch.tensor([0, 0.1, 0.8, 0.4])
>>> target = torch.tensor([0, 1, 1, 0])
>>> pr_curve = BinnedPrecisionRecallCurve(num_classes=1, thresholds=5)
>>> precision, recall, thresholds = pr_curve(pred, target)
>>> precision
tensor([0.5000, 0.5000, 1.0000, 1.0000, 1.0000, 1.0000])
>>> recall
tensor([1.0000, 0.5000, 0.5000, 0.5000, 0.0000, 0.0000])
>>> thresholds
tensor([0.0000, 0.2500, 0.5000, 0.7500, 1.0000])
```

Example (multiclass case):

```
>>> pred = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                      [0.05, 0.75, 0.05, 0.05, 0.05],
...                      [0.05, 0.05, 0.75, 0.05, 0.05],
...                      [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> pr_curve = BinnedPrecisionRecallCurve(num_classes=5, thresholds=3)
>>> precision, recall, thresholds = pr_curve(pred, target)
>>> precision
[tensor([0.2500, 1.0000, 1.0000, 1.0000]),
 tensor([0.2500, 1.0000, 1.0000, 1.0000]),
 tensor([2.5000e-01, 1.0000e-06, 1.0000e+00, 1.0000e+00]),
 tensor([2.5000e-01, 1.0000e-06, 1.0000e+00, 1.0000e+00]),
 tensor([2.5000e-07, 1.0000e+00, 1.0000e+00, 1.0000e+00])]
>>> recall
[tensor([1.0000, 1.0000, 0.0000, 0.0000]),
 tensor([1.0000, 1.0000, 0.0000, 0.0000]),
 tensor([1.0000, 0.0000, 0.0000, 0.0000]),
 tensor([1.0000, 0.0000, 0.0000, 0.0000]),
 tensor([0., 0., 0., 0.])]
>>> thresholds
[tensor([0.0000, 0.5000, 1.0000]),
 tensor([0.0000, 0.5000, 1.0000]),
 tensor([0.0000, 0.5000, 1.0000]),
 tensor([0.0000, 0.5000, 1.0000]),
 tensor([0.0000, 0.5000, 1.0000])]
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Returns float tensor of size n\_classes.

**Return type** Union[Tuple[Tensor, Tensor, Tensor], Tuple[List[Tensor], List[Tensor], List[Tensor]]]

**update(preds, target)**

**Args** preds: (n\_samples, n\_classes) tensor target: (n\_samples, n\_classes) tensor

**Return type** None

## 1.21 Binned Recall At Fixed Precision

### 1.21.1 Module Interface

**class** torchmetrics.BinnedRecallAtFixedPrecision(*num\_classes*, *min\_precision*, *thresholds*=100, *\*\*kwargs*)

Computes the highest possible recall value given the minimum precision thresholds provided.

Computation is performed in constant-memory by computing precision and recall for *thresholds* buckets/thresholds (evenly distributed between 0 and 1).

Forward accepts

- **preds** (float tensor): (N, ...) (binary) or (N, C, ...) (multiclass) tensor with probabilities, where C is the number of classes.
- **target** (long tensor): (N, ...) with integer labels

#### Parameters

- **num\_classes** (`int`) – integer with number of classes. Provide 1 for binary problems.
- **min\_precision** (`float`) – float value specifying minimum precision threshold.
- **thresholds** (`Union[int, Tensor, List[float]]`) – list or tensor with specific thresholds or a number of bins from linear sampling. It is used for computation will lead to more detailed curve and accurate estimates, but will be slower and consume more memory
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises** `ValueError` – If *thresholds* is not a list or tensor

**Example (binary case):**

```
>>> from torchmetrics import BinnedRecallAtFixedPrecision
>>> pred = torch.tensor([0, 0.2, 0.5, 0.8])
>>> target = torch.tensor([0, 1, 1, 0])
>>> average_precision = BinnedRecallAtFixedPrecision(num_classes=1,
↳ thresholds=10, min_precision=0.5)
>>> average_precision(pred, target)
(tensor(1.0000), tensor(0.1111))
```

**Example (multiclass case):**

```
>>> pred = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                      [0.05, 0.75, 0.05, 0.05, 0.05],
...                      [0.05, 0.05, 0.75, 0.05, 0.05],
...                      [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> average_precision = BinnedRecallAtFixedPrecision(num_classes=5,
↳ thresholds=10, min_precision=0.5)
>>> average_precision(pred, target)
(tensor([1.0000, 1.0000, 0.0000, 0.0000, 0.0000]),
tensor([6.6667e-01, 6.6667e-01, 1.0000e+06, 1.0000e+06, 1.0000e+06]))
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Returns float tensor of size n\_classes.

**Return type** `Tuple[Tensor, Tensor]`

## 1.22 Calibration Error

### 1.22.1 Module Interface

**class** `torchmetrics.CalibrationError(n_bins=15, norm='l1', **kwargs)`

---

**Note:** From v0.10 an `'binary_*`, `'multiclass_*`, `'multilabel_*` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

---

**Computes the Top-label Calibration Error** Three different norms are implemented, each corresponding to variations on the calibration error metric.

L1 norm (Expected Calibration Error)

$$\text{ECE} = \sum_i^N b_i \| (p_i - c_i) \|$$

Infinity norm (Maximum Calibration Error)

$$\text{MCE} = \max_i (p_i - c_i)$$

L2 norm (Root Mean Square Calibration Error)

$$\text{RMSCE} = \sqrt{\sum_i^N b_i (p_i - c_i)^2}$$

Where  $p_i$  is the top-1 prediction accuracy in bin  $i$ ,  $c_i$  is the average confidence of predictions in bin  $i$ , and  $b_i$  is the fraction of data points in bin  $i$ .

---

**Note:** L2-norm debiasing is not yet supported.

---

#### Parameters

- **n\_bins** (`int`) – Number of bins to use when computing probabilities and accuracies.
- **norm** (`str`) – Norm used to compare empirical and expected probability bins. Defaults to “l1”, or Expected Calibration Error.
- **debias** – Applies debiasing term, only implemented for l2 norm. Defaults to True.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

**compute()**

Computes calibration error across all confidences and accuracies.

**Returns** Calibration error across previously collected examples.

**Return type** Tensor

**update(preds, target)**

Computes top-level confidences and accuracies for the input probabilities and appends them to internal state.

**Parameters**

- **preds** (Tensor) – Model output probabilities.
- **target** (Tensor) – Ground-truth target class labels.

**Return type** None

**BinaryCalibrationError**

```
class torchmetrics.classification.BinaryCalibrationError(n_bins=15, norm='l1',
                                                         ignore_index=None, validate_args=True,
                                                         **kwargs)
```

Computes the [Top-label Calibration Error](#) for binary tasks. The expected calibration error can be used to quantify how well a given model is calibrated e.g. how well the predicted output probabilities of the model matches the actual probabilities of the ground truth distribution.

Three different norms are implemented, each corresponding to variations on the calibration error metric.

$$\text{ECE} = \sum_i^N b_i \| (p_i - c_i) \|, \text{L1 norm (Expected Calibration Error)}$$

$$\text{MCE} = \max_i (p_i - c_i), \text{Infinity norm (Maximum Calibration Error)}$$

$$\text{RMSCE} = \sqrt{\sum_i^N b_i (p_i - c_i)^2}, \text{L2 norm (Root Mean Square Calibration Error)}$$

Where  $p_i$  is the top-1 prediction accuracy in bin  $i$ ,  $c_i$  is the average confidence of predictions in bin  $i$ , and  $b_i$  is the fraction of data points in bin  $i$ . Bins are constructed in a uniform way in the  $[0,1]$  range.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain  $\{0,1\}$  values (except if `ignore_index` is specified).

Additional dimension ... will be flattened into the batch dimension.

**Parameters**

- **n\_bins** (int) – Number of bins to use when computing the metric.
- **norm** (Literal['l1', 'l2', 'max']) – Norm used to compare empirical and expected probability bins.

- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torchmetrics.classification import BinaryCalibrationError
>>> preds = torch.tensor([0.25, 0.25, 0.55, 0.75, 0.75])
>>> target = torch.tensor([0, 0, 1, 1, 1])
>>> metric = BinaryCalibrationError(n_bins=2, norm='l1')
>>> metric(preds, target)
tensor(0.2900)
>>> metric = BinaryCalibrationError(n_bins=2, norm='l2')
>>> metric(preds, target)
tensor(0.2918)
>>> metric = BinaryCalibrationError(n_bins=2, norm='max')
>>> metric(preds, target)
tensor(0.3167)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

#### `compute()`

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tensor`

#### `update(preds, target)`

Override this method to update the state variables of your metric class.

**Return type** `None`

### MulticlassCalibrationError

```
class torchmetrics.classification.MulticlassCalibrationError(num_classes, n_bins=15, norm='l1',
                                                            ignore_index=None,
                                                            validate_args=True, **kwargs)
```

Computes the [Top-label Calibration Error](#) for multiclass tasks. The expected calibration error can be used to quantify how well a given model is calibrated e.g. how well the predicted output probabilities of the model matches the actual probabilities of the ground truth distribution.

Three different norms are implemented, each corresponding to variations on the calibration error metric.

$$\text{ECE} = \sum_i^N b_i \|(p_i - c_i)\|, \text{L1 norm (Expected Calibration Error)}$$

$$\text{MCE} = \max_i (p_i - c_i), \text{Infinity norm (Maximum Calibration Error)}$$

$$\text{RMSCE} = \sqrt{\sum_i^N b_i (p_i - c_i)^2}, \text{L2 norm (Root Mean Square Calibration Error)}$$

Where  $p_i$  is the top-1 prediction accuracy in bin  $i$ ,  $c_i$  is the average confidence of predictions in bin  $i$ , and  $b_i$  is the fraction of data points in bin  $i$ . Bins are constructed in a uniform way in the  $[0,1]$  range.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the  $[0, n\_classes-1]$  range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

#### Parameters

- **num\_classes** (int) – Integer specifying the number of classes
- **n\_bins** (int) – Number of bins to use when computing the metric.
- **norm** (Literal['l1', 'l2', 'max']) – Norm used to compare empirical and expected probability bins.
- **ignore\_index** (Optional[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (bool) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.
- **kwargs** (Any) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Example

```
>>> from torchmetrics.classification import MulticlassCalibrationError
>>> preds = torch.tensor([[0.25, 0.20, 0.55],
...                       [0.55, 0.05, 0.40],
...                       [0.10, 0.30, 0.60],
...                       [0.90, 0.05, 0.05]])
>>> target = torch.tensor([0, 1, 2, 0])
>>> metric = MulticlassCalibrationError(num_classes=3, n_bins=3, norm='l1')
>>> metric(preds, target)
tensor(0.2000)
>>> metric = MulticlassCalibrationError(num_classes=3, n_bins=3, norm='l2')
>>> metric(preds, target)
tensor(0.2082)
>>> metric = MulticlassCalibrationError(num_classes=3, n_bins=3, norm='max')
>>> metric(preds, target)
tensor(0.2333)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### compute()

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** Tensor

#### update(preds, target)

Override this method to update the state variables of your metric class.

Return type `None`

## 1.22.2 Functional Interface

```
torchmetrics.functional.calibration_error(preds, target, n_bins=15, norm='l1', task=None,
                                         num_classes=None, ignore_index=None,
                                         validate_args=True)
```

---

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as a single entrypoint to calling the three specialized versions.

---

Computes the Top-label Calibration Error

Three different norms are implemented, each corresponding to variations on the calibration error metric.

L1 norm (Expected Calibration Error)

$$\text{ECE} = \sum_i^N b_i \| (p_i - c_i) \|$$

Infinity norm (Maximum Calibration Error)

$$\text{MCE} = \max_i (p_i - c_i)$$

L2 norm (Root Mean Square Calibration Error)

$$\text{RMSCE} = \sqrt{\sum_i^N b_i (p_i - c_i)^2}$$

Where  $p_i$  is the top-1 prediction accuracy in bin  $i$ ,  $c_i$  is the average confidence of predictions in bin  $i$ , and  $b_i$  is the fraction of data points in bin  $i$ .

### Parameters

- **preds** (`Tensor`) – Model output probabilities.
- **target** (`Tensor`) – Ground-truth target class labels.
- **n\_bins** (`int`) – Number of bins to use when computing t.
- **norm** (`Literal['l1', 'l2', 'max']`) – Norm used to compare empirical and expected probability bins. Defaults to “l1”, or Expected Calibration Error.

Return type `Tensor`



## binary\_calibration\_error

```
torchmetrics.functional.classification.binary_calibration_error(preds, target, n_bins=15,
                                                                norm='l1', ignore_index=None,
                                                                validate_args=True)
```

Computes the [Top-label Calibration Error](#) for binary tasks. The expected calibration error can be used to quantify how well a given model is calibrated e.g. how well the predicted output probabilities of the model matches the actual probabilities of the ground truth distribution.

Three different norms are implemented, each corresponding to variations on the calibration error metric.

$$\text{ECE} = \sum_i^N b_i \| (p_i - c_i) \|, \text{L1 norm (Expected Calibration Error)}$$

$$\text{MCE} = \max_i (p_i - c_i), \text{Infinity norm (Maximum Calibration Error)}$$

$$\text{RMSCE} = \sqrt{\sum_i^N b_i (p_i - c_i)^2}, \text{L2 norm (Root Mean Square Calibration Error)}$$

Where  $p_i$  is the top-1 prediction accuracy in bin  $i$ ,  $c_i$  is the average confidence of predictions in bin  $i$ , and  $b_i$  is the fraction of data points in bin  $i$ . Bins are constructed in a uniform way in the  $[0,1]$  range.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain  $\{0,1\}$  values (except if `ignore_index` is specified).

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **n\_bins** ([int](#)) – Number of bins to use when computing the metric.
- **norm** ([Literal](#)['l1', 'l2', 'max']) – Norm used to compare empirical and expected probability bins.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Example

```
>>> from torchmetrics.functional.classification import binary_calibration_error
>>> preds = torch.tensor([0.25, 0.25, 0.55, 0.75, 0.75])
>>> target = torch.tensor([0, 0, 1, 1, 1])
>>> binary_calibration_error(preds, target, n_bins=2, norm='l1')
tensor(0.2900)
>>> binary_calibration_error(preds, target, n_bins=2, norm='l2')
tensor(0.2918)
>>> binary_calibration_error(preds, target, n_bins=2, norm='max')
tensor(0.3167)
```

Return type `Tensor`

### multiclass\_calibration\_error

`torchmetrics.functional.classification.multiclass_calibration_error(preds, target, num_classes, n_bins=15, norm='l1', ignore_index=None, validate_args=True)`

Computes the Top-label Calibration Error for multiclass tasks. The expected calibration error can be used to quantify how well a given model is calibrated e.g. how well the predicted output probabilities of the model matches the actual probabilities of the ground truth distribution.

Three different norms are implemented, each corresponding to variations on the calibration error metric.

$$\text{ECE} = \sum_i^N b_i \| (p_i - c_i) \|, \text{L1 norm (Expected Calibration Error)}$$

$$\text{MCE} = \max_i (p_i - c_i), \text{Infinity norm (Maximum Calibration Error)}$$

$$\text{RMSCE} = \sqrt{\sum_i^N b_i (p_i - c_i)^2}, \text{L2 norm (Root Mean Square Calibration Error)}$$

Where  $p_i$  is the top-1 prediction accuracy in bin  $i$ ,  $c_i$  is the average confidence of predictions in bin  $i$ , and  $b_i$  is the fraction of data points in bin  $i$ . Bins are constructed in an uniform way in the  $[0,1]$  range.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the  $[0, \text{num\_classes}-1]$  range (except if `ignore_index` is specified).

Additional dimension ... will be flattened into the batch dimension.

#### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **num\_classes** (`int`) – Integer specifying the number of classes
- **n\_bins** (`int`) – Number of bins to use when computing the metric.

- **norm** (`Literal['l1', 'l2', 'max']`) – Norm used to compare empirical and expected probability bins.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Example

```
>>> from torchmetrics.functional.classification import multiclass_calibration_error
>>> preds = torch.tensor([[0.25, 0.20, 0.55],
...                       [0.55, 0.05, 0.40],
...                       [0.10, 0.30, 0.60],
...                       [0.90, 0.05, 0.05]])
>>> target = torch.tensor([0, 1, 2, 0])
>>> multiclass_calibration_error(preds, target, num_classes=3, n_bins=3, norm='l1')
tensor(0.2000)
>>> multiclass_calibration_error(preds, target, num_classes=3, n_bins=3, norm='l2')
tensor(0.2082)
>>> multiclass_calibration_error(preds, target, num_classes=3, n_bins=3, norm='max')
tensor(0.2333)
```

Return type `Tensor`

## 1.23 Cohen Kappa

### 1.23.1 Module Interface

#### CohenKappa

**class** `torchmetrics.CohenKappa`(*num\_classes*, *weights=None*, *threshold=0.5*, *\*\*kwargs*)

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the *task* argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

Calculates [Cohen's kappa score](#) that measures inter-annotator agreement. It is defined as

$$\kappa = (p_o - p_e) / (1 - p_e)$$

where  $p_o$  is the empirical probability of agreement and  $p_e$  is the expected agreement when both annotators assign labels randomly. Note that  $p_e$  is estimated using a per-annotator empirical prior over the class labels.

Works with binary, multiclass, and multilabel data. Accepts probabilities from a model output or integer class values in prediction. Works with multi-dimensional preds and target.

**Forward accepts**

- **preds** (float or long tensor): (N, ...) or (N, C, ...) where C is the number of classes
- **target** (long tensor): (N, ...)

If preds and target are the same shape and preds is a float tensor, we use the `self.threshold` argument to convert into integer labels. This is the case for binary and multi-label probabilities or logits.

If preds has an extra dimension as in the case of multi-class scores we perform an `argmax` on `dim=1`.

#### Parameters

- **num\_classes** (`int`) – Number of classes in the dataset.
- **weights** (`Optional[str]`) – Weighting type to calculate the score. Choose from:
  - `None` or `'none'`: no weighting
  - `'linear'`: linear weighting
  - `'quadratic'`: quadratic weighting
- **threshold** (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Example

```
>>> from torchmetrics import CohenKappa
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0, 1, 0, 0])
>>> cohenkappa = CohenKappa(num_classes=2)
>>> cohenkappa(preds, target)
tensor(0.5000)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

#### `compute()`

Computes cohen kappa score.

**Return type** `Tensor`

#### `update(preds, target)`

Update state with predictions and targets.

#### Parameters

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

## BinaryCohenKappa

`class torchmetrics.classification.BinaryCohenKappa(threshold=0.5, ignore_index=None, weights=None, validate_args=True, **kwargs)`

Calculates [Cohen's kappa score](#) that measures inter-annotator agreement for binary tasks. It is defined as

$$\kappa = (p_o - p_e) / (1 - p_e)$$

where  $p_o$  is the empirical probability of agreement and  $p_e$  is the expected agreement when both annotators assign labels randomly. Note that  $p_e$  is estimated using a per-annotator empirical prior over the class labels.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **threshold** (float) – Threshold for transforming probability to binary (0,1) predictions
- **ignore\_index** (Optional[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **weights** (Optional[Literal['linear', 'quadratic', 'none']]) – Weighting type to calculate the score. Choose from:
  - None or 'none': no weighting
  - 'linear': linear weighting
  - 'quadratic': quadratic weighting
- **validate\_args** (bool) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.
- **kwargs** (Any) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Example (preds is int tensor):

```
>>> from torchmetrics.classification import BinaryCohenKappa
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0, 1, 0, 0])
>>> metric = BinaryCohenKappa()
>>> metric(preds, target)
tensor(0.5000)
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import BinaryCohenKappa
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0.35, 0.85, 0.48, 0.01])
>>> metric = BinaryCohenKappa()
>>> metric(preds, target)
tensor(0.5000)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

## MulticlassCohenKappa

```
class torchmetrics.classification.MulticlassCohenKappa(num_classes, ignore_index=None,
                                                       weights=None, validate_args=True,
                                                       **kwargs)
```

Calculates [Cohen's kappa score](#) that measures inter-annotator agreement for multiclass tasks. It is defined as

$$\kappa = (p_o - p_e) / (1 - p_e)$$

where  $p_o$  is the empirical probability of agreement and  $p_e$  is the expected agreement when both annotators assign labels randomly. Note that  $p_e$  is estimated using a per-annotator empirical prior over the class labels.

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **num\_classes** (`int`) – Integer specifying the number of classes
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **weights** (`Optional[Literal['linear', 'quadratic', 'none']]`) – Weighting type to calculate the score. Choose from:
  - None or 'none': no weighting
  - 'linear': linear weighting
  - 'quadratic': quadratic weighting
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Example (pred is integer tensor):

```
>>> from torchmetrics.classification import MulticlassCohenKappa
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> metric = MulticlassCohenKappa(num_classes=3)
>>> metric(preds, target)
tensor(0.6364)
```

Example (pred is float tensor):

```
>>> from torchmetrics.classification import MulticlassCohenKappa
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
```

(continues on next page)

(continued from previous page)

```
... ]])
>>> metric = MulticlassCohenKappa(num_classes=3)
>>> metric(preds, target)
tensor(0.6364)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

## 1.23.2 Functional Interface

### cohen\_kappa

`torchmetrics.functional.cohen_kappa(preds, target, num_classes, weights=None, threshold=0.5, task=None, ignore_index=None, validate_args=True)`

---

**Note:** From v0.10 an `'binary_*`, `'multiclass_*`, `'multilabel_*` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

---

Calculates [Cohen's kappa score](#) that measures inter-annotator agreement.

It is defined as

$$\kappa = (p_o - p_e) / (1 - p_e)$$

where  $p_o$  is the empirical probability of agreement and  $p_e$  is the expected agreement when both annotators assign labels randomly. Note that  $p_e$  is estimated using a per-annotator empirical prior over the class labels.

#### Parameters

- **preds** ([Tensor](#)) – (float or long tensor), Either a (N, ...) tensor with labels or (N, C, ...) where C is the number of classes, tensor with labels/probabilities
- **target** ([Tensor](#)) – target (long tensor), tensor with shape (N, ...) with ground true labels
- **num\_classes** ([int](#)) – Number of classes in the dataset.
- **weights** ([OptionalLiteral](#)['linear', 'quadratic', 'none']) – Weighting type to calculate the score. Choose from:
  - None or 'none': no weighting
  - 'linear': linear weighting
  - 'quadratic': quadratic weighting
- **threshold** ([float](#)) – Threshold value for binary or multi-label probabilities.

### Example

```
>>> from torchmetrics.functional import cohen_kappa
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0, 1, 0, 0])
>>> cohen_kappa(preds, target, num_classes=2)
tensor(0.5000)
```

Return type `Tensor`

### binary\_cohen\_kappa

`torchmetrics.functional.classification.binary_cohen_kappa(preds, target, threshold=0.5, weights=None, ignore_index=None, validate_args=True)`

Calculates [Cohen's kappa score](#) that measures inter-annotator agreement for binary tasks. It is defined as

$$\kappa = (p_o - p_e) / (1 - p_e)$$

where  $p_o$  is the empirical probability of agreement and  $p_e$  is the expected agreement when both annotators assign labels randomly. Note that  $p_e$  is estimated using a per-annotator empirical prior over the class labels.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

Additional dimension ... will be flattened into the batch dimension.

#### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **threshold** (`float`) – Threshold for transforming probability to binary (0,1) predictions
- **weights** (`Optional[Literal['linear', 'quadratic', 'none']]`) – Weighting type to calculate the score. Choose from:
  - None or 'none': no weighting
  - 'linear': linear weighting
  - 'quadratic': quadratic weighting
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Example (preds is int tensor):



```
>>> from torchmetrics.functional.classification import binary_cohen_kappa
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0, 1, 0, 0])
>>> binary_cohen_kappa(preds, target)
tensor(0.5000)
```

Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import binary_cohen_kappa
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0.35, 0.85, 0.48, 0.01])
>>> binary_cohen_kappa(preds, target)
tensor(0.5000)
```

Return type `Tensor`

## multiclass\_cohen\_kappa

`torchmetrics.functional.classification.multiclass_cohen_kappa(preds, target, num_classes, weights=None, ignore_index=None, validate_args=True)`

Calculates Cohen's kappa score that measures inter-annotator agreement for multiclass tasks. It is defined as

$$\kappa = (p_o - p_e) / (1 - p_e)$$

where  $p_o$  is the empirical probability of agreement and  $p_e$  is the expected agreement when both annotators assign labels randomly. Note that  $p_e$  is estimated using a per-annotator empirical prior over the class labels.

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **num\_classes** (`int`) – Integer specifying the number of classes
- **weights** (`Optional[Literal['linear', 'quadratic', 'none']]`) – Weighting type to calculate the score. Choose from:
  - None or 'none': no weighting
  - 'linear': linear weighting
  - 'quadratic': quadratic weighting
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

- **kwargs** – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Example (pred is integer tensor):

```
>>> from torchmetrics.functional.classification import multiclass_cohen_kappa
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> multiclass_cohen_kappa(preds, target, num_classes=3)
tensor(0.6364)
```

Example (pred is float tensor):

```
>>> from torchmetrics.functional.classification import multiclass_cohen_kappa
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> multiclass_cohen_kappa(preds, target, num_classes=3)
tensor(0.6364)
```

Return type `Tensor`

## 1.24 Confusion Matrix

### 1.24.1 Module Interface

#### ConfusionMatrix

```
class torchmetrics.ConfusionMatrix(num_classes, normalize=None, threshold=0.5, multilabel=False,
                                   **kwargs)
```

---

**Note:** From v0.10 an `'binary_*`, `'multiclass_*`, `'multilabel_*` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

---

Computes the [confusion matrix](#).

Works with binary, multiclass, and multilabel data. Accepts probabilities or logits from a model output or integer class values in prediction. Works with multi-dimensional preds and target, but it should be noted that additional dimensions will be flattened.

Forward accepts

- **preds** (float or long tensor): (N, ...) or (N, C, ...) where C is the number of classes
- **target** (long tensor): (N, ...)

If preds and target are the same shape and preds is a float tensor, we use the `self.threshold` argument to convert into integer labels. This is the case for binary and multi-label probabilities or logits.

If preds has an extra dimension as in the case of multi-class scores we perform an `argmax` on `dim=1`.

If working with multilabel data, setting the `is_multilabel` argument to `True` will make sure that a [confusion matrix gets calculated per label](#).

### Parameters

- **num\_classes** (`int`) – Number of classes in the dataset.
- **normalize** (`Optional[str]`) – Normalization mode for confusion matrix. Choose from:
  - `None` or `'none'`: no normalization (default)
  - `'true'`: normalization over the targets (most commonly used)
  - `'pred'`: normalization over the predictions
  - `'all'`: normalization over the whole matrix
- **threshold** (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
- **multilabel** (`bool`) – determines if data is multilabel or not.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example (binary data):

```
>>> from torchmetrics import ConfusionMatrix
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0, 1, 0, 0])
>>> confmat = ConfusionMatrix(num_classes=2)
>>> confmat(preds, target)
tensor([[2, 0],
        [1, 1]])
```

### Example (multiclass data):

```
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> confmat = ConfusionMatrix(num_classes=3)
>>> confmat(preds, target)
tensor([[1, 1, 0],
        [0, 1, 0],
        [0, 0, 1]])
```

### Example (multilabel data):

```
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> confmat = ConfusionMatrix(num_classes=3, multilabel=True)
>>> confmat(preds, target)
tensor([[[1, 0], [0, 1]],
        [[1, 0], [1, 0]],
        [[0, 1], [0, 1]]])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes confusion matrix.

**Return type** `Tensor`

**Returns** If `multilabel=False` this will be a `[n_classes, n_classes]` tensor and if `multilabel=True` this will be a `[n_classes, 2, 2]` tensor.

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

## BinaryConfusionMatrix

```
class torchmetrics.classification.BinaryConfusionMatrix(threshold=0.5, ignore_index=None,
                                                         normalize=None, validate_args=True,
                                                         **kwargs)
```

Computes the `confusion matrix` for binary tasks.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **target** (int tensor): (N, ...)

Additional dimension ... will be flattened into the batch dimension.

**Parameters**

- **threshold** (`float`) – Threshold for transforming probability to binary (0,1) predictions
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **normalize** (`Optional[Literal['true', 'pred', 'all', 'none']]`) – Normalization mode for confusion matrix. Choose from:
  - `None` or `'none'`: no normalization (default)
  - `'true'`: normalization over the targets (most commonly used)
  - `'pred'`: normalization over the predictions
  - `'all'`: normalization over the whole matrix
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import BinaryConfusionMatrix
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0, 1, 0, 0])
>>> metric = BinaryConfusionMatrix()
>>> metric(preds, target)
tensor([[2, 0],
        [1, 1]])
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import BinaryConfusionMatrix
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0.35, 0.85, 0.48, 0.01])
>>> metric = BinaryConfusionMatrix()
>>> metric(preds, target)
tensor([[2, 0],
        [1, 1]])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes confusion matrix.

Returns an [2,2] matrix.

**Return type** `Tensor`

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels

**Return type** `None`

## MulticlassConfusionMatrix

```
class torchmetrics.classification.MulticlassConfusionMatrix(num_classes, ignore_index=None,
                                                            normalize=None, validate_args=True,
                                                            **kwargs)
```

Computes the `confusion matrix` for multiclass tasks.

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

Additional dimension ... will be flattened into the batch dimension.

**Parameters**

- **num\_classes** (`int`) – Integer specifying the number of classes

- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **normalize** (`Optional[Literal['none', 'true', 'pred', 'all']]`) – Normalization mode for confusion matrix. Choose from:
  - `None` or `'none'`: no normalization (default)
  - `'true'`: normalization over the targets (most commonly used)
  - `'pred'`: normalization over the predictions
  - `'all'`: normalization over the whole matrix
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (pred is integer tensor):**

```
>>> from torchmetrics.classification import MulticlassConfusionMatrix
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> metric = MulticlassConfusionMatrix(num_classes=3)
>>> metric(preds, target)
tensor([[1, 1, 0],
        [0, 1, 0],
        [0, 0, 1]])
```

**Example (pred is float tensor):**

```
>>> from torchmetrics.classification import MulticlassConfusionMatrix
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> metric = MulticlassConfusionMatrix(num_classes=3)
>>> metric(preds, target)
tensor([[1, 1, 0],
        [0, 1, 0],
        [0, 0, 1]])
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

**compute()**

Computes confusion matrix.

Returns an `[num_classes, num_classes]` matrix.

**Return type** `Tensor`

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels

Return type `None`

## MultilabelConfusionMatrix

```
class torchmetrics.classification.MultilabelConfusionMatrix(num_labels, threshold=0.5,
                                                            ignore_index=None,
                                                            normalize=None, validate_args=True,
                                                            **kwargs)
```

Computes the [confusion matrix](#) for multilabel tasks.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **num\_classes** – Integer specifying the number of labels
- **threshold** (`float`) – Threshold for transforming probability to binary (0,1) predictions
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **normalize** (`Optional[Literal['none', 'true', 'pred', 'all']]`) – Normalization mode for confusion matrix. Choose from:
  - `None` or `'none'`: no normalization (default)
  - `'true'`: normalization over the targets (most commonly used)
  - `'pred'`: normalization over the predictions
  - `'all'`: normalization over the whole matrix
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Example (preds is int tensor):

```
>>> from torchmetrics.classification import MultilabelConfusionMatrix
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelConfusionMatrix(num_labels=3)
>>> metric(preds, target)
tensor([[[1, 0], [0, 1]],
        [[1, 0], [1, 0]],
        [[0, 1], [0, 1]]])
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import MultilabelConfusionMatrix
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelConfusionMatrix(num_labels=3)
>>> metric(preds, target)
tensor([[[1, 0], [0, 1]],
        [[1, 0], [1, 0]],
        [[0, 1], [0, 1]])])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes confusion matrix.

Returns an [num\_labels,2,2] matrix.

**Return type** `Tensor`

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels

**Return type** `None`

## 1.24.2 Functional Interface

### `confusion_matrix`

`torchmetrics.functional.confusion_matrix(preds, target, num_classes, normalize=None, threshold=0.5, multilabel=False, task=None, num_labels=None, ignore_index=None, validate_args=True)`

---

**Note:** From v0.10 an `'binary_*`, `'multiclass_*`, `'multilabel_*` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entryptpoint to calling the three specialized versions.

---

Computes the `confusion matrix`. Works with binary, multiclass, and multilabel data. Accepts probabilities or logits from a model output or integer class values in prediction. Works with multi-dimensional preds and target, but it should be noted that additional dimensions will be flattened.

If preds and target are the same shape and preds is a float tensor, we use the `self.threshold` argument to convert into integer labels. This is the case for binary and multi-label probabilities or logits.

If preds has an extra dimension as in the case of multi-class scores we perform an `argmax` on `dim=1`.

If working with multilabel data, setting the `is_multilabel` argument to `True` will make sure that a `confusion matrix` gets calculated per label.

**Parameters**



- **preds** (`Tensor`) – (float or long tensor), Either a (N, ...) tensor with labels or (N, C, ...) where C is the number of classes, tensor with labels/logits/probabilities
- **target** (`Tensor`) – target (long tensor), tensor with shape (N, ...) with ground true labels
- **num\_classes** (`int`) – Number of classes in the dataset.
- **normalize** (`Optional[Literal['true', 'pred', 'all', 'none']]`) – Normalization mode for confusion matrix. Choose from:
  - None or 'none': no normalization (default)
  - 'true': normalization over the targets (most commonly used)
  - 'pred': normalization over the predictions
  - 'all': normalization over the whole matrix
- **threshold** (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
- **multilabel** (`bool`) – determines if data is multilabel or not.

Example (binary data):

```
>>> from torchmetrics import ConfusionMatrix
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0, 1, 0, 0])
>>> confmat = ConfusionMatrix(num_classes=2)
>>> confmat(preds, target)
tensor([[2, 0],
        [1, 1]])
```

Example (multiclass data):

```
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> confmat = ConfusionMatrix(num_classes=3)
>>> confmat(preds, target)
tensor([[1, 1, 0],
        [0, 1, 0],
        [0, 0, 1]])
```

Example (multilabel data):

```
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> confmat = ConfusionMatrix(num_classes=3, multilabel=True)
>>> confmat(preds, target)
tensor([[1, 0], [0, 1]],
        [[1, 0], [1, 0]],
        [[0, 1], [0, 1]])
```

Return type `Tensor`

## binary\_confusion\_matrix

```
torchmetrics.functional.classification.binary_confusion_matrix(preds, target, threshold=0.5,
                                                                normalize=None,
                                                                ignore_index=None,
                                                                validate_args=True)
```

Computes the [confusion matrix](#) for binary tasks.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **normalize** ([Optional](#)[[Literal](#)['true', 'pred', 'all', 'none']]) – Normalization mode for confusion matrix. Choose from:
  - None or 'none': no normalization (default)
  - 'true': normalization over the targets (most commonly used)
  - 'pred': normalization over the predictions
  - 'all': normalization over the whole matrix
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.

**Return type** [Tensor](#)

**Returns** A [2, 2] tensor

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import binary_confusion_matrix
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0, 1, 0, 0])
>>> binary_confusion_matrix(preds, target)
tensor([[2, 0],
        [1, 1]])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import binary_confusion_matrix
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0.35, 0.85, 0.48, 0.01])
>>> binary_confusion_matrix(preds, target)
```

(continues on next page)

(continued from previous page)

```
tensor([[2, 0],
        [1, 1]])
```

## multiclass\_confusion\_matrix

`torchmetrics.functional.classification.multiclass_confusion_matrix(preds, target, num_classes, normalize=None, ignore_index=None, validate_args=True)`

Computes the [confusion matrix](#) for multiclass tasks.

Accepts the following input tensors:

- **preds** (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **normalize** ([Optional](#)[[Literal](#)['true', 'pred', 'all', 'none']]) – Normalization mode for confusion matrix. Choose from:
  - None or 'none': no normalization (default)
  - 'true': normalization over the targets (most commonly used)
  - 'pred': normalization over the predictions
  - 'all': normalization over the whole matrix
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.

**Return type** [Tensor](#)

**Returns** A [num\_classes, num\_classes] tensor

**Example (pred is integer tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_confusion_
    matrix
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> multiclass_confusion_matrix(preds, target, num_classes=3)
tensor([[1, 1, 0],
        [0, 1, 0],
        [0, 0, 1]])
```

Example (pred is float tensor):

```
>>> from torchmetrics.functional.classification import multiclass_confusion_
    matrix
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> multiclass_confusion_matrix(preds, target, num_classes=3)
tensor([[1, 1, 0],
        [0, 1, 0],
        [0, 0, 1]])
```

## multilabel\_confusion\_matrix

`torchmetrics.functional.classification.multilabel_confusion_matrix`(*preds, target, num\_labels, threshold=0.5, normalize=None, ignore\_index=None, validate\_args=True*)

Computes the [confusion matrix](#) for multilabel tasks.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **target** (int tensor): (N, C, ...)

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **normalize** ([Optional](#)[[Literal](#)['true', 'pred', 'all', 'none']]) – Normalization mode for confusion matrix. Choose from:
  - None or 'none': no normalization (default)
  - 'true': normalization over the targets (most commonly used)
  - 'pred': normalization over the predictions
  - 'all': normalization over the whole matrix
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Return type** `Tensor`

**Returns** A `[num_labels, 2, 2]` tensor

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_confusion_
      ↪matrix
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_confusion_matrix(preds, target, num_labels=3)
tensor([[[1, 0], [0, 1]],
        [[1, 0], [1, 0]],
        [[0, 1], [0, 1]]])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_confusion_
      ↪matrix
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_confusion_matrix(preds, target, num_labels=3)
tensor([[[1, 0], [0, 1]],
        [[1, 0], [1, 0]],
        [[0, 1], [0, 1]]])
```

## 1.25 Coverage Error

### 1.25.1 Module Interface

**class** `torchmetrics.CoverageError`(\*\**kwargs*)

Computes multilabel coverage error [1]. The score measure how far we need to go through the ranked scores to cover all true labels. The best value is equal to the average number of labels in the target tensor per sample.

**Parameters** *kwargs* (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

**Example**

```
>>> from torchmetrics import CoverageError
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand(10, 5)
>>> target = torch.randint(2, (10, 5))
>>> metric = CoverageError()
>>> metric(preds, target)
tensor(3.9000)
```

## References

[1] Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). Mining multi-label data. In Data mining and knowledge discovery handbook (pp. 667-685). Springer US.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes the multilabel coverage error.

**Return type** `Tensor`

**update**(*preds*, *target*, *sample\_weight=None*)

### Parameters

- **preds** (`Tensor`) – tensor of shape  $[N, L]$  where  $N$  is the number of samples and  $L$  is the number of labels. Should either be probabilities of the positive class or corresponding logits
- **target** (`Tensor`) – tensor of shape  $[N, L]$  where  $N$  is the number of samples and  $L$  is the number of labels. Should only contain binary labels.
- **sample\_weight** (`Optional[Tensor]`) – tensor of shape  $N$  where  $N$  is the number of samples. How much each sample should be weighted in the final score.

**Return type** `None`

**class** `torchmetrics.classification.MultilabelCoverageError`(*num\_labels*, *ignore\_index=None*, *validate\_args=True*, *\*\*kwargs*)

Computes multilabel coverage error [1]. The score measure how far we need to go through the ranked scores to cover all true labels. The best value is equal to the average number of labels in the target tensor per sample.

Accepts the following input tensors:

- **preds** (float tensor):  $(N, C, \dots)$ . Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor):  $(N, C, \dots)$ . Target should be a tensor containing ground truth labels, and therefore only contain  $\{0,1\}$  values (except if *ignore\_index* is specified).

Additional dimension  $\dots$  will be flattened into the batch dimension.

### Parameters

- **num\_labels** (`int`) – Integer specifying the number of labels
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Example

```
>>> from torchmetrics.classification import MultilabelCoverageError
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand(10, 5)
>>> target = torch.randint(2, (10, 5))
>>> metric = MultilabelCoverageError(num_labels=5)
>>> metric(preds, target)
tensor(3.9000)
```

### References

[1] Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). Mining multi-label data. In Data mining and knowledge discovery handbook (pp. 667-685). Springer US.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### **compute()**

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tensor`

#### **update(preds, target)**

Override this method to update the state variables of your metric class.

**Return type** `None`

## 1.25.2 Functional Interface

`torchmetrics.functional.coverage_error(preds, target, sample_weight=None)`

Computes multilabel coverage error [1]. The score measure how far we need to go through the ranked scores to cover all true labels. The best value is equal to the average number of labels in the target tensor per sample.

#### **Parameters**

- **preds** (`Tensor`) – tensor of shape  $[N, L]$  where  $N$  is the number of samples and  $L$  is the number of labels. Should either be probabilities of the positive class or corresponding logits
- **target** (`Tensor`) – tensor of shape  $[N, L]$  where  $N$  is the number of samples and  $L$  is the number of labels. Should only contain binary labels.
- **sample\_weight** (`Optional[Tensor]`) – tensor of shape  $N$  where  $N$  is the number of samples. How much each sample should be weighted in the final score.

### Example

```
>>> from torchmetrics.functional import coverage_error
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand(10, 5)
>>> target = torch.randint(2, (10, 5))
>>> coverage_error(preds, target)
tensor(3.9000)
```

### References

[1] Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). Mining multi-label data. In Data mining and knowledge discovery handbook (pp. 667-685). Springer US.

**Return type** `Tensor`

`torchmetrics.functional.classification.multilabel_coverage_error(preds, target, num_labels, ignore_index=None, validate_args=True)`

Computes multilabel coverage error [1]. The score measure how far we need to go through the ranked scores to cover all true labels. The best value is equal to the average number of labels in the target tensor per sample.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

#### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **num\_labels** (`int`) – Integer specifying the number of labels
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Example

```
>>> from torchmetrics.functional.classification import multilabel_coverage_error
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand(10, 5)
>>> target = torch.randint(2, (10, 5))
>>> multilabel_coverage_error(preds, target, num_labels=5)
tensor(3.9000)
```



## References

[1] Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). Mining multi-label data. In Data mining and knowledge discovery handbook (pp. 667-685). Springer US.

**Return type** `Tensor`

## 1.26 Dice

### 1.26.1 Module Interface

```
class torchmetrics.Dice(zero_division=0, num_classes=None, threshold=0.5, average='micro',
                        mdmc_average='global', ignore_index=None, top_k=None, multiclass=None,
                        **kwargs)
```

Computes `Dice`:

$$\text{Dice} = \frac{2 * \text{TP}}{2 * \text{TP} + \text{FP} + \text{FN}}$$

Where TP and FP represent the number of true positives and false positives respectively.

It is recommend set `ignore_index` to index of background class.

The reduction method (how the precision scores are aggregated) is controlled by the `average` parameter, and additionally by the `mdmc_average` parameter in the multi-dimensional multi-class case. Accepts all inputs listed in *Input types*.

#### Parameters

- **num\_classes** (`Optional[int]`) – Number of classes. Necessary for 'macro', 'weighted' and None average methods.
- **threshold** (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
- **zero\_division** (`int`) – The value to use for the score if denominator equals zero.
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied. Should be one of the following:
  - 'micro' [default]: Calculate the metric globally, across all samples and classes.
  - 'macro': Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
  - 'weighted': Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (`tp + fn`).
  - 'none' or None: Calculate the metric for each class separately, and return the metric for every class.
  - 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

---

**Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_average`.

---

- **mdmc\_average** ([Optional\[str\]](#)) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
  - `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - `'samplewise'`: In this case, the statistics are computed separately for each sample on the `N` axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... (see [Input types](#)) as the `N` dimension within the sample, and computing the metric for the sample based on that.
  - `'global'`: In this case the `N` and ... dimensions of the inputs (see [Input types](#)) are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were `(N_X, C)`. From here on the `average` parameter applies as usual.
- **ignore\_index** ([Optional\[int\]](#)) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or `'none'`, the score for the ignored class will be returned as `nan`.
- **top\_k** ([Optional\[int\]](#)) – Number of the highest probability or logit score predictions considered finding the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (`None`) will be interpreted as 1 for these inputs. Should be left at default (`None`) for all other types of inputs.
- **multiclass** ([Optional\[bool\]](#)) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's [documentation section](#) for a more detailed explanation and examples.
- **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

- **ValueError** – If `average` is none of `"micro"`, `"macro"`, `"weighted"`, `"samples"`, `"none"`, `None`.
- **ValueError** – If `mdmc_average` is not one of `None`, `"samplewise"`, `"global"`.
- **ValueError** – If `average` is set but `num_classes` is not provided.
- **ValueError** – If `num_classes` is set and `ignore_index` is not in the range `[0, num_classes)`.

#### Example

```
>>> import torch
>>> from torchmetrics import Dice
>>> preds = torch.tensor([2, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> dice = Dice(average='micro')
>>> dice(preds, target)
tensor(0.2500)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

#### `compute()`

Computes the dice score based on inputs passed in to `update` previously.

#### Returns

- If `average` in `['micro', 'macro', 'weighted', 'samples']`, a one-element tensor will be returned
- If `average` in `['none', None]`, the shape will be `(C,)`, where `C` stands for the number of classes

**Return type** The shape of the returned tensor depends on the `average` parameter

## 1.26.2 Functional Interface

`torchmetrics.functional.dice(preds, target, zero_division=0, average='micro', mdmc_average='global', threshold=0.5, top_k=None, num_classes=None, multiclass=None, ignore_index=None)`

Computes [Dice](#):

$$\text{Dice} = \frac{2 * \text{TP}}{2 * \text{TP} + \text{FP} + \text{FN}}$$

Where TP and FN represent the number of true positives and false negatives respectively.

It is recommend set `ignore_index` to index of background class.

The reduction method (how the recall scores are aggregated) is controlled by the `average` parameter, and additionally by the `mdmc_average` parameter in the multi-dimensional multi-class case. Accepts all inputs listed in [Input types](#).

### Parameters

- **preds** ([Tensor](#)) – Predictions from model (probabilities, logits or labels)
- **target** ([Tensor](#)) – Ground truth values
- **zero\_division** ([int](#)) – The value to use for the score if denominator equals zero
- **average** ([Optional\[str\]](#)) – Defines the reduction that is applied. Should be one of the following:
  - `'micro'` [default]: Calculate the metric globally, across all samples and classes.
  - `'macro'`: Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
  - `'weighted'`: Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (`tp + fn`).
  - `'none'` or `None`: Calculate the metric for each class separately, and return the metric for every class.
  - `'samples'`: Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

---

**Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_average`.

---



---

**Note:** If `'none'` and a given class doesn't occur in the `preds` or `target`, the value for the class will be `nan`.

---

- **mdmc\_average** ([Optional\[str\]](#)) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
  - `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - `'samplewise'`: In this case, the statistics are computed separately for each sample on the `N` axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... (see [Input types](#)) as the `N` dimension within the sample, and computing the metric for the sample based on that.
  - `'global'`: In this case the `N` and ... dimensions of the inputs (see [Input types](#)) are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were `(N_X, C)`. From here on the `average` parameter applies as usual.
- **ignore\_index** ([Optional\[int\]](#)) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or `'none'`, the score for the ignored class will be returned as `nan`.
- **num\_classes** ([Optional\[int\]](#)) – Number of classes. Necessary for `'macro'`, `'weighted'` and `None` average methods.
- **threshold** ([float](#)) – Threshold for transforming probability or logit predictions to binary `(0,1)` predictions, in the case of binary or multi-label inputs. Default value of `0.5` corresponds to input being probabilities.
- **top\_k** ([Optional\[int\]](#)) – Number of the highest probability or logit score predictions considered finding the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (`None`) will be interpreted as `1` for these inputs.

Should be left at default (`None`) for all other types of inputs.
- **multiclass** ([Optional\[bool\]](#)) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's [documentation section](#) for a more detailed explanation and examples.

**Return type** [Tensor](#)

#### Returns

The shape of the returned tensor depends on the `average` parameter

- If `average` in `['micro', 'macro', 'weighted', 'samples']`, a one-element tensor will be returned
- If `average` in `['none', None]`, the shape will be `(C,)`, where `C` stands for the number of classes

#### Raises

- **ValueError** – If `average` is not one of `"micro", "macro", "weighted", "samples", "none"` or `None`
- **ValueError** – If `mdmc_average` is not one of `None, "samplewise", "global"`.
- **ValueError** – If `average` is set but `num_classes` is not provided.
- **ValueError** – If `num_classes` is set and `ignore_index` is not in the range `[0, num_classes)`.

### Example

```
>>> from torchmetrics.functional import dice
>>> preds = torch.tensor([2, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> dice(preds, target, average='micro')
tensor(0.2500)
```

## 1.27 Dice Score

### 1.27.1 Functional Interface (was deprecated in v0.9)

`torchmetrics.functional.dice_score(preds, target, bg=False, nan_score=0.0, no_fg_score=0.0, reduction='elementwise_mean')`

Compute dice score from prediction scores.

Supports only “macro” approach, which mean calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).

Deprecated since version v0.9: The *dice\_score* function was deprecated in v0.9 and will be removed in v0.10. Use *dice* function instead.

#### Parameters

- **preds** (`Tensor`) – estimated probabilities
- **target** (`Tensor`) – ground-truth labels
- **bg** (`bool`) – whether to also compute dice for the background
- **nan\_score** (`float`) – score to return, if a NaN occurs during computation
- **no\_fg\_score** (`float`) – (default, 0.0) score to return, if no foreground pixel was found in target

Deprecated since version v0.9: All different from default options will be changed to default.

- **reduction** (`Literal`['elementwise\_mean', 'sum', 'none', None]) – (default, 'elementwise\_mean') a method to reduce metric score over labels.

Deprecated since version v0.9: All different from default options will be changed to default.

- 'elementwise\_mean': takes the mean (default)
- 'sum': takes the sum
- 'none' or None: no reduction will be applied

**Return type** `Tensor`

**Returns** Tensor containing dice score

### Example

```
>>> from torchmetrics.functional import dice_score
>>> pred = torch.tensor([[0.85, 0.05, 0.05, 0.05],
...                      [0.05, 0.85, 0.05, 0.05],
...                      [0.05, 0.05, 0.85, 0.05],
...                      [0.05, 0.05, 0.05, 0.85]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> dice_score(pred, target)
tensor(0.3333)
```

## 1.28 Exact Match

### 1.28.1 Module Interface

#### MultilabelExactMatch

```
class torchmetrics.classification.MultilabelExactMatch(num_labels, threshold=0.5,
                                                         multidim_average='global',
                                                         ignore_index=None, validate_args=True,
                                                         **kwargs)
```

Computes Exact match (also known as subset accuracy) for multilabel tasks. Exact Match is a stricter version of accuracy where all labels have to match exactly for the sample to be correctly classified.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

#### Parameters

- **num\_labels** (**int**) – Integer specifying the number of labels
- **threshold** (**float**) – Threshold for transforming probability to binary (0,1) predictions
- **multidim\_average** (**Literal**['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (**Optional**[**int**]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (**bool**) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

#### Returns

- If *multidim\_average* is set to **global**:

- If average='micro'/'macro'/'weighted', the output will be a scalar tensor
- If average=None/'none', the shape will be (C,)
- If multidim\_average is set to samplewise:
  - If average='micro'/'macro'/'weighted', the shape will be (N,)
  - If average=None/'none', the shape will be (N, C)

**Return type** The returned shape depends on the average and multidim\_average arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import MultilabelExactMatch
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelExactMatch(num_labels=3)
>>> metric(preds, target)
tensor(0.5000)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import MultilabelExactMatch
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelExactMatch(num_labels=3)
>>> metric(preds, target)
tensor(0.5000)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import MultilabelExactMatch
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> metric = MultilabelExactMatch(num_labels=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0., 0.] )
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tensor`

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Tensor with predictions

- **target** (`Tensor`) – Tensor with true labels

Return type `None`

## 1.28.2 Functional Interface

### `multilabel_exact_match`

```
torchmetrics.functional.classification.multilabel_exact_match(preds, target, num_labels,
                                                             threshold=0.5,
                                                             multidim_average='global',
                                                             ignore_index=None,
                                                             validate_args=True)
```

Computes Exact match (also known as subset accuracy) for multilabel tasks. Exact Match is a stricter version of accuracy where all labels have to match exactly for the sample to be correctly classified.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

#### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **num\_labels** (`int`) – Integer specifying the number of labels
- **threshold** (`float`) – Threshold for transforming probability to binary (0,1) predictions
- **multidim\_average** (`Literal`['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional`[`int`]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

#### Returns

- If *multidim\_average* is set to **global**:
  - If *average*='micro'/'macro'/'weighted', the output will be a scalar tensor
  - If *average*=None/'none', the shape will be (C,)
- If *multidim\_average* is set to **samplewise**:
  - If *average*='micro'/'macro'/'weighted', the shape will be (N,)
  - If *average*=None/'none', the shape will be (N, C)



**Return type** The returned shape depends on the average and multidim\_average arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_exact_match
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_exact_match(preds, target, num_labels=3)
tensor(0.5000)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_exact_match
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_exact_match(preds, target, num_labels=3)
tensor(0.5000)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import multilabel_exact_match
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> multilabel_exact_match(preds, target, num_labels=3, multidim_average=
↪ 'samplewise')
tensor([0., 0.])
```

## 1.29 F-1 Score

### 1.29.1 Module Interface

#### F1Score

```
class torchmetrics.F1Score(num_classes=None, threshold=0.5, average='micro', mdmc_average=None,
                           ignore_index=None, top_k=None, multiclass=None, **kwargs)
```

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

Computes F1 metric.

F1 metrics correspond to a harmonic mean of the precision and recall scores. Works with binary, multiclass, and multilabel data. Accepts logits or probabilities from a model output or integer class values in prediction. Works with multi-dimensional preds and target.

Forward accepts

- **preds** (float or long tensor): (N, ...) or (N, C, ...) where C is the number of classes
- **target** (long tensor): (N, ...)

If preds and target are the same shape and preds is a float tensor, we use the `self.threshold` argument. This is the case for binary and multi-label logits.

If preds has an extra dimension as in the case of multi-class scores we perform an `argmax` on `dim=1`.

#### Parameters

- **num\_classes** (`Optional[int]`) – Number of classes. Necessary for 'macro', 'weighted' and None average methods.
- **threshold** (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied. Should be one of the following:
  - 'micro' [default]: Calculate the metric globally, across all samples and classes.
  - 'macro': Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
  - 'weighted': Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (`tp + fn`).
  - 'none' or None: Calculate the metric for each class separately, and return the metric for every class.
  - 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

---

**Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_average`.

---

- **mdmc\_average** (`Optional[str]`) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
  - None [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - 'samplewise': In this case, the statistics are computed separately for each sample on the N axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... (see *Input types*) as the N dimension within the sample, and computing the metric for the sample based on that.
  - 'global': In this case the N and ... dimensions of the inputs (see *Input types*) are flattened into a new N\_X sample axis, i.e. the inputs are treated as if they were (N\_X, C). From here on the `average` parameter applies as usual.
- **ignore\_index** (`Optional[int]`) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or 'none', the score for the ignored class will be returned as nan.

- **top\_k** (`Optional[int]`) – Number of the highest probability or logit score predictions considered finding the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (`None`) will be interpreted as 1 for these inputs. Should be left at default (`None`) for all other types of inputs.
- **multiclass** (`Optional[bool]`) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter’s [documentation section](#) for a more detailed explanation and examples.
- **kwargs** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> import torch
>>> from torchmetrics import F1Score
>>> target = torch.tensor([0, 1, 2, 0, 1, 2])
>>> preds = torch.tensor([0, 2, 1, 0, 0, 1])
>>> f1 = F1Score(num_classes=3)
>>> f1(preds, target)
tensor(0.3333)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

### BinaryF1Score

```
class torchmetrics.classification.BinaryF1Score(threshold=0.5, multidim_average='global',
                                                ignore_index=None, validate_args=True, **kwargs)
```

Computes F-1 score for binary tasks:

$$F_1 = 2 \frac{\text{precision} * \text{recall}}{(\text{precision}) + \text{recall}}$$

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

#### Parameters

- **threshold** (`float`) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Returns** If `multidim_average` is set to `global`, the metric returns a scalar value. If `multidim_average` is set to `samplewise`, the metric returns (N,) vector consisting of a scalar value per sample.

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import BinaryF1Score
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0, 0, 1, 1, 0, 1])
>>> metric = BinaryF1Score()
>>> metric(preds, target)
tensor(0.6667)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import BinaryF1Score
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> metric = BinaryF1Score()
>>> metric(preds, target)
tensor(0.6667)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import BinaryF1Score
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> metric = BinaryF1Score(multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.5000, 0.0000])
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

## MulticlassF1Score

```
class torchmetrics.classification.MulticlassF1Score(num_classes, top_k=1, average='macro',
                                                    multidim_average='global', ignore_index=None,
                                                    validate_args=True, **kwargs)
```

Computes F-1 score for multiclass tasks:

$$F_1 = 2 \frac{\text{precision} * \text{recall}}{(\text{precision}) + \text{recall}}$$

Accepts the following input tensors:

- `preds`: (N, ...) (int tensor) or (N, C, ...) (float tensor). If `preds` is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- `target` (int tensor): (N, ...)

The influence of the additional dimension . . . (if present) will be determined by the *multidim\_average* argument.

### Parameters

- **preds** – Tensor with predictions
- **target** – Tensor with true labels
- **num\_classes** (`int`) – Integer specifying the number of classes
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: Calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: Calculates statistic for each label and applies no reduction
- **top\_k** (`int`) – Number of highest probability or logit score predictions considered to find the correct label. Only works when `preds` contain probabilities/logits.
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions . . . should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the `N` axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be `(C,)`
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

**Return type** The returned shape depends on the `average` and `multidim_average` arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import MulticlassF1Score
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> metric = MulticlassF1Score(num_classes=3)
>>> metric(preds, target)
tensor(0.7778)
>>> metric = MulticlassF1Score(num_classes=3, average=None)
```

(continues on next page)

(continued from previous page)

```
>>> metric(preds, target)
tensor([0.6667, 0.6667, 1.0000])
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import MulticlassF1Score
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> metric = MulticlassF1Score(num_classes=3)
>>> metric(preds, target)
tensor(0.7778)
>>> metric = MulticlassF1Score(num_classes=3, average=None)
>>> metric(preds, target)
tensor([0.6667, 0.6667, 1.0000])
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import MulticlassF1Score
>>> target = torch.tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = torch.tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> metric = MulticlassF1Score(num_classes=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.4333, 0.2667])
>>> metric = MulticlassF1Score(num_classes=3, multidim_average='samplewise',
...                             average=None)
>>> metric(preds, target)
tensor([[0.8000, 0.0000, 0.5000],
        [0.0000, 0.4000, 0.4000]])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

## MultilabelF1Score

```
class torchmetrics.classification.MultilabelF1Score(num_labels, threshold=0.5, average='macro',
                                                    multidim_average='global', ignore_index=None,
                                                    validate_args=True, **kwargs)
```

Computes F-1 score for multilabel tasks:

$$F_1 = 2 \frac{\text{precision} * \text{recall}}{(\text{precision}) + \text{recall}}$$

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

The influence of the additional dimension . . . (if present) will be determined by the *multidim\_average* argument.

### Parameters

- **num\_labels** (`int`) – Integer specifying the number of labels
- **threshold** (`float`) – Threshold for transforming probability to binary (0,1) predictions
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: Calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: Calculates statistic for each label and applies no reduction
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions . . . should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be `(C,)`
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

**Return type** The returned shape depends on the `average` and `multidim_average` arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import MultilabelF1Score
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelF1Score(num_labels=3)
>>> metric(preds, target)
tensor(0.5556)
>>> metric = MultilabelF1Score(num_labels=3, average=None)
>>> metric(preds, target)
tensor([1.0000, 0.0000, 0.6667])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import MultilabelF1Score
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelF1Score(num_labels=3)
>>> metric(preds, target)
tensor(0.5556)
>>> metric = MultilabelF1Score(num_labels=3, average=None)
>>> metric(preds, target)
tensor([1.0000, 0.0000, 0.6667])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import MultilabelF1Score
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.78], [0.45, 0.37]],
...     ]
... )
>>> metric = MultilabelF1Score(num_labels=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.4444, 0.0000])
>>> metric = MultilabelF1Score(num_labels=3, multidim_average='samplewise',
...                             average=None)
>>> metric(preds, target)
tensor([[0.6667, 0.6667, 0.0000],
        [0.0000, 0.0000, 0.0000]])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

## 1.29.2 Functional Interface

### f1\_score

`torchmetrics.functional.f1_score(preds, target, beta=1.0, average='micro', mdc_average=None, ignore_index=None, num_classes=None, threshold=0.5, top_k=None, multiclass=None, task=None, num_labels=None, multidim_average='global', validate_args=True)`

---

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as a single entrypoint to calling the three specialized versions.

---

Computes F1 metric. F1 metrics correspond to a equally weighted average of the precision and recall scores.

Works with binary, multiclass, and multilabel data. Accepts probabilities or logits from a model output or integer class values in prediction. Works with multi-dimensional preds and target.



If `preds` and `target` are the same shape and `preds` is a float tensor, we use the `self.threshold` argument to convert into integer labels. This is the case for binary and multi-label probabilities or logits.

If `preds` has an extra dimension as in the case of multi-class scores we perform an `argmax` on `dim=1`.

The reduction method (how the precision scores are aggregated) is controlled by the `average` parameter, and additionally by the `mdmc_average` parameter in the multi-dimensional multi-class case. Accepts all inputs listed in *Input types*.

### Parameters

- **preds** (`Tensor`) – Predictions from model (probabilities, logits or labels)
- **target** (`Tensor`) – Ground truth values
- **beta** (`float`) – it is ignored
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied. Should be one of the following:
  - 'micro' [default]: Calculate the metric globally, across all samples and classes.
  - 'macro': Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
  - 'weighted': Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (`tp + fn`).
  - 'none' or `None`: Calculate the metric for each class separately, and return the metric for every class.
  - 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

---

**Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_average`.

---



---

**Note:** If 'none' and a given class doesn't occur in the `preds` or `target`, the value for the class will be `nan`.

---

- **mdmc\_average** (`Optional[str]`) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
  - `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - 'samplewise': In this case, the statistics are computed separately for each sample on the `N` axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... (see *Input types*) as the `N` dimension within the sample, and computing the metric for the sample based on that.
  - 'global': In this case the `N` and ... dimensions of the inputs (see *Input types*) are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were `(N_X, C)`. From here on the `average` parameter applies as usual.
- **ignore\_index** (`Optional[int]`) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or 'none', the score for the ignored class will be returned as `nan`.

- **num\_classes** (`Optional[int]`) – Number of classes. Necessary for 'macro', 'weighted' and None average methods.
- **threshold** (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
- **top\_k** (`Optional[int]`) – Number of highest probability or logit score predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (None) will be interpreted as 1 for these inputs.  
Should be left at default (None) for all other types of inputs.
- **multiclass** (`Optional[bool]`) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's [documentation section](#) for a more detailed explanation and examples.

**Return type** `Tensor`

#### Returns

The shape of the returned tensor depends on the average parameter

- If average in ['micro', 'macro', 'weighted', 'samples'], a one-element tensor will be returned
- If average in ['none', None], the shape will be (C,), where C stands for the number of classes

#### Example

```
>>> from torchmetrics.functional import f1_score
>>> target = torch.tensor([0, 1, 2, 0, 1, 2])
>>> preds = torch.tensor([0, 2, 1, 0, 0, 1])
>>> f1_score(preds, target, num_classes=3)
tensor(0.3333)
```

### binary\_f1\_score

```
torchmetrics.functional.classification.binary_f1_score(preds, target, threshold=0.5,
                                                         multidim_average='global',
                                                         ignore_index=None, validate_args=True)
```

Computes F-1 score for binary tasks:

$$F_1 = 2 \frac{\text{precision} * \text{recall}}{(\text{precision}) + \text{recall}}$$

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

#### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **threshold** (`float`) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (`Literal`['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional`[`int`]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Return type** `Tensor`

**Returns** If `multidim_average` is set to `global`, the metric returns a scalar value. If `multidim_average` is set to `samplewise`, the metric returns (N,) vector consisting of a scalar value per sample.

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import binary_f1_score
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0, 0, 1, 1, 0, 1])
>>> binary_f1_score(preds, target)
tensor(0.6667)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import binary_f1_score
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> binary_f1_score(preds, target)
tensor(0.6667)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import binary_f1_score
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> binary_f1_score(preds, target, multidim_average='samplewise')
tensor([0.5000, 0.0000])
```

## multiclass\_f1\_score

```
torchmetrics.functional.classification.multiclass_f1_score(preds, target, num_classes,
                                                           average='macro', top_k=1,
                                                           multidim_average='global',
                                                           ignore_index=None,
                                                           validate_args=True)
```

Computes F-1 score for multiclass tasks:

$$F_1 = 2 \frac{\text{precision} * \text{recall}}{(\text{precision}) + \text{recall}}$$

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: Calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: Calculates statistic for each label and applies no reduction
- **top\_k** ([int](#)) – Number of highest probability or logit score predictions considered to find the correct label. Only works when **preds** contain probabilities/logits.
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

### Returns

- If **multidim\_average** is set to **global**:
  - If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor
  - If **average**=None/'none', the shape will be (C,)

- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

**Return type** The returned shape depends on the `average` and `multidim_average` arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_f1_score
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> multiclass_f1_score(preds, target, num_classes=3)
tensor(0.7778)
>>> multiclass_f1_score(preds, target, num_classes=3, average=None)
tensor([0.6667, 0.6667, 1.0000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_f1_score
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> multiclass_f1_score(preds, target, num_classes=3)
tensor(0.7778)
>>> multiclass_f1_score(preds, target, num_classes=3, average=None)
tensor([0.6667, 0.6667, 1.0000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import multiclass_f1_score
>>> target = torch.tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = torch.tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> multiclass_f1_score(preds, target, num_classes=3, multidim_average=
↳ 'samplewise')
tensor([0.4333, 0.2667])
>>> multiclass_f1_score(preds, target, num_classes=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[0.8000, 0.0000, 0.5000],
        [0.0000, 0.4000, 0.4000]])
```

## multilabel\_f1\_score

```
torchmetrics.functional.classification.multilabel_f1_score(preds, target, num_labels,
                                                           threshold=0.5, average='macro',
                                                           multidim_average='global',
                                                           ignore_index=None,
                                                           validate_args=True)
```

Computes F-1 score for multilabel tasks:

$$F_1 = 2 \frac{\text{precision} * \text{recall}}{(\text{precision}) + \text{recall}}$$

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: Calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: Calculates statistic for each label and applies no reduction
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

### Returns

- If **multidim\_average** is set to **global**:
  - If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor
  - If **average**=None/'none', the shape will be (C,)

- If multidim\_average is set to samplewise:
  - If average='micro'/'macro'/'weighted', the shape will be (N,)
  - If average=None/'none', the shape will be (N, C)

**Return type** The returned shape depends on the average and multidim\_average arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_f1_score
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_f1_score(preds, target, num_labels=3)
tensor(0.5556)
>>> multilabel_f1_score(preds, target, num_labels=3, average=None)
tensor([1.0000, 0.0000, 0.6667])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_f1_score
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_f1_score(preds, target, num_labels=3)
tensor(0.5556)
>>> multilabel_f1_score(preds, target, num_labels=3, average=None)
tensor([1.0000, 0.0000, 0.6667])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import multilabel_f1_score
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> multilabel_f1_score(preds, target, num_labels=3, multidim_average=
↳ 'samplewise')
tensor([0.4444, 0.0000])
>>> multilabel_f1_score(preds, target, num_labels=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[0.6667, 0.6667, 0.0000],
        [0.0000, 0.0000, 0.0000]])
```

## 1.30 F-Beta Score

### 1.30.1 Module Interface

#### FBetaScore

```
class torchmetrics.FBetaScore(num_classes=None, beta=1.0, threshold=0.5, average='micro',
                              mdmc_average=None, ignore_index=None, top_k=None, multiclass=None,
                              **kwargs)
```

---

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

---

Computes [F-score](#), specifically:

$$F_{\beta} = (1 + \beta^2) * \frac{\text{precision} * \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}}$$

Where  $\beta$  is some positive real factor. Works with binary, multiclass, and multilabel data. Accepts logit scores or probabilities from a model output or integer class values in prediction. Works with multi-dimensional preds and target.

Forward accepts

- **preds** (float or long tensor): (N, ...) or (N, C, ...) where C is the number of classes
- **target** (long tensor): (N, ...)

If preds and target are the same shape and preds is a float tensor, we use the `self.threshold` argument to convert into integer labels. This is the case for binary and multi-label logits and probabilities.

If preds has an extra dimension as in the case of multi-class scores we perform an `argmax` on `dim=1`.

#### Parameters

- **num\_classes** ([Optional\[int\]](#)) – Number of classes. Necessary for 'macro', 'weighted' and None average methods.
- **beta** ([float](#)) – Beta coefficient in the F measure.
- **threshold** ([float](#)) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
- **average** ([Optional\[Literal\['micro', 'macro', 'weighted', 'none'\]\]](#)) – Defines the reduction that is applied. Should be one of the following:
  - 'micro' [default]: Calculate the metric globally, across all samples and classes.
  - 'macro': Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
  - 'weighted': Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (`tp + fn`).



- 'none' or None: Calculate the metric for each class separately, and return the metric for every class.
- 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

---

**Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_average`.

---



---

**Note:** If 'none' and a given class doesn't occur in the `preds` or `target`, the value for the class will be nan.

---

- **mdmc\_average** (Optional[str]) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
  - None [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - 'samplewise': In this case, the statistics are computed separately for each sample on the N axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... (see [Input types](#)) as the N dimension within the sample, and computing the metric for the sample based on that.
  - 'global': In this case the N and ... dimensions of the inputs (see [Input types](#)) are flattened into a new N\_X sample axis, i.e. the inputs are treated as if they were (N\_X, C). From here on the `average` parameter applies as usual.
- **ignore\_index** (Optional[int]) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or 'none', the score for the ignored class will be returned as nan.
- **top\_k** (Optional[int]) – Number of the highest probability or logit score predictions considered finding the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (None) will be interpreted as 1 for these inputs.  
Should be left at default (None) for all other types of inputs.
- **multiclass** (Optional[bool]) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's [documentation section](#) for a more detailed explanation and examples.
- **kwargs** (Any) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises** `ValueError` – If `average` is none of "micro", "macro", "weighted", "none", None.

### Example

```
>>> import torch
>>> from torchmetrics import FBetaScore
>>> target = torch.tensor([0, 1, 2, 0, 1, 2])
>>> preds = torch.tensor([0, 2, 1, 0, 0, 1])
>>> f_beta = FBetaScore(num_classes=3, beta=0.5)
>>> f_beta(preds, target)
tensor(0.3333)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes f-beta over state.

**Return type** `Tensor`

## BinaryFBetaScore

```
class torchmetrics.classification.BinaryFBetaScore(beta, threshold=0.5, multidim_average='global',
                                                    ignore_index=None, validate_args=True,
                                                    **kwargs)
```

Computes `F-score` metric for binary tasks:

$$F_{\beta} = (1 + \beta^2) * \frac{\text{precision} * \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}}$$

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

### Parameters

- **beta** (`float`) – Weighting between precision and recall in calculation. Setting to 1 corresponds to equal weight
- **threshold** (`float`) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (`Literal`['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional`[`int`]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Returns** If *multidim\_average* is set to *global*, the metric returns a scalar value. If *multidim\_average* is set to *samplewise*, the metric returns (N,) vector consisting of a scalar value per sample.

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import BinaryFBetaScore
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0, 0, 1, 1, 0, 1])
>>> metric = BinaryFBetaScore(beta=2.0)
>>> metric(preds, target)
tensor(0.6667)
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import BinaryFBetaScore
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> metric = BinaryFBetaScore(beta=2.0)
>>> metric(preds, target)
tensor(0.6667)
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import BinaryFBetaScore
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.78], [0.45, 0.37]],
...     ]
... )
>>> metric = BinaryFBetaScore(beta=2.0, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.5882, 0.0000])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes the final statistics.

**Return type** `Tensor`

**Returns**

The metric returns a tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to  $[tp, fp, tn, fn, sup]$  ( $sup$  stands for support and equals  $tp + fn$ ). The shape depends on the `multidim_average` parameter:

- If `multidim_average` is set to `global`, the shape will be  $(5,)$
- If `multidim_average` is set to `samplewise`, the shape will be  $(N, 5)$

## MulticlassFBetaScore

```
class torchmetrics.classification.MulticlassFBetaScore(beta, num_classes, top_k=1,
                                                       average='macro',
                                                       multidim_average='global',
                                                       ignore_index=None, validate_args=True,
                                                       **kwargs)
```

Computes `F-score` metric for multiclass tasks:

$$F_{\beta} = (1 + \beta^2) * \frac{\text{precision} * \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}}$$

Accepts the following input tensors:

- `preds`:  $(N, \dots)$  (int tensor) or  $(N, C, \dots)$  (float tensor). If `preds` is a floating point we apply `torch.argmax` along the `C` dimension to automatically convert probabilities/logits into an int tensor.

- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

#### Parameters

- **beta** (float) – Weighting between precision and recall in calculation. Setting to 1 corresponds to equal weight
- **num\_classes** (int) – Integer specifying the number of classes
- **average** (Optional[Literal['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - micro: Sum statistics over all labels
  - macro: Calculate statistics for each label and average them
  - weighted: Calculates statistics for each label and computes weighted average using their support
  - "none" or None: Calculates statistic for each label and applies no reduction
- **top\_k** (int) – Number of highest probability or logit score predictions considered to find the correct label. Only works when preds contain probabilities/logits.
- **multidim\_average** (Literal['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - global: Additional dimensions are flattened along the batch dimension
  - samplewise: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (Optional[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (bool) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.

#### Returns

- If multidim\_average is set to global:
  - If average='micro'/'macro'/'weighted', the output will be a scalar tensor
  - If average=None/'none', the shape will be (C,)
- If multidim\_average is set to samplewise:
  - If average='micro'/'macro'/'weighted', the shape will be (N,)
  - If average=None/'none', the shape will be (N, C)

**Return type** The returned shape depends on the average and multidim\_average arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import MulticlassFBetaScore
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> metric = MulticlassFBetaScore(beta=2.0, num_classes=3)
>>> metric(preds, target)
tensor(0.7963)
>>> metric = MulticlassFBetaScore(beta=2.0, num_classes=3, average=None)
```

(continues on next page)

(continued from previous page)

```
>>> metric(preds, target)
tensor([0.5556, 0.8333, 1.0000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import MulticlassFBetaScore
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> metric = MulticlassFBetaScore(beta=2.0, num_classes=3)
>>> metric(preds, target)
tensor(0.7963)
>>> metric = MulticlassFBetaScore(beta=2.0, num_classes=3, average=None)
>>> metric(preds, target)
tensor([0.5556, 0.8333, 1.0000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import MulticlassFBetaScore
>>> target = torch.tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = torch.tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> metric = MulticlassFBetaScore(beta=2.0, num_classes=3, multidim_average=
↳ 'samplewise')
>>> metric(preds, target)
tensor([0.4697, 0.2706])
>>> metric = MulticlassFBetaScore(beta=2.0, num_classes=3, multidim_average=
↳ 'samplewise', average=None)
>>> metric(preds, target)
tensor([[0.9091, 0.0000, 0.5000],
        [0.0000, 0.3571, 0.4545]])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes the final statistics.

**Return type** `Tensor`

**Returns**

The metric returns a tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to  $[tp, fp, tn, fn, sup]$  ( $sup$  stands for support and equals  $tp + fn$ ). The shape depends on `average` and `multidim_average` parameters:

- If `multidim_average` is set to `global`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(5,)$
- If `average=None/'none'`, the shape will be  $(C, 5)$
- If `multidim_average` is set to `samplewise`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(N, 5)$

- If `average=None/'none'`, the shape will be `(N, C, 5)`

## MultilabelFBetaScore

```
class torchmetrics.classification.MultilabelFBetaScore(beta, num_labels, threshold=0.5,
                                                       average='macro',
                                                       multidim_average='global',
                                                       ignore_index=None, validate_args=True,
                                                       **kwargs)
```

Computes [F-score](#) metric for multilabel tasks:

$$F_{\beta} = (1 + \beta^2) * \frac{\text{precision} * \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}}$$

Accepts the following input tensors:

- **preds** (int or float tensor): `(N, C, ...)`. If preds is a floating point tensor with values outside `[0,1]` range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **target** (int tensor): `(N, C, ...)`

The influence of the additional dimension ... (if present) will be determined by the `multidim_average` argument.

### Parameters

- **beta** ([float](#)) – Weighting between precision and recall in calculation. Setting to 1 corresponds to equal weight
- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: Calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: Calculates statistic for each label and applies no reduction
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the `N` axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor

- If average=None/'none', the shape will be (C,)
- If multidim\_average is set to samplewise:
  - If average='micro'/'macro'/'weighted', the shape will be (N,)
  - If average=None/'none', the shape will be (N, C)

**Return type** The returned shape depends on the average and multidim\_average arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import MultilabelFBetaScore
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelFBetaScore(beta=2.0, num_labels=3)
>>> metric(preds, target)
tensor(0.6111)
>>> metric = MultilabelFBetaScore(beta=2.0, num_labels=3, average=None)
>>> metric(preds, target)
tensor([1.0000, 0.0000, 0.8333])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import MultilabelFBetaScore
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelFBetaScore(beta=2.0, num_labels=3)
>>> metric(preds, target)
tensor(0.6111)
>>> metric = MultilabelFBetaScore(beta=2.0, num_labels=3, average=None)
>>> metric(preds, target)
tensor([1.0000, 0.0000, 0.8333])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import MultilabelFBetaScore
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> metric = MultilabelFBetaScore(num_labels=3, beta=2.0, multidim_average=
↳ 'samplewise')
>>> metric(preds, target)
tensor([0.5556, 0.0000])
>>> metric = MultilabelFBetaScore(num_labels=3, beta=2.0, multidim_average=
↳ 'samplewise', average=None)
>>> metric(preds, target)
tensor([[0.8333, 0.8333, 0.0000],
        [0.0000, 0.0000, 0.0000]])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes the final statistics.

**Return type** `Tensor`

**Returns**

The metric returns a tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to  $[tp, fp, tn, fn, sup]$  (sup stands for support and equals  $tp + fn$ ). The shape depends on `average` and `multidim_average` parameters:

- If `multidim_average` is set to `global`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(5,)$
- If `average=None/'none'`, the shape will be  $(C, 5)$
- If `multidim_average` is set to `samplewise`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(N, 5)$
- If `average=None/'none'`, the shape will be  $(N, C, 5)$

## 1.30.2 Functional Interface

**fbeta\_score**

```
torchmetrics.functional.fbeta_score(preds, target, beta=1.0, average='micro', mdc_average=None,
                                     ignore_index=None, num_classes=None, threshold=0.5,
                                     top_k=None, multiclass=None, task=None, num_labels=None,
                                     multidim_average='global', validate_args=True)
```

---

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

---

Computes  $f_{\beta}$  metric.

$$F_{\beta} = (1 + \beta^2) * \frac{\text{precision} * \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}}$$

Works with binary, multiclass, and multilabel data. Accepts probabilities or logits from a model output or integer class values in prediction. Works with multi-dimensional preds and target.

If preds and target are the same shape and preds is a float tensor, we use the `self.threshold` argument to convert into integer labels. This is the case for binary and multi-label logits or probabilities.

If preds has an extra dimension as in the case of multi-class scores we perform an `argmax` on `dim=1`.

The reduction method (how the precision scores are aggregated) is controlled by the `average` parameter, and additionally by the `mdmc_average` parameter in the multi-dimensional multi-class case. Accepts all inputs listed in *Input types*.

**Parameters**

- **preds** (`Tensor`) – Predictions from model (probabilities, logits or labels)
- **target** (`Tensor`) – Ground truth values



- **beta** (`float`) – beta coefficient
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied. Should be one of the following:
  - 'micro' [default]: Calculate the metric globally, across all samples and classes.
  - 'macro': Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
  - 'weighted': Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support ( $tp + fn$ ).
  - 'none' or `None`: Calculate the metric for each class separately, and return the metric for every class.
  - 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

---

**Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_average`.

---



---

**Note:** If 'none' and a given class doesn't occur in the `preds` or `target`, the value for the class will be `nan`.

---

- **mdmc\_average** (`Optional[str]`) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
  - `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - 'samplewise': In this case, the statistics are computed separately for each sample on the `N` axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... (see *Input types*) as the `N` dimension within the sample, and computing the metric for the sample based on that.
  - 'global': In this case the `N` and ... dimensions of the inputs (see *Input types*) are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were `(N_X, C)`. From here on the `average` parameter applies as usual.
- **ignore\_index** (`Optional[int]`) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or 'none', the score for the ignored class will be returned as `nan`.
- **num\_classes** (`Optional[int]`) – Number of classes. Necessary for 'macro', 'weighted' and `None` average methods.
- **threshold** (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
- **top\_k** (`Optional[int]`) – Number of highest probability or logit score predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (`None`) will be interpreted as 1 for these inputs.  
Should be left at default (`None`) for all other types of inputs.

- **multiclass** (Optional[bool]) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter’s [documentation section](#) for a more detailed explanation and examples.

**Return type** `Tensor`

#### Returns

The shape of the returned tensor depends on the average parameter

- If average in ['micro', 'macro', 'weighted', 'samples'], a one-element tensor will be returned
- If average in ['none', None], the shape will be (C,), where C stands for the number of classes

#### Example

```
>>> from torchmetrics.functional import fbeta_score
>>> target = torch.tensor([0, 1, 2, 0, 1, 2])
>>> preds = torch.tensor([0, 2, 1, 0, 0, 1])
>>> fbeta_score(preds, target, num_classes=3, beta=0.5)
tensor(0.3333)
```

### binary\_fbeta\_score

`torchmetrics.functional.classification.binary_fbeta_score(preds, target, beta, threshold=0.5, multidim_average='global', ignore_index=None, validate_args=True)`

Computes [F-score](#) metric for binary tasks:

$$F_{\beta} = (1 + \beta^2) * \frac{\text{precision} * \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}}$$

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

#### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **beta** (`float`) – Weighting between precision and recall in calculation. Setting to 1 corresponds to equal weight
- **threshold** (`float`) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (`Literal`['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension

- **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Return type** `Tensor`

**Returns** If `multidim_average` is set to `global`, the metric returns a scalar value. If `multidim_average` is set to `samplewise`, the metric returns (N,) vector consisting of a scalar value per sample.

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import binary_fbeta_score
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0, 0, 1, 1, 0, 1])
>>> binary_fbeta_score(preds, target, beta=2.0)
tensor(0.6667)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import binary_fbeta_score
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> binary_fbeta_score(preds, target, beta=2.0)
tensor(0.6667)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import binary_fbeta_score
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> binary_fbeta_score(preds, target, beta=2.0, multidim_average='samplewise')
tensor([0.5882, 0.0000])
```

## multiclass\_fbeta\_score

`torchmetrics.functional.classification.multiclass_fbeta_score(preds, target, beta, num_classes, average='macro', top_k=1, multidim_average='global', ignore_index=None, validate_args=True)`

Computes **F-score** metric for multiclass tasks:

$$F_{\beta} = (1 + \beta^2) * \frac{\text{precision} * \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}}$$

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

#### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **beta** ([float](#)) – Weighting between precision and recall in calculation. Setting to 1 corresponds to equal weight
- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: Calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: Calculates statistic for each label and applies no reduction
- **top\_k** ([int](#)) – Number of highest probability or logit score predictions considered to find the correct label. Only works when **preds** contain probabilities/logits.
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

#### Returns

- If **multidim\_average** is set to **global**:
  - If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor
  - If **average**=None/'none', the shape will be (C,)
- If **multidim\_average** is set to **samplewise**:
  - If **average**='micro'/'macro'/'weighted', the shape will be (N,)
  - If **average**=None/'none', the shape will be (N, C)

**Return type** The returned shape depends on the **average** and **multidim\_average** arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_fbeta_score
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> multiclass_fbeta_score(preds, target, beta=2.0, num_classes=3)
tensor(0.7963)
>>> multiclass_fbeta_score(preds, target, beta=2.0, num_classes=3, average=None)
tensor([0.5556, 0.8333, 1.0000])
```

Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import multiclass_fbeta_score
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> multiclass_fbeta_score(preds, target, beta=2.0, num_classes=3)
tensor(0.7963)
>>> multiclass_fbeta_score(preds, target, beta=2.0, num_classes=3, average=None)
tensor([0.5556, 0.8333, 1.0000])
```

Example (multidim tensors):

```
>>> from torchmetrics.functional.classification import multiclass_fbeta_score
>>> target = torch.tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = torch.tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> multiclass_fbeta_score(preds, target, beta=2.0, num_classes=3, multidim_
↪average='samplewise')
tensor([0.4697, 0.2706])
>>> multiclass_fbeta_score(preds, target, beta=2.0, num_classes=3, multidim_
↪average='samplewise', average=None)
tensor([[0.9091, 0.0000, 0.5000],
        [0.0000, 0.3571, 0.4545]])
```

## multilabel\_fbeta\_score

`torchmetrics.functional.classification.multilabel_fbeta_score(preds, target, beta, num_labels, threshold=0.5, average='macro', multidim_average='global', ignore_index=None, validate_args=True)`

Computes **F-score** metric for multilabel tasks:

$$F_{\beta} = (1 + \beta^2) * \frac{\text{precision} * \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}}$$

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.

- **target** (int tensor): (N, C, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

#### Parameters

- **preds** (Tensor) – Tensor with predictions
- **target** (Tensor) – Tensor with true labels
- **beta** (float) – Weighting between precision and recall in calculation. Setting to 1 corresponds to equal weight
- **num\_labels** (int) – Integer specifying the number of labels
- **threshold** (float) – Threshold for transforming probability to binary (0,1) predictions
- **average** (Optional[Literal['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - micro: Sum statistics over all labels
  - macro: Calculate statistics for each label and average them
  - weighted: Calculates statistics for each label and computes weighted average using their support
  - "none" or None: Calculates statistic for each label and applies no reduction
- **multidim\_average** (Literal['global', 'samplewise']) – Defines how additional dimensions ... should be handled. Should be one of the following:
  - global: Additional dimensions are flattened along the batch dimension
  - samplewise: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (Optional[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (bool) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.

#### Returns

- If *multidim\_average* is set to global:
  - If *average*='micro'/'macro'/'weighted', the output will be a scalar tensor
  - If *average*=None/'none', the shape will be (C,)
- If *multidim\_average* is set to samplewise:
  - If *average*='micro'/'macro'/'weighted', the shape will be (N,)
  - If *average*=None/'none', the shape will be (N, C)

**Return type** The returned shape depends on the *average* and *multidim\_average* arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_fbeta_score
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_fbeta_score(preds, target, beta=2.0, num_labels=3)
tensor(0.6111)
```

(continues on next page)

(continued from previous page)

```
>>> multilabel_fbeta_score(preds, target, beta=2.0, num_labels=3, average=None)
tensor([1.0000, 0.0000, 0.8333])
```

Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import multilabel_fbeta_score
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_fbeta_score(preds, target, beta=2.0, num_labels=3)
tensor(0.6111)
>>> multilabel_fbeta_score(preds, target, beta=2.0, num_labels=3, average=None)
tensor([1.0000, 0.0000, 0.8333])
```

Example (multidim tensors):

```
>>> from torchmetrics.functional.classification import multilabel_fbeta_score
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> multilabel_fbeta_score(preds, target, num_labels=3, beta=2.0, multidim_
↳average='samplewise')
tensor([0.5556, 0.0000])
>>> multilabel_fbeta_score(preds, target, num_labels=3, beta=2.0, multidim_
↳average='samplewise', average=None)
tensor([[0.8333, 0.8333, 0.0000],
        [0.0000, 0.0000, 0.0000]])
```

## 1.31 Hamming Distance

### 1.31.1 Module Interface

#### HammingDistance

**class** torchmetrics.HammingDistance(*threshold=0.5, \*\*kwargs*)

**Note:** From v0.10 an `'binary_*`, `'multiclass_*`, `'multilabel_*` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

Computes the average *Hamming distance* (also known as Hamming loss) between targets and predictions:

$$\text{Hamming distance} = \frac{1}{N \cdot L} \sum_i^N \sum_l^L 1(y_{il} \neq \hat{y}_{il})$$

Where  $y$  is a tensor of target values,  $\hat{y}$  is a tensor of predictions, and  $\bullet_{il}$  refers to the  $l$ -th label of the  $i$ -th sample of that tensor.

This is the same as 1-accuracy for binary data, while for all other types of inputs it treats each possible label separately - meaning that, for example, multi-class data is treated as if it were multi-label.

Accepts all input types listed in *Input types*.

#### Parameters

- **threshold** (`float`) – Threshold for transforming probability or logit predictions to binary (0, 1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
- **kwargs** (`Any`) – Additional keyword arguments, see *Advanced metric settings* for more info.

**Raises** `ValueError` – If threshold is not between 0 and 1.

#### Example

```
>>> from torchmetrics import HammingDistance
>>> target = torch.tensor([[0, 1], [1, 1]])
>>> preds = torch.tensor([[0, 1], [0, 1]])
>>> hamming_distance = HammingDistance()
>>> hamming_distance(preds, target)
tensor(0.2500)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### `compute()`

Computes hamming distance based on inputs passed in to `update` previously.

**Return type** `Tensor`

#### `update(preds, target)`

Update state with predictions and targets.

See *Input types* for more information on input types.

#### Parameters

- **preds** (`Tensor`) – Predictions from model (probabilities, logits or labels)
- **target** (`Tensor`) – Ground truth labels

**Return type** `None`

### BinaryHammingDistance

```
class torchmetrics.classification.BinaryHammingDistance(threshold=0.5, multidim_average='global',
                                                         ignore_index=None, validate_args=True,
                                                         **kwargs)
```

Computes the average *Hamming distance* (also known as Hamming loss) for binary tasks:

$$\text{Hamming distance} = \frac{1}{N \cdot L} \sum_i^N \sum_l^L 1(y_{il} \neq \hat{y}_{il})$$

Where  $y$  is a tensor of target values,  $\hat{y}$  is a tensor of predictions, and  $\bullet_{il}$  refers to the  $l$ -th label of the  $i$ -th sample of that tensor.



Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

#### Parameters

- **threshold** (*float*) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (*Literal*['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - *global*: Additional dimensions are flattened along the batch dimension
  - *samplewise*: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (*Optional*[*int*]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (*bool*) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

**Returns** If *multidim\_average* is set to *global*, the metric returns a scalar value. If *multidim\_average* is set to *samplewise*, the metric returns (N,) vector consisting of a scalar value per sample.

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import BinaryHammingDistance
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0, 0, 1, 1, 0, 1])
>>> metric = BinaryHammingDistance()
>>> metric(preds, target)
tensor(0.3333)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import BinaryHammingDistance
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> metric = BinaryHammingDistance()
>>> metric(preds, target)
tensor(0.3333)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import BinaryHammingDistance
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
```

(continues on next page)

(continued from previous page)

```
... )
>>> metric = BinaryHammingDistance(multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.6667, 0.8333])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes the final statistics.

**Return type** `Tensor`

**Returns**

The metric returns a tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to  $[tp, fp, tn, fn, sup]$  ( $sup$  stands for support and equals  $tp + fn$ ). The shape depends on the `multidim_average` parameter:

- If `multidim_average` is set to `global`, the shape will be  $(5,)$
- If `multidim_average` is set to `samplewise`, the shape will be  $(N, 5)$

## MulticlassHammingDistance

```
class torchmetrics.classification.MulticlassHammingDistance(num_classes, top_k=1,
                                                            average='macro',
                                                            multidim_average='global',
                                                            ignore_index=None,
                                                            validate_args=True, **kwargs)
```

Computes the average *Hamming distance* (also known as Hamming loss) for multiclass tasks:

$$\text{Hamming distance} = \frac{1}{N \cdot L} \sum_i^N \sum_l^L 1(y_{il} \neq \hat{y}_{il})$$

Where  $y$  is a tensor of target values,  $\hat{y}$  is a tensor of predictions, and  $\bullet_{il}$  refers to the  $l$ -th label of the  $i$ -th sample of that tensor.

Accepts the following input tensors:

- **preds**:  $(N, \dots)$  (int tensor) or  $(N, C, \dots)$  (float tensor). If `preds` is a floating point we apply `torch.argmax` along the  $C$  dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor):  $(N, \dots)$

The influence of the additional dimension  $\dots$  (if present) will be determined by the `multidim_average` argument.

**Parameters**

- **num\_classes** (`int`) – Integer specifying the number of classes
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: Calculates statistics for each label and computes weighted average using their support

- "none" or None: Calculates statistic for each label and applies no reduction
- **top\_k** (`int`) – Number of highest probability or logit score predictions considered to find the correct label. Only works when preds contain probabilities/logits.
- **multidim\_average** (`Literal`['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - global: Additional dimensions are flattened along the batch dimension
  - samplewise: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional`[`int`]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be (C,)
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be (N,)
  - If `average=None/'none'`, the shape will be (N, C)

**Return type** The returned shape depends on the `average` and `multidim_average` arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import MulticlassHammingDistance
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> metric = MulticlassHammingDistance(num_classes=3)
>>> metric(preds, target)
tensor(0.1667)
>>> metric = MulticlassHammingDistance(num_classes=3, average=None)
>>> metric(preds, target)
tensor([0.5000, 0.0000, 0.0000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import MulticlassHammingDistance
>>> target = target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> metric = MulticlassHammingDistance(num_classes=3)
>>> metric(preds, target)
tensor(0.1667)
```

(continues on next page)

(continued from previous page)

```
>>> metric = MulticlassHammingDistance(num_classes=3, average=None)
>>> metric(preds, target)
tensor([0.5000, 0.0000, 0.0000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import MulticlassHammingDistance
>>> target = torch.tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = torch.tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> metric = MulticlassHammingDistance(num_classes=3, multidim_average=
↳ 'samplewise')
>>> metric(preds, target)
tensor([0.5000, 0.7222])
>>> metric = MulticlassHammingDistance(num_classes=3, multidim_average=
↳ 'samplewise', average=None)
>>> metric(preds, target)
tensor([[0.0000, 1.0000, 0.5000],
        [1.0000, 0.6667, 0.5000]])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes the final statistics.

**Return type** `Tensor`**Returns**

The metric returns a tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to  $[tp, fp, tn, fn, sup]$  (sup stands for support and equals  $tp + fn$ ). The shape depends on `average` and `multidim_average` parameters:

- If `multidim_average` is set to `global`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(5,)$
- If `average=None/'none'`, the shape will be  $(C, 5)$
- If `multidim_average` is set to `samplewise`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(N, 5)$
- If `average=None/'none'`, the shape will be  $(N, C, 5)$

## MultilabelHammingDistance

```
class torchmetrics.classification.MultilabelHammingDistance(num_labels, threshold=0.5,
                                                             average='macro',
                                                             multidim_average='global',
                                                             ignore_index=None,
                                                             validate_args=True, **kwargs)
```

Computes the average *Hamming distance* (also known as Hamming loss) for multilabel tasks:

$$\text{Hamming distance} = \frac{1}{N \cdot L} \sum_i^N \sum_l^L 1(y_{il} \neq \hat{y}_{il})$$

Where  $y$  is a tensor of target values,  $\hat{y}$  is a tensor of predictions, and  $\bullet_{il}$  refers to the  $l$ -th label of the  $i$ -th sample of that tensor.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

### Parameters

- **num\_labels** (**int**) – Integer specifying the number of labels
- **threshold** (**float**) – Threshold for transforming probability to binary (0,1) predictions
- **average** (**Optional**[**Literal**['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: Calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: Calculates statistic for each label and applies no reduction
- **multidim\_average** (**Literal**['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (**Optional**[**int**]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (**bool**) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

### Returns

- If **multidim\_average** is set to **global**:
  - If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor
  - If **average**=None/'none', the shape will be (C,)
- If **multidim\_average** is set to **samplewise**:
  - If **average**='micro'/'macro'/'weighted', the shape will be (N,)
  - If **average**=None/'none', the shape will be (N, C)

**Return type** The returned shape depends on the **average** and **multidim\_average** arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import MultilabelHammingDistance
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelHammingDistance(num_labels=3)
>>> metric(preds, target)
tensor(0.3333)
>>> metric = MultilabelHammingDistance(num_labels=3, average=None)
>>> metric(preds, target)
tensor([0.0000, 0.5000, 0.5000])
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import MultilabelHammingDistance
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelHammingDistance(num_labels=3)
>>> metric(preds, target)
tensor(0.3333)
>>> metric = MultilabelHammingDistance(num_labels=3, average=None)
>>> metric(preds, target)
tensor([0.0000, 0.5000, 0.5000])
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import MultilabelHammingDistance
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> metric = MultilabelHammingDistance(num_labels=3, multidim_average=
↳ 'samplewise')
>>> metric(preds, target)
tensor([0.6667, 0.8333])
>>> metric = MultilabelHammingDistance(num_labels=3, multidim_average=
↳ 'samplewise', average=None)
>>> metric(preds, target)
tensor([[0.5000, 0.5000, 1.0000],
        [1.0000, 1.0000, 0.5000]])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes the final statistics.

**Return type** `Tensor`

**Returns**

The metric returns a tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to  $[tp, fp, tn, fn, sup]$  ( $sup$  stands for support and equals  $tp + fn$ ). The shape depends on `average` and `multidim_average` parameters:

- If `multidim_average` is set to `global`

- If `average='micro'/'macro'/'weighted'`, the shape will be  $(5,)$
- If `average=None/'none'`, the shape will be  $(C, 5)$
- If `multidim_average` is set to `samplewise`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(N, 5)$
- If `average=None/'none'`, the shape will be  $(N, C, 5)$

### 1.31.2 Functional Interface

#### hamming\_distance

`torchmetrics.functional.hamming_distance(preds, target, threshold=0.5, task=None, num_classes=None, num_labels=None, average='macro', top_k=1, multidim_average='global', ignore_index=None, validate_args=True)`

---

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entryptpoint to calling the three specialized versions.

---

Computes the average *Hamming distance* (also known as Hamming loss) between targets and predictions:

$$\text{Hamming distance} = \frac{1}{N \cdot L} \sum_i^N \sum_l^L 1(y_{il} \neq \hat{y}_{il})$$

Where  $y$  is a tensor of target values,  $\hat{y}$  is a tensor of predictions, and  $\bullet_{il}$  refers to the  $l$ -th label of the  $i$ -th sample of that tensor.

This is the same as 1-accuracy for binary data, while for all other types of inputs it treats each possible label separately - meaning that, for example, multi-class data is treated as if it were multi-label.

Accepts all input types listed in *Input types*.

#### Parameters

- **preds** ([Tensor](#)) – Predictions from model (probabilities, logits or labels)
- **target** ([Tensor](#)) – Ground truth
- **threshold** ([float](#)) – Threshold for transforming probability or logit predictions to binary  $(0,1)$  predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.

### Example

```
>>> from torchmetrics.functional import hamming_distance
>>> target = torch.tensor([[0, 1], [1, 1]])
>>> preds = torch.tensor([[0, 1], [0, 1]])
>>> hamming_distance(preds, target)
tensor(0.2500)
```

Return type `Tensor`

### binary\_hamming\_distance

`torchmetrics.functional.classification.binary_hamming_distance(preds, target, threshold=0.5, multidim_average='global', ignore_index=None, validate_args=True)`

Computes the average *Hamming distance* (also known as Hamming loss) for binary tasks:

$$\text{Hamming distance} = \frac{1}{N \cdot L} \sum_i^N \sum_l^L 1(y_{il} \neq \hat{y}_{il})$$

Where  $y$  is a tensor of target values,  $\hat{y}$  is a tensor of predictions, and  $\bullet_{il}$  refers to the  $l$ -th label of the  $i$ -th sample of that tensor.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

#### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **threshold** (`float`) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (`Literal`['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional`[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

Return type `Tensor`



**Returns** If `multidim_average` is set to `global`, the metric returns a scalar value. If `multidim_average` is set to `samplewise`, the metric returns (N,) vector consisting of a scalar value per sample.

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import binary_hamming_distance
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0, 0, 1, 1, 0, 1])
>>> binary_hamming_distance(preds, target)
tensor(0.3333)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import binary_hamming_distance
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> binary_hamming_distance(preds, target)
tensor(0.3333)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import binary_hamming_distance
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> binary_hamming_distance(preds, target, multidim_average='samplewise')
tensor([0.6667, 0.8333])
```

## multiclass\_hamming\_distance

`torchmetrics.functional.classification.multiclass_hamming_distance(preds, target, num_classes, average='macro', top_k=1, multidim_average='global', ignore_index=None, validate_args=True)`

Computes the average *Hamming distance* (also known as Hamming loss) for multiclass tasks:

$$\text{Hamming distance} = \frac{1}{N \cdot L} \sum_i^N \sum_l^L 1(y_{il} \neq \hat{y}_{il})$$

Where  $y$  is a tensor of target values,  $\hat{y}$  is a tensor of predictions, and  $\bullet_{il}$  refers to the  $l$ -th label of the  $i$ -th sample of that tensor.

Accepts the following input tensors:

- `preds`: (N, ...) (int tensor) or (N, C, ...) (float tensor). If `preds` is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- `target` (int tensor): (N, ...)

The influence of the additional dimension . . . (if present) will be determined by the *multidim\_average* argument.

### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **num\_classes** (`int`) – Integer specifying the number of classes
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: Calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: Calculates statistic for each label and applies no reduction
- **top\_k** (`int`) – Number of highest probability or logit score predictions considered to find the correct label. Only works when `preds` contain probabilities/logits.
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions . . . should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the `N` axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be `(C,)`
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

**Return type** The returned shape depends on the `average` and `multidim_average` arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_hamming_
↳ distance
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> multiclass_hamming_distance(preds, target, num_classes=3)
tensor(0.1667)
>>> multiclass_hamming_distance(preds, target, num_classes=3, average=None)
tensor([0.5000, 0.0000, 0.0000])
```

Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import multiclass_hamming_
↳ distance
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> multiclass_hamming_distance(preds, target, num_classes=3)
tensor(0.1667)
>>> multiclass_hamming_distance(preds, target, num_classes=3, average=None)
tensor([0.5000, 0.0000, 0.0000])
```

Example (multidim tensors):

```
>>> from torchmetrics.functional.classification import multiclass_hamming_
↳ distance
>>> target = torch.tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = torch.tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> multiclass_hamming_distance(preds, target, num_classes=3, multidim_average=
↳ 'samplewise')
tensor([0.5000, 0.7222])
>>> multiclass_hamming_distance(preds, target, num_classes=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[0.0000, 1.0000, 0.5000],
        [1.0000, 0.6667, 0.5000]])
```

## multilabel\_hamming\_distance

`torchmetrics.functional.classification.multilabel_hamming_distance(preds, target, num_labels, threshold=0.5, average='macro', multidim_average='global', ignore_index=None, validate_args=True)`

Computes the average *Hamming distance* (also known as Hamming loss) for multilabel tasks:

$$\text{Hamming distance} = \frac{1}{N \cdot L} \sum_i^N \sum_l^L 1(y_{il} \neq \hat{y}_{il})$$

Where  $y$  is a tensor of target values,  $\hat{y}$  is a tensor of predictions, and  $\bullet_{il}$  refers to the  $l$ -th label of the  $i$ -th sample of that tensor.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **num\_labels** (`int`) – Integer specifying the number of labels
- **threshold** (`float`) – Threshold for transforming probability to binary (0,1) predictions
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: Calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: Calculates statistic for each label and applies no reduction
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be `(C,)`
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

**Return type** The returned shape depends on the `average` and `multidim_average` arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_hamming_
↳ distance
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_hamming_distance(preds, target, num_labels=3)
tensor(0.3333)
>>> multilabel_hamming_distance(preds, target, num_labels=3, average=None)
tensor([0.0000, 0.5000, 0.5000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_hamming_
↳ distance
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_hamming_distance(preds, target, num_labels=3)
tensor(0.3333)
>>> multilabel_hamming_distance(preds, target, num_labels=3, average=None)
tensor([0.0000, 0.5000, 0.5000])
```

Example (multidim tensors):

```
>>> from torchmetrics.functional.classification import multilabel_hamming_
↳ distance
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> multilabel_hamming_distance(preds, target, num_labels=3, multidim_average=
↳ 'samplewise')
tensor([0.6667, 0.8333])
>>> multilabel_hamming_distance(preds, target, num_labels=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[0.5000, 0.5000, 1.0000],
        [1.0000, 1.0000, 0.5000]])
```

## 1.32 Hinge Loss

### 1.32.1 Module Interface

**class** torchmetrics.HingeLoss(*squared=False, multiclass\_mode=None, \*\*kwargs*)

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entypoint to calling the three specialized versions.

Computes the mean *Hinge loss*, typically used for Support Vector Machines (SVMs).

In the binary case it is defined as:

$$\text{Hinge loss} = \max(0, 1 - y \times \hat{y})$$

Where  $y \in -1, 1$  is the target, and  $\hat{y} \in \mathbb{R}$  is the prediction.

In the multi-class case, when `multiclass_mode=None` (default), `multiclass_mode=MulticlassMode.CRAMMER_SINGER` or `multiclass_mode="crammer-singer"`, this metric will compute the multi-class hinge

loss defined by Crammer and Singer as:

$$\text{Hinge loss} = \max \left( 0, 1 - \hat{y}_y + \max_{i \neq y}(\hat{y}_i) \right)$$

Where  $y \in 0, \dots, C$  is the target class (where  $C$  is the number of classes), and  $\hat{y} \in \mathbb{R}^C$  is the predicted output per class.

In the multi-class case when `multiclass_mode=MulticlassMode.ONE_VS_ALL` or `multiclass_mode='one-vs-all'`, this metric will use a one-vs-all approach to compute the hinge loss, giving a vector of  $C$  outputs where each entry pits that class against all remaining classes.

This metric can optionally output the mean of the squared hinge loss by setting `squared=True`

Only accepts inputs with preds shape of (N) (binary) or (N, C) (multi-class) and target shape of (N).

#### Parameters

- **squared** (`bool`) – If True, this will compute the squared hinge loss. Otherwise, computes the regular hinge loss (default).
- **multiclass\_mode** (`Union[str, MulticlassMode, None]`) – Which approach to use for multi-class inputs (has no effect in the binary case). `None` (default), `MulticlassMode.CRAMMER_SINGER` or `"crammer-singer"`, uses the Crammer Singer multi-class hinge loss. `MulticlassMode.ONE_VS_ALL` or `"one-vs-all"` computes the hinge loss in a one-vs-all fashion.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises** `ValueError` – If `multiclass_mode` is not: `None`, `MulticlassMode.CRAMMER_SINGER`, `"crammer-singer"`, `MulticlassMode.ONE_VS_ALL` or `"one-vs-all"`.

#### Example (binary case):

```
>>> import torch
>>> from torchmetrics import HingeLoss
>>> target = torch.tensor([0, 1, 1])
>>> preds = torch.tensor([-2.2, 2.4, 0.1])
>>> hinge = HingeLoss()
>>> hinge(preds, target)
tensor(0.3000)
```

#### Example (default / multiclass case):

```
>>> target = torch.tensor([0, 1, 2])
>>> preds = torch.tensor([[-1.0, 0.9, 0.2], [0.5, -1.1, 0.8], [2.2, -0.5, 0.3]])
>>> hinge = HingeLoss()
>>> hinge(preds, target)
tensor(2.9000)
```

#### Example (multiclass example, one vs all mode):

```
>>> target = torch.tensor([0, 1, 2])
>>> preds = torch.tensor([[-1.0, 0.9, 0.2], [0.5, -1.1, 0.8], [2.2, -0.5, 0.3]])
>>> hinge = HingeLoss(multiclass_mode="one-vs-all")
>>> hinge(preds, target)
tensor([2.2333, 1.5000, 1.2333])
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

**compute()**

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tensor`

**update(preds, target)**

Override this method to update the state variables of your metric class.

**Return type** `None`

**BinaryHingeLoss**

```
class torchmetrics.classification.BinaryHingeLoss(squared=False, ignore_index=None,
                                                  validate_args=True, **kwargs)
```

Computes the mean *Hinge loss* typically used for Support Vector Machines (SVMs) for binary tasks. It is defined as:

$$\text{Hinge loss} = \max(0, 1 - y \times \hat{y})$$

Where  $y \in -1, 1$  is the target, and  $\hat{y} \in \mathbb{R}$  is the prediction.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

**Parameters**

- **squared** (`bool`) – If True, this will compute the squared hinge loss. Otherwise, computes the regular hinge loss.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see *Advanced metric settings* for more info.

**Example**

```
>>> from torchmetrics.classification import BinaryHingeLoss
>>> preds = torch.tensor([0.25, 0.25, 0.55, 0.75, 0.75])
>>> target = torch.tensor([0, 0, 1, 1, 1])
>>> metric = BinaryHingeLoss()
>>> metric(preds, target)
tensor(0.6900)
>>> metric = BinaryHingeLoss(squared=True)
>>> metric(preds, target)
tensor(0.6905)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tensor`

**update(preds, target)**

Override this method to update the state variables of your metric class.

**Return type** `None`

## MulticlassHingeLoss

```
class torchmetrics.classification.MulticlassHingeLoss(num_classes, squared=False,
                                                    multiclass_mode='crammer-singer',
                                                    ignore_index=None, validate_args=True,
                                                    **kwargs)
```

Computes the mean *Hinge loss* typically used for Support Vector Machines (SVMs) for multiclass tasks

The metric can be computed in two ways. Either, the definition by Crammer and Singer is used:

$$\text{Hinge loss} = \max \left( 0, 1 - \hat{y}_y + \max_{i \neq y}(\hat{y}_i) \right)$$

Where  $y \in 0, \dots, C$  is the target class (where  $C$  is the number of classes), and  $\hat{y} \in \mathbb{R}^C$  is the predicted output per class. Alternatively, the metric can also be computed in one-vs-all approach, where each class is valued against all other classes in a binary fashion.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the [0, n\_classes-1] range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **num\_classes** (`int`) – Integer specifying the number of classes
- **squared** (`bool`) – If True, this will compute the squared hinge loss. Otherwise, computes the regular hinge loss.
- **multiclass\_mode** (`Literal`['crammer-singer', 'one-vs-all']) – Determines how to compute the metric
- **ignore\_index** (`Optional`[`int`]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see *Advanced metric settings* for more info.



### Example

```
>>> from torchmetrics.classification import MulticlassHingeLoss
>>> preds = torch.tensor([[0.25, 0.20, 0.55],
...                        [0.55, 0.05, 0.40],
...                        [0.10, 0.30, 0.60],
...                        [0.90, 0.05, 0.05]])
>>> target = torch.tensor([0, 1, 2, 0])
>>> metric = MulticlassHingeLoss(num_classes=3)
>>> metric(preds, target)
tensor(0.9125)
>>> metric = MulticlassHingeLoss(num_classes=3, squared=True)
>>> metric(preds, target)
tensor(1.1131)
>>> metric = MulticlassHingeLoss(num_classes=3, multiclass_mode='one-vs-all')
>>> metric(preds, target)
tensor([0.8750, 1.1250, 1.1000])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### `compute()`

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tensor`

#### `update(preds, target)`

Override this method to update the state variables of your metric class.

**Return type** `None`

## 1.32.2 Functional Interface

`torchmetrics.functional.hinge_loss(preds, target, squared=False, multiclass_mode=None, task=None, num_classes=None, ignore_index=None, validate_args=True)`

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entryptpoint to calling the three specialized versions.

Computes the mean *Hinge loss* typically used for Support Vector Machines (SVMs).

In the binary case it is defined as:

$$\text{Hinge loss} = \max(0, 1 - y \times \hat{y})$$

Where  $y \in -1, 1$  is the target, and  $\hat{y} \in \mathbb{R}$  is the prediction.

In the multi-class case, when `multiclass_mode=None` (default), `multiclass_mode=MulticlassMode.CRAMMER_SINGER` or `multiclass_mode="crammer-singer"`, this metric will compute the multi-class hinge loss defined by Crammer and Singer as:

$$\text{Hinge loss} = \max \left( 0, 1 - \hat{y}_y + \max_{i \neq y} (\hat{y}_i) \right)$$

Where  $y \in 0, \dots, C$  is the target class (where  $C$  is the number of classes), and  $\hat{y} \in \mathbb{R}^C$  is the predicted output per class.

In the multi-class case when `multiclass_mode=MulticlassMode.ONE_VS_ALL` or `multiclass_mode='one-vs-all'`, this metric will use a one-vs-all approach to compute the hinge loss, giving a vector of  $C$  outputs where each entry pits that class against all remaining classes.

This metric can optionally output the mean of the squared hinge loss by setting `squared=True`

Only accepts inputs with preds shape of  $(N)$  (binary) or  $(N, C)$  (multi-class) and target shape of  $(N)$ .

#### Parameters

- **preds** (`Tensor`) – Predictions from model (as float outputs from decision function).
- **target** (`Tensor`) – Ground truth labels.
- **squared** (`bool`) – If True, this will compute the squared hinge loss. Otherwise, computes the regular hinge loss (default).
- **multiclass\_mode** (`Optional[Literal['crammer-singer', 'one-vs-all']]`) – Which approach to use for multi-class inputs (has no effect in the binary case). None (default), `MulticlassMode.CRAMMER_SINGER` or `"crammer-singer"`, uses the Crammer Singer multi-class hinge loss. `MulticlassMode.ONE_VS_ALL` or `"one-vs-all"` computes the hinge loss in a one-vs-all fashion.

#### Raises

- **ValueError** – If preds shape is not of size  $(N)$  or  $(N, C)$ .
- **ValueError** – If target shape is not of size  $(N)$ .
- **ValueError** – If `multiclass_mode` is not: None, `MulticlassMode.CRAMMER_SINGER`, `"crammer-singer"`, `MulticlassMode.ONE_VS_ALL` or `"one-vs-all"`.

#### Example (binary case):

```
>>> import torch
>>> from torchmetrics.functional import hinge_loss
>>> target = torch.tensor([0, 1, 1])
>>> preds = torch.tensor([-2.2, 2.4, 0.1])
>>> hinge_loss(preds, target)
tensor(0.3000)
```

#### Example (default / multiclass case):

```
>>> target = torch.tensor([0, 1, 2])
>>> preds = torch.tensor([[-1.0, 0.9, 0.2], [0.5, -1.1, 0.8], [2.2, -0.5, 0.3]])
>>> hinge_loss(preds, target)
tensor(2.9000)
```

#### Example (multiclass example, one vs all mode):

```
>>> target = torch.tensor([0, 1, 2])
>>> preds = torch.tensor([[-1.0, 0.9, 0.2], [0.5, -1.1, 0.8], [2.2, -0.5, 0.3]])
>>> hinge_loss(preds, target, multiclass_mode="one-vs-all")
tensor([2.2333, 1.5000, 1.2333])
```

**Return type** `Tensor`

## binary\_hinge\_loss

```
torchmetrics.functional.classification.binary_hinge_loss(preds, target, squared=False,
                                                         ignore_index=None,
                                                         validate_args=False)
```

Computes the mean *Hinge loss* typically used for Support Vector Machines (SVMs) for binary tasks. It is defined as:

$$\text{Hinge loss} = \max(0, 1 - y \times \hat{y})$$

Where  $y \in -1, 1$  is the target, and  $\hat{y} \in \mathbb{R}$  is the prediction.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **preds** (*Tensor*) – Tensor with predictions
- **target** (*Tensor*) – Tensor with true labels
- **squared** (*bool*) – If True, this will compute the squared hinge loss. Otherwise, computes the regular hinge loss.
- **ignore\_index** (*Optional[int]*) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (*bool*) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

### Example

```
>>> from torchmetrics.functional.classification import binary_hinge_loss
>>> preds = torch.tensor([0.25, 0.25, 0.55, 0.75, 0.75])
>>> target = torch.tensor([0, 0, 1, 1, 1])
>>> binary_hinge_loss(preds, target)
tensor(0.6900)
>>> binary_hinge_loss(preds, target, squared=True)
tensor(0.6905)
```

**Return type** *Tensor*

## multiclass\_hinge\_loss

```
torchmetrics.functional.classification.multiclass_hinge_loss(preds, target, num_classes,
                                                             squared=False,
                                                             multiclass_mode='crammer-singer',
                                                             ignore_index=None,
                                                             validate_args=False)
```

Computes the mean *Hinge loss* typically used for Support Vector Machines (SVMs) for multiclass tasks

The metric can be computed in two ways. Either, the definition by Crammer and Singer is used:

$$\text{Hinge loss} = \max \left( 0, 1 - \hat{y}_y + \max_{i \neq y}(\hat{y}_i) \right)$$

Where  $y \in 0, \dots, C$  is the target class (where  $C$  is the number of classes), and  $\hat{y} \in \mathbb{R}^C$  is the predicted output per class. Alternatively, the metric can also be computed in one-vs-all approach, where each class is valued against all other classes in a binary fashion.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the [0, n\_classes-1] range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **preds** (Tensor) – Tensor with predictions
- **target** (Tensor) – Tensor with true labels
- **num\_classes** (int) – Integer specifying the number of classes
- **squared** (bool) – If True, this will compute the squared hinge loss. Otherwise, computes the regular hinge loss.
- **multiclass\_mode** (Literal['crammer-singer', 'one-vs-all']) – Determines how to compute the metric
- **ignore\_index** (Optional[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (bool) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.

### Example

```
>>> from torchmetrics.functional.classification import multiclass_hinge_loss
>>> preds = torch.tensor([[0.25, 0.20, 0.55],
...                       [0.55, 0.05, 0.40],
...                       [0.10, 0.30, 0.60],
...                       [0.90, 0.05, 0.05]])
>>> target = torch.tensor([0, 1, 2, 0])
>>> multiclass_hinge_loss(preds, target, num_classes=3)
tensor(0.9125)
```

(continues on next page)

(continued from previous page)

```
>>> multiclass_hinge_loss(preds, target, num_classes=3, squared=True)
tensor(1.1131)
>>> multiclass_hinge_loss(preds, target, num_classes=3, multiclass_mode='one-vs-all
↪')
tensor([0.8750, 1.1250, 1.1000])
```

**Return type** `Tensor`

## 1.33 Jaccard Index

### 1.33.1 Module Interface

#### CohenKappa

**class** `torchmetrics.JaccardIndex`(*num\_classes*, *average*='macro', *ignore\_index*=None, *absent\_score*=0.0, *threshold*=0.5, *multilabel*=False, \*\*kwargs)

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the *task* argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entryptpoint to calling the three specialized versions.

Computes Intersection over union, or *Jaccard index*:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Where: *A* and *B* are both tensors of the same size, containing integer class values. They may be subject to conversion from input data (see description below). Note that it is different from box IoU.

Works with binary, multiclass and multi-label data. Accepts probabilities from a model output or integer class values in prediction. Works with multi-dimensional preds and target.

Forward accepts

- **preds** (float or long tensor): (N, ...) or (N, C, ...) where C is the number of classes
- **target** (long tensor): (N, ...)

If preds and target are the same shape and preds is a float tensor, we use the `self.threshold` argument to convert into integer labels. This is the case for binary and multi-label probabilities.

If preds has an extra dimension as in the case of multi-class scores we perform an `argmax` on `dim=1`.

#### Parameters

- **num\_classes** (`int`) – Number of classes in the dataset.
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied. Should be one of the following:
  - `'macro'` [default]: Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).

- 'micro': Calculate the metric globally, across all samples and classes.
- 'weighted': Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support ( $tp + fn$ ).
- 'none' or None: Calculate the metric for each class separately, and return the metric for every class. Note that if a given class doesn't occur in the *preds* or *target*, the value for the class will be nan.
- **ignore\_index** (Optional[int]) – optional int specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. Has no effect if given an int that is not in the range  $[0, \text{num\_classes}-1]$ . By default, no index is ignored, and all classes are used.
- **absent\_score** (float) – score to use for an individual class, if no instances of the class index were present in *preds* AND no instances of the class index were present in *target*. For example, if we have 3 classes,  $[0, 0]$  for *preds*, and  $[0, 2]$  for *target*, then class 1 would be assigned the *absent\_score*.
- **threshold** (float) – Threshold value for binary or multi-label probabilities.
- **multilabel** (bool) – determines if data is multilabel or not.
- **kwargs** (Any) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torchmetrics import JaccardIndex
>>> target = torch.randint(0, 2, (10, 25, 25))
>>> pred = torch.tensor(target)
>>> pred[2:5, 7:13, 9:15] = 1 - pred[2:5, 7:13, 9:15]
>>> jaccard = JaccardIndex(num_classes=2)
>>> jaccard(pred, target)
tensor(0.9660)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes intersection over union (IoU)

**Return type** Tensor

### BinaryJaccardIndex

**class** torchmetrics.classification.BinaryJaccardIndex(*threshold=0.5, ignore\_index=None, validate\_args=True, \*\*kwargs*)

Calculates the Jaccard index for binary tasks. The *Jaccard index* (also known as the intersection over union or jaccard similarity coefficient) is an statistic that can be used to determine the similarity and diversity of a sample set. It is defined as the size of the intersection divided by the union of the sample sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside  $[0, 1]$  range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.

- **target** (int tensor): (N, ...)

Additional dimension ... will be flattened into the batch dimension.

#### Parameters

- **threshold** (float) – Threshold for transforming probability to binary (0,1) predictions
- **ignore\_index** (Optional[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **normalize** – Normalization mode for confusion matrix. Choose from:
  - None or 'none': no normalization (default)
  - 'true': normalization over the targets (most commonly used)
  - 'pred': normalization over the predictions
  - 'all': normalization over the whole matrix
- **validate\_args** (bool) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.
- **kwargs** (Any) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Example (preds is int tensor):

```
>>> from torchmetrics.classification import BinaryJaccardIndex
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0, 1, 0, 0])
>>> metric = BinaryJaccardIndex()
>>> metric(preds, target)
tensor(0.5000)
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import BinaryJaccardIndex
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0.35, 0.85, 0.48, 0.01])
>>> metric = BinaryJaccardIndex()
>>> metric(preds, target)
tensor(0.5000)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

### MulticlassJaccardIndex

```
class torchmetrics.classification.MulticlassJaccardIndex(num_classes, average='macro',
                                                         ignore_index=None, validate_args=True,
                                                         **kwargs)
```

Calculates the Jaccard index for multiclass tasks. The *Jaccard index* (also known as the intersection over union or jaccard similarity coefficient) is an statistic that can be used to determine the similarity and diversity of a sample set. It is defined as the size of the intersection divided by the union of the sample sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

Additional dimension ... will be flattened into the batch dimension.

#### Parameters

- **num\_classes** (`int`) – Integer specifying the number of classes
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: Calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: Calculates statistic for each label and applies no reduction
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Example (pred is integer tensor):

```
>>> from torchmetrics.classification import MulticlassJaccardIndex
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> metric = MulticlassJaccardIndex(num_classes=3)
>>> metric(preds, target)
tensor(0.6667)
```

Example (pred is float tensor):

```
>>> from torchmetrics.classification import MulticlassJaccardIndex
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> metric = MulticlassJaccardIndex(num_classes=3)
>>> metric(preds, target)
tensor(0.6667)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.



## MultilabelJaccardIndex

```
class torchmetrics.classification.MultilabelJaccardIndex(num_labels, threshold=0.5,
                                                         average='macro', ignore_index=None,
                                                         validate_args=True, **kwargs)
```

Calculates the Jaccard index for multilabel tasks. The *Jaccard index* (also known as the intersection over union or jaccard similarity coefficient) is an statistic that can be used to determine the similarity and diversity of a sample set. It is defined as the size of the intersection divided by the union of the sample sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **num\_classes** – Integer specifying the number of labels
- **threshold** (**float**) – Threshold for transforming probability to binary (0,1) predictions
- **ignore\_index** (**Optional[int]**) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **average** (**Optional[Literal['micro', 'macro', 'weighted', 'none']]**) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: Calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: Calculates statistic for each label and applies no reduction
- **validate\_args** (**bool**) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.
- **kwargs** (**Any**) – Additional keyword arguments, see *Advanced metric settings* for more info.

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import MultilabelJaccardIndex
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelJaccardIndex(num_labels=3)
>>> metric(preds, target)
tensor(0.5000)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import MultilabelJaccardIndex
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelJaccardIndex(num_labels=3)
>>> metric(preds, target)
tensor(0.5000)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

### 1.33.2 Functional Interface

#### jaccard\_index

`torchmetrics.functional.jaccard_index(preds, target, num_classes, average='macro', ignore_index=None, absent_score=0.0, threshold=0.5, task=None, num_labels=None, validate_args=True)`

---

**Note:** From v0.10 an `'binary_*`, `'multiclass_*`, `'multilabel_*` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entypoint to calling the three specialized versions.

---

Computes *Jaccard index*

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Where:  $A$  and  $B$  are both tensors of the same size, containing integer class values. They may be subject to conversion from input data (see description below).

Note that it is different from box IoU.

If `preds` and `target` are the same shape and `preds` is a float tensor, we use the `self.threshold` argument to convert into integer labels. This is the case for binary and multi-label probabilities.

If `pred` has an extra dimension as in the case of multi-class scores we perform an `argmax` on `dim=1`.

#### Parameters

- **preds** (`Tensor`) – tensor containing predictions from model (probabilities, or labels) with shape `[N, d1, d2, ...]`
- **target** (`Tensor`) – tensor containing ground truth labels with shape `[N, d1, d2, ...]`
- **num\_classes** (`int`) – Specify the number of classes
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied. Should be one of the following:
  - `'macro'` [default]: Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
  - `'micro'`: Calculate the metric globally, across all samples and classes.
  - `'weighted'`: Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (`tp + fn`).

- 'none' or None: Calculate the metric for each class separately, and return the metric for every class. Note that if a given class doesn't occur in the *preds* or *target*, the value for the class will be nan.
- **ignore\_index** (Optional[int]) – optional int specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. Has no effect if given an int that is not in the range [0, num\_classes-1], where num\_classes is either given or derived from pred and target. By default, no index is ignored, and all classes are used.
- **absent\_score** (float) – score to use for an individual class, if no instances of the class index were present in *preds* AND no instances of the class index were present in *target*. For example, if we have 3 classes, [0, 0] for *preds*, and [0, 2] for *target*, then class 1 would be assigned the *absent\_score*.
- **threshold** (float) – Threshold value for binary or multi-label probabilities.

**Return type** Tensor

#### Returns

The shape of the returned tensor depends on the *average* parameter

- If *average* in ['micro', 'macro', 'weighted'], a one-element tensor will be returned
- If *average* in ['none', None], the shape will be (C, ), where C stands for the number of classes

#### Example

```
>>> from torchmetrics.functional import jaccard_index
>>> target = torch.randint(0, 2, (10, 25, 25))
>>> pred = torch.tensor(target)
>>> pred[2:5, 7:13, 9:15] = 1 - pred[2:5, 7:13, 9:15]
>>> jaccard_index(pred, target, num_classes=2)
tensor(0.9660)
```

### binary\_jaccard\_index

`torchmetrics.functional.classification.binary_jaccard_index(preds, target, threshold=0.5, ignore_index=None, validate_args=True)`

Calculates the Jaccard index for binary tasks. The *Jaccard index* (also known as the intersection over union or jaccard similarity coefficient) is an statistic that can be used to determine the similarity and diversity of a sample set. It is defined as the size of the intersection divided by the union of the sample sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Accepts the following input tensors:

- *preds* (int or float tensor): (N, ...). If *preds* is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in *threshold*.
- *target* (int tensor): (N, ...)

Additional dimension ... will be flattened into the batch dimension.

#### Parameters

- **threshold** (`float`) – Threshold for transforming probability to binary (0,1) predictions
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **normalize** – Normalization mode for confusion matrix. Choose from:
  - None or 'none': no normalization (default)
  - 'true': normalization over the targets (most commonly used)
  - 'pred': normalization over the predictions
  - 'all': normalization over the whole matrix
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Example (preds is int tensor):

```
>>> from torchmetrics.functional.classification import binary_jaccard_index
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0, 1, 0, 0])
>>> binary_jaccard_index(preds, target)
tensor(0.5000)
```

Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import binary_jaccard_index
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0.35, 0.85, 0.48, 0.01])
>>> binary_jaccard_index(preds, target)
tensor(0.5000)
```

Return type `Tensor`

### multiclass\_jaccard\_index

```
torchmetrics.functional.classification.multiclass_jaccard_index(preds, target, num_classes,
                                                                average='macro',
                                                                ignore_index=None,
                                                                validate_args=True)
```

Calculates the Jaccard index for multiclass tasks. The *Jaccard index* (also known as the intersection over union or jaccard similarity coefficient) is an statistic that can be used to determine the similarity and diversity of a sample set. It is defined as the size of the intersection divided by the union of the sample sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Accepts the following input tensors:

- **preds**: (`N, ...`) (int tensor) or (`N, C, ...`) (float tensor). If preds is a floating point we apply `torch.argmax` along the `C` dimension to automatically convert probabilities/logits into an int tensor.

- **target** (int tensor): (N, ...)

Additional dimension ... will be flattened into the batch dimension.

#### Parameters

- **num\_classes** (`int`) – Integer specifying the number of classes
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: Calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: Calculates statistic for each label and applies no reduction
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (pred is integer tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_jaccard_index
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> multiclass_jaccard_index(preds, target, num_classes=3)
tensor(0.6667)
```

**Example (pred is float tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_jaccard_index
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> multiclass_jaccard_index(preds, target, num_classes=3)
tensor(0.6667)
```

**Return type** `Tensor`

## multilabel\_jaccard\_index

```
torchmetrics.functional.classification.multilabel_jaccard_index(preds, target, num_labels,
                                                                threshold=0.5, average='macro',
                                                                ignore_index=None,
                                                                validate_args=True)
```

Calculates the Jaccard index for multilabel tasks. The *Jaccard index* (also known as the intersection over union or jaccard similarity coefficient) is an statistic that can be used to determine the similarity and diversity of a sample set. It is defined as the size of the intersection divided by the union of the sample sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **num\_classes** – Integer specifying the number of labels
- **threshold** (**float**) – Threshold for transforming probability to binary (0,1) predictions
- **average** (**Optional**[**Literal**['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: Calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: Calculates statistic for each label and applies no reduction
- **ignore\_index** (**Optional**[**int**]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (**bool**) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.
- **kwargs** – Additional keyword arguments, see *Advanced metric settings* for more info.

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_jaccard_index
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_jaccard_index(preds, target, num_labels=3)
tensor(0.5000)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_jaccard_index
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_jaccard_index(preds, target, num_labels=3)
tensor(0.5000)
```

**Return type** `Tensor`

## 1.34 Label Ranking Average Precision

### 1.34.1 Module Interface

**class** `torchmetrics.LabelRankingAveragePrecision(**kwargs)`

Computes label ranking average precision score for multilabel data [1].

The score is the average over each ground truth label assigned to each sample of the ratio of true vs. total labels with lower score. Best score is 1.

**Parameters** `kwargs` (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

#### Example

```
>>> from torchmetrics import LabelRankingAveragePrecision
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand(10, 5)
>>> target = torch.randint(2, (10, 5))
>>> metric = LabelRankingAveragePrecision()
>>> metric(preds, target)
tensor(0.7744)
```

#### References

[1] Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). Mining multi-label data. In Data mining and knowledge discovery handbook (pp. 667-685). Springer US.

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

**compute()**

Computes the label ranking average precision score.

**Return type** `Tensor`

**update**(*preds, target, sample\_weight=None*)

**Parameters**

- **preds** (`Tensor`) – tensor of shape `[N,L]` where `N` is the number of samples and `L` is the number of labels. Should either be probabilities of the positive class or corresponding logits
- **target** (`Tensor`) – tensor of shape `[N,L]` where `N` is the number of samples and `L` is the number of labels. Should only contain binary labels.

- **sample\_weight** (`Optional[Tensor]`) – tensor of shape N where N is the number of samples. How much each sample should be weighted in the final score.

**Return type** `None`

```
class torchmetrics.classification.MultilabelRankingAveragePrecision(num_labels,
                                                                    ignore_index=None,
                                                                    validate_args=True,
                                                                    **kwargs)
```

Computes label ranking average precision score for multilabel data [1]. The score is the average over each ground truth label assigned to each sample of the ratio of true vs. total labels with lower score. Best score is 1.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if `ignore_index` is specified).

Additional dimension ... will be flattened into the batch dimension.

#### Parameters

- **num\_labels** (`int`) – Integer specifying the number of labels
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

#### Example

```
>>> from torchmetrics.classification import MultilabelRankingAveragePrecision
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand(10, 5)
>>> target = torch.randint(2, (10, 5))
>>> metric = MultilabelRankingAveragePrecision(num_labels=5)
>>> metric(preds, target)
tensor(0.7744)
```

#### References

[1] Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). Mining multi-label data. In Data mining and knowledge discovery handbook (pp. 667-685). Springer US.

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

#### `compute()`

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tensor`



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

Override this method to update the state variables of your metric class.

**Return type** `None`

## 1.34.2 Functional Interface

`torchmetrics.functional.label_ranking_average_precision`(*preds*, *target*, *sample\_weight=None*)

Computes label ranking average precision score for multilabel data [1]. The score is the average over each ground truth label assigned to each sample of the ratio of true vs. total labels with lower score. Best score is 1.

### Parameters

- **preds** (`Tensor`) – tensor of shape  $[N, L]$  where  $N$  is the number of samples and  $L$  is the number of labels. Should either be probabilities of the positive class or corresponding logits
- **target** (`Tensor`) – tensor of shape  $[N, L]$  where  $N$  is the number of samples and  $L$  is the number of labels. Should only contain binary labels.
- **sample\_weight** (`Optional[Tensor]`) – tensor of shape  $N$  where  $N$  is the number of samples. How much each sample should be weighted in the final score.

### Example

```
>>> from torchmetrics.functional import label_ranking_average_precision
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand(10, 5)
>>> target = torch.randint(2, (10, 5))
>>> label_ranking_average_precision(preds, target)
tensor(0.7744)
```

### References

[1] Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). Mining multi-label data. In Data mining and knowledge discovery handbook (pp. 667-685). Springer US.

**Return type** `Tensor`

`torchmetrics.functional.classification.multilabel_ranking_average_precision`(*preds*, *target*, *num\_labels*, *ignore\_index=None*, *validate\_args=True*)

Computes label ranking average precision score for multilabel data [1]. The score is the average over each ground truth label assigned to each sample of the ratio of true vs. total labels with lower score. Best score is 1.

Accepts the following input tensors:

- **preds** (float tensor):  $(N, C, \dots)$ . Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor):  $(N, C, \dots)$ . Target should be a tensor containing ground truth labels, and therefore only contain  $\{0,1\}$  values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

#### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

#### Example

```
>>> from torchmetrics.functional.classification import multilabel_ranking_average_
    ↪precision
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand(10, 5)
>>> target = torch.randint(2, (10, 5))
>>> multilabel_ranking_average_precision(preds, target, num_labels=5)
tensor(0.7744)
```

#### References

[1] Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). Mining multi-label data. In Data mining and knowledge discovery handbook (pp. 667-685). Springer US.

**Return type** [Tensor](#)

## 1.35 Label Ranking Loss

### 1.35.1 Module Interface

**class** `torchmetrics.LabelRankingLoss(**kwargs)`

Computes the label ranking loss for multilabel data [1]. The score corresponds to the average number of label pairs that are incorrectly ordered given some predictions weighted by the size of the label set and the number of labels not in the label set. The best score is 0.

**Parameters** **kwargs** ([Any](#)) – Additional keyword arguments, see *Advanced metric settings* for more info.

## Example

```
>>> from torchmetrics import LabelRankingLoss
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand(10, 5)
>>> target = torch.randint(2, (10, 5))
>>> metric = LabelRankingLoss()
>>> metric(preds, target)
tensor(0.4167)
```

## References

[1] Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). Mining multi-label data. In Data mining and knowledge discovery handbook (pp. 667-685). Springer US.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

### `compute()`

Computes the label ranking loss.

**Return type** `Tensor`

### `update(preds, target, sample_weight=None)`

#### Parameters

- **preds** (`Tensor`) – tensor of shape  $[N, L]$  where  $N$  is the number of samples and  $L$  is the number of labels. Should either be probabilities of the positive class or corresponding logits
- **target** (`Tensor`) – tensor of shape  $[N, L]$  where  $N$  is the number of samples and  $L$  is the number of labels. Should only contain binary labels.
- **sample\_weight** (`Optional[Tensor]`) – tensor of shape  $N$  where  $N$  is the number of samples. How much each sample should be weighted in the final score.

**Return type** `None`

```
class torchmetrics.classification.MultilabelRankingLoss(num_labels, ignore_index=None,
                                                         validate_args=True, **kwargs)
```

Computes the label ranking loss for multilabel data [1]. The score corresponds to the average number of label pairs that are incorrectly ordered given some predictions weighted by the size of the label set and the number of labels not in the label set. The best score is 0.

Accepts the following input tensors:

- **preds** (float tensor):  $(N, C, \dots)$ . Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0, 1]$  range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor):  $(N, C, \dots)$ . Target should be a tensor containing ground truth labels, and therefore only contain  $\{0, 1\}$  values (except if `ignore_index` is specified).

Additional dimension  $\dots$  will be flattened into the batch dimension.

#### Parameters

- **preds** – Tensor with predictions
- **target** – Tensor with true labels

- **num\_labels** (`int`) – Integer specifying the number of labels
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Example

```
>>> from torchmetrics.classification import MultilabelRankingLoss
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand(10, 5)
>>> target = torch.randint(2, (10, 5))
>>> metric = MultilabelRankingLoss(num_labels=5)
>>> metric(preds, target)
tensor(0.4167)
```

### References

[1] Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). Mining multi-label data. In Data mining and knowledge discovery handbook (pp. 667-685). Springer US.

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

#### **compute()**

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tensor`

#### **update(preds, target)**

Override this method to update the state variables of your metric class.

**Return type** `None`

## 1.35.2 Functional Interface

`torchmetrics.functional.label_ranking_loss(preds, target, sample_weight=None)`

Computes the label ranking loss for multilabel data [1]. The score corresponds to the average number of label pairs that are incorrectly ordered given some predictions weighted by the size of the label set and the number of labels not in the label set. The best score is 0.

#### **Parameters**

- **preds** (`Tensor`) – tensor of shape `[N,L]` where `N` is the number of samples and `L` is the number of labels. Should either be probabilities of the positive class or corresponding logits
- **target** (`Tensor`) – tensor of shape `[N,L]` where `N` is the number of samples and `L` is the number of labels. Should only contain binary labels.
- **sample\_weight** (`Optional[Tensor]`) – tensor of shape `N` where `N` is the number of samples. How much each sample should be weighted in the final score.

## Example

```
>>> from torchmetrics.functional import label_ranking_loss
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand(10, 5)
>>> target = torch.randint(2, (10, 5))
>>> label_ranking_loss(preds, target)
tensor(0.4167)
```

## References

[1] Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). Mining multi-label data. In Data mining and knowledge discovery handbook (pp. 667-685). Springer US.

**Return type** `Tensor`

```
torchmetrics.functional.classification.multilabel_ranking_loss(preds, target, num_labels,
                                                                ignore_index=None,
                                                                validate_args=True)
```

Computes the label ranking loss for multilabel data [1]. The score corresponds to the average number of label pairs that are incorrectly ordered given some predictions weighted by the size of the label set and the number of labels not in the label set. The best score is 0.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **num\_labels** (`int`) – Integer specifying the number of labels
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Example

```
>>> from torchmetrics.functional.classification import multilabel_ranking_loss
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand(10, 5)
>>> target = torch.randint(2, (10, 5))
>>> multilabel_ranking_loss(preds, target, num_labels=5)
tensor(0.4167)
```

### References

[1] Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). Mining multi-label data. In Data mining and knowledge discovery handbook (pp. 667-685). Springer US.

**Return type** `Tensor`

## 1.36 Matthews Correlation Coefficient

### 1.36.1 Module Interface

#### MatthewsCorrCoef

**class** torchmetrics.**MatthewsCorrCoef**(*num\_classes*, *threshold=0.5*, *\*\*kwargs*)

---

**Note:** From v0.10 an `'binary_*`, `'multiclass_*`, `'multilabel_*` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entryptoint to calling the three specialized versions.

---

Calculates [Matthews correlation coefficient](#) that measures the general correlation or quality of a classification.

In the binary case it is defined as:

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP) * (TP + FN) * (TN + FP) * (TN + FN)}}$$

where TP, TN, FP and FN are respectively the true positives, true negatives, false positives and false negatives. Also works in the case of multi-label or multi-class input.

---

**Note:** This metric produces a multi-dimensional output, so it can not be directly logged.

---

Forward accepts

- **preds** (float or long tensor): (N, ...) or (N, C, ...) where C is the number of classes
- **target** (long tensor): (N, ...)

If preds and target are the same shape and preds is a float tensor, we use the `self.threshold` argument to convert into integer labels. This is the case for binary and multi-label probabilities.

If preds has an extra dimension as in the case of multi-class scores we perform an `argmax` on `dim=1`.

**Parameters**

- **num\_classes** (`int`) – Number of classes in the dataset.
- **threshold** (`float`) – Threshold value for binary or multi-label probabilities.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example**

```
>>> from torchmetrics import MatthewsCorrCoef
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0, 1, 0, 0])
>>> matthews_corrcoef = MatthewsCorrCoef(num_classes=2)
>>> matthews_corrcoef(preds, target)
tensor(0.5774)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes matthews correlation coefficient.

**Return type** `Tensor`

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

**BinaryMatthewsCorrCoef**

```
class torchmetrics.classification.BinaryMatthewsCorrCoef(threshold=0.5, ignore_index=None,
                                                         validate_args=True, **kwargs)
```

Calculates [Matthews correlation coefficient](#) for binary tasks. This metric measures the general correlation or quality of a classification.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

Additional dimension ... will be flattened into the batch dimension.

**Parameters**

- **threshold** (`float`) – Threshold for transforming probability to binary (0,1) predictions
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **normalize** – Normalization mode for confusion matrix. Choose from:

- None or 'none': no normalization (default)
- 'true': normalization over the targets (most commonly used)
- 'pred': normalization over the predictions
- 'all': normalization over the whole matrix
- **validate\_args** (*bool*) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Example (preds is int tensor):

```
>>> from torchmetrics.classification import BinaryMatthewsCorrCoef
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0, 1, 0, 0])
>>> metric = BinaryMatthewsCorrCoef()
>>> metric(preds, target)
tensor(0.5774)
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import BinaryMatthewsCorrCoef
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0.35, 0.85, 0.48, 0.01])
>>> metric = BinaryMatthewsCorrCoef()
>>> metric(preds, target)
tensor(0.5774)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

## MulticlassMatthewsCorrCoef

```
class torchmetrics.classification.MulticlassMatthewsCorrCoef(num_classes, ignore_index=None,
                                                             validate_args=True, **kwargs)
```

Calculates [Matthews correlation coefficient](#) for multiclass tasks. This metric measures the general correlation or quality of a classification.

Accepts the following input tensors:

- **preds**: (`N, ...`) (int tensor) or (`N, C, ...`) (float tensor). If preds is a floating point we apply `torch.argmax` along the `C` dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (`N, ...`)

Additional dimension `...` will be flattened into the batch dimension.

### Parameters

- **num\_classes** (*int*) – Integer specifying the number of classes
- **ignore\_index** (*Optional[int]*) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **normalize** – Normalization mode for confusion matrix. Choose from:
  - None or 'none': no normalization (default)



- 'true': normalization over the targets (most commonly used)
- 'pred': normalization over the predictions
- 'all': normalization over the whole matrix
- **validate\_args** (*bool*) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (pred is integer tensor):**

```
>>> from torchmetrics.classification import MulticlassMatthewsCorrCoef
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> metric = MulticlassMatthewsCorrCoef(num_classes=3)
>>> metric(preds, target)
tensor(0.7000)
```

**Example (pred is float tensor):**

```
>>> from torchmetrics.classification import MulticlassMatthewsCorrCoef
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> metric = MulticlassMatthewsCorrCoef(num_classes=3)
>>> metric(preds, target)
tensor(0.7000)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

## MultilabelMatthewsCorrCoef

```
class torchmetrics.classification.MultilabelMatthewsCorrCoef(num_labels, threshold=0.5,
                                                             ignore_index=None,
                                                             validate_args=True, **kwargs)
```

Calculates [Matthews correlation coefficient](#) for multilabel tasks. This metric measures the general correlation or quality of a classification.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **num\_classes** – Integer specifying the number of labels

- **threshold** (`float`) – Threshold for transforming probability to binary (0,1) predictions
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **normalize** – Normalization mode for confusion matrix. Choose from:
  - None or 'none': no normalization (default)
  - 'true': normalization over the targets (most commonly used)
  - 'pred': normalization over the predictions
  - 'all': normalization over the whole matrix
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Example (preds is int tensor):

```
>>> from torchmetrics.classification import MultilabelMatthewsCorrCoef
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelMatthewsCorrCoef(num_labels=3)
>>> metric(preds, target)
tensor(0.3333)
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import MultilabelMatthewsCorrCoef
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelMatthewsCorrCoef(num_labels=3)
>>> metric(preds, target)
tensor(0.3333)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

## 1.36.2 Functional Interface

### `matthews_corrcoef`

`torchmetrics.functional.matthews_corrcoef(preds, target, num_classes, threshold=0.5, task=None, num_labels=None, ignore_index=None, validate_args=True)`

---

**Note:** From v0.10 an `'binary_*`, `'multiclass_*`, `'multilabel_*` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

---

Calculates [Matthews correlation coefficient](#) that measures the general correlation or quality of a classification. In the binary case it is defined as:

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP) * (TP + FN) * (TN + FP) * (TN + FN)}}$$

where TP, TN, FP and FN are respectively the true positives, true negatives, false positives and false negatives. Also works in the case of multi-label or multi-class input.

#### Parameters

- **preds** ([Tensor](#)) – (float or long tensor), Either a (N, ...) tensor with labels or (N, C, ...) where C is the number of classes, tensor with labels/probabilities
- **target** ([Tensor](#)) – target (long tensor), tensor with shape (N, ...) with ground true labels
- **num\_classes** ([int](#)) – Number of classes in the dataset.
- **threshold** ([float](#)) – Threshold value for binary or multi-label probabilities.

#### Example

```
>>> from torchmetrics.functional import matthews_corrcoef
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0, 1, 0, 0])
>>> matthews_corrcoef(preds, target, num_classes=2)
tensor(0.5774)
```

Return type [Tensor](#)

### binary\_matthews\_corrcoef

`torchmetrics.functional.classification.binary_matthews_corrcoef(preds, target, threshold=0.5, ignore_index=None, validate_args=True)`

Calculates [Matthews correlation coefficient](#) for binary tasks. This metric measures the general correlation or quality of a classification.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

Additional dimension ... will be flattened into the batch dimension.

#### Parameters

- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **normalize** – Normalization mode for confusion matrix. Choose from:
  - None or 'none': no normalization (default)

- 'true': normalization over the targets (most commonly used)
- 'pred': normalization over the predictions
- 'all': normalization over the whole matrix
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import binary_matthews_corrcoef
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0, 1, 0, 0])
>>> binary_matthews_corrcoef(preds, target)
tensor(0.5774)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import binary_matthews_corrcoef
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0.35, 0.85, 0.48, 0.01])
>>> binary_matthews_corrcoef(preds, target)
tensor(0.5774)
```

**Return type** `Tensor`

## **multiclass\_matthews\_corrcoef**

`torchmetrics.functional.classification.multiclass_matthews_corrcoef(preds, target, num_classes, ignore_index=None, validate_args=True)`

Calculates [Matthews correlation coefficient](#) for multiclass tasks. This metric measures the general correlation or quality of a classification.

Accepts the following input tensors:

- **preds**: (`N, ...`) (int tensor) or (`N, C, ...`) (float tensor). If preds is a floating point we apply `torch.argmax` along the `C` dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (`N, ...`)

Additional dimension `...` will be flattened into the batch dimension.

### **Parameters**

- **num\_classes** (`int`) – Integer specifying the number of classes
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **normalize** – Normalization mode for confusion matrix. Choose from:
  - `None` or `'none'`: no normalization (default)
  - `'true'`: normalization over the targets (most commonly used)
  - `'pred'`: normalization over the predictions

- 'all': normalization over the whole matrix
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Example (pred is integer tensor):

```
>>> from torchmetrics.functional.classification import multiclass_matthews_
    <-corrcoef
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> multiclass_matthews_corrcoef(preds, target, num_classes=3)
tensor(0.7000)
```

Example (pred is float tensor):

```
>>> from torchmetrics.functional.classification import multiclass_matthews_
    <-corrcoef
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> multiclass_matthews_corrcoef(preds, target, num_classes=3)
tensor(0.7000)
```

Return type `Tensor`

### multilabel\_matthews\_corrcoef

`torchmetrics.functional.classification.multilabel_matthews_corrcoef(preds, target, num_labels, threshold=0.5, ignore_index=None, validate_args=True)`

Calculates [Matthews correlation coefficient](#) for multilabel tasks. This metric measures the general correlation or quality of a classification.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

Additional dimension ... will be flattened into the batch dimension.

#### Parameters

- **num\_classes** – Integer specifying the number of labels
- **threshold** (`float`) – Threshold for transforming probability to binary (0,1) predictions

- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **normalize** – Normalization mode for confusion matrix. Choose from:
  - `None` or `'none'`: no normalization (default)
  - `'true'`: normalization over the targets (most commonly used)
  - `'pred'`: normalization over the predictions
  - `'all'`: normalization over the whole matrix
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Example (preds is int tensor):

```
>>> from torchmetrics.functional.classification import multilabel_matthews_
    ↪ corrcoef
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_matthews_corrcoef(preds, target, num_labels=3)
tensor(0.3333)
```

Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import multilabel_matthews_
    ↪ corrcoef
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_matthews_corrcoef(preds, target, num_labels=3)
tensor(0.3333)
```

Return type `Tensor`

## 1.37 Precision

### 1.37.1 Module Interface

```
class torchmetrics.Precision(num_classes=None, threshold=0.5, average='micro', mdmc_average=None,
                             ignore_index=None, top_k=None, multiclass=None, **kwargs)
```

---

**Note:** From v0.10 an `'binary_*`, `'multiclass_*`, `'multilabel_*` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

---

Computes [Precision](#):

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Where TP and FP represent the number of true positives and false positives respectively. With the use of `top_k` parameter, this metric can generalize to Precision@K.

The reduction method (how the precision scores are aggregated) is controlled by the `average` parameter, and additionally by the `mdmc_average` parameter in the multi-dimensional multi-class case. Accepts all inputs listed in *Input types*.

### Parameters

- **num\_classes** (`Optional[int]`) – Number of classes. Necessary for 'macro', 'weighted' and None average methods.
- **threshold** (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied. Should be one of the following:
  - 'micro' [default]: Calculate the metric globally, across all samples and classes.
  - 'macro': Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
  - 'weighted': Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (`tp + fn`).
  - 'none' or None: Calculate the metric for each class separately, and return the metric for every class.
  - 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

---

**Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_average`.

---

- **mdmc\_average** (`Optional[str]`) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
  - None [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - 'samplewise': In this case, the statistics are computed separately for each sample on the N axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... (see *Input types*) as the N dimension within the sample, and computing the metric for the sample based on that.
  - 'global': In this case the N and ... dimensions of the inputs (see *Input types*) are flattened into a new N\_X sample axis, i.e. the inputs are treated as if they were (N\_X, C). From here on the `average` parameter applies as usual.
- **ignore\_index** (`Optional[int]`) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or 'none', the score for the ignored class will be returned as nan.
- **top\_k** (`Optional[int]`) – Number of the highest probability or logit score predictions considered finding the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (None) will be interpreted as 1 for these inputs. Should be left at default (None) for all other types of inputs.

- **multiclass** (*Optional[bool]*) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter’s [documentation section](#) for a more detailed explanation and examples.
- **kwargs** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises** **ValueError** – If average is none of "micro", "macro", "weighted", "samples", "none", None.

### Example

```
>>> import torch
>>> from torchmetrics import Precision
>>> preds = torch.tensor([2, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> precision = Precision(average='macro', num_classes=3)
>>> precision(preds, target)
tensor(0.1667)
>>> precision = Precision(average='micro')
>>> precision(preds, target)
tensor(0.2500)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### **compute()**

Computes the precision score based on inputs passed in to update previously.

#### **Returns**

- If average in ['micro', 'macro', 'weighted', 'samples'], a one-element tensor will be returned
- If average in ['none', None], the shape will be (C,), where C stands for the number of classes

**Return type** The shape of the returned tensor depends on the average parameter

### BinaryPrecision

```
class torchmetrics.classification.BinaryPrecision(threshold=0.5, multidim_average='global',
                                                  ignore_index=None, validate_args=True,
                                                  **kwargs)
```

Computes [Precision](#) for binary tasks:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Where TP and FP represent the number of true positives and false positives respectively.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.



**Parameters**

- **threshold** (`float`) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (`Literal`['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the `N` axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional`[`int`]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – `bool` indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Returns** If `multidim_average` is set to `global`, the metric returns a scalar value. If `multidim_average` is set to `samplewise`, the metric returns (`N,`) vector consisting of a scalar value per sample.

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import BinaryPrecision
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0, 0, 1, 1, 0, 1])
>>> metric = BinaryPrecision()
>>> metric(preds, target)
tensor(0.6667)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import BinaryPrecision
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> metric = BinaryPrecision()
>>> metric(preds, target)
tensor(0.6667)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import BinaryPrecision
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> metric = BinaryPrecision(multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.4000, 0.0000])
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

**compute()**

Computes the final statistics.

**Return type** `Tensor`

#### Returns

The metric returns a tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to `[tp, fp, tn, fn, sup]` (`sup` stands for support and equals `tp + fn`). The shape depends on the `multidim_average` parameter:

- If `multidim_average` is set to `global`, the shape will be  $(5,)$
- If `multidim_average` is set to `samplewise`, the shape will be  $(N, 5)$

## MulticlassPrecision

```
class torchmetrics.classification.MulticlassPrecision(num_classes, top_k=1, average='macro',
                                                    multidim_average='global',
                                                    ignore_index=None, validate_args=True,
                                                    **kwargs)
```

Computes `Precision` for multiclass tasks

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Where TP and FP represent the number of true positives and false positives respectively.

Accepts the following input tensors:

- `preds`:  $(N, \dots)$  (int tensor) or  $(N, C, \dots)$  (float tensor). If `preds` is a floating point we apply `torch.argmax` along the `C` dimension to automatically convert probabilities/logits into an int tensor.
- `target` (int tensor):  $(N, \dots)$

The influence of the additional dimension  $\dots$  (if present) will be determined by the `multidim_average` argument.

#### Parameters

- **`num_classes`** (`int`) – Integer specifying the number of classes
- **`average`** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: Calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: Calculates statistic for each label and applies no reduction
- **`top_k`** (`int`) – Number of highest probability or logit score predictions considered to find the correct label. Only works when `preds` contain probabilities/logits.
- **`multidim_average`** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions  $\dots$  should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the `N` axis. The statistics in this case are calculated over the additional dimensions.
- **`ignore_index`** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation

- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

#### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be `(C,)`
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

**Return type** The returned shape depends on the `average` and `multidim_average` arguments

#### Example (preds is int tensor):

```
>>> from torchmetrics.classification import MulticlassPrecision
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> metric = MulticlassPrecision(num_classes=3)
>>> metric(preds, target)
tensor(0.8333)
>>> metric = MulticlassPrecision(num_classes=3, average=None)
>>> metric(preds, target)
tensor([1.0000, 0.5000, 1.0000])
```

#### Example (preds is float tensor):

```
>>> from torchmetrics.classification import MulticlassPrecision
>>> target = target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> metric = MulticlassPrecision(num_classes=3)
>>> metric(preds, target)
tensor(0.8333)
>>> metric = MulticlassPrecision(num_classes=3, average=None)
>>> metric(preds, target)
tensor([1.0000, 0.5000, 1.0000])
```

#### Example (multidim tensors):

```
>>> from torchmetrics.classification import MulticlassPrecision
>>> target = torch.tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = torch.tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> metric = MulticlassPrecision(num_classes=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.3889, 0.2778])
>>> metric = MulticlassPrecision(num_classes=3, multidim_average='samplewise',
↪ average=None)
```

(continues on next page)

(continued from previous page)

```
>>> metric(preds, target)
tensor([[0.6667, 0.0000, 0.5000],
        [0.0000, 0.5000, 0.3333]])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes the final statistics.

**Return type** `Tensor`

**Returns**

The metric returns a tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to `[tp, fp, tn, fn, sup]` (`sup` stands for support and equals `tp + fn`). The shape depends on `average` and `multidim_average` parameters:

- If `multidim_average` is set to `global`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(5,)$
- If `average=None/'none'`, the shape will be  $(C, 5)$
- If `multidim_average` is set to `samplewise`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(N, 5)$
- If `average=None/'none'`, the shape will be  $(N, C, 5)$

## MultilabelPrecision

```
class torchmetrics.classification.MultilabelPrecision(num_labels, threshold=0.5, average='macro',
                                                    multidim_average='global',
                                                    ignore_index=None, validate_args=True,
                                                    **kwargs)
```

Computes `Precision` for multilabel tasks

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Where TP and FP represent the number of true positives and false positives respectively.

Accepts the following input tensors:

- **preds** (int or float tensor):  $(N, C, \dots)$ . If `preds` is a floating point tensor with values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **target** (int tensor):  $(N, C, \dots)$

The influence of the additional dimension  $\dots$  (if present) will be determined by the `multidim_average` argument.

**Parameters**

- **num\_labels** (`int`) – Integer specifying the number of labels
- **threshold** (`float`) – Threshold for transforming probability to binary (0,1) predictions
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:

- micro: Sum statistics over all labels
- macro: Calculate statistics for each label and average them
- weighted: Calculates statistics for each label and computes weighted average using their support
- "none" or None: Calculates statistic for each label and applies no reduction
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - global: Additional dimensions are flattened along the batch dimension
  - samplewise: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

#### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be `(C,)`
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

**Return type** The returned shape depends on the `average` and `multidim_average` arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import MultilabelPrecision
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelPrecision(num_labels=3)
>>> metric(preds, target)
tensor(0.5000)
>>> metric = MultilabelPrecision(num_labels=3, average=None)
>>> metric(preds, target)
tensor([1.0000, 0.0000, 0.5000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import MultilabelPrecision
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelPrecision(num_labels=3)
>>> metric(preds, target)
tensor(0.5000)
>>> metric = MultilabelPrecision(num_labels=3, average=None)
>>> metric(preds, target)
tensor([1.0000, 0.0000, 0.5000])
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import MultilabelPrecision
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> metric = MultilabelPrecision(num_labels=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.3333, 0.0000])
>>> metric = MultilabelPrecision(num_labels=3, multidim_average='samplewise',
...                               average=None)
>>> metric(preds, target)
tensor([[0.5000, 0.5000, 0.0000],
        [0.0000, 0.0000, 0.0000]])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes the final statistics.

**Return type** `Tensor`

**Returns**

The metric returns a tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to  $[tp, fp, tn, fn, sup]$  ( $sup$  stands for support and equals  $tp + fn$ ). The shape depends on `average` and `multidim_average` parameters:

- If `multidim_average` is set to `global`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(5,)$
- If `average=None/'none'`, the shape will be  $(C, 5)$
- If `multidim_average` is set to `samplewise`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(N, 5)$
- If `average=None/'none'`, the shape will be  $(N, C, 5)$

### 1.37.2 Functional Interface

`torchmetrics.functional.precision(preds, target, average='micro', mdmc_average=None, ignore_index=None, num_classes=None, threshold=0.5, top_k=None, multiclass=None, task=None, num_labels=None, multidim_average='global', validate_args=True)`

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

Computes Precision

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Where TP and FP represent the number of true positives and false positives respectively. With the use of `top_k` parameter, this metric can generalize to Precision@K.

The reduction method (how the precision scores are aggregated) is controlled by the `average` parameter, and additionally by the `mdmc_average` parameter in the multi-dimensional multi-class case. Accepts all inputs listed in *Input types*.

#### Parameters

- **preds** (*Tensor*) – Predictions from model (probabilities, logits or labels)
- **target** (*Tensor*) – Ground truth values
- **average** (*Optional[Literal['micro', 'macro', 'weighted', 'none']*) – Defines the reduction that is applied. Should be one of the following:
  - 'micro' [default]: Calculate the metric globally, across all samples and classes.
  - 'macro': Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
  - 'weighted': Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (`tp + fn`).
  - 'none' or `None`: Calculate the metric for each class separately, and return the metric for every class.
  - 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

---

**Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_average`.

---



---

**Note:** If 'none' and a given class doesn't occur in the `preds` or `target`, the value for the class will be nan.

---

- **mdmc\_average** (*Optional[str]*) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
  - `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - 'samplewise': In this case, the statistics are computed separately for each sample on the N axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... (see *Input types*) as the N dimension within the sample, and computing the metric for the sample based on that.
  - 'global': In this case the N and ... dimensions of the inputs (see *Input types*) are flattened into a new N\_X sample axis, i.e. the inputs are treated as if they were (N\_X, C). From here on the `average` parameter applies as usual.
- **ignore\_index** (*Optional[int]*) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or 'none', the score for the ignored class will be returned as nan.

- **num\_classes** (`Optional[int]`) – Number of classes. Necessary for 'macro', 'weighted' and None average methods.
- **threshold** (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
- **top\_k** (`Optional[int]`) – Number of highest probability or logit score predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (None) will be interpreted as 1 for these inputs.  
Should be left at default (None) for all other types of inputs.
- **multiclass** (`Optional[bool]`) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's [documentation section](#) for a more detailed explanation and examples.

**Return type** `Tensor`

#### Returns

The shape of the returned tensor depends on the average parameter

- If average in ['micro', 'macro', 'weighted', 'samples'], a one-element tensor will be returned
- If average in ['none', None], the shape will be (C,), where C stands for the number of classes

#### Raises

- **ValueError** – If average is not one of "micro", "macro", "weighted", "samples", "none" or None
- **ValueError** – If mdmc\_average is not one of None, "samplewise", "global".
- **ValueError** – If average is set but num\_classes is not provided.
- **ValueError** – If num\_classes is set and ignore\_index is not in the range [0, num\_classes).

#### Example

```
>>> from torchmetrics.functional import precision
>>> preds = torch.tensor([2, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> precision(preds, target, average='macro', num_classes=3)
tensor(0.1667)
>>> precision(preds, target, average='micro')
tensor(0.2500)
```



## binary\_precision

`torchmetrics.functional.classification.binary_precision(preds, target, threshold=0.5, multidim_average='global', ignore_index=None, validate_args=True)`

Computes [Precision](#) for binary tasks:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Where TP and FP represent the number of true positives and false positives respectively.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

### Return type [Tensor](#)

**Returns** If *multidim\_average* is set to **global**, the metric returns a scalar value. If *multidim\_average* is set to **samplewise**, the metric returns (N,) vector consisting of a scalar value per sample.

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import binary_precision
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0, 0, 1, 1, 0, 1])
>>> binary_precision(preds, target)
tensor(0.6667)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import binary_precision
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> binary_precision(preds, target)
tensor(0.6667)
```

Example (multidim tensors):

```
>>> from torchmetrics.functional.classification import binary_precision
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> binary_precision(preds, target, multidim_average='samplewise')
tensor([0.4000, 0.0000])
```

## multiclass\_precision

`torchmetrics.functional.classification.multiclass_precision(preds, target, num_classes, average='macro', top_k=1, multidim_average='global', ignore_index=None, validate_args=True)`

Computes [Precision](#) for multiclass tasks

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Where TP and FP represent the number of true positives and false positives respectively.

Accepts the following input tensors:

- **preds** (`N, ...`) (int tensor) or (`N, C, ...`) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (`N, ...`)

The influence of the additional dimension ... (if present) will be determined by the `multidim_average` argument.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_classes** (`int`) – Integer specifying the number of classes
- **average** ([Optional\[Literal\['micro', 'macro', 'weighted', 'none'\]\]](#)) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: Calculates statistics for each label and computes weighted average using their support

- "none" or None: Calculates statistic for each label and applies no reduction
- **top\_k** (`int`) – Number of highest probability or logit score predictions considered to find the correct label. Only works when preds contain probabilities/logits.
- **multidim\_average** (`Literal`['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - global: Additional dimensions are flattened along the batch dimension
  - samplewise: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional`[`int`]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.

### Returns

- If multidim\_average is set to global:
  - If average='micro'/'macro'/'weighted', the output will be a scalar tensor
  - If average=None/'none', the shape will be (C,)
- If multidim\_average is set to samplewise:
  - If average='micro'/'macro'/'weighted', the shape will be (N,)
  - If average=None/'none', the shape will be (N, C)

**Return type** The returned shape depends on the average and multidim\_average arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_precision
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> multiclass_precision(preds, target, num_classes=3)
tensor(0.8333)
>>> multiclass_precision(preds, target, num_classes=3, average=None)
tensor([1.0000, 0.5000, 1.0000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_precision
>>> target = target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> multiclass_precision(preds, target, num_classes=3)
tensor(0.8333)
>>> multiclass_precision(preds, target, num_classes=3, average=None)
tensor([1.0000, 0.5000, 1.0000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import multiclass_precision
>>> target = torch.tensor([[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]])
>>> preds = torch.tensor([[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]])
>>> multiclass_precision(preds, target, num_classes=3, multidim_average=
↳ 'samplewise')
tensor([0.3889, 0.2778])
>>> multiclass_precision(preds, target, num_classes=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[0.6667, 0.0000, 0.5000],
        [0.0000, 0.5000, 0.3333]])
```

## multilabel\_precision

```
torchmetrics.functional.classification.multilabel_precision(preds, target, num_labels,
                                                            threshold=0.5, average='macro',
                                                            multidim_average='global',
                                                            ignore_index=None,
                                                            validate_args=True)
```

Computes [Precision](#) for multilabel tasks

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Where TP and FP represent the number of true positives and false positives respectively.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: Calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: Calculates statistic for each label and applies no reduction
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension

- **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be (C,)
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be (N,)
  - If `average=None/'none'`, the shape will be (N, C)

**Return type** The returned shape depends on the `average` and `multidim_average` arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_precision
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_precision(preds, target, num_labels=3)
tensor(0.5000)
>>> multilabel_precision(preds, target, num_labels=3, average=None)
tensor([1.0000, 0.0000, 0.5000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_precision
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_precision(preds, target, num_labels=3)
tensor(0.5000)
>>> multilabel_precision(preds, target, num_labels=3, average=None)
tensor([1.0000, 0.0000, 0.5000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import multilabel_precision
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> multilabel_precision(preds, target, num_labels=3, multidim_average=
↪ 'samplewise')
tensor([0.3333, 0.0000])
>>> multilabel_precision(preds, target, num_labels=3, multidim_average=
↪ 'samplewise', average=None)
```

(continues on next page)

(continued from previous page)

```
tensor([[0.5000, 0.5000, 0.0000],
        [0.0000, 0.0000, 0.0000]])
```

## 1.38 Precision Recall

### 1.38.1 Functional Interface

```
torchmetrics.functional.precision_recall(preds, target, average='micro', mdmc_average=None,
                                         ignore_index=None, num_classes=None, threshold=0.5,
                                         top_k=None, multiclass=None)
```

Computes Precision

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where  $\text{TP}$ ,  $\text{FN}$  and  $\text{FP}$  represent the number of true positives, false negatives and false positives respectively. With the use of `top_k` parameter, this metric can generalize to Recall@K and Precision@K.

The reduction method (how the recall scores are aggregated) is controlled by the `average` parameter, and additionally by the `mdmc_average` parameter in the multi-dimensional multi-class case. Accepts all inputs listed in *Input types*.

#### Parameters

- **preds** (*Tensor*) – Predictions from model (probabilities, logits or labels)
- **target** (*Tensor*) – Ground truth values
- **average** (*Optional*[*Literal*['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied. Should be one of the following:
  - 'micro' [default]: Calculate the metric globally, across all samples and classes.
  - 'macro': Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
  - 'weighted': Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (`tp + fn`).
  - 'none' or `None`: Calculate the metric for each class separately, and return the metric for every class.
  - 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

---

**Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_average`.

---



---

**Note:** If 'none' and a given class doesn't occur in the `preds` or `target`, the value for the class will be nan.

---

- **mdmc\_average** (`Optional[str]`) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
  - `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - `'samplewise'`: In this case, the statistics are computed separately for each sample on the `N` axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... (see [Input types](#)) as the `N` dimension within the sample, and computing the metric for the sample based on that.
  - `'global'`: In this case the `N` and ... dimensions of the inputs (see [Input types](#)) are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were `(N_X, C)`. From here on the `average` parameter applies as usual.
- **ignore\_index** (`Optional[int]`) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or `'none'`, the score for the ignored class will be returned as `nan`.
- **num\_classes** (`Optional[int]`) – Number of classes. Necessary for `'macro'`, `'weighted'` and `None` average methods.
- **threshold** (`float`) – Threshold for transforming probability or logit predictions to binary `(0,1)` predictions, in the case of binary or multi-label inputs. Default value of `0.5` corresponds to input being probabilities.
- **top\_k** (`Optional[int]`) – Number of highest probability or logit score predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (`None`) will be interpreted as `1` for these inputs.  
Should be left at default (`None`) for all other types of inputs.
- **multiclass** (`Optional[bool]`) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's [documentation section](#) for a more detailed explanation and examples.

### Returns

precision and recall. Their shape depends on the `average` parameter

- If `average` in `['micro', 'macro', 'weighted', 'samples']`, they are a single element tensor
- If `average` in `['none', None]`, they are a tensor of shape `(C, )`, where `C` stands for the number of classes

**Return type** The function returns a tuple with two elements

### Raises

- **ValueError** – If `average` is not one of `"micro", "macro", "weighted", "samples", "none"` or `None`
- **ValueError** – If `mdmc_average` is not one of `None, "samplewise", "global"`.
- **ValueError** – If `average` is set but `num_classes` is not provided.
- **ValueError** – If `num_classes` is set and `ignore_index` is not in the range `[0, num_classes)`.

### Example

```
>>> from torchmetrics.functional import precision_recall
>>> preds = torch.tensor([2, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> precision_recall(preds, target, average='macro', num_classes=3)
(tensor(0.1667), tensor(0.3333))
>>> precision_recall(preds, target, average='micro')
(tensor(0.2500), tensor(0.2500))
```

## 1.39 Precision Recall Curve

### 1.39.1 Module Interface

**class** torchmetrics.PrecisionRecallCurve(num\_classes=None, pos\_label=None, \*\*kwargs)

---

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entypoint to calling the three specialized versions.

---

Computes precision-recall pairs for different thresholds. Works for both binary and multiclass problems. In the case of multiclass, the values will be calculated based on a one-vs-the-rest approach.

Forward accepts

- **preds** (float tensor): (N, ...) (binary) or (N, C, ...) (multiclass) tensor with probabilities, where C is the number of classes.
- **target** (long tensor): (N, ...) or (N, C, ...) with integer labels

#### Parameters

- **num\_classes** (Optional[int]) – integer with number of classes for multi-label and multi-class problems. Should be set to None for binary problems
- **pos\_label** (Optional[int]) – integer determining the positive class. Default is None which for binary problem is translated to 1. For multiclass problems this argument should not be set as we iteratively change it in the range `[0, num_classes-1]`
- **kwargs** (Any) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (binary case):**

```
>>> from torchmetrics import PrecisionRecallCurve
>>> pred = torch.tensor([0, 0.1, 0.8, 0.4])
>>> target = torch.tensor([0, 1, 1, 0])
>>> pr_curve = PrecisionRecallCurve(pos_label=1)
>>> precision, recall, thresholds = pr_curve(pred, target)
>>> precision
tensor([0.6667, 0.5000, 1.0000, 1.0000])
```

(continues on next page)



(continued from previous page)

```
>>> recall
tensor([1.0000, 0.5000, 0.5000, 0.0000])
>>> thresholds
tensor([0.1000, 0.4000, 0.8000])
```

**Example (multiclass case):**

```
>>> pred = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                      [0.05, 0.75, 0.05, 0.05, 0.05],
...                      [0.05, 0.05, 0.75, 0.05, 0.05],
...                      [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> pr_curve = PrecisionRecallCurve(num_classes=5)
>>> precision, recall, thresholds = pr_curve(pred, target)
>>> precision
[tensor([1., 1.]), tensor([1., 1.]), tensor([0.2500, 0.0000, 1.0000]),
 tensor([0.2500, 0.0000, 1.0000]), tensor([0., 1.])]
>>> recall
[tensor([1., 0.]), tensor([1., 0.]), tensor([1., 0., 0.]), tensor([1., 0., 0.]),
 ↪ tensor([nan, 0.])]
>>> thresholds
[tensor(0.7500), tensor(0.7500), tensor([0.0500, 0.7500]), tensor([0.0500, 0.
 ↪ 7500]), tensor(0.0500)]
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Compute the precision-recall curve.

**Return type** `Union[Tuple[Tensor, Tensor, Tensor], Tuple[List[Tensor], List[Tensor], List[Tensor]]]`

**Returns**

3-element tuple containing

**precision:** tensor where element *i* is the precision of predictions with `score >= thresholds[i]` and the last element is 1. If multiclass, this is a list of such tensors, one for each class.

**recall:** tensor where element *i* is the recall of predictions with `score >= thresholds[i]` and the last element is 0. If multiclass, this is a list of such tensors, one for each class.

**thresholds:** Thresholds used for computing precision/recall scores

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** ([Tensor](#)) – Predictions from model
- **target** ([Tensor](#)) – Ground truth values

**Return type** `None`

## BinaryPrecisionRecallCurve

```
class torchmetrics.classification.BinaryPrecisionRecallCurve(thresholds=None,
                                                            ignore_index=None,
                                                            validate_args=True, **kwargs)
```

Computes the precision-recall curve for binary tasks. The curve consist of multiple pairs of precision and recall values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

### Parameters

- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.
- **kwargs** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Returns

a tuple of 3 tensors containing:

- precision: an 1d tensor of size (n\_thresholds+1, ) with precision values
- recall: an 1d tensor of size (n\_thresholds+1, ) with recall values
- thresholds: an 1d tensor of size (n\_thresholds, ) with increasing threshold values

**Return type** (`tuple`)

## Example

```

>>> from torchmetrics.classification import BinaryPrecisionRecallCurve
>>> preds = torch.tensor([0, 0.5, 0.7, 0.8])
>>> target = torch.tensor([0, 1, 1, 0])
>>> metric = BinaryPrecisionRecallCurve(thresholds=None)
>>> metric(preds, target)
(tensor([0.6667, 0.5000, 0.0000, 1.0000]),
 tensor([1.0000, 0.5000, 0.0000, 0.0000]),
 tensor([0.5000, 0.7000, 0.8000]))
>>> metric = BinaryPrecisionRecallCurve(thresholds=5)
>>> metric(preds, target)
(tensor([0.5000, 0.6667, 0.6667, 0.0000, 0.0000, 1.0000]),
 tensor([1., 1., 1., 0., 0., 0.]),
 tensor([0.0000, 0.2500, 0.5000, 0.7500, 1.0000]))

```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

### compute()

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tuple[Tensor, Tensor, Tensor]`

### update(preds, target)

Override this method to update the state variables of your metric class.

**Return type** `None`

## MulticlassPrecisionRecallCurve

```

class torchmetrics.classification.MulticlassPrecisionRecallCurve(num_classes, thresholds=None,
                                                                ignore_index=None,
                                                                validate_args=True, **kwargs)

```

Computes the precision-recall curve for multiclass tasks. The curve consist of multiple pairs of precision and recall values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the [0, n\_classes-1] range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

### Parameters

- **num\_classes** (`int`) – Integer specifying the number of classes

- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an `int` (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an `list` of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d `tensor` of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Returns

a tuple of either 3 tensors or 3 lists containing

- precision: if `thresholds=None` a list for each class is returned with an 1d tensor of size (`n_thresholds+1,` ) with precision values (length may differ between classes). If `thresholds` is set to something else, then a single 2d tensor of size (`n_classes, n_thresholds+1`) with precision values is returned.
- recall: if `thresholds=None` a list for each class is returned with an 1d tensor of size (`n_thresholds+1,` ) with recall values (length may differ between classes). If `thresholds` is set to something else, then a single 2d tensor of size (`n_classes, n_thresholds+1`) with recall values is returned.
- thresholds: if `thresholds=None` a list for each class is returned with an 1d tensor of size (`n_thresholds,` ) with increasing threshold values (length may differ between classes). If `threshold` is set to something else, then a single 1d tensor of size (`n_thresholds,` ) is returned with shared threshold values for all classes.

**Return type** (`tuple`)

### Example

```
>>> from torchmetrics.classification import MulticlassPrecisionRecallCurve
>>> preds = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                       [0.05, 0.75, 0.05, 0.05, 0.05],
...                       [0.05, 0.05, 0.75, 0.05, 0.05],
...                       [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> metric = MulticlassPrecisionRecallCurve(num_classes=5, thresholds=None)
>>> precision, recall, thresholds = metric(preds, target)
>>> precision
[tensor([1., 1.]), tensor([1., 1.]), tensor([0.2500, 0.0000, 1.0000]),
 tensor([0.2500, 0.0000, 1.0000]), tensor([0., 1.])]
>>> recall
[tensor([1., 0.]), tensor([1., 0.]), tensor([1., 0., 0.]), tensor([1., 0., 0.]),
 → tensor([nan, 0.])]
>>> thresholds
[tensor(0.7500), tensor(0.7500), tensor([0.0500, 0.7500]), tensor([0.0500, 0.7500]),
 → tensor(0.0500)]
```

(continues on next page)

(continued from previous page)

```
>>> metric = MulticlassPrecisionRecallCurve(num_classes=5, thresholds=5)
>>> metric(preds, target)
(tensor([[0.2500, 1.0000, 1.0000, 1.0000, 0.0000, 1.0000],
        [0.2500, 1.0000, 1.0000, 1.0000, 0.0000, 1.0000],
        [0.2500, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000],
        [0.2500, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000]]),
 tensor([[1., 1., 1., 1., 0., 0.],
        [1., 1., 1., 1., 0., 0.],
        [1., 0., 0., 0., 0., 0.],
        [1., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0.])),
 tensor([0.0000, 0.2500, 0.5000, 0.7500, 1.0000]))
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** Union[Tuple[Tensor, Tensor, Tensor], Tuple[List[Tensor], List[Tensor], List[Tensor]]]

**update(preds, target)**

Override this method to update the state variables of your metric class.

**Return type** None

## MultilabelPrecisionRecallCurve

```
class torchmetrics.classification.MultilabelPrecisionRecallCurve(num_labels, thresholds=None,
                                                                ignore_index=None,
                                                                validate_args=True, **kwargs)
```

Computes the precision-recall curve for multilabel tasks. The curve consist of multiple pairs of precision and recall values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

### Parameters

- **preds** – Tensor with predictions

- **target** – Tensor with true labels
- **num\_labels** (`int`) – Integer specifying the number of labels
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an `int` (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an `list` of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d `tensor` of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

a tuple of either 3 tensors or 3 lists containing

- precision: if `thresholds=None` a list for each label is returned with an 1d tensor of size (`n_thresholds+1,` ) with precision values (length may differ between labels). If `thresholds` is set to something else, then a single 2d tensor of size (`n_labels, n_thresholds+1`) with precision values is returned.
- recall: if `thresholds=None` a list for each label is returned with an 1d tensor of size (`n_thresholds+1,` ) with recall values (length may differ between labels). If `thresholds` is set to something else, then a single 2d tensor of size (`n_labels, n_thresholds+1`) with recall values is returned.
- thresholds: if `thresholds=None` a list for each label is returned with an 1d tensor of size (`n_thresholds,` ) with increasing threshold values (length may differ between labels). If `threshold` is set to something else, then a single 1d tensor of size (`n_thresholds,` ) is returned with shared threshold values for all labels.

**Return type** (`tuple`)

### Example

```
>>> from torchmetrics.classification import MultilabelPrecisionRecallCurve
>>> preds = torch.tensor([[0.75, 0.05, 0.35],
...                       [0.45, 0.75, 0.05],
...                       [0.05, 0.55, 0.75],
...                       [0.05, 0.65, 0.05]])
>>> target = torch.tensor([[1, 0, 1],
...                        [0, 0, 0],
...                        [0, 1, 1],
...                        [1, 1, 1]])
>>> metric = MultilabelPrecisionRecallCurve(num_labels=3, thresholds=None)
>>> precision, recall, thresholds = metric(preds, target)
>>> precision
(tensor([0.5000, 0.5000, 1.0000, 1.0000]), tensor([0.6667, 0.5000, 0.0000, 1.0000]),
 tensor([0.7500, 1.0000, 1.0000, 1.0000]))
>>> recall
```

(continues on next page)

(continued from previous page)

```
[tensor([1.0000, 0.5000, 0.5000, 0.0000]), tensor([1.0000, 0.5000, 0.0000, 0.0000]),
 tensor([1.0000, 0.6667, 0.3333, 0.0000])]
>>> thresholds
[tensor([0.0500, 0.4500, 0.7500]), tensor([0.5500, 0.6500, 0.7500]),
 tensor([0.0500, 0.3500, 0.7500])]
>>> metric = MultilabelPrecisionRecallCurve(num_labels=3, thresholds=5)
>>> metric(preds, target)
(tensor([[0.5000, 0.5000, 1.0000, 1.0000, 0.0000, 1.0000],
        [0.5000, 0.6667, 0.6667, 0.0000, 0.0000, 1.0000],
        [0.7500, 1.0000, 1.0000, 1.0000, 0.0000, 1.0000]]),
 tensor([[1.0000, 0.5000, 0.5000, 0.5000, 0.0000, 0.0000],
        [1.0000, 1.0000, 1.0000, 0.0000, 0.0000, 0.0000],
        [1.0000, 0.6667, 0.3333, 0.3333, 0.0000, 0.0000]]),
 tensor([0.0000, 0.2500, 0.5000, 0.7500, 1.0000]))
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### **compute()**

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** Union[Tuple[Tensor, Tensor, Tensor], Tuple[List[Tensor], List[Tensor], List[Tensor]]]

#### **update(preds, target)**

Override this method to update the state variables of your metric class.

**Return type** None

## 1.39.2 Functional Interface

`torchmetrics.functional.precision_recall_curve(preds, target, num_classes=None, pos_label=None, sample_weights=None, task=None, num_labels=None, thresholds=None, ignore_index=None, validate_args=True)`

**Note:** From v0.10 an `'binary_*`, `'multiclass_*`, `'multilabel_*` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

Computes precision-recall pairs for different thresholds.

#### **Parameters**

- **preds** (Tensor) – predictions from model (probabilities).
- **target** (Tensor) – ground truth labels.
- **num\_classes** (Optional[int]) – integer with number of classes for multi-label and multi-class problems. Should be set to None for binary problems.

- **pos\_label** (`Optional[int]`) – integer determining the positive class. Default is `None` which for binary problem is translated to 1. For multiclass problems this argument should not be set as we iteratively change it in the range `[0, num_classes-1]`.
- **sample\_weights** (`Optional[Sequence]`) – sample weights for each data point.

**Return type** `Union[Tuple[Tensor, Tensor, Tensor], Tuple[List[Tensor], List[Tensor], List[Tensor]]]`

#### Returns

3-element tuple containing

**precision:** tensor where element `i` is the precision of predictions with `score >= thresholds[i]` and the last element is 1. If multiclass, this is a list of such tensors, one for each class.

**recall:** tensor where element `i` is the recall of predictions with `score >= thresholds[i]` and the last element is 0. If multiclass, this is a list of such tensors, one for each class.

**thresholds:** Thresholds used for computing precision/recall scores.

#### Raises

- **ValueError** – If preds and target don't have the same number of dimensions, or one additional dimension for preds.
- **ValueError** – If the number of classes deduced from preds is not the same as the `num_classes` provided.

#### Example (binary case):

```
>>> from torchmetrics.functional import precision_recall_curve
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 1, 0])
>>> precision, recall, thresholds = precision_recall_curve(pred, target, pos_
↳ label=1)
>>> precision
tensor([0.6667, 0.5000, 0.0000, 1.0000])
>>> recall
tensor([1.0000, 0.5000, 0.0000, 0.0000])
>>> thresholds
tensor([1, 2, 3])
```

#### Example (multiclass case):

```
>>> pred = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                      [0.05, 0.75, 0.05, 0.05, 0.05],
...                      [0.05, 0.05, 0.75, 0.05, 0.05],
...                      [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> precision, recall, thresholds = precision_recall_curve(pred, target, num_
↳ classes=5)
>>> precision
[tensor([1., 1.]), tensor([1., 1.]), tensor([0.2500, 0.0000, 1.0000]),
 tensor([0.2500, 0.0000, 1.0000]), tensor([0., 1.])]
>>> recall
[tensor([1., 0.]), tensor([1., 0.]), tensor([1., 0., 0.]), tensor([1., 0., 0.]),
↳ tensor([nan, 0.])]
```

(continues on next page)



(continued from previous page)

```
>>> thresholds
[tensor([0.7500]), tensor([0.7500]), tensor([0.0500, 0.7500]), tensor([0.0500, 0.7500]), tensor([0.0500])]
```

## binary\_precision\_recall\_curve

```
torchmetrics.functional.classification.binary_precision_recall_curve(preds, target,
                                                                    thresholds=None,
                                                                    ignore_index=None,
                                                                    validate_args=True)
```

Computes the precision-recall curve for binary tasks. The curve consist of multiple pairs of precision and recall values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **thresholds** ([Union\[int, List\[float\], Tensor, None\]](#)) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

### Returns

a tuple of 3 tensors containing:

- precision: an 1d tensor of size (n\_thresholds+1, ) with precision values
- recall: an 1d tensor of size (n\_thresholds+1, ) with recall values
- thresholds: an 1d tensor of size (n\_thresholds, ) with increasing threshold values

Return type (tuple)

### Example

```
>>> from torchmetrics.functional.classification import binary_precision_recall_curve
>>> preds = torch.tensor([0, 0.5, 0.7, 0.8])
>>> target = torch.tensor([0, 1, 1, 0])
>>> binary_precision_recall_curve(preds, target, thresholds=None)
(tensor([0.6667, 0.5000, 0.0000, 1.0000]),
 tensor([1.0000, 0.5000, 0.0000, 0.0000]),
 tensor([0.5000, 0.7000, 0.8000]))
>>> binary_precision_recall_curve(preds, target, thresholds=5)
(tensor([0.5000, 0.6667, 0.6667, 0.0000, 0.0000, 1.0000]),
 tensor([1., 1., 1., 0., 0., 0.]),
 tensor([0.0000, 0.2500, 0.5000, 0.7500, 1.0000]))
```

### multiclass\_precision\_recall\_curve

`torchmetrics.functional.classification.multiclass_precision_recall_curve(preds, target, num_classes, thresholds=None, ignore_index=None, validate_args=True)`

Computes the precision-recall curve for multiclass tasks. The curve consist of multiple pairs of precision and recall values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the [0, n\_classes-1] range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

#### Parameters

- **preds** (Tensor) – Tensor with predictions
- **target** (Tensor) – Tensor with true labels
- **num\_classes** (int) – Integer specifying the number of classes
- **thresholds** (Union[int, List[float], Tensor, None]) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.

- If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
- If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (*bool*) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

### Returns

a tuple of either 3 tensors or 3 lists containing

- **precision**: if *thresholds=None* a list for each class is returned with an 1d tensor of size (*n\_thresholds*+1, ) with precision values (length may differ between classes). If *thresholds* is set to something else, then a single 2d tensor of size (*n\_classes*, *n\_thresholds*+1) with precision values is returned.
- **recall**: if *thresholds=None* a list for each class is returned with an 1d tensor of size (*n\_thresholds*+1, ) with recall values (length may differ between classes). If *thresholds* is set to something else, then a single 2d tensor of size (*n\_classes*, *n\_thresholds*+1) with recall values is returned.
- **thresholds**: if *thresholds=None* a list for each class is returned with an 1d tensor of size (*n\_thresholds*, ) with increasing threshold values (length may differ between classes). If *threshold* is set to something else, then a single 1d tensor of size (*n\_thresholds*, ) is returned with shared threshold values for all classes.

**Return type** (*tuple*)

### Example

```
>>> from torchmetrics.functional.classification import multiclass_precision_recall_
↳ curve
>>> preds = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                       [0.05, 0.75, 0.05, 0.05, 0.05],
...                       [0.05, 0.05, 0.75, 0.05, 0.05],
...                       [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> precision, recall, thresholds = multiclass_precision_recall_curve(
...     preds, target, num_classes=5, thresholds=None
... )
>>> precision
[tensor([1., 1.]), tensor([1., 1.]), tensor([0.2500, 0.0000, 1.0000]),
 tensor([0.2500, 0.0000, 1.0000]), tensor([0., 1.])]
>>> recall
[tensor([1., 0.]), tensor([1., 0.]), tensor([1., 0., 0.]), tensor([1., 0., 0.]),
↳ tensor([nan, 0.])]
>>> thresholds
[tensor([0.7500]), tensor([0.7500]), tensor([0.0500, 0.7500]), tensor([0.0500, 0.
↳ 7500]), tensor([0.0500])]
>>> multiclass_precision_recall_curve(
...     preds, target, num_classes=5, thresholds=5
... )
(tensor([[0.2500, 1.0000, 1.0000, 1.0000, 0.0000, 1.0000],
        [0.2500, 1.0000, 1.0000, 1.0000, 0.0000, 1.0000],
```

(continues on next page)

(continued from previous page)

```

        [0.2500, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000],
        [0.2500, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000]]),
    tensor([[1., 1., 1., 1., 0., 0.],
           [1., 1., 1., 1., 0., 0.],
           [1., 0., 0., 0., 0., 0.],
           [1., 0., 0., 0., 0., 0.],
           [0., 0., 0., 0., 0., 0.])),
    tensor([0.0000, 0.2500, 0.5000, 0.7500, 1.0000]))

```

## multilabel\_precision\_recall\_curve

```

torchmetrics.functional.classification.multilabel_precision_recall_curve(preds, target,
                                                                           num_labels,
                                                                           thresholds=None,
                                                                           ignore_index=None,
                                                                           validate_args=True)

```

Computes the precision-recall curve for multilabel tasks. The curve consist of multiple pairs of precision and recall values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **thresholds** ([Union\[int, List\[float\], Tensor, None\]](#)) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.

- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

a tuple of either 3 tensors or 3 lists containing

- precision: if *thresholds*=`None` a list for each label is returned with an 1d tensor of size (`n_thresholds+1`, ) with precision values (length may differ between labels). If *thresholds* is set to something else, then a single 2d tensor of size (`n_labels`, `n_thresholds+1`) with precision values is returned.
- recall: if *thresholds*=`None` a list for each label is returned with an 1d tensor of size (`n_thresholds+1`, ) with recall values (length may differ between labels). If *thresholds* is set to something else, then a single 2d tensor of size (`n_labels`, `n_thresholds+1`) with recall values is returned.
- thresholds: if *thresholds*=`None` a list for each label is returned with an 1d tensor of size (`n_thresholds`, ) with increasing threshold values (length may differ between labels). If *threshold* is set to something else, then a single 1d tensor of size (`n_thresholds`, ) is returned with shared threshold values for all labels.

**Return type** (`tuple`)

### Example

```
>>> from torchmetrics.functional.classification import multilabel_precision_recall_
↳ curve
>>> preds = torch.tensor([[0.75, 0.05, 0.35],
...                        [0.45, 0.75, 0.05],
...                        [0.05, 0.55, 0.75],
...                        [0.05, 0.65, 0.05]])
>>> target = torch.tensor([[1, 0, 1],
...                        [0, 0, 0],
...                        [0, 1, 1],
...                        [1, 1, 1]])
>>> precision, recall, thresholds = multilabel_precision_recall_curve(
...     preds, target, num_labels=3, thresholds=None
... )
>>> precision
(tensor([0.5000, 0.5000, 1.0000, 1.0000]), tensor([0.6667, 0.5000, 0.0000, 1.0000]),
 tensor([0.7500, 1.0000, 1.0000, 1.0000]))
>>> recall
(tensor([1.0000, 0.5000, 0.5000, 0.0000]), tensor([1.0000, 0.5000, 0.0000, 0.0000]),
 tensor([1.0000, 0.6667, 0.3333, 0.0000]))
>>> thresholds
(tensor([0.0500, 0.4500, 0.7500]), tensor([0.5500, 0.6500, 0.7500]),
 tensor([0.0500, 0.3500, 0.7500]))
>>> multilabel_precision_recall_curve(
...     preds, target, num_labels=3, thresholds=5
... )
(tensor([[0.5000, 0.5000, 1.0000, 1.0000, 0.0000, 1.0000],
         [0.5000, 0.6667, 0.6667, 0.0000, 0.0000, 1.0000],
         [0.7500, 1.0000, 1.0000, 1.0000, 0.0000, 1.0000]]),
 tensor([[1.0000, 0.5000, 0.5000, 0.5000, 0.0000, 0.0000],
```

(continues on next page)

(continued from previous page)

```
[1.0000, 1.0000, 1.0000, 0.0000, 0.0000, 0.0000],
 [1.0000, 0.6667, 0.3333, 0.3333, 0.0000, 0.0000]]),
 tensor([0.0000, 0.2500, 0.5000, 0.7500, 1.0000]))
```

## 1.40 Recall

### 1.40.1 Module Interface

```
class torchmetrics.Recall(num_classes=None, threshold=0.5, average='micro', mdmc_average=None,
                          ignore_index=None, top_k=None, multiclass=None, **kwargs)
```

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as a single entrypoint to calling the three specialized versions.

Computes *Recall*:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where TP and FN represent the number of true positives and false negatives respectively. With the use of `top_k` parameter, this metric can generalize to Recall@K.

The reduction method (how the recall scores are aggregated) is controlled by the `average` parameter, and additionally by the `mdmc_average` parameter in the multi-dimensional multi-class case. Accepts all inputs listed in *Input types*.

#### Parameters

- **num\_classes** (`Optional[int]`) – Number of classes. Necessary for 'macro', 'weighted' and None average methods.
- **threshold** (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied. Should be one of the following:
  - 'micro' [default]: Calculate the metric globally, across all samples and classes.
  - 'macro': Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
  - 'weighted': Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (`tp + fn`).
  - 'none' or None: Calculate the metric for each class separately, and return the metric for every class.
  - 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

---

**Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_average`.

---

- **`mdmc_average`** ([Optional\[str\]](#)) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
  - `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - `'samplewise'`: In this case, the statistics are computed separately for each sample on the `N` axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... (see [Input types](#)) as the `N` dimension within the sample, and computing the metric for the sample based on that.
  - `'global'`: In this case the `N` and ... dimensions of the inputs (see [Input types](#)) are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were `(N_X, C)`. From here on the `average` parameter applies as usual.
- **`ignore_index`** ([Optional\[int\]](#)) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or `'none'`, the score for the ignored class will be returned as `nan`.
- **`top_k`** ([Optional\[int\]](#)) – Number of the highest probability or logit score predictions considered finding the correct label, relevant only for (multi-dimensional) multi-class. The default value (`None`) will be interpreted as 1 for these inputs.  
Should be left at default (`None`) for all other types of inputs.
- **`multiclass`** ([Optional\[bool\]](#)) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's [documentation section](#) for a more detailed explanation and examples.
- **`kwargs`** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises** [ValueError](#) – If `average` is none of "micro", "macro", "weighted", "samples", "none", `None`.

### Example

```
>>> import torch
>>> from torchmetrics import Recall
>>> preds = torch.tensor([2, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> recall = Recall(average='macro', num_classes=3)
>>> recall(preds, target)
tensor(0.3333)
>>> recall = Recall(average='micro')
>>> recall(preds, target)
tensor(0.2500)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

#### **`compute()`**

Computes the recall score based on inputs passed in to `update` previously.

#### **Returns**

- If `average` in `['micro', 'macro', 'weighted', 'samples']`, a one-element tensor will be returned
- If `average` in `['none', None]`, the shape will be `(C,)`, where `C` stands for the number of classes

**Return type** The shape of the returned tensor depends on the `average` parameter

## BinaryRecall

```
class torchmetrics.classification.BinaryRecall(threshold=0.5, multidim_average='global',
                                              ignore_index=None, validate_args=True, **kwargs)
```

Computes *Recall* for binary tasks:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where TP and FN represent the number of true positives and false negatives respectively.

Accepts the following input tensors:

- `preds` (int or float tensor): `(N, ...)`. If `preds` is a floating point tensor with values outside `[0,1]` range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- `target` (int tensor): `(N, ...)`

The influence of the additional dimension ... (if present) will be determined by the `multidim_average` argument.

### Parameters

- **`threshold`** (`float`) – Threshold for transforming probability to binary `{0,1}` predictions
- **`multidim_average`** (`Literal``['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the `N` axis. The statistics in this case are calculated over the additional dimensions.
- **`ignore_index`** (`Optional``[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **`validate_args`** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Returns** If `multidim_average` is set to `global`, the metric returns a scalar value. If `multidim_average` is set to `samplewise`, the metric returns `(N,)` vector consisting of a scalar value per sample.

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import BinaryRecall
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0, 0, 1, 1, 0, 1])
>>> metric = BinaryRecall()
>>> metric(preds, target)
tensor(0.6667)
```

**Example (preds is float tensor):**



```
>>> from torchmetrics.classification import BinaryRecall
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> metric = BinaryRecall()
>>> metric(preds, target)
tensor(0.6667)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import BinaryRecall
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> metric = BinaryRecall(multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.6667, 0.0000])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes the final statistics.

**Return type** `Tensor`

**Returns**

The metric returns a tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to  $[tp, fp, tn, fn, sup]$  ( $sup$  stands for support and equals  $tp + fn$ ). The shape depends on the `multidim_average` parameter:

- If `multidim_average` is set to `global`, the shape will be  $(5,)$
- If `multidim_average` is set to `samplewise`, the shape will be  $(N, 5)$

## MulticlassRecall

```
class torchmetrics.classification.MulticlassRecall(num_classes, top_k=1, average='macro',
                                                    multidim_average='global', ignore_index=None,
                                                    validate_args=True, **kwargs)
```

Computes *Recall* for multiclass tasks:

$$\text{Recall} = \frac{TP}{TP + FN}$$

Where TP and FN represent the number of true positives and false negatives respectively.

Accepts the following input tensors:

- `preds`:  $(N, \dots)$  (int tensor) or  $(N, C, \dots)$  (float tensor). If `preds` is a floating point we apply `torch.argmax` along the `C` dimension to automatically convert probabilities/logits into an int tensor.
- `target` (int tensor):  $(N, \dots)$

The influence of the additional dimension  $\dots$  (if present) will be determined by the `multidim_average` argument.

### Parameters

- **num\_classes** (`int`) – Integer specifying the number of classes
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: Calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: Calculates statistic for each label and applies no reduction
- **top\_k** (`int`) – Number of highest probability or logit score predictions considered to find the correct label. Only works when `preds` contain probabilities/logits.
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the `N` axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – `bool` indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be `(C,)`
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

**Return type** The returned shape depends on the `average` and `multidim_average` arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import MulticlassRecall
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> metric = MulticlassRecall(num_classes=3)
>>> metric(preds, target)
tensor(0.8333)
>>> metric = MulticlassRecall(num_classes=3, average=None)
>>> metric(preds, target)
tensor([0.5000, 1.0000, 1.0000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import MulticlassRecall
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> metric = MulticlassRecall(num_classes=3)
>>> metric(preds, target)
tensor(0.8333)
>>> metric = MulticlassRecall(num_classes=3, average=None)
>>> metric(preds, target)
tensor([0.5000, 1.0000, 1.0000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import MulticlassRecall
>>> target = torch.tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = torch.tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> metric = MulticlassRecall(num_classes=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.5000, 0.2778])
>>> metric = MulticlassRecall(num_classes=3, multidim_average='samplewise',
...                             average=None)
>>> metric(preds, target)
tensor([[1.0000, 0.0000, 0.5000],
        [0.0000, 0.3333, 0.5000]])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes the final statistics.

**Return type** `Tensor`

**Returns**

The metric returns a tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to  $[tp, fp, tn, fn, sup]$  ( $sup$  stands for support and equals  $tp + fn$ ). The shape depends on `average` and `multidim_average` parameters:

- If `multidim_average` is set to `global`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(5,)$
- If `average=None/'none'`, the shape will be  $(C, 5)$
- If `multidim_average` is set to `samplewise`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(N, 5)$
- If `average=None/'none'`, the shape will be  $(N, C, 5)$

## MultilabelRecall

```
class torchmetrics.classification.MultilabelRecall(num_labels, threshold=0.5, average='macro',
                                                    multidim_average='global', ignore_index=None,
                                                    validate_args=True, **kwargs)
```

Computes *Recall* for multilabel tasks:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where TP and FN represent the number of true positives and false negatives respectively.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

### Parameters

- **num\_labels** (**int**) – Integer specifying the number of labels
- **threshold** (**float**) – Threshold for transforming probability to binary (0,1) predictions
- **average** (**Optional[Literal['micro', 'macro', 'weighted', 'none']]**) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: Calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: Calculates statistic for each label and applies no reduction
- **multidim\_average** (**Literal['global', 'samplewise']**) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (**Optional[int]**) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (**bool**) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

### Returns

- If **multidim\_average** is set to **global**:
  - If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor
  - If **average**=None/'none', the shape will be (C,)
- If **multidim\_average** is set to **samplewise**:
  - If **average**='micro'/'macro'/'weighted', the shape will be (N,)

– If `average=None/'none'`, the shape will be (N, C)

**Return type** The returned shape depends on the `average` and `multidim_average` arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import MultilabelRecall
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelRecall(num_labels=3)
>>> metric(preds, target)
tensor(0.6667)
>>> metric = MultilabelRecall(num_labels=3, average=None)
>>> metric(preds, target)
tensor([1., 0., 1.])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import MultilabelRecall
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelRecall(num_labels=3)
>>> metric(preds, target)
tensor(0.6667)
>>> metric = MultilabelRecall(num_labels=3, average=None)
>>> metric(preds, target)
tensor([1., 0., 1.])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import MultilabelRecall
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> metric = MultilabelRecall(num_labels=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.6667, 0.0000])
>>> metric = MultilabelRecall(num_labels=3, multidim_average='samplewise',
...                             average=None)
>>> metric(preds, target)
tensor([[1., 1., 0.],
        [0., 0., 0.]])
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

**compute()**

Computes the final statistics.

**Return type** `Tensor`

**Returns**

The metric returns a tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to  $[tp, fp, tn, fn, sup]$  ( $sup$  stands for support and equals  $tp + fn$ ). The shape depends on `average` and `multidim_average` parameters:

- If `multidim_average` is set to `global`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(5,)$
- If `average=None/'none'`, the shape will be  $(C, 5)$
- If `multidim_average` is set to `samplewise`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(N, 5)$
- If `average=None/'none'`, the shape will be  $(N, C, 5)$

## 1.40.2 Functional Interface

`torchmetrics.functional.recall(preds, target, average='micro', mdmc_average=None, ignore_index=None, num_classes=None, threshold=0.5, top_k=None, multiclass=None, task=None, num_labels=None, multidim_average='global', validate_args=True)`

---

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entypoint to calling the three specialized versions.

---

Computes *Recall*

$$\text{Recall} = \frac{TP}{TP + FN}$$

Where  $TP$  and  $FN$  represent the number of true positives and false negatives respectively. With the use of `top_k` parameter, this metric can generalize to  $\text{Recall}@K$ .

The reduction method (how the recall scores are aggregated) is controlled by the `average` parameter, and additionally by the `mdmc_average` parameter in the multi-dimensional multi-class case. Accepts all inputs listed in *Input types*.

### Parameters

- **preds** (*Tensor*) – Predictions from model (probabilities, logits or labels)
- **target** (*Tensor*) – Ground truth values
- **average** (*Optional[Literal['micro', 'macro', 'weighted', 'none']]*) – Defines the reduction that is applied. Should be one of the following:
  - `'micro'` [default]: Calculate the metric globally, across all samples and classes.
  - `'macro'`: Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
  - `'weighted'`: Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support ( $tp + fn$ ).
  - `'none'` or `None`: Calculate the metric for each class separately, and return the metric for every class.

- 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

---

**Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_average`.

---



---

**Note:** If 'none' and a given class doesn't occur in the `preds` or `target`, the value for the class will be `nan`.

---

- **mdmc\_average** (`Optional[str]`) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
  - `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - 'samplewise': In this case, the statistics are computed separately for each sample on the `N` axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... (see *Input types*) as the `N` dimension within the sample, and computing the metric for the sample based on that.
  - 'global': In this case the `N` and ... dimensions of the inputs (see *Input types*) are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were `(N_X, C)`. From here on the `average` parameter applies as usual.
- **ignore\_index** (`Optional[int]`) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or 'none', the score for the ignored class will be returned as `nan`.
- **num\_classes** (`Optional[int]`) – Number of classes. Necessary for 'macro', 'weighted' and `None` average methods.
- **threshold** (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
- **top\_k** (`Optional[int]`) – Number of the highest probability or logit score predictions considered finding the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (`None`) will be interpreted as 1 for these inputs.  
Should be left at default (`None`) for all other types of inputs.
- **multiclass** (`Optional[bool]`) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.

**Return type** `Tensor`

### Returns

The shape of the returned tensor depends on the `average` parameter

- If `average` in `['micro', 'macro', 'weighted', 'samples']`, a one-element tensor will be returned
- If `average` in `['none', None]`, the shape will be `(C,)`, where `C` stands for the number of classes

### Raises

- **ValueError** – If average is not one of "micro", "macro", "weighted", "samples", "none" or None
- **ValueError** – If mdmc\_average is not one of None, "samplewise", "global".
- **ValueError** – If average is set but num\_classes is not provided.
- **ValueError** – If num\_classes is set and ignore\_index is not in the range [0, num\_classes).

### Example

```
>>> from torchmetrics.functional import recall
>>> preds = torch.tensor([2, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> recall(preds, target, average='macro', num_classes=3)
tensor(0.3333)
>>> recall(preds, target, average='micro')
tensor(0.2500)
```

### binary\_recall

`torchmetrics.functional.classification.binary_recall(preds, target, threshold=0.5, multidim_average='global', ignore_index=None, validate_args=True)`

Computes *Recall* for binary tasks:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where TP and FN represent the number of true positives and false negatives respectively.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

#### Parameters

- **preds** (**Tensor**) – Tensor with predictions
- **target** (**Tensor**) – Tensor with true labels
- **threshold** (**float**) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (**Literal**['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (**Optional**[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation



- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Return type** `Tensor`

**Returns** If `multidim_average` is set to `global`, the metric returns a scalar value. If `multidim_average` is set to `samplewise`, the metric returns (N,) vector consisting of a scalar value per sample.

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import binary_recall
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0, 0, 1, 1, 0, 1])
>>> binary_recall(preds, target)
tensor(0.6667)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import binary_recall
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> binary_recall(preds, target)
tensor(0.6667)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import binary_recall
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> binary_recall(preds, target, multidim_average='samplewise')
tensor([0.6667, 0.0000])
```

## multiclass\_recall

`torchmetrics.functional.classification.multiclass_recall(preds, target, num_classes, average='macro', top_k=1, multidim_average='global', ignore_index=None, validate_args=True)`

Computes *Recall* for multiclass tasks:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where TP and FN represent the number of true positives and false negatives respectively.

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

The influence of the additional dimension . . . (if present) will be determined by the *multidim\_average* argument.

### Parameters

- **preds** (*Tensor*) – Tensor with predictions
- **target** (*Tensor*) – Tensor with true labels
- **num\_classes** (*int*) – Integer specifying the number of classes
- **average** (*Optional[Literal['micro', 'macro', 'weighted', 'none']]*) – Defines the reduction that is applied over labels. Should be one of the following:
  - *micro*: Sum statistics over all labels
  - *macro*: Calculate statistics for each label and average them
  - *weighted*: Calculates statistics for each label and computes weighted average using their support
  - *"none"* or *None*: Calculates statistic for each label and applies no reduction
- **top\_k** (*int*) – Number of highest probability or logit score predictions considered to find the correct label. Only works when *preds* contain probabilities/logits.
- **multidim\_average** (*Literal['global', 'samplewise']*) – Defines how additionally dimensions . . . should be handled. Should be one of the following:
  - *global*: Additional dimensions are flattened along the batch dimension
  - *samplewise*: Statistic will be calculated independently for each sample on the *N* axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (*Optional[int]*) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (*bool*) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

### Returns

- If *multidim\_average* is set to *global*:
  - If *average*='micro'/'macro'/'weighted', the output will be a scalar tensor
  - If *average*=None/'none', the shape will be (C,)
- If *multidim\_average* is set to *samplewise*:
  - If *average*='micro'/'macro'/'weighted', the shape will be (N,)
  - If *average*=None/'none', the shape will be (N, C)

**Return type** The returned shape depends on the *average* and *multidim\_average* arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_recall
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> multiclass_recall(preds, target, num_classes=3)
tensor(0.8333)
>>> multiclass_recall(preds, target, num_classes=3, average=None)
tensor([0.5000, 1.0000, 1.0000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_recall
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> multiclass_recall(preds, target, num_classes=3)
tensor(0.8333)
>>> multiclass_recall(preds, target, num_classes=3, average=None)
tensor([0.5000, 1.0000, 1.0000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import multiclass_recall
>>> target = torch.tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = torch.tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> multiclass_recall(preds, target, num_classes=3, multidim_average='samplewise
↳')
tensor([0.5000, 0.2778])
>>> multiclass_recall(preds, target, num_classes=3, multidim_average='samplewise
↳', average=None)
tensor([[1.0000, 0.0000, 0.5000],
        [0.0000, 0.3333, 0.5000]])
```

## multilabel\_recall

`torchmetrics.functional.classification.multilabel_recall(preds, target, num_labels, threshold=0.5, average='macro', multidim_average='global', ignore_index=None, validate_args=True)`

Computes *Recall* for multilabel tasks:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where TP and FN represent the number of true positives and false negatives respectively.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions

- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: Calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: Calculates statistic for each label and applies no reduction
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the `N` axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – `bool` indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

#### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be `(C,)`
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

**Return type** The returned shape depends on the `average` and `multidim_average` arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_recall
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_recall(preds, target, num_labels=3)
tensor(0.6667)
>>> multilabel_recall(preds, target, num_labels=3, average=None)
tensor([1., 0., 1.])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_recall
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_recall(preds, target, num_labels=3)
tensor(0.6667)
>>> multilabel_recall(preds, target, num_labels=3, average=None)
tensor([1., 0., 1.])
```

**Example (multidim tensors):**

```

>>> from torchmetrics.functional.classification import multilabel_recall
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> multilabel_recall(preds, target, num_labels=3, multidim_average='samplewise
↪')
tensor([0.6667, 0.0000])
>>> multilabel_recall(preds, target, num_labels=3, multidim_average='samplewise
↪', average=None)
tensor([[1., 1., 0.],
        [0., 0., 0.]])

```

## 1.41 Recall At Fixed Precision

### 1.41.1 Module Interface

#### BinaryRecallAtFixedPrecision

```

class torchmetrics.classification.BinaryRecallAtFixedPrecision(min_precision, thresholds=None,
                                                                ignore_index=None,
                                                                validate_args=True, **kwargs)

```

Computes the highest possible recall value given the minimum precision thresholds provided. This is done by first calculating the precision-recall curve for different thresholds and then find the recall for a given precision level.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{\text{samples}})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{\text{thresholds}})$  (constant memory).

#### Parameters

- **min\_precision** (float) – float value specifying minimum precision threshold.
- **thresholds** (Union[int, List[float], Tensor, None]) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.

- If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
- If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (*bool*) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Returns

a tuple of 2 tensors containing:

- recall: an scalar tensor with the maximum recall for the given precision level
- threshold: an scalar tensor with the corresponding threshold level

**Return type** (*tuple*)

### Example

```
>>> from torchmetrics.classification import BinaryRecallAtFixedPrecision
>>> preds = torch.tensor([0, 0.5, 0.7, 0.8])
>>> target = torch.tensor([0, 1, 1, 0])
>>> metric = BinaryRecallAtFixedPrecision(min_precision=0.5, thresholds=None)
>>> metric(preds, target)
(tensor(1.), tensor(0.5000))
>>> metric = BinaryRecallAtFixedPrecision(min_precision=0.5, thresholds=5)
>>> metric(preds, target)
(tensor(1.), tensor(0.5000))
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

### compute()

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tuple[Tensor, Tensor]`

## MulticlassRecallAtFixedPrecision

```
class torchmetrics.classification.MulticlassRecallAtFixedPrecision(num_classes, min_precision,
                                                                    thresholds=None,
                                                                    ignore_index=None,
                                                                    validate_args=True,
                                                                    **kwargs)
```

Computes the highest possible recall value given the minimum precision thresholds provided. This is done by first calculating the precision-recall curve for different thresholds and then find the recall for a given precision level.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply softmax per sample.

- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the [0, n\_classes-1] range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

### Parameters

- **num\_classes** (int) – Integer specifying the number of classes
- **min\_precision** (float) – float value specifying minimum precision threshold.
- **thresholds** (Union[int, List[float], Tensor, None]) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (bool) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.
- **kwargs** (Any) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Returns

a tuple of either 2 tensors or 2 lists containing

- recall: an 1d tensor of size (n\_classes, ) with the maximum recall for the given precision level per class
- thresholds: an 1d tensor of size (n\_classes, ) with the corresponding threshold level per class

**Return type** (tuple)

### Example

```
>>> from torchmetrics.classification import MulticlassRecallAtFixedPrecision
>>> preds = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                       [0.05, 0.75, 0.05, 0.05, 0.05],
...                       [0.05, 0.05, 0.75, 0.05, 0.05],
...                       [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> metric = MulticlassRecallAtFixedPrecision(num_classes=5, min_precision=0.5,
-> thresholds=None)
>>> metric(preds, target)
(tensor([1., 1., 0., 0., 0.]), tensor([7.5000e-01, 7.5000e-01, 1.0000e+06, 1.
-> 0000e+06, 1.0000e+06]))
>>> metric = MulticlassRecallAtFixedPrecision(num_classes=5, min_precision=0.5,
-> thresholds=5)
```

(continues on next page)

(continued from previous page)

```
>>> metric(preds, target)
(tensor([1., 1., 0., 0., 0.]), tensor([7.5000e-01, 7.5000e-01, 1.0000e+06, 1.
↪ 0000e+06, 1.0000e+06]))
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tuple[Tensor, Tensor]`

## MultilabelRecallAtFixedPrecision

```
class torchmetrics.classification.MultilabelRecallAtFixedPrecision(num_labels, min_precision,
                                                                    thresholds=None,
                                                                    ignore_index=None,
                                                                    validate_args=True,
                                                                    **kwargs)
```

Computes the highest possible recall value given the minimum precision thresholds provided. This is done by first calculating the precision-recall curve for different thresholds and then find the recall for a given precision level.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

### Parameters

- **num\_labels** (`int`) – Integer specifying the number of labels
- **min\_precision** (`float`) – float value specifying minimum precision threshold.
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.



- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Returns**

a tuple of either 2 tensors or 2 lists containing

- recall: an 1d tensor of size (n\_classes, ) with the maximum recall for the given precision level per class
- thresholds: an 1d tensor of size (n\_classes, ) with the corresponding threshold level per class

**Return type** (`tuple`)

**Example**

```
>>> from torchmetrics.classification import MultilabelRecallAtFixedPrecision
>>> preds = torch.tensor([[0.75, 0.05, 0.35],
...                       [0.45, 0.75, 0.05],
...                       [0.05, 0.55, 0.75],
...                       [0.05, 0.65, 0.05]])
>>> target = torch.tensor([[1, 0, 1],
...                        [0, 0, 0],
...                        [0, 1, 1],
...                        [1, 1, 1]])
>>> metric = MultilabelRecallAtFixedPrecision(num_labels=3, min_precision=0.5,
↳ thresholds=None)
>>> metric(preds, target)
(tensor([1., 1., 1.]), tensor([0.0500, 0.5500, 0.0500]))
>>> metric = MultilabelRecallAtFixedPrecision(num_labels=3, min_precision=0.5,
↳ thresholds=5)
>>> metric(preds, target)
(tensor([1., 1., 1.]), tensor([0.0000, 0.5000, 0.0000]))
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tuple[Tensor, Tensor]`

## 1.41.2 Functional Interface

### binary\_recall\_at\_fixed\_precision

```
torchmetrics.functional.classification.binary_recall_at_fixed_precision(preds, target,
                                                                           min_precision,
                                                                           thresholds=None,
                                                                           ignore_index=None,
                                                                           validate_args=True)
```

Computes the highest possible recall value given the minimum precision thresholds provided. This is done by first calculating the precision-recall curve for different thresholds and then find the recall for a given precision level.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

#### Parameters

- **preds** (**Tensor**) – Tensor with predictions
- **target** (**Tensor**) – Tensor with true labels
- **min\_precision** (**float**) – float value specifying minimum precision threshold.
- **thresholds** (**Union[int, List[float], Tensor, None]**) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (**bool**) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

#### Returns

a tuple of 2 tensors containing:

- recall: an scalar tensor with the maximum recall for the given precision level
- threshold: an scalar tensor with the corresponding threshold level

**Return type** (**tuple**)

#### Example

```
>>> from torchmetrics.functional.classification import binary_recall_at_fixed_
    ↳ precision
>>> preds = torch.tensor([0, 0.5, 0.7, 0.8])
>>> target = torch.tensor([0, 1, 1, 0])
>>> binary_recall_at_fixed_precision(preds, target, min_precision=0.5,
    ↳ thresholds=None)
(tensor(1.), tensor(0.5000))
>>> binary_recall_at_fixed_precision(preds, target, min_precision=0.5, thresholds=5)
(tensor(1.), tensor(0.5000))
```

## multiclass\_recall\_at\_fixed\_precision

```
torchmetrics.functional.classification.multiclass_recall_at_fixed_precision(preds, target,
                                                                            num_classes,
                                                                            min_precision,
                                                                            thresh-
                                                                            olds=None,
                                                                            ig-
                                                                            nore_index=None,
                                                                            vali-
                                                                            date_args=True)
```

Computes the highest possible recall value given the minimum precision thresholds provided. This is done by first calculating the precision-recall curve for different thresholds and then find the recall for a given precision level.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the [0, n\_classes-1] range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **min\_precision** ([float](#)) – float value specifying minimum precision threshold.
- **thresholds** ([Union\[int, List\[float\], Tensor, None\]](#)) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

### Returns

a tuple of either 2 tensors or 2 lists containing

- recall: an 1d tensor of size (n\_classes, ) with the maximum recall for the given precision level per class
- thresholds: an 1d tensor of size (n\_classes, ) with the corresponding threshold level per class

**Return type** (tuple)

### Example

```
>>> from torchmetrics.functional.classification import multiclass_recall_at_fixed_
    precision
>>> preds = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                        [0.05, 0.75, 0.05, 0.05, 0.05],
...                        [0.05, 0.05, 0.75, 0.05, 0.05],
...                        [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> multiclass_recall_at_fixed_precision(preds, target, num_classes=5, min_
    precision=0.5, thresholds=None)
(tensor([1., 1., 0., 0., 0.]), tensor([7.5000e-01, 7.5000e-01, 1.0000e+06, 1.
    0000e+06, 1.0000e+06]))
>>> multiclass_recall_at_fixed_precision(preds, target, num_classes=5, min_
    precision=0.5, thresholds=5)
(tensor([1., 1., 0., 0., 0.]), tensor([7.5000e-01, 7.5000e-01, 1.0000e+06, 1.
    0000e+06, 1.0000e+06]))
```

### multilabel\_recall\_at\_fixed\_precision

`torchmetrics.functional.classification.multilabel_recall_at_fixed_precision(preds, target, num_labels, min_precision, thresholds=None, ignore_index=None, validate_args=True)`

Computes the highest possible recall value given the minimum precision thresholds provided. This is done by first calculating the precision-recall curve for different thresholds and then find the recall for a given precision level.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if `ignore_index` is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the `thresholds` argument to `None` will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{\text{samples}})$  whereas setting the `thresholds` argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{\text{thresholds}} \times n_{\text{labels}})$  (constant memory).

### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **num\_labels** (`int`) – Integer specifying the number of labels
- **min\_precision** (`float`) – float value specifying minimum precision threshold.
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

### Returns

a tuple of either 2 tensors or 2 lists containing

- recall: an 1d tensor of size (n\_classes, ) with the maximum recall for the given precision level per class
- thresholds: an 1d tensor of size (n\_classes, ) with the corresponding threshold level per class

**Return type** (`tuple`)

### Example

```
>>> from torchmetrics.functional.classification import multilabel_recall_at_fixed_
↪precision
>>> preds = torch.tensor([[0.75, 0.05, 0.35],
...                       [0.45, 0.75, 0.05],
...                       [0.05, 0.55, 0.75],
...                       [0.05, 0.65, 0.05]])
>>> target = torch.tensor([[1, 0, 1],
...                        [0, 0, 0],
...                        [0, 1, 1],
...                        [1, 1, 1]])
>>> multilabel_recall_at_fixed_precision(preds, target, num_labels=3, min_
↪precision=0.5, thresholds=None)
(tensor([1., 1., 1.]), tensor([0.0500, 0.5500, 0.0500]))
>>> multilabel_recall_at_fixed_precision(preds, target, num_labels=3, min_
↪precision=0.5, thresholds=5)
(tensor([1., 1., 1.]), tensor([0.0000, 0.5000, 0.0000]))
```

## 1.42 ROC

### 1.42.1 Module Interface

**class** torchmetrics.ROC(num\_classes=None, pos\_label=None, \*\*kwargs)

---

**Note:** From v0.10 an `'binary_*`, `'multiclass_*`, `'multilabel_*` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

---

Computes the Receiver Operating Characteristic (ROC). Works for both binary, multiclass and multilabel problems. In the case of multiclass, the values will be calculated based on a one-vs-the-rest approach.

Forward accepts

- **preds** (float tensor): (N, ...) (binary) or (N, C, ...) (multiclass/multilabel) tensor with probabilities, where C is the number of classes/labels.
- **target** (long tensor): (N, ...) or (N, C, ...) with integer labels

---

**Note:** If either the positive class or negative class is completely missing in the target tensor, the roc values are not well-defined in this case and a tensor of zeros will be returned (either fpr or tpr depending on what class is missing) together with a warning.

---

#### Parameters

- **num\_classes** (Optional[int]) – integer with number of classes for multi-label and multi-class problems. Should be set to None for binary problems
- **pos\_label** (Optional[int]) – integer determining the positive class. Default is None which for binary problem is translated to 1. For multiclass problems this argument should not be set as we iteratively change it in the range [0, num\_classes-1]
- **kwargs** (Any) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (binary case):**

```
>>> from torchmetrics import ROC
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 1, 1])
>>> roc = ROC(pos_label=1)
>>> fpr, tpr, thresholds = roc(pred, target)
>>> fpr
tensor([0., 0., 0., 0., 1.])
>>> tpr
tensor([0.0000, 0.3333, 0.6667, 1.0000, 1.0000])
>>> thresholds
tensor([4, 3, 2, 1, 0])
```

**Example (multiclass case):**

```

>>> pred = torch.tensor([[0.75, 0.05, 0.05, 0.05],
...                        [0.05, 0.75, 0.05, 0.05],
...                        [0.05, 0.05, 0.75, 0.05],
...                        [0.05, 0.05, 0.05, 0.75]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> roc = ROC(num_classes=4)
>>> fpr, tpr, thresholds = roc(pred, target)
>>> fpr
[tensor([0., 0., 1.]), tensor([0., 0., 1.]), tensor([0.0000, 0.3333, 1.0000]),
 tensor([0.0000, 0.3333, 1.0000])]
>>> tpr
[tensor([0., 1., 1.]), tensor([0., 1., 1.]), tensor([0., 0., 1.]), tensor([0.,
 0., 1.])]
>>> thresholds
[tensor([1.7500, 0.7500, 0.0500]),
 tensor([1.7500, 0.7500, 0.0500]),
 tensor([1.7500, 0.7500, 0.0500]),
 tensor([1.7500, 0.7500, 0.0500])]

```

**Example (multilabel case):**

```

>>> pred = torch.tensor([[0.8191, 0.3680, 0.1138],
...                       [0.3584, 0.7576, 0.1183],
...                       [0.2286, 0.3468, 0.1338],
...                       [0.8603, 0.0745, 0.1837]])
>>> target = torch.tensor([[1, 1, 0], [0, 1, 0], [0, 0, 0], [0, 1, 1]])
>>> roc = ROC(num_classes=3, pos_label=1)
>>> fpr, tpr, thresholds = roc(pred, target)
>>> fpr
[tensor([0.0000, 0.3333, 0.3333, 0.6667, 1.0000]),
 tensor([0., 0., 0., 1., 1.]),
 tensor([0.0000, 0.0000, 0.3333, 0.6667, 1.0000])]
>>> tpr
[tensor([0., 0., 1., 1., 1.]),
 tensor([0.0000, 0.3333, 0.6667, 0.6667, 1.0000]),
 tensor([0., 1., 1., 1., 1.])]
>>> thresholds
[tensor([1.8603, 0.8603, 0.8191, 0.3584, 0.2286]),
 tensor([1.7576, 0.7576, 0.3680, 0.3468, 0.0745]),
 tensor([1.1837, 0.1837, 0.1338, 0.1183, 0.1138])]

```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Compute the receiver operating characteristic.

**Return type** Union[Tuple[Tensor, Tensor, Tensor], Tuple[List[Tensor], List[Tensor], List[Tensor]]]

**Returns**

3-element tuple containing

**fpr: tensor with false positive rates.** If multiclass, this is a list of such tensors, one for each class.

**tpr: tensor with true positive rates.** If multiclass, this is a list of such tensors, one for each class.

**thresholds:** thresholds used for computing false- and true-positive rates

**update**(preds, target)

Update state with predictions and targets.

#### Parameters

- **preds** ([Tensor](#)) – Predictions from model
- **target** ([Tensor](#)) – Ground truth values

**Return type** [None](#)

## BinaryROC

```
class torchmetrics.classification.BinaryROC(thresholds=None, ignore_index=None,
                                             validate_args=True, **kwargs)
```

Computes the Receiver Operating Characteristic (ROC) for binary tasks. The curve consist of multiple pairs of true positive rate (TPR) and false positive rate (FPR) values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

Note that outputted thresholds will be in reversed order to ensure that they corresponds to both fpr and tpr which are sorted in reversed order during their calculation, such that they are monotome increasing.

#### Parameters

- **thresholds** ([Union\[int, List\[float\], Tensor, None\]](#)) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.
- **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.



**Returns**

a tuple of 3 tensors containing:

- fpr: an 1d tensor of size (n\_thresholds+1, ) with false positive rate values
- tpr: an 1d tensor of size (n\_thresholds+1, ) with true positive rate values
- thresholds: an 1d tensor of size (n\_thresholds, ) with decreasing threshold values

**Return type** (tuple)

**Example**

```
>>> from torchmetrics.classification import BinaryROC
>>> preds = torch.tensor([0, 0.5, 0.7, 0.8])
>>> target = torch.tensor([0, 1, 1, 0])
>>> metric = BinaryROC(thresholds=None)
>>> metric(preds, target)
(tensor([0.0000, 0.5000, 0.5000, 0.5000, 1.0000]),
 tensor([0.0000, 0.0000, 0.5000, 1.0000, 1.0000]),
 tensor([1.0000, 0.8000, 0.7000, 0.5000, 0.0000]))
>>> metric = BinaryROC(thresholds=5)
>>> metric(preds, target)
(tensor([0.0000, 0.5000, 0.5000, 0.5000, 1.0000]),
 tensor([0., 0., 1., 1., 1.]),
 tensor([1.0000, 0.7500, 0.5000, 0.2500, 0.0000]))
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** Tuple[Tensor, Tensor, Tensor]

**MulticlassROC**

```
class torchmetrics.classification.MulticlassROC(num_classes, thresholds=None, ignore_index=None,
                                                validate_args=True, **kwargs)
```

Computes the Receiver Operating Characteristic (ROC) for binary tasks. The curve consist of multiple pairs of true positive rate (TPR) and false positive rate (FPR) values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the [0, n\_classes-1] range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an

integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

Note that outputted thresholds will be in reversed order to ensure that they corresponds to both fpr and tpr which are sorted in reversed order during their calculation, such that they are monotome increasing.

### Parameters

- **num\_classes** (`int`) – Integer specifying the number of classes
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an `int` (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an `list` of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d `tensor` of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Returns

a tuple of either 3 tensors or 3 lists containing

- fpr: if `thresholds=None` a list for each class is returned with an 1d tensor of size  $(n_{thresholds}+1, )$  with false positive rate values (length may differ between classes). If `thresholds` is set to something else, then a single 2d tensor of size  $(n_{classes}, n_{thresholds}+1)$  with false positive rate values is returned.
- tpr: if `thresholds=None` a list for each class is returned with an 1d tensor of size  $(n_{thresholds}+1, )$  with true positive rate values (length may differ between classes). If `thresholds` is set to something else, then a single 2d tensor of size  $(n_{classes}, n_{thresholds}+1)$  with true positive rate values is returned.
- thresholds: if `thresholds=None` a list for each class is returned with an 1d tensor of size  $(n_{thresholds}, )$  with decreasing threshold values (length may differ between classes). If `threshold` is set to something else, then a single 1d tensor of size  $(n_{thresholds}, )$  is returned with shared threshold values for all classes.

**Return type** (`tuple`)

### Example

```
>>> from torchmetrics.classification import MulticlassROC
>>> preds = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                       [0.05, 0.75, 0.05, 0.05, 0.05],
...                       [0.05, 0.05, 0.75, 0.05, 0.05],
...                       [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> metric = MulticlassROC(num_classes=5, thresholds=None)
>>> fpr, tpr, thresholds = metric(preds, target)
```

(continues on next page)

(continued from previous page)

```

>>> fpr
[tensor([0., 0., 1.]), tensor([0., 0., 1.]), tensor([0.0000, 0.3333, 1.0000]),
 tensor([0.0000, 0.3333, 1.0000]), tensor([0., 1.])]
>>> tpr
[tensor([0., 1., 1.]), tensor([0., 1., 1.]), tensor([0., 0., 1.]), tensor([0., 0., 1.]),
 tensor([0., 0.])]
>>> thresholds
[tensor([1.0000, 0.7500, 0.0500]), tensor([1.0000, 0.7500, 0.0500]),
 tensor([1.0000, 0.7500, 0.0500]), tensor([1.0000, 0.7500, 0.0500]), tensor([1.0000,
 0.0500])]
>>> metric = MulticlassROC(num_classes=5, thresholds=5)
>>> metric(preds, target)
(tensor([[0.0000, 0.0000, 0.0000, 0.0000, 1.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 1.0000],
        [0.0000, 0.3333, 0.3333, 0.3333, 1.0000],
        [0.0000, 0.3333, 0.3333, 0.3333, 1.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 1.0000]]),
 tensor([[0., 1., 1., 1., 1.],
        [0., 1., 1., 1., 1.],
        [0., 0., 0., 0., 1.],
        [0., 0., 0., 0., 1.],
        [0., 0., 0., 0., 0.]])
 tensor([1.0000, 0.7500, 0.5000, 0.2500, 0.0000]))

```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### **compute()**

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tuple[Tensor, Tensor, Tensor]`

## MultilabelROC

**class** `torchmetrics.classification.MultilabelROC(num_labels, thresholds=None, ignore_index=None, validate_args=True, **kwargs)`

Computes the Receiver Operating Characteristic (ROC) for binary tasks. The curve consist of multiple pairs of true positive rate (TPR) and false positive rate (FPR) values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an

integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

Note that outputted thresholds will be in reversed order to ensure that they corresponds to both fpr and tpr which are sorted in reversed order during their calculation, such that they are monotome increasing.

#### Parameters

- **num\_labels** (`int`) – Integer specifying the number of labels
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Returns

a tuple of either 3 tensors or 3 lists containing

- fpr: if *thresholds=None* a list for each label is returned with an 1d tensor of size  $(n_{thresholds}+1, )$  with false positive rate values (length may differ between labels). If *thresholds* is set to something else, then a single 2d tensor of size  $(n_{labels}, n_{thresholds}+1)$  with false positive rate values is returned.
- tpr: if *thresholds=None* a list for each label is returned with an 1d tensor of size  $(n_{thresholds}+1, )$  with true positive rate values (length may differ between labels). If *thresholds* is set to something else, then a single 2d tensor of size  $(n_{labels}, n_{thresholds}+1)$  with true positive rate values is returned.
- thresholds: if *thresholds=None* a list for each label is returned with an 1d tensor of size  $(n_{thresholds}, )$  with decreasing threshold values (length may differ between labels). If *threshold* is set to something else, then a single 1d tensor of size  $(n_{thresholds}, )$  is returned with shared threshold values for all labels.

**Return type** (`tuple`)

#### Example

```
>>> from torchmetrics.classification import MultilabelROC
>>> preds = torch.tensor([[0.75, 0.05, 0.35],
...                       [0.45, 0.75, 0.05],
...                       [0.05, 0.55, 0.75],
...                       [0.05, 0.65, 0.05]])
>>> target = torch.tensor([[1, 0, 1],
...                        [0, 0, 0],
...                        [0, 1, 1],
```

(continues on next page)

(continued from previous page)

```

... [1, 1, 1]])
>>> metric = MultilabelROC(num_labels=3, thresholds=None)
>>> fpr, tpr, thresholds = metric(preds, target)
>>> fpr
(tensor([0.0000, 0.0000, 0.5000, 1.0000]),
 tensor([0.0000, 0.5000, 0.5000, 0.5000, 1.0000]),
 tensor([0., 0., 0., 1.]))
>>> tpr
(tensor([0.0000, 0.5000, 0.5000, 1.0000]),
 tensor([0.0000, 0.0000, 0.5000, 1.0000, 1.0000]),
 tensor([0.0000, 0.3333, 0.6667, 1.0000]))
>>> thresholds
(tensor([1.0000, 0.7500, 0.4500, 0.0500]),
 tensor([1.0000, 0.7500, 0.6500, 0.5500, 0.0500]),
 tensor([1.0000, 0.7500, 0.3500, 0.0500]))
>>> metric = MultilabelROC(num_labels=3, thresholds=5)
>>> metric(preds, target)
(tensor([[0.0000, 0.0000, 0.0000, 0.5000, 1.0000],
         [0.0000, 0.5000, 0.5000, 0.5000, 1.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 1.0000]]),
 tensor([[0.0000, 0.5000, 0.5000, 0.5000, 1.0000],
         [0.0000, 0.0000, 1.0000, 1.0000, 1.0000],
         [0.0000, 0.3333, 0.3333, 0.6667, 1.0000]]),
 tensor([1.0000, 0.7500, 0.5000, 0.2500, 0.0000]))

```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tuple[Tensor, Tensor, Tensor]`

## 1.42.2 Functional Interface

`torchmetrics.functional.roc(preds, target, num_classes=None, pos_label=None, sample_weights=None, task=None, thresholds=None, num_labels=None, ignore_index=None, validate_args=True)`

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entryptoint to calling the three specialized versions.

Computes the Receiver Operating Characteristic (ROC). Works with both binary, multiclass and multilabel input.

**Note:** If either the positive class or negative class is completely missing in the target tensor, the roc values are not well-defined in this case and a tensor of zeros will be returned (either fpr or tpr depending on what class is missing) together with a warning.

**Parameters**

- **preds** (`Tensor`) – predictions from model (logits or probabilities)
- **target** (`Tensor`) – ground truth values
- **num\_classes** (`Optional[int]`) – integer with number of classes for multi-label and multi-class problems. Should be set to `None` for binary problems.
- **pos\_label** (`Optional[int]`) – integer determining the positive class. Default is `None` which for binary problem is translated to 1. For multiclass problems this argument should not be set as we iteratively change it in the range `[0, num_classes-1]`
- **sample\_weights** (`Optional[Sequence]`) – sample weights for each data point

**Return type** `Union[Tuple[Tensor, Tensor, Tensor], Tuple[List[Tensor], List[Tensor], List[Tensor]]]`

**Returns**

3-element tuple containing

**fpr: tensor with false positive rates.** If multiclass or multilabel, this is a list of such tensors, one for each class/label.

**tpr: tensor with true positive rates.** If multiclass or multilabel, this is a list of such tensors, one for each class/label.

**thresholds: tensor with thresholds used for computing false- and true postive rates** If multiclass or multilabel, this is a list of such tensors, one for each class/label.

**Example (binary case):**

```
>>> from torchmetrics.functional import roc
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 1, 1])
>>> fpr, tpr, thresholds = roc(pred, target, pos_label=1)
>>> fpr
tensor([0., 0., 0., 0., 1.])
>>> tpr
tensor([0.0000, 0.3333, 0.6667, 1.0000, 1.0000])
>>> thresholds
tensor([4, 3, 2, 1, 0])
```

**Example (multiclass case):**

```
>>> from torchmetrics.functional import roc
>>> pred = torch.tensor([[0.75, 0.05, 0.05, 0.05],
...                      [0.05, 0.75, 0.05, 0.05],
...                      [0.05, 0.05, 0.75, 0.05],
...                      [0.05, 0.05, 0.05, 0.75]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> fpr, tpr, thresholds = roc(pred, target, num_classes=4)
>>> fpr
[tensor([0., 0., 1.]), tensor([0., 0., 1.]), tensor([0.0000, 0.3333, 1.0000]),
↪ tensor([0.0000, 0.3333, 1.0000])]
>>> tpr
[tensor([0., 1., 1.]), tensor([0., 1., 1.]), tensor([0., 0., 1.]), tensor([0.,
↪ 0., 1.])]
```

(continues on next page)

(continued from previous page)

```
>>> thresholds
[tensor([1.7500, 0.7500, 0.0500]),
 tensor([1.7500, 0.7500, 0.0500]),
 tensor([1.7500, 0.7500, 0.0500]),
 tensor([1.7500, 0.7500, 0.0500])]
```

**Example (multilabel case):**

```
>>> from torchmetrics.functional import roc
>>> pred = torch.tensor([[0.8191, 0.3680, 0.1138],
...                      [0.3584, 0.7576, 0.1183],
...                      [0.2286, 0.3468, 0.1338],
...                      [0.8603, 0.0745, 0.1837]])
>>> target = torch.tensor([[1, 1, 0], [0, 1, 0], [0, 0, 0], [0, 1, 1]])
>>> fpr, tpr, thresholds = roc(pred, target, num_classes=3, pos_label=1)
>>> fpr
[tensor([0.0000, 0.3333, 0.3333, 0.6667, 1.0000]),
 tensor([0., 0., 0., 1., 1.]),
 tensor([0.0000, 0.0000, 0.3333, 0.6667, 1.0000])]
>>> tpr
[tensor([0., 0., 1., 1., 1.]), tensor([0.0000, 0.3333, 0.6667, 0.6667, 1.0000]),
 ↪ tensor([0., 1., 1., 1., 1.])]
>>> thresholds
[tensor([1.8603, 0.8603, 0.8191, 0.3584, 0.2286]),
 tensor([1.7576, 0.7576, 0.3680, 0.3468, 0.0745]),
 tensor([1.1837, 0.1837, 0.1338, 0.1183, 0.1138])]
```

## binary\_roc

`torchmetrics.functional.classification.binary_roc(preds, target, thresholds=None, ignore_index=None, validate_args=True)`

Computes the Receiver Operating Characteristic (ROC) for binary tasks. The curve consist of multiple pairs of true positive rate (TPR) and false positive rate (FPR) values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if `ignore_index` is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the `thresholds` argument to `None` will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the `thresholds` argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

Note that outputted thresholds will be in reversed order to ensure that they corresponds to both fpr and tpr which are sorted in reversed order during their calculation, such that they are monotome increasing.

### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an `int` (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an `list` of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d `tensor` of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

a tuple of 3 tensors containing:

- fpr: an 1d tensor of size (n\_thresholds+1, ) with false positive rate values
- tpr: an 1d tensor of size (n\_thresholds+1, ) with true positive rate values
- thresholds: an 1d tensor of size (n\_thresholds, ) with decreasing threshold values

**Return type** (`tuple`)

### Example

```
>>> from torchmetrics.functional.classification import binary_roc
>>> preds = torch.tensor([0, 0.5, 0.7, 0.8])
>>> target = torch.tensor([0, 1, 1, 0])
>>> binary_roc(preds, target, thresholds=None)
(tensor([0.0000, 0.5000, 0.5000, 0.5000, 1.0000]),
 tensor([0.0000, 0.0000, 0.5000, 1.0000, 1.0000]),
 tensor([1.0000, 0.8000, 0.7000, 0.5000, 0.0000]))
>>> binary_roc(preds, target, thresholds=5)
(tensor([0.0000, 0.5000, 0.5000, 0.5000, 1.0000]),
 tensor([0., 0., 1., 1., 1.]),
 tensor([1.0000, 0.7500, 0.5000, 0.2500, 0.0000]))
```

### multiclass\_roc

`torchmetrics.functional.classification.multiclass_roc(preds, target, num_classes, thresholds=None, ignore_index=None, validate_args=True)`

Computes the Receiver Operating Characteristic (ROC) for binary tasks. The curve consist of multiple pairs of true positive rate (TPR) and false positive rate (FPR) values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

Accepts the following input tensors:



- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the [0, n\_classes-1] range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

Note that outputted thresholds will be in reversed order to ensure that they corresponds to both fpr and tpr which are sorted in reversed order during their calculation, such that they are monotome increasing.

### Parameters

- **preds** (**Tensor**) – Tensor with predictions
- **target** (**Tensor**) – Tensor with true labels
- **num\_classes** (**int**) – Integer specifying the number of classes
- **thresholds** (**Union[int, List[float], Tensor, None]**) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (**bool**) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

### Returns

a tuple of either 3 tensors or 3 lists containing

- **fpr**: if *thresholds=None* a list for each class is returned with an 1d tensor of size (n\_thresholds+1, ) with false positive rate values (length may differ between classes). If *thresholds* is set to something else, then a single 2d tensor of size (n\_classes, n\_thresholds+1) with false positive rate values is returned.
- **tpr**: if *thresholds=None* a list for each class is returned with an 1d tensor of size (n\_thresholds+1, ) with true positive rate values (length may differ between classes). If *thresholds* is set to something else, then a single 2d tensor of size (n\_classes, n\_thresholds+1) with true positive rate values is returned.
- **thresholds**: if *thresholds=None* a list for each class is returned with an 1d tensor of size (n\_thresholds, ) with decreasing threshold values (length may differ between classes). If *threshold* is set to something else, then a single 1d tensor of size (n\_thresholds, ) is returned with shared threshold values for all classes.

**Return type** (**tuple**)

### Example

```

>>> from torchmetrics.functional.classification import multiclass_roc
>>> preds = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                       [0.05, 0.75, 0.05, 0.05, 0.05],
...                       [0.05, 0.05, 0.75, 0.05, 0.05],
...                       [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> fpr, tpr, thresholds = multiclass_roc(
...     preds, target, num_classes=5, thresholds=None
... )
>>> fpr
[tensor([0., 0., 1.]), tensor([0., 0., 1.]), tensor([0.0000, 0.3333, 1.0000]),
 tensor([0.0000, 0.3333, 1.0000]), tensor([0., 1.])]
>>> tpr
[tensor([0., 1., 1.]), tensor([0., 1., 1.]), tensor([0., 0., 1.]), tensor([0., 0., 1.]),
 tensor([0., 0., 1.]), tensor([0., 0.])]
>>> thresholds
[tensor([1.0000, 0.7500, 0.0500]), tensor([1.0000, 0.7500, 0.0500]),
 tensor([1.0000, 0.7500, 0.0500]), tensor([1.0000, 0.7500, 0.0500]),
 tensor([1.0000, 0.7500, 0.0500])]
>>> multiclass_roc(
...     preds, target, num_classes=5, thresholds=5
... )
(tensor([[0.0000, 0.0000, 0.0000, 0.0000, 1.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 1.0000],
        [0.0000, 0.3333, 0.3333, 0.3333, 1.0000],
        [0.0000, 0.3333, 0.3333, 0.3333, 1.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 1.0000]]),
 tensor([[0., 1., 1., 1., 1.],
        [0., 1., 1., 1., 1.],
        [0., 0., 0., 0., 1.],
        [0., 0., 0., 0., 1.],
        [0., 0., 0., 0., 0.]])
 tensor([1.0000, 0.7500, 0.5000, 0.2500, 0.0000]))

```

### multilabel\_roc

`torchmetrics.functional.classification.multilabel_roc(preds, target, num_labels, thresholds=None, ignore_index=None, validate_args=True)`

Computes the Receiver Operating Characteristic (ROC) for binary tasks. The curve consist of multiple pairs of true positive rate (TPR) and false positive rate (FPR) values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

Note that outputted thresholds will be in reversed order to ensure that they corresponds to both fpr and tpr which are sorted in reversed order during their calculation, such that they are monotome increasing.

### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **num\_labels** (`int`) – Integer specifying the number of labels
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

### Returns

a tuple of either 3 tensors or 3 lists containing

- fpr: if *thresholds=None* a list for each label is returned with an 1d tensor of size  $(n_{thresholds}+1, )$  with false positive rate values (length may differ between labels). If *thresholds* is set to something else, then a single 2d tensor of size  $(n_{labels}, n_{thresholds}+1)$  with false positive rate values is returned.
- tpr: if *thresholds=None* a list for each label is returned with an 1d tensor of size  $(n_{thresholds}+1, )$  with true positive rate values (length may differ between labels). If *thresholds* is set to something else, then a single 2d tensor of size  $(n_{labels}, n_{thresholds}+1)$  with true positive rate values is returned.
- thresholds: if *thresholds=None* a list for each label is returned with an 1d tensor of size  $(n_{thresholds}, )$  with decreasing threshold values (length may differ between labels). If *threshold* is set to something else, then a single 1d tensor of size  $(n_{thresholds}, )$  is returned with shared threshold values for all labels.

**Return type** (`tuple`)

### Example

```
>>> from torchmetrics.functional.classification import multilabel_roc
>>> preds = torch.tensor([[0.75, 0.05, 0.35],
...                       [0.45, 0.75, 0.05],
...                       [0.05, 0.55, 0.75],
...                       [0.05, 0.65, 0.05]])
>>> target = torch.tensor([[1, 0, 1],
...                        [0, 0, 0],
...                        [0, 1, 1],
...                        [1, 1, 1]])
>>> fpr, tpr, thresholds = multilabel_roc(
...     preds, target, num_labels=3, thresholds=None
... )
>>> fpr
[tensor([0.0000, 0.0000, 0.5000, 1.0000]),
 tensor([0.0000, 0.5000, 0.5000, 0.5000, 1.0000]),
 tensor([0., 0., 0., 1.])]
>>> tpr
[tensor([0.0000, 0.5000, 0.5000, 1.0000]),
 tensor([0.0000, 0.0000, 0.5000, 1.0000, 1.0000]),
 tensor([0.0000, 0.3333, 0.6667, 1.0000])]
>>> thresholds
[tensor([1.0000, 0.7500, 0.4500, 0.0500]),
 tensor([1.0000, 0.7500, 0.6500, 0.5500, 0.0500]),
 tensor([1.0000, 0.7500, 0.3500, 0.0500])]
>>> multilabel_roc(
...     preds, target, num_labels=3, thresholds=5
... )
(tensor([[0.0000, 0.0000, 0.0000, 0.5000, 1.0000],
         [0.0000, 0.5000, 0.5000, 0.5000, 1.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 1.0000]]),
 tensor([[0.0000, 0.5000, 0.5000, 0.5000, 1.0000],
         [0.0000, 0.0000, 1.0000, 1.0000, 1.0000],
         [0.0000, 0.3333, 0.3333, 0.6667, 1.0000]]),
 tensor([1.0000, 0.7500, 0.5000, 0.2500, 0.0000]))
```

## 1.43 Specificity

### 1.43.1 Module Interface

```
class torchmetrics.Specificity(num_classes=None, threshold=0.5, average='micro', mdmc_average=None,
                               ignore_index=None, top_k=None, multiclass=None, **kwargs)
```

---

**Note:** From v0.10 an `'binary_*`, `'multiclass_*`, `'multilabel_*` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

---

Computes *Specificity*:

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Where TN and FP represent the number of true negatives and false positives respectively. With the use of `top_k` parameter, this metric can generalize to `Specificity@K`.

The reduction method (how the specificity scores are aggregated) is controlled by the `average` parameter, and additionally by the `mdmc_average` parameter in the multi-dimensional multi-class case. Accepts all inputs listed in *Input types*.

#### Parameters

- **num\_classes** (`Optional[int]`) – Number of classes. Necessary for 'macro', 'weighted' and None average methods.
- **threshold** (`float`) – Threshold probability value for transforming probability predictions to binary (0,1) predictions, in the case of binary or multi-label inputs.
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied. Should be one of the following:
  - 'micro' [default]: Calculate the metric globally, across all samples and classes.
  - 'macro': Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
  - 'weighted': Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (`tn + fp`).
  - 'none' or None: Calculate the metric for each class separately, and return the metric for every class.
  - 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

---

**Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_average`.

---

- **mdmc\_average** (`Optional[str]`) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
  - None [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - 'samplewise': In this case, the statistics are computed separately for each sample on the N axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... (see *Input types*) as the N dimension within the sample, and computing the metric for the sample based on that.
  - 'global': In this case the N and ... dimensions of the inputs (see *Input types*) are flattened into a new N\_X sample axis, i.e. the inputs are treated as if they were (N\_X, C). From here on the `average` parameter applies as usual.
- **ignore\_index** (`Optional[int]`) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or 'none', the score for the ignored class will be returned as `nan`.
- **top\_k** (`Optional[int]`) – Number of the highest probability entries for each sample to convert to 1s - relevant only for inputs with probability predictions. If this parameter is set

for multi-label inputs, it will take precedence over `threshold`. For (multi-dim) multi-class inputs, this parameter defaults to 1.

Should be left unset (`None`) for inputs with label predictions.

- **multiclass** (`Optional[bool]`) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter’s [documentation section](#) for a more detailed explanation and examples.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises** `ValueError` – If average is none of "micro", "macro", "weighted", "samples", "none", `None`.

### Example

```
>>> from torchmetrics import Specificity
>>> preds = torch.tensor([2, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> specificity = Specificity(average='macro', num_classes=3)
>>> specificity(preds, target)
tensor(0.6111)
>>> specificity = Specificity(average='micro')
>>> specificity(preds, target)
tensor(0.6250)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

#### `compute()`

Computes the specificity score based on inputs passed in to update previously.

##### Returns

- If average in ['micro', 'macro', 'weighted', 'samples'], a one-element tensor will be returned
- If average in ['none', `None`], the shape will be (C, ), where C stands for the number of classes

**Return type** The shape of the returned tensor depends on the average parameter

### BinarySpecificity

```
class torchmetrics.classification.BinarySpecificity(threshold=0.5, multidim_average='global',
                                                    ignore_index=None, validate_args=True,
                                                    **kwargs)
```

Computes *Specificity* for binary tasks:

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Where TN and FP represent the number of true negatives and false positives respectively.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.

- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

#### Parameters

- **threshold** (float) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (Literal['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (Optional[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (bool) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.

**Returns** If *multidim\_average* is set to *global*, the metric returns a scalar value. If *multidim\_average* is set to *samplewise*, the metric returns (N,) vector consisting of a scalar value per sample.

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import BinarySpecificity
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0, 0, 1, 1, 0, 1])
>>> metric = BinarySpecificity()
>>> metric(preds, target)
tensor(0.6667)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import BinarySpecificity
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> metric = BinarySpecificity()
>>> metric(preds, target)
tensor(0.6667)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import BinarySpecificity
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> metric = BinarySpecificity(multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.0000, 0.3333])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes the final statistics.

**Return type** `Tensor`

**Returns**

The metric returns a tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to `[tp, fp, tn, fn, sup]` (sup stands for support and equals `tp + fn`). The shape depends on the `multidim_average` parameter:

- If `multidim_average` is set to `global`, the shape will be  $(5,)$
- If `multidim_average` is set to `samplewise`, the shape will be  $(N, 5)$

## MulticlassSpecificity

```
class torchmetrics.classification.MulticlassSpecificity(num_classes, top_k=1, average='macro',
                                                         multidim_average='global',
                                                         ignore_index=None, validate_args=True,
                                                         **kwargs)
```

Computes *Specificity* for multiclass tasks:

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Where TN and FP represent the number of true negatives and false positives respectively.

Accepts the following input tensors:

- `preds` ( $(N, \dots)$  (int tensor) or  $(N, C, \dots)$  (float tensor). If `preds` is a floating point we apply `torch.argmax` along the `C` dimension to automatically convert probabilities/logits into an int tensor.
- `target` (int tensor):  $(N, \dots)$

The influence of the additional dimension  $\dots$  (if present) will be determined by the `multidim_average` argument.

**Parameters**

- **num\_classes** (`int`) – Integer specifying the number of classes
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: Calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: Calculates statistic for each label and applies no reduction
- **top\_k** (`int`) – Number of highest probability or logit score predictions considered to find the correct label. Only works when `preds` contain probabilities/logits.
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions  $\dots$  should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the `N` axis. The statistics in this case are calculated over the additional dimensions.



- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be `(C,)`
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

**Return type** The returned shape depends on the `average` and `multidim_average` arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import MulticlassSpecificity
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> metric = MulticlassSpecificity(num_classes=3)
>>> metric(preds, target)
tensor(0.8889)
>>> metric = MulticlassSpecificity(num_classes=3, average=None)
>>> metric(preds, target)
tensor([1.0000, 0.6667, 1.0000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import MulticlassSpecificity
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> metric = MulticlassSpecificity(num_classes=3)
>>> metric(preds, target)
tensor(0.8889)
>>> metric = MulticlassSpecificity(num_classes=3, average=None)
>>> metric(preds, target)
tensor([1.0000, 0.6667, 1.0000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import MulticlassSpecificity
>>> target = torch.tensor([[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]])
>>> preds = torch.tensor([[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]])
>>> metric = MulticlassSpecificity(num_classes=3, multidim_average='samplewise')
>>> metric(preds, target)
```

(continues on next page)

(continued from previous page)

```

tensor([0.7500, 0.6556])
>>> metric = MulticlassSpecificity(num_classes=3, multidim_average='samplewise',
    ↪ average=None)
>>> metric(preds, target)
tensor([[0.7500, 0.7500, 0.7500],
        [0.8000, 0.6667, 0.5000]])

```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### **compute()**

Computes the final statistics.

**Return type** `Tensor`

#### **Returns**

The metric returns a tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to  $[tp, fp, tn, fn, sup]$  ( $sup$  stands for support and equals  $tp + fn$ ). The shape depends on `average` and `multidim_average` parameters:

- If `multidim_average` is set to `global`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(5,)$
- If `average=None/'none'`, the shape will be  $(C, 5)$
- If `multidim_average` is set to `samplewise`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(N, 5)$
- If `average=None/'none'`, the shape will be  $(N, C, 5)$

## MultilabelSpecificity

```

class torchmetrics.classification.MultilabelSpecificity(num_labels, threshold=0.5,
                                                         average='macro',
                                                         multidim_average='global',
                                                         ignore_index=None, validate_args=True,
                                                         **kwargs)

```

Computes *Specificity* for multilabel tasks

$$\text{Specificity} = \frac{TN}{TN + FP}$$

Where TN and FP represent the number of true negatives and false positives respectively.

Accepts the following input tensors:

- **preds** (int or float tensor):  $(N, C, \dots)$ . If preds is a floating point tensor with values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **target** (int tensor):  $(N, C, \dots)$

The influence of the additional dimension  $\dots$  (if present) will be determined by the `multidim_average` argument.

#### **Parameters**

- **num\_labels** (`int`) – Integer specifying the number of labels

- **threshold** (`float`) – Threshold for transforming probability to binary (0,1) predictions
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: Calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: Calculates statistic for each label and applies no reduction
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

#### Returns

- If `multidim_average` is set to `global` - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor - If `average=None/'none'`, the shape will be (C,)
- If `multidim_average` is set to `samplewise` - If `average='micro'/'macro'/'weighted'`, the shape will be (N,) - If `average=None/'none'`, the shape will be (N, C)

**Return type** The returned shape depends on the `average` and `multidim_average` arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import MultilabelSpecificity
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelSpecificity(num_labels=3)
>>> metric(preds, target)
tensor(0.6667)
>>> metric = MultilabelSpecificity(num_labels=3, average=None)
>>> metric(preds, target)
tensor([1., 1., 0.])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import MultilabelSpecificity
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelSpecificity(num_labels=3)
>>> metric(preds, target)
tensor(0.6667)
>>> metric = MultilabelSpecificity(num_labels=3, average=None)
```

(continues on next page)

(continued from previous page)

```
>>> metric(preds, target)
tensor([1., 1., 0.])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import MultilabelSpecificity
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> metric = MultilabelSpecificity(num_labels=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.0000, 0.3333])
>>> metric = MultilabelSpecificity(num_labels=3, multidim_average='samplewise',
... ↪average=None)
>>> metric(preds, target)
tensor([[0., 0., 0.],
        [0., 0., 1.]])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes the final statistics.

**Return type** `Tensor`

**Returns**

The metric returns a tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to  $[tp, fp, tn, fn, sup]$  ( $sup$  stands for support and equals  $tp + fn$ ). The shape depends on `average` and `multidim_average` parameters:

- If `multidim_average` is set to `global`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(5,)$
- If `average=None/'none'`, the shape will be  $(C, 5)$
- If `multidim_average` is set to `samplewise`
- If `average='micro'/'macro'/'weighted'`, the shape will be  $(N, 5)$
- If `average=None/'none'`, the shape will be  $(N, C, 5)$

### 1.43.2 Functional Interface

```
torchmetrics.functional.specificity(preds, target, average='micro', mdmc_average=None,
                                     ignore_index=None, num_classes=None, threshold=0.5,
                                     top_k=None, multiclass=None, task=None, num_labels=None,
                                     multidim_average='global', validate_args=True)
```

---

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

---

Computes *Specificity*

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Where TN and FP represent the number of true negatives and false positives respectively. With the use of `top_k` parameter, this metric can generalize to Specificity@K.

The reduction method (how the specificity scores are aggregated) is controlled by the `average` parameter, and additionally by the `mdmc_average` parameter in the multi-dimensional multi-class case. Accepts all inputs listed in *Input types*.

#### Parameters

- **preds** (*Tensor*) – Predictions from model (probabilities, or labels)
- **target** (*Tensor*) – Ground truth values
- **average** (*Optional*[*Literal*['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied. Should be one of the following:
  - 'micro' [default]: Calculate the metric globally, across all samples and classes.
  - 'macro': Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
  - 'weighted': Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (`tn + fp`).
  - 'none' or `None`: Calculate the metric for each class separately, and return the metric for every class.
  - 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

---

**Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_average`.

---



---

**Note:** If 'none' and a given class doesn't occur in the `preds` or `target`, the value for the class will be nan.

---

- **mdmc\_average** (*Optional*[*str*]) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:

- `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
- `'samplewise'`: In this case, the statistics are computed separately for each sample on the `N` axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... (see [Input types](#)) as the `N` dimension within the sample, and computing the metric for the sample based on that.
- `'global'`: In this case the `N` and ... dimensions of the inputs (see [Input types](#)) are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were `(N_X, C)`. From here on the `average` parameter applies as usual.
- **`ignore_index`** ([Optional\[int\]](#)) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or `'none'`, the score for the ignored class will be returned as `nan`.
- **`num_classes`** ([Optional\[int\]](#)) – Number of classes. Necessary for `'macro'`, `'weighted'` and `None` average methods.
- **`threshold`** ([float](#)) – Threshold probability value for transforming probability predictions to binary (0,1) predictions, in the case of binary or multi-label inputs
- **`top_k`** ([Optional\[int\]](#)) – Number of highest probability entries for each sample to convert to 1s - relevant only for inputs with probability predictions. If this parameter is set for multi-label inputs, it will take precedence over `threshold`. For (multi-dim) multi-class inputs, this parameter defaults to 1.  
  
Should be left unset (`None`) for inputs with label predictions.
- **`multiclass`** ([Optional\[bool\]](#)) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's [documentation section](#) for a more detailed explanation and examples.

**Return type** [Tensor](#)

#### Returns

The shape of the returned tensor depends on the `average` parameter

- If `average` in `['micro', 'macro', 'weighted', 'samples']`, a one-element tensor will be returned
- If `average` in `['none', None]`, the shape will be `(C,)`, where `C` stands for the number of classes

#### Raises

- **`ValueError`** – If `average` is not one of `"micro"`, `"macro"`, `"weighted"`, `"samples"`, `"none"` or `None`
- **`ValueError`** – If `mdmc_average` is not one of `None`, `"samplewise"`, `"global"`.
- **`ValueError`** – If `average` is set but `num_classes` is not provided.
- **`ValueError`** – If `num_classes` is set and `ignore_index` is not in the range `[0, num_classes)`.

### Example

```
>>> from torchmetrics.functional import specificity
>>> preds = torch.tensor([2, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> specificity(preds, target, average='macro', num_classes=3)
tensor(0.6111)
>>> specificity(preds, target, average='micro')
tensor(0.6250)
```

### binary\_specificity

`torchmetrics.functional.classification.binary_specificity(preds, target, threshold=0.5, multidim_average='global', ignore_index=None, validate_args=True)`

Computes *Specificity* for binary tasks:

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Where TN and FP represent the number of true negatives and false positives respectively.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

#### Parameters

- **preds** (**Tensor**) – Tensor with predictions
- **target** (**Tensor**) – Tensor with true labels
- **threshold** (**float**) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (**Literal**['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (**Optional**[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (**bool**) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.

#### Return type **Tensor**

**Returns** If *multidim\_average* is set to *global*, the metric returns a scalar value. If *multidim\_average* is set to *samplewise*, the metric returns (N,) vector consisting of a scalar value per sample.

Example (preds is int tensor):

```
>>> from torchmetrics.functional.classification import binary_specificity
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0, 0, 1, 1, 0, 1])
>>> binary_specificity(preds, target)
tensor(0.6667)
```

Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import binary_specificity
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> binary_specificity(preds, target)
tensor(0.6667)
```

Example (multidim tensors):

```
>>> from torchmetrics.functional.classification import binary_specificity
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> binary_specificity(preds, target, multidim_average='samplewise')
tensor([0.0000, 0.3333])
```

## multiclass\_specificity

`torchmetrics.functional.classification.multiclass_specificity(preds, target, num_classes, average='macro', top_k=1, multidim_average='global', ignore_index=None, validate_args=True)`

Computes *Specificity* for multiclass tasks:

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Where TN and FP represent the number of true negatives and false positives respectively.

Accepts the following input tensors:

- **preds** (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

### Parameters

- **preds** (*Tensor*) – Tensor with predictions
- **target** (*Tensor*) – Tensor with true labels



- **num\_classes** (`int`) – Integer specifying the number of classes
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: Calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: Calculates statistic for each label and applies no reduction
- **top\_k** (`int`) – Number of highest probability or logit score predictions considered to find the correct label. Only works when `preds` contain probabilities/logits.
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the `N` axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

#### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be `(C,)`
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

**Return type** The returned shape depends on the `average` and `multidim_average` arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_specificity
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> multiclass_specificity(preds, target, num_classes=3)
tensor(0.8889)
>>> multiclass_specificity(preds, target, num_classes=3, average=None)
tensor([1.0000, 0.6667, 1.0000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_specificity
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
```

(continues on next page)

(continued from previous page)

```

...   [0.22, 0.61, 0.17],
...   [0.71, 0.09, 0.20],
...   [0.05, 0.82, 0.13],
... ])
>>> multiclass_specificity(preds, target, num_classes=3)
tensor(0.8889)
>>> multiclass_specificity(preds, target, num_classes=3, average=None)
tensor([1.0000, 0.6667, 1.0000])

```

**Example (multidim tensors):**

```

>>> from torchmetrics.functional.classification import multiclass_specificity
>>> target = torch.tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = torch.tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> multiclass_specificity(preds, target, num_classes=3, multidim_average=
    ↳ 'samplewise')
tensor([0.7500, 0.6556])
>>> multiclass_specificity(preds, target, num_classes=3, multidim_average=
    ↳ 'samplewise', average=None)
tensor([[0.7500, 0.7500, 0.7500],
        [0.8000, 0.6667, 0.5000]])

```

## multilabel\_specificity

`torchmetrics.functional.classification.multilabel_specificity(preds, target, num_labels, threshold=0.5, average='macro', multidim_average='global', ignore_index=None, validate_args=True)`

Computes *Specificity* for multilabel tasks

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Where TN and FP represent the number of true negatives and false positives respectively.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in *threshold*.
- **target** (int tensor): (N, C, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

### Parameters

- **preds** (**Tensor**) – Tensor with predictions
- **target** (**Tensor**) – Tensor with true labels
- **num\_labels** (**int**) – Integer specifying the number of labels
- **threshold** (**float**) – Threshold for transforming probability to binary (0,1) predictions
- **average** (**Optional[Literal['micro', 'macro', 'weighted', 'none']]**) – Defines the reduction that is applied over labels. Should be one of the following:

- micro: Sum statistics over all labels
- macro: Calculate statistics for each label and average them
- weighted: Calculates statistics for each label and computes weighted average using their support
- "none" or None: Calculates statistic for each label and applies no reduction
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - global: Additional dimensions are flattened along the batch dimension
  - samplewise: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

#### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be `(C,)`
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

**Return type** The returned shape depends on the `average` and `multidim_average` arguments

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_specificity
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_specificity(preds, target, num_labels=3)
tensor(0.6667)
>>> multilabel_specificity(preds, target, num_labels=3, average=None)
tensor([1., 1., 0.])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_specificity
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_specificity(preds, target, num_labels=3)
tensor(0.6667)
>>> multilabel_specificity(preds, target, num_labels=3, average=None)
tensor([1., 1., 0.])
```

**Example (multidim tensors):**

```

>>> from torchmetrics.functional.classification import multilabel_specificity
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> multilabel_specificity(preds, target, num_labels=3, multidim_average=
↳ 'samplewise')
tensor([0.0000, 0.3333])
>>> multilabel_specificity(preds, target, num_labels=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[0., 0., 0.],
        [0., 0., 1.]])

```

## 1.44 Stat Scores

### 1.44.1 Module Interface

#### StatScores

```

class torchmetrics.StatScores(threshold=0.5, top_k=None, reduce='micro', num_classes=None,
                              ignore_index=None, mdmc_reduce=None, multiclass=None, task=None,
                              average='macro', num_labels=None, multidim_average='global',
                              validate_args=True, **kwargs)

```

---

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

---

Computes the number of true positives, false positives, true negatives, false negatives. Related to [Type I and Type II errors](#) and the [confusion matrix](#).

The reduction method (how the statistics are aggregated) is controlled by the `reduce` parameter, and additionally by the `mdmc_reduce` parameter in the multi-dimensional multi-class case.

Accepts all inputs listed in [Input types](#).

#### Parameters

- **threshold** (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
- **top\_k** (`Optional[int]`) – Number of the highest probability or logit score predictions considered finding the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (None) will be interpreted as 1 for these inputs. Should be left at default (None) for all other types of inputs.

- **reduce** (*str*) – Defines the reduction that is applied. Should be one of the following:
  - 'micro' [default]: Counts the statistics by summing over all [sample, class] combinations (globally). Each statistic is represented by a single integer.
  - 'macro': Counts the statistics for each class separately (over all samples). Each statistic is represented by a (C,) tensor. Requires num\_classes to be set.
  - 'samples': Counts the statistics for each sample separately (over all classes). Each statistic is represented by a (N, ) 1d tensor.

---

**Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of mdmc\_reduce.

---

- **num\_classes** (*Optional[int]*) – Number of classes. Necessary for (multi-dimensional) multi-class or multi-label data.
- **ignore\_index** (*Optional[int]*) – Specify a class (label) to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and reduce='macro', the class statistics for the ignored class will all be returned as -1.
- **mdmc\_reduce** (*Optional[str]*) – Defines how the multi-dimensional multi-class inputs are handled. Should be one of the following:
  - None [default]: Should be left unchanged if your data is not multi-dimensional multi-class (see *Input types* for the definition of input types).
  - 'samplewise': In this case, the statistics are computed separately for each sample on the N axis, and then the outputs are concatenated together. In each sample the extra axes ... are flattened to become the sub-sample axis, and statistics for each sample are computed by treating the sub-sample axis as the N axis for that sample.
  - 'global': In this case the N and ... dimensions of the inputs are flattened into a new N\_X sample axis, i.e. the inputs are treated as if they were (N\_X, C). From here on the reduce parameter applies as usual.
- **multiclass** (*Optional[bool]*) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.
- **kwargs** (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

#### Raises

- **ValueError** – If reduce is none of "micro", "macro" or "samples".
- **ValueError** – If mdmc\_reduce is none of None, "samplewise", "global".
- **ValueError** – If reduce is set to "macro" and num\_classes is not provided.
- **ValueError** – If num\_classes is set and ignore\_index is not in the range  $0 \leq \text{ignore\_index} < \text{num\_classes}$ .

### Example

```
>>> from torchmetrics.classification import StatScores
>>> preds = torch.tensor([1, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> stat_scores = StatScores(reduce='macro', num_classes=3)
>>> stat_scores(preds, target)
tensor([[0, 1, 2, 1, 1],
        [1, 1, 1, 1, 2],
        [1, 0, 3, 0, 1]])
>>> stat_scores = StatScores(reduce='micro')
>>> stat_scores(preds, target)
tensor([2, 2, 6, 2, 4])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### **compute()**

Computes the stat scores based on inputs passed in to update previously.

**Return type** `Tensor`

#### **Returns**

The metric returns a tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to  $[tp, fp, tn, fn, sup]$  ( $sup$  stands for support and equals  $tp + fn$ ). The shape depends on the `reduce` and `mdmc_reduce` (in case of multi-dimensional multi-class data) parameters:

- If the data is not multi-dimensional multi-class, then
  - If `reduce='micro'`, the shape will be  $(5, )$
  - If `reduce='macro'`, the shape will be  $(C, 5)$ , where  $C$  stands for the number of classes
  - If `reduce='samples'`, the shape will be  $(N, 5)$ , where  $N$  stands for the number of samples
- If the data is multi-dimensional multi-class and `mdmc_reduce='global'`, then
  - If `reduce='micro'`, the shape will be  $(5, )$
  - If `reduce='macro'`, the shape will be  $(C, 5)$
  - If `reduce='samples'`, the shape will be  $(N \times X, 5)$ , where  $X$  stands for the product of sizes of all “extra” dimensions of the data (i.e. all dimensions except for  $C$  and  $N$ )
- If the data is multi-dimensional multi-class and `mdmc_reduce='samplewise'`, then
  - If `reduce='micro'`, the shape will be  $(N, 5)$
  - If `reduce='macro'`, the shape will be  $(N, C, 5)$
  - If `reduce='samples'`, the shape will be  $(N, X, 5)$

#### **update(preds, target)**

Update state with predictions and targets.

See [Input types](#) for more information on input types.

#### **Parameters**

- **preds** (`Tensor`) – Predictions from model (probabilities, logits or labels)
- **target** (`Tensor`) – Ground truth values

Return type `None`

## BinaryStatScores

```
class torchmetrics.classification.BinaryStatScores(threshold=0.5, multidim_average='global',
                                                    ignore_index=None, validate_args=True,
                                                    **kwargs)
```

Computes the number of true positives, false positives, true negatives, false negatives and the support for binary tasks. Related to [Type I and Type II errors](#).

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

### Parameters

- **threshold** (`float`) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (`Literal`[‘global’, ‘samplewise’]) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional`[`int`]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Example (preds is int tensor):

```
>>> from torchmetrics.classification import BinaryStatScores
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0, 0, 1, 1, 0, 1])
>>> metric = BinaryStatScores()
>>> metric(preds, target)
tensor([2, 1, 2, 1, 3])
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import BinaryStatScores
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> metric = BinaryStatScores()
>>> metric(preds, target)
tensor([2, 1, 2, 1, 3])
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import BinaryStatScores
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> metric = BinaryStatScores(multidim_average='samplewise')
>>> metric(preds, target)
tensor([[2, 3, 0, 1, 3],
        [0, 2, 1, 3, 3]])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

## MulticlassStatScores

```
class torchmetrics.classification.MulticlassStatScores(num_classes, top_k=1, average='macro',
                                                       multidim_average='global',
                                                       ignore_index=None, validate_args=True,
                                                       **kwargs)
```

Computes the number of true positives, false positives, true negatives, false negatives and the support for multi-class tasks. Related to [Type I and Type II errors](#).

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

### Parameters

- **num\_classes** (`int`) – Integer specifying the number of classes
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: Calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: Calculates statistic for each label and applies no reduction
- **top\_k** (`int`) – Number of highest probability or logit score predictions considered to find the correct label. Only works when **preds** contain probabilities/logits.
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.



- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Example (preds is int tensor):

```
>>> from torchmetrics.classification import MulticlassStatScores
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> metric = MulticlassStatScores(num_classes=3, average='micro')
>>> metric(preds, target)
tensor([3, 1, 7, 1, 4])
>>> metric = MulticlassStatScores(num_classes=3, average=None)
>>> metric(preds, target)
tensor([[1, 0, 2, 1, 2],
        [1, 1, 2, 0, 1],
        [1, 0, 3, 0, 1]])
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import MulticlassStatScores
>>> target = target = torch.tensor([2, 1, 0, 0])
>>> preds = preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> metric = MulticlassStatScores(num_classes=3, average='micro')
>>> metric(preds, target)
tensor([3, 1, 7, 1, 4])
>>> metric = MulticlassStatScores(num_classes=3, average=None)
>>> metric(preds, target)
tensor([[1, 0, 2, 1, 2],
        [1, 1, 2, 0, 1],
        [1, 0, 3, 0, 1]])
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import MulticlassStatScores
>>> target = torch.tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = torch.tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> metric = MulticlassStatScores(num_classes=3, multidim_average="samplewise",
...                               average='micro')
>>> metric(preds, target)
tensor([[3, 3, 9, 3, 6],
        [2, 4, 8, 4, 6]])
>>> metric = MulticlassStatScores(num_classes=3, multidim_average="samplewise",
...                               average=None)
>>> metric(preds, target)
tensor([[[2, 1, 3, 0, 2],
```

(continues on next page)

(continued from previous page)

```
[0, 1, 3, 2, 2],
[1, 1, 3, 1, 2]],
[[0, 1, 4, 1, 1],
[1, 1, 2, 2, 3],
[1, 2, 2, 1, 2]]])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

## MultilabelStatScores

```
class torchmetrics.classification.MultilabelStatScores(num_labels, threshold=0.5,
                                                       average='macro',
                                                       multidim_average='global',
                                                       ignore_index=None, validate_args=True,
                                                       **kwargs)
```

Computes the number of true positives, false positives, true negatives, false negatives and the support for multi-label tasks. Related to [Type I and Type II errors](#).

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

### Parameters

- **num\_labels** (**int**) – Integer specifying the number of labels
- **threshold** (**float**) – Threshold for transforming probability to binary (0,1) predictions
- **average** (**Optional[Literal['micro', 'macro', 'weighted', 'none']]**) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: Calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: Calculates statistic for each label and applies no reduction
- **multidim\_average** (**Literal['global', 'samplewise']**) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (**Optional[int]**) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (**bool**) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

- **kwargs** (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import MultilabelStatScores
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelStatScores(num_labels=3, average='micro')
>>> metric(preds, target)
tensor([2, 1, 2, 1, 3])
>>> metric = MultilabelStatScores(num_labels=3, average=None)
>>> metric(preds, target)
tensor([[1, 0, 1, 0, 1],
        [0, 0, 1, 1, 1],
        [1, 1, 0, 0, 1]])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import MultilabelStatScores
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelStatScores(num_labels=3, average='micro')
>>> metric(preds, target)
tensor([2, 1, 2, 1, 3])
>>> metric = MultilabelStatScores(num_labels=3, average=None)
>>> metric(preds, target)
tensor([[1, 0, 1, 0, 1],
        [0, 0, 1, 1, 1],
        [1, 1, 0, 0, 1]])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import MultilabelStatScores
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> metric = MultilabelStatScores(num_labels=3, multidim_average='samplewise',
...                               average='micro')
>>> metric(preds, target)
tensor([[2, 3, 0, 1, 3],
        [0, 2, 1, 3, 3]])
>>> metric = MultilabelStatScores(num_labels=3, multidim_average='samplewise',
...                               average=None)
>>> metric(preds, target)
tensor([[[1, 1, 0, 0, 1],
         [1, 1, 0, 0, 1],
         [0, 1, 0, 1, 1]],
        [[0, 0, 0, 2, 2],
         [0, 2, 0, 0, 0],
         [0, 0, 1, 1, 1]])
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

## 1.44.2 Functional Interface

### `stat_scores`

```
torchmetrics.functional.stat_scores(preds, target, reduce='micro', mdmc_reduce=None,
                                     num_classes=None, top_k=None, threshold=0.5, multiclass=None,
                                     ignore_index=None, task=None, num_labels=None, average='micro',
                                     multidim_average='global', validate_args=True)
```

---

**Note:** From v0.10 an `'binary_*'`, `'multiclass_*'`, `'multilabel_*'` version now exist of each classification metric. Moving forward we recommend using these versions. This base metric will still work as it did prior to v0.10 until v0.11. From v0.11 the `task` argument introduced in this metric will be required and the general order of arguments may change, such that this metric will just function as an single entrypoint to calling the three specialized versions.

---

Computes the number of true positives, false positives, true negatives, false negatives. Related to [Type I and Type II errors](#) and the [confusion matrix](#).

The reduction method (how the statistics are aggregated) is controlled by the `reduce` parameter, and additionally by the `mdmc_reduce` parameter in the multi-dimensional multi-class case. Accepts all inputs listed in [Input types](#).

#### Parameters

- **preds** ([Tensor](#)) – Predictions from model (probabilities, logits or labels)
- **target** ([Tensor](#)) – Ground truth values
- **threshold** ([float](#)) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
- **top\_k** ([Optional\[int\]](#)) – Number of highest probability or logit score predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (None) will be interpreted as 1 for these inputs.  
Should be left at default (None) for all other types of inputs.
- **reduce** ([str](#)) – Defines the reduction that is applied. Should be one of the following:
  - `'micro'` [default]: Counts the statistics by summing over all [sample, class] combinations (globally). Each statistic is represented by a single integer.
  - `'macro'`: Counts the statistics for each class separately (over all samples). Each statistic is represented by a (C, ) tensor. Requires `num_classes` to be set.
  - `'samples'`: Counts the statistics for each sample separately (over all classes). Each statistic is represented by a (N, ) 1d tensor.

---

**Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_reduce`.

---

- **num\_classes** ([Optional\[int\]](#)) – Number of classes. Necessary for (multi-dimensional) multi-class or multi-label data.

- **ignore\_index** (`Optional[int]`) – Specify a class (label) to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `reduce='macro'`, the class statistics for the ignored class will all be returned as -1.
- **mdmc\_reduce** (`Optional[str]`) – Defines how the multi-dimensional multi-class inputs are handled. Should be one of the following:
  - `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class (see [Input types](#) for the definition of input types).
  - `'samplewise'`: In this case, the statistics are computed separately for each sample on the `N` axis, and then the outputs are concatenated together. In each sample the extra axes ... are flattened to become the sub-sample axis, and statistics for each sample are computed by treating the sub-sample axis as the `N` axis for that sample.
  - `'global'`: In this case the `N` and ... dimensions of the inputs are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were `(N_X, C)`. From here on the `reduce` parameter applies as usual.
- **multiclass** (`Optional[bool]`) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's [documentation section](#) for a more detailed explanation and examples.

**Return type** `Tensor`

#### Returns

The metric returns a tensor of shape `(..., 5)`, where the last dimension corresponds to `[tp, fp, tn, fn, sup]` (`sup` stands for support and equals `tp + fn`). The shape depends on the `reduce` and `mdmc_reduce` (in case of multi-dimensional multi-class data) parameters:

- If the data is not multi-dimensional multi-class, then
  - If `reduce='micro'`, the shape will be `(5, )`
  - If `reduce='macro'`, the shape will be `(C, 5)`, where `C` stands for the number of classes
  - If `reduce='samples'`, the shape will be `(N, 5)`, where `N` stands for the number of samples
- If the data is multi-dimensional multi-class and `mdmc_reduce='global'`, then
  - If `reduce='micro'`, the shape will be `(5, )`
  - If `reduce='macro'`, the shape will be `(C, 5)`
  - If `reduce='samples'`, the shape will be `(N*X, 5)`, where `X` stands for the product of sizes of all “extra” dimensions of the data (i.e. all dimensions except for `C` and `N`)
- If the data is multi-dimensional multi-class and `mdmc_reduce='samplewise'`, then
  - If `reduce='micro'`, the shape will be `(N, 5)`
  - If `reduce='macro'`, the shape will be `(N, C, 5)`
  - If `reduce='samples'`, the shape will be `(N, X, 5)`

#### Raises

- **ValueError** – If `reduce` is none of `"micro"`, `"macro"` or `"samples"`.
- **ValueError** – If `mdmc_reduce` is none of `None`, `"samplewise"`, `"global"`.
- **ValueError** – If `reduce` is set to `"macro"` and `num_classes` is not provided.

- **ValueError** – If `num_classes` is set and `ignore_index` is not in the range `[0, num_classes)`.
- **ValueError** – If `ignore_index` is used with binary data.
- **ValueError** – If inputs are multi-dimensional multi-class and `mdmc_reduce` is not provided.

### Example

```
>>> from torchmetrics.functional import stat_scores
>>> preds = torch.tensor([1, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> stat_scores(preds, target, reduce='macro', num_classes=3)
tensor([[0, 1, 2, 1, 1],
        [1, 1, 1, 1, 2],
        [1, 0, 3, 0, 1]])
>>> stat_scores(preds, target, reduce='micro')
tensor([2, 2, 6, 2, 4])
```

### binary\_stat\_scores

`torchmetrics.functional.classification.binary_stat_scores(preds, target, threshold=0.5, multidim_average='global', ignore_index=None, validate_args=True)`

Computes the number of true positives, false positives, true negatives, false negatives and the support for binary tasks. Related to [Type I and Type II errors](#).

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If `preds` is a floating point tensor with values outside `[0,1]` range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the `multidim_average` argument.

#### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary `{0,1}` predictions
- **multidim\_average** ([Literal](#)`['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the `N` axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)`[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation

- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Return type** `Tensor`

#### Returns

The metric returns a tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to `[tp, fp, tn, fn, sup]` (sup stands for support and equals `tp + fn`). The shape depends on the `multidim_average` parameter:

- If `multidim_average` is set to `global`, the shape will be  $(5,)$
- If `multidim_average` is set to `samplewise`, the shape will be  $(N, 5)$

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import binary_stat_scores
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0, 0, 1, 1, 0, 1])
>>> binary_stat_scores(preds, target)
tensor([2, 1, 2, 1, 3])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import binary_stat_scores
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> binary_stat_scores(preds, target)
tensor([2, 1, 2, 1, 3])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import binary_stat_scores
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> binary_stat_scores(preds, target, multidim_average='samplewise')
tensor([[2, 3, 0, 1, 3],
        [0, 2, 1, 3, 3]])
```

## multiclass\_stat\_scores

`torchmetrics.functional.classification.multiclass_stat_scores`(*preds, target, num\_classes, average='macro', top\_k=1, multidim\_average='global', ignore\_index=None, validate\_args=True*)

Computes the number of true positives, false positives, true negatives, false negatives and the support for multi-class tasks. Related to [Type I and Type II errors](#).

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

#### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: Calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: Calculates statistic for each label and applies no reduction
- **top\_k** ([int](#)) – Number of highest probability or logit score predictions considered to find the correct label. Only works when preds contain probabilities/logits.
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

#### Return type [Tensor](#)

#### Returns

The metric returns a tensor of shape (..., 5), where the last dimension corresponds to [tp, fp, tn, fn, sup] (sup stands for support and equals tp + fn). The shape depends on average and multidim\_average parameters:

- If **multidim\_average** is set to **global**:
  - If **average**='micro'/'macro'/'weighted', the shape will be (5,)
  - If **average**=None/'none', the shape will be (C, 5)
- If **multidim\_average** is set to **samplewise**:
  - If **average**='micro'/'macro'/'weighted', the shape will be (N, 5)
  - If **average**=None/'none', the shape will be (N, C, 5)

**Example (preds is int tensor):**



```
>>> from torchmetrics.functional.classification import multiclass_stat_scores
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> multiclass_stat_scores(preds, target, num_classes=3, average='micro')
tensor([3, 1, 7, 1, 4])
>>> multiclass_stat_scores(preds, target, num_classes=3, average=None)
tensor([[1, 0, 2, 1, 2],
        [1, 1, 2, 0, 1],
        [1, 0, 3, 0, 1]])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_stat_scores
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([
...     [0.16, 0.26, 0.58],
...     [0.22, 0.61, 0.17],
...     [0.71, 0.09, 0.20],
...     [0.05, 0.82, 0.13],
... ])
>>> multiclass_stat_scores(preds, target, num_classes=3, average='micro')
tensor([3, 1, 7, 1, 4])
>>> multiclass_stat_scores(preds, target, num_classes=3, average=None)
tensor([[1, 0, 2, 1, 2],
        [1, 1, 2, 0, 1],
        [1, 0, 3, 0, 1]])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import multiclass_stat_scores
>>> target = torch.tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = torch.tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> multiclass_stat_scores(preds, target, num_classes=3, multidim_average=
↳ 'samplewise', average='micro')
tensor([[3, 3, 9, 3, 6],
        [2, 4, 8, 4, 6]])
>>> multiclass_stat_scores(preds, target, num_classes=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[[2, 1, 3, 0, 2],
         [0, 1, 3, 2, 2],
         [1, 1, 3, 1, 2]],
        [[0, 1, 4, 1, 1],
         [1, 1, 2, 2, 3],
         [1, 2, 2, 1, 2]]])
```

## multilabel\_stat\_scores

```
torchmetrics.functional.classification.multilabel_stat_scores(preds, target, num_labels,
                                                             threshold=0.5, average='macro',
                                                             multidim_average='global',
                                                             ignore_index=None,
                                                             validate_args=True)
```

Computes the number of true positives, false positives, true negatives, false negatives and the support for multi-label tasks. Related to [Type I and Type II errors](#).

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

The influence of the additional dimension ... (if present) will be determined by the *multidim\_average* argument.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: Calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: Calculates statistic for each label and applies no reduction
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

### Return type [Tensor](#)

### Returns

The metric returns a tensor of shape (... , 5), where the last dimension corresponds to [tp, fp, tn, fn, sup] (sup stands for support and equals tp + fn). The shape depends on **average** and **multidim\_average** parameters:

- If **multidim\_average** is set to **global**:

- If average='micro'/'macro'/'weighted', the shape will be (5,)
- If average=None/'none', the shape will be (C, 5)
- If multidim\_average is set to samplewise:
  - If average='micro'/'macro'/'weighted', the shape will be (N, 5)
  - If average=None/'none', the shape will be (N, C, 5)

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_stat_scores
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_stat_scores(preds, target, num_labels=3, average='micro')
tensor([2, 1, 2, 1, 3])
>>> multilabel_stat_scores(preds, target, num_labels=3, average=None)
tensor([[1, 0, 1, 0, 1],
        [0, 0, 1, 1, 1],
        [1, 1, 0, 0, 1]])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_stat_scores
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_stat_scores(preds, target, num_labels=3, average='micro')
tensor([2, 1, 2, 1, 3])
>>> multilabel_stat_scores(preds, target, num_labels=3, average=None)
tensor([[1, 0, 1, 0, 1],
        [0, 0, 1, 1, 1],
        [1, 1, 0, 0, 1]])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import multilabel_stat_scores
>>> target = torch.tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = torch.tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> multilabel_stat_scores(preds, target, num_labels=3, multidim_average=
↳ 'samplewise', average='micro')
tensor([[2, 3, 0, 1, 3],
        [0, 2, 1, 3, 3]])
>>> multilabel_stat_scores(preds, target, num_labels=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[[1, 1, 0, 0, 1],
        [1, 1, 0, 0, 1],
        [0, 1, 0, 1, 1]],
        [[0, 0, 0, 2, 2],
        [0, 2, 0, 0, 0],
        [0, 0, 1, 1, 1]]])
```

## 1.45 Error Relative Global Dim. Synthesis (ERGAS)

### 1.45.1 Module Interface

```
class torchmetrics.image.ergas.ErrorRelativeGlobalDimensionlessSynthesis(ratio=4, reduction='elementwise_mean', **kwargs)
```

Relative dimensionless global error synthesis (ERGAS) is used to calculate the accuracy of Pan sharpened image considering normalized average error of each band of the result image (ErrorRelativeGlobalDimensionlessSynthesis).

#### Parameters

- **ratio** (`Union[int, float]`) – ratio of high resolution to low resolution
- **reduction** (`Literal['elementwise_mean', 'sum', 'none', None]`) – a method to reduce metric score over labels.
  - 'elementwise\_mean': takes the mean (default)
  - 'sum': takes the sum
  - 'none' or None: no reduction will be applied
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Returns** Tensor with ErrorRelativeGlobalDimensionlessSynthesis score

#### Example

```
>>> import torch
>>> from torchmetrics import ErrorRelativeGlobalDimensionlessSynthesis
>>> preds = torch.rand([16, 1, 16, 16], generator=torch.manual_seed(42))
>>> target = preds * 0.75
>>> ergas = ErrorRelativeGlobalDimensionlessSynthesis()
>>> torch.round(ergas(preds, target))
tensor(154.)
```

#### References

[1] Qian Du; Nicholas H. Younan; Roger King; Vijay P. Shah, “On the Performance Evaluation of Pan-Sharpening Techniques” in IEEE Geoscience and Remote Sensing Letters, vol. 4, no. 4, pp. 518-522, 15 October 2007, doi: 10.1109/LGRS.2007.896328.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes explained variance over state.

**Return type** `Tensor`

**update(preds, target)**

Update state with predictions and targets.

#### Parameters

- **preds** (`Tensor`) – Predictions from model

- **target** (`Tensor`) – Ground truth values

Return type `None`

## 1.45.2 Functional Interface

`torchmetrics.functional.error_relative_global_dimensionless_synthesis(preds, target, ratio=4, reduction='elementwise_mean')`

Erreur Relative Globale Adimensionnelle de Synthèse.

### Parameters

- **preds** (`Tensor`) – estimated image
- **target** (`Tensor`) – ground truth image
- **ratio** (`Union[int, float]`) – ratio of high resolution to low resolution
- **reduction** (`Literal['elementwise_mean', 'sum', 'none', None]`) – a method to reduce metric score over labels.
  - 'elementwise\_mean': takes the mean (default)
  - 'sum': takes the sum
  - 'none' or `None`: no reduction will be applied

Return type `Tensor`

Returns Tensor with RelativeG score

### Raises

- **TypeError** – If `preds` and `target` don't have the same data type.
- **ValueError** – If `preds` and `target` don't have `BxCxHxW` shape.

### Example

```
>>> from torchmetrics.functional import error_relative_global_dimensionless_
    ↳ synthesis
>>> preds = torch.rand([16, 1, 16, 16], generator=torch.manual_seed(42))
>>> target = preds * 0.75
>>> ergds = error_relative_global_dimensionless_synthesis(preds, target)
>>> torch.round(ergds)
tensor(154.)
```

## References

[1] Qian Du; Nicholas H. Younan; Roger King; Vijay P. Shah, “On the Performance Evaluation of Pan-Sharpening Techniques” in IEEE Geoscience and Remote Sensing Letters, vol. 4, no. 4, pp. 518-522, 15 October 2007, doi: 10.1109/LGRS.2007.896328.

## 1.46 Frechet Inception Distance (FID)

### 1.46.1 Module Interface

```
class torchmetrics.image.fid.FrechetInceptionDistance(feature=2048, reset_real_features=True,
                                                    **kwargs)
```

Calculates Fréchet inception distance (**FID**) which is used to access the quality of generated images. Given by

$$FID = |\mu - \mu_w| + \text{tr}(\Sigma + \Sigma_w - 2(\Sigma\Sigma_w)^{\frac{1}{2}})$$

where  $\mathcal{N}(\mu, \Sigma)$  is the multivariate normal distribution estimated from Inception v3 [1] features calculated on real life images and  $\mathcal{N}(\mu_w, \Sigma_w)$  is the multivariate normal distribution estimated from Inception v3 features calculated on generated (fake) images. The metric was originally proposed in [1].

Using the default feature extraction (Inception v3 using the original weights from [2]), the input is expected to be mini-batches of 3-channel RGB images of shape (3 x H x W) with dtype uint8. All images will be resized to 299 x 299 which is the size of the original training data. The boolean flag `real` determines if the images should update the statistics of the real distribution or the fake distribution.

---

**Note:** using this metrics requires you to have `scipy` install. Either install as `pip install torchmetrics[image]` or `pip install scipy`

---

---

**Note:** using this metric with the default feature extractor requires that `torch-fidelity` is installed. Either install as `pip install torchmetrics[image]` or `pip install torch-fidelity`

---

### Parameters

- **feature** (`Union[int, Module]`) – Either an integer or `nn.Module`:
  - an integer will indicate the inceptionv3 feature layer to choose. Can be one of the following: 64, 192, 768, 2048
  - an `nn.Module` for using a custom feature extractor. Expects that its forward method returns an `[N, d]` matrix where `N` is the batch size and `d` is the feature size.
- **reset\_real\_features** (`bool`) – Whether to also reset the real features. Since in many cases the real dataset does not change, the features can be cached to avoid recomputing them which is costly. Set this to `False` if your dataset does not change.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

## References

- [1] Rethinking the Inception Architecture for Computer Vision Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, Zbigniew Wojna <https://arxiv.org/abs/1512.00567>
- [2] GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium, Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, Sepp Hochreiter <https://arxiv.org/abs/1706.08500>

### Raises

- **ValueError** – If `feature` is set to an `int` (default settings) and `torch-fidelity` is not installed
- **ValueError** – If `feature` is set to an `int` not in `[64, 192, 768, 2048]`
- **TypeError** – If `feature` is not an `str`, `int` or `torch.nn.Module`
- **ValueError** – If `reset_real_features` is not an `bool`

## Example

```
>>> import torch
>>> _ = torch.manual_seed(123)
>>> from torchmetrics.image.fid import FrechetInceptionDistance
>>> fid = FrechetInceptionDistance(feature=64)
>>> # generate two slightly overlapping image intensity distributions
>>> imgs_dist1 = torch.randint(0, 200, (100, 3, 299, 299), dtype=torch.uint8)
>>> imgs_dist2 = torch.randint(100, 255, (100, 3, 299, 299), dtype=torch.uint8)
>>> fid.update(imgs_dist1, real=True)
>>> fid.update(imgs_dist2, real=False)
>>> fid.compute()
tensor(12.7202)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

### `compute()`

Calculate FID score based on accumulated extracted features from the two distributions.

**Return type** `Tensor`

### `reset()`

This method automatically resets the metric state variables to their default value.

**Return type** `None`

### `update(imgs, real)`

Update the state with extracted features.

#### Parameters

- **imgs** (`Tensor`) – tensor with images feed to the feature extractor
- **real** (`bool`) – bool indicating if `imgs` belong to the real or the fake distribution

**Return type** `None`

## 1.47 Image Gradients

### 1.47.1 Functional Interface

`torchmetrics.functional.image_gradients(img)`

Computes [Gradient Computation of Image](#) of a given image using finite difference.

**Parameters** `img` ([Tensor](#)) – An (N, C, H, W) input tensor where C is the number of image channels

**Return type** [Tuple](#)[[Tensor](#), [Tensor](#)]

**Returns** Tuple of (dy, dx) with each gradient of shape [N, C, H, W]

**Raises**

- [TypeError](#) – If `img` is not of the type `torch.Tensor`.
- [RuntimeError](#) – If `img` is not a 4D tensor.

#### Example

```
>>> from torchmetrics.functional import image_gradients
>>> image = torch.arange(0, 1*1*5*5, dtype=torch.float32)
>>> image = torch.reshape(image, (1, 1, 5, 5))
>>> dy, dx = image_gradients(image)
>>> dy[0, 0, :, :]
tensor([[5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [0., 0., 0., 0., 0.]])
```

---

**Note:** The implementation follows the 1-step finite difference method as followed by the TF implementation. The values are organized such that the gradient of  $[I(x+1, y) - I(x, y)]$  are at the (x, y) location

---

## 1.48 Inception Score

### 1.48.1 Module Interface

**class** `torchmetrics.image.inception.InceptionScore`(*feature='logits\_unbiased', splits=10, \*\*kwargs*)

Calculates the Inception Score (IS) which is used to access how realistic generated images are. It is defined as

$$IS = \exp(\mathbb{E}_x KL(p(y|x)||p(y)))$$

where  $KL(p(y|x)||p(y))$  is the KL divergence between the conditional distribution  $p(y|x)$  and the marginal distribution  $p(y)$ . Both the conditional and marginal distribution is calculated from features extracted from the images. The score is calculated on random splits of the images such that both a mean and standard deviation of the score are returned. The metric was originally proposed in [1].



Using the default feature extraction (Inception v3 using the original weights from [2]), the input is expected to be mini-batches of 3-channel RGB images of shape (3 x H x W) with dtype uint8. All images will be resized to 299 x 299 which is the size of the original training data.

---

**Note:** using this metric with the default feature extractor requires that `torch-fidelity` is installed. Either install as `pip install torchmetrics[image]` or `pip install torch-fidelity`

---

### Parameters

- **feature** (`Union[str, int, Module]`) – Either an str, integer or `nn.Module`:
  - an str or integer will indicate the inceptionv3 feature layer to choose. Can be one of the following: 'logits\_unbiased', 64, 192, 768, 2048
  - an `nn.Module` for using a custom feature extractor. Expects that its forward method returns an `[N, d]` matrix where N is the batch size and d is the feature size.
- **splits** (`int`) – integer determining how many splits the inception score calculation should be split among
- **kwargs** (`Any`) – Additional keyword arguments, see *Advanced metric settings* for more info.

### References

- [1] Improved Techniques for Training GANs Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, Xi Chen <https://arxiv.org/abs/1606.03498>
- [2] GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium, Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, Sepp Hochreiter <https://arxiv.org/abs/1706.08500>

### Raises

- **ValueError** – If feature is set to an str or int and `torch-fidelity` is not installed
- **ValueError** – If feature is set to an str or int and not one of ['logits\_unbiased', 64, 192, 768, 2048]
- **TypeError** – If feature is not an str, int or `torch.nn.Module`

### Example

```
>>> import torch
>>> _ = torch.manual_seed(123)
>>> from torchmetrics.image.inception import InceptionScore
>>> inception = InceptionScore()
>>> # generate some images
>>> imgs = torch.randint(0, 255, (100, 3, 299, 299), dtype=torch.uint8)
>>> inception.update(imgs)
>>> inception.compute()
(tensor(1.0544), tensor(0.0117))
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

### compute()

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tuple[Tensor, Tensor]`

**update**(*imgs*)

Update the state with extracted features.

**Parameters** *imgs* ([Tensor](#)) – tensor with images feed to the feature extractor

**Return type** `None`

## 1.49 Kernel Inception Distance

### 1.49.1 Module Interface

```
class torchmetrics.image.kid.KernelInceptionDistance(feature=2048, subsets=100, subset_size=1000,
                                                    degree=3, gamma=None, coef=1.0,
                                                    reset_real_features=True, **kwargs)
```

Calculates Kernel Inception Distance (KID) which is used to access the quality of generated images. Given by

$$KID = MMD(f_{real}, f_{fake})^2$$

where *MMD* is the maximum mean discrepancy and  $I_{real}, I_{fake}$  are extracted features from real and fake images, see [1] for more details. In particular, calculating the MMD requires the evaluation of a polynomial kernel function *k*

$$k(x, y) = (\gamma * x^T y + coef)^{degree}$$

which controls the distance between two features. In practise the MMD is calculated over a number of subsets to be able to both get the mean and standard deviation of KID.

Using the default feature extraction (Inception v3 using the original weights from [2]), the input is expected to be mini-batches of 3-channel RGB images of shape (3 x H x W) with dtype uint8. All images will be resized to 299 x 299 which is the size of the original training data.

---

**Note:** using this metric with the default feature extractor requires that `torch-fidelity` is installed. Either install as `pip install torchmetrics[image]` or `pip install torch-fidelity`

---

#### Parameters

- **feature** ([Union](#)[[str](#), [int](#), [Module](#)]) – Either an str, integer or `nn.Module`:
  - an str or integer will indicate the inceptionv3 feature layer to choose. Can be one of the following: ‘logits\_unbiased’, 64, 192, 768, 2048
  - an `nn.Module` for using a custom feature extractor. Expects that its forward method returns an `[N, d]` matrix where N is the batch size and d is the feature size.
- **subsets** ([int](#)) – Number of subsets to calculate the mean and standard deviation scores over
- **subset\_size** ([int](#)) – Number of randomly picked samples in each subset
- **degree** ([int](#)) – Degree of the polynomial kernel function
- **gamma** ([Optional](#)[[float](#)]) – Scale-length of polynomial kernel. If set to `None` will be automatically set to the feature size

- **coef** (*float*) – Bias term in the polynomial kernel.
- **reset\_real\_features** (*bool*) – Whether to also reset the real features. Since in many cases the real dataset does not change, the features can be cached to avoid recomputing them which is costly. Set this to `False` if your dataset does not change.
- **kwargs** (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

## References

- [1] Demystifying MMD GANs Mikołaj Bińkowski, Danica J. Sutherland, Michael Arbel, Arthur Gretton <https://arxiv.org/abs/1801.01401>
- [2] GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium, Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, Sepp Hochreiter <https://arxiv.org/abs/1706.08500>

### Raises

- **ValueError** – If `feature` is set to an `int` (default settings) and `torch-fidelity` is not installed
- **ValueError** – If `feature` is set to an `int` not in `[64, 192, 768, 2048]`
- **ValueError** – If `subsets` is not an integer larger than 0
- **ValueError** – If `subset_size` is not an integer larger than 0
- **ValueError** – If `degree` is not an integer larger than 0
- **ValueError** – If `gamma` is neither `None` or a float larger than 0
- **ValueError** – If `coef` is not a float larger than 0
- **ValueError** – If `reset_real_features` is not a `bool`

## Example

```
>>> import torch
>>> _ = torch.manual_seed(123)
>>> from torchmetrics.image.kid import KernelInceptionDistance
>>> kid = KernelInceptionDistance(subset_size=50)
>>> # generate two slightly overlapping image intensity distributions
>>> imgs_dist1 = torch.randint(0, 200, (100, 3, 299, 299), dtype=torch.uint8)
>>> imgs_dist2 = torch.randint(100, 255, (100, 3, 299, 299), dtype=torch.uint8)
>>> kid.update(imgs_dist1, real=True)
>>> kid.update(imgs_dist2, real=False)
>>> kid_mean, kid_std = kid.compute()
>>> print((kid_mean, kid_std))
(tensor(0.0337), tensor(0.0023))
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

### compute()

Calculate KID score based on accumulated extracted features from the two distributions. Returns a tuple of mean and standard deviation of KID scores calculated on subsets of extracted features.

Implementation inspired by [Fid Score](#)

**Return type** `Tuple[Tensor, Tensor]`

**reset()**

This method automatically resets the metric state variables to their default value.

**Return type** `None`

**update**(*imgs, real*)

Update the state with extracted features.

**Parameters**

- **imgs** (`Tensor`) – tensor with images feed to the feature extractor
- **real** (`bool`) – bool indicating if *imgs* belong to the real or the fake distribution

**Return type** `None`

## 1.50 Learned Perceptual Image Patch Similarity (LPIPS)

### 1.50.1 Module Interface

```
class torchmetrics.image.lpip.LearnedPerceptualImagePatchSimilarity(net_type='alex',  
                                                                    reduction='mean',  
                                                                    normalize=False,  
                                                                    **kwargs)
```

The Learned Perceptual Image Patch Similarity (*LPIPS*) is used to judge the perceptual similarity between two images. LPIPS essentially computes the similarity between the activations of two image patches for some pre-defined network. This measure has been shown to match human perception well. A low LPIPS score means that image patches are perceptual similar.

Both input image patches are expected to have shape  $[N, 3, H, W]$ . The minimum size of  $H, W$  depends on the chosen backbone (see *net\_type* arg).

---

**Note:** using this metrics requires you to have `lpips` package installed. Either install as `pip install torchmetrics[image]` or `pip install lpips`

---

---

**Note:** this metric is not scriptable when using `torch<1.8`. Please update your pytorch installation if this is a issue.

---

**Parameters**

- **net\_type** (`str`) – str indicating backbone network type to use. Choose between `'alex'`, `'vgg'` or `'squeeze'`
- **reduction** (`Literal['sum', 'mean']`) – str indicating how to reduce over the batch dimension. Choose between `'sum'` or `'mean'`.
- **normalize** (`bool`) – by default this is `False` meaning that the input is expected to be in the  $[-1, 1]$  range. If set to `True` will instead expect input to be in the  $[0, 1]$  range.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises**

- **ModuleNotFoundError** – If `lpips` package is not installed

- **ValueError** – If `net_type` is not one of "vgg", "alex" or "squeeze"
- **ValueError** – If `reduction` is not one of "mean" or "sum"

### Example

```
>>> import torch
>>> _ = torch.manual_seed(123)
>>> from torchmetrics.image.lpip import LearnedPerceptualImagePatchSimilarity
>>> lpips = LearnedPerceptualImagePatchSimilarity(net_type='vgg')
>>> img1 = torch.rand(10, 3, 100, 100)
>>> img2 = torch.rand(10, 3, 100, 100)
>>> lpips(img1, img2)
tensor(0.3566, grad_fn=<SqueezeBackward0>)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

#### `compute()`

Compute final perceptual similarity metric.

**Return type** `Tensor`

#### `update(img1, img2)`

Update internal states with lpips score.

#### Parameters

- **img1** (`Tensor`) – tensor with images of shape `[N, 3, H, W]`
- **img2** (`Tensor`) – tensor with images of shape `[N, 3, H, W]`

**Return type** `None`

## 1.51 Multi-Scale SSIM

### 1.51.1 Module Interface

```
class torchmetrics.MultiScaleStructuralSimilarityIndexMeasure(gaussian_kernel=True,
                                                              kernel_size=11, sigma=1.5,
                                                              reduction='elementwise_mean',
                                                              data_range=None, k1=0.01,
                                                              k2=0.03, betas=(0.0448, 0.2856,
                                                              0.3001, 0.2363, 0.1333),
                                                              normalize='relu', **kwargs)
```

Computes `MultiScaleSSIM`, Multi-scale Structural Similarity Index Measure, which is a generalization of Structural Similarity Index Measure by incorporating image details at different resolution scores.

#### Parameters

- **gaussian\_kernel** (`bool`) – If `True` (default), a gaussian kernel is used, if `false` a uniform kernel is used
- **kernel\_size** (`Union[int, Sequence[int]]`) – size of the gaussian kernel
- **sigma** (`Union[float, Sequence[float]]`) – Standard deviation of the gaussian kernel

- **reduction** (`Literal`['elementwise\_mean', 'sum', 'none', None]) – a method to reduce metric score over labels.
  - 'elementwise\_mean': takes the mean
  - 'sum': takes the sum
  - 'none' or None: no reduction will be applied
- **data\_range** (`Optional`[float]) – Range of the image. If None, it is determined from the image (max - min)
- **k1** (float) – Parameter of structural similarity index measure.
- **k2** (float) – Parameter of structural similarity index measure.
- **betas** (`Tuple`[float, ...]) – Exponent parameters for individual similarities and contrastive sensitivities returned by different image resolutions.
- **normalize** (`Literal`['relu', 'simple', None]) – When `MultiScaleStructuralSimilarityIndexMeasure` loss is used for training, it is desirable to use normalizes to improve the training stability. This *normalize* argument is out of scope of the original implementation [1], and it is adapted from <https://github.com/jorge-pessoa/pytorch-msssim> instead.
- **kwargs** (`Any`) – Additional keyword arguments, see *Advanced metric settings* for more info.

**Returns** Tensor with Multi-Scale SSIM score

**Raises**

- **ValueError** – If `kernel_size` is not an int or a Sequence of ints with size 2 or 3.
- **ValueError** – If `betas` is not a tuple of floats with length 2.
- **ValueError** – If `normalize` is neither *None*, *ReLU* nor *simple*.

## Example

```
>>> from torchmetrics import MultiScaleStructuralSimilarityIndexMeasure
>>> import torch
>>> preds = torch.rand([3, 3, 256, 256], generator=torch.manual_seed(42))
>>> target = preds * 0.75
>>> ms_ssim = MultiScaleStructuralSimilarityIndexMeasure(data_range=1.0)
>>> ms_ssim(preds, target)
tensor(0.9627)
```

## References

[1] Multi-Scale Structural Similarity For Image Quality Assessment by Zhou Wang, Eero P. Simoncelli and Alan C. Bovik [MultiScaleSSIM](#)

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

**compute()**

Computes MS-SSIM over state.

**Return type** `Tensor`

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

Update state with predictions and targets.

**Parameters**

- **preds** (*Tensor*) – Predictions from model of shape [N, C, H, W]
- **target** (*Tensor*) – Ground truth values of shape [N, C, H, W]

**Return type** *None*

## 1.51.2 Functional Interface

```
torchmetrics.functional.multiscale_structural_similarity_index_measure(preds, target,
                                                                    gaussian_kernel=True,
                                                                    sigma=1.5,
                                                                    kernel_size=11, reduc-
                                                                    tion='elementwise_mean',
                                                                    data_range=None,
                                                                    k1=0.01, k2=0.03,
                                                                    betas=(0.0448, 0.2856,
                                                                    0.3001, 0.2363,
                                                                    0.1333),
                                                                    normalize='relu')
```

Computes *MultiScaleSSIM*, Multi-scale Structural Similarity Index Measure, which is a generalization of Structural Similarity Index Measure by incorporating image details at different resolution scores.

**Parameters**

- **preds** (*Tensor*) – Predictions from model of shape [N, C, H, W]
- **target** (*Tensor*) – Ground truth values of shape [N, C, H, W]
- **kernel\_size** (*Union[int, Sequence[int]]*) – size of the gaussian kernel
- **sigma** (*Union[float, Sequence[float]]*) – Standard deviation of the gaussian kernel
- **reduction** (*Literal['elementwise\_mean', 'sum', 'none', None]*) – a method to reduce metric score over labels.
  - 'elementwise\_mean': takes the mean
  - 'sum': takes the sum
  - 'none' or *None*: no reduction will be applied
- **data\_range** (*Optional[float]*) – Range of the image. If *None*, it is determined from the image (max - min)
- **k1** (*float*) – Parameter of structural similarity index measure.
- **k2** (*float*) – Parameter of structural similarity index measure.
- **betas** (*Tuple[float, ...]*) – Exponent parameters for individual similarities and contrastive sensitivities returned by different image resolutions.
- **normalize** (*Optional[Literal['relu', 'simple']]*) – When *MultiScaleSSIM* loss is used for training, it is desirable to use *normalize* to improve the training stability. This *normalize* argument is out of scope of the original implementation [1], and it is adapted from <https://github.com/jorge-pessoa/pytorch-msssim> instead.

**Return type** *Tensor*

**Returns** Tensor with Multi-Scale SSIM score

**Raises**

- **TypeError** – If preds and target don't have the same data type.
- **ValueError** – If preds and target don't have BxCxHxW shape.
- **ValueError** – If the length of kernel\_size or sigma is not 2.
- **ValueError** – If one of the elements of kernel\_size is not an odd positive number.
- **ValueError** – If one of the elements of sigma is not a positive number.

### Example

```
>>> from torchmetrics.functional import multiscale_structural_similarity_index_
    ↪measure
>>> preds = torch.rand([3, 3, 256, 256], generator=torch.manual_seed(42))
>>> target = preds * 0.75
>>> multiscale_structural_similarity_index_measure(preds, target, data_range=1.0)
tensor(0.9627)
```

### References

[1] Multi-Scale Structural Similarity For Image Quality Assessment by Zhou Wang, Eero P. Simoncelli and Alan C. Bovik [MultiScaleSSIM](#)

## 1.52 Peak Signal-to-Noise Ratio (PSNR)

### 1.52.1 Module Interface

```
class torchmetrics.PeakSignalNoiseRatio(data_range=None, base=10.0, reduction='elementwise_mean',
                                         dim=None, **kwargs)
```

Computes Computes Peak Signal-to-Noise Ratio (PSNR):

$$\text{PSNR}(I, J) = 10 * \log_{10} \left( \frac{\max(I)^2}{\text{MSE}(I, J)} \right)$$

Where MSE denotes the [mean-squared-error](#) function.

#### Parameters

- **data\_range** ([Optional\[float\]](#)) – the range of the data. If None, it is determined from the data (max - min). The data\_range must be given when dim is not None.
- **base** ([float](#)) – a base of a logarithm to use.
- **reduction** ([Literal](#)['elementwise\_mean', 'sum', 'none', None]) – a method to reduce metric score over labels.
  - 'elementwise\_mean': takes the mean (default)
  - 'sum': takes the sum
  - 'none' or None: no reduction will be applied



- **dim** (`Union[int, Tuple[int, ...], None]`) – Dimensions to reduce PSNR scores over, provided as either an integer or a list of integers. Default is `None` meaning scores will be reduced across all dimensions and all batches.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises** `ValueError` – If `dim` is not `None` and `data_range` is not given.

### Example

```
>>> from torchmetrics import PeakSignalNoiseRatio
>>> psnr = PeakSignalNoiseRatio()
>>> preds = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
>>> target = torch.tensor([[3.0, 2.0], [1.0, 0.0]])
>>> psnr(preds, target)
tensor(2.5527)
```

---

**Note:** Half precision is only support on GPU for this metric

---

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

**compute()**

Compute peak signal-to-noise ratio over state.

**Return type** `Tensor`

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

## 1.52.2 Functional Interface

`torchmetrics.functional.peak_signal_noise_ratio(preds, target, data_range=None, base=10.0, reduction='elementwise_mean', dim=None)`

Computes the peak signal-to-noise ratio.

**Parameters**

- **preds** (`Tensor`) – estimated signal
- **target** (`Tensor`) – ground truth signal
- **data\_range** (`Optional[float]`) – the range of the data. If `None`, it is determined from the data (`max - min`). `data_range` must be given when `dim` is not `None`.
- **base** (`float`) – a base of a logarithm to use
- **reduction** (`Literal['elementwise_mean', 'sum', 'none', None]`) – a method to reduce metric score over labels.
  - `'elementwise_mean'`: takes the mean (default)

- 'sum': takes the sum
- 'none' or None: no reduction will be applied
- **dim** (`Union[int, Tuple[int, ...], None]`) – Dimensions to reduce PSNR scores over provided as either an integer or a list of integers. Default is None meaning scores will be reduced across all dimensions.

**Return type** `Tensor`

**Returns** Tensor with PSNR score

**Raises** `ValueError` – If dim is not None and data\_range is not provided.

### Example

```
>>> from torchmetrics.functional import peak_signal_noise_ratio
>>> pred = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
>>> target = torch.tensor([[3.0, 2.0], [1.0, 0.0]])
>>> peak_signal_noise_ratio(pred, target)
tensor(2.5527)
```

---

**Note:** Half precision is only support on GPU for this metric

---

## 1.53 Spectral Angle Mapper

### 1.53.1 Module Interface

**class** `torchmetrics.SpectralAngleMapper`(*reduction='elementwise\_mean', \*\*kwargs*)

The Spectral Angle Mapper determines the spectral similarity between image spectra and reference spectra by calculating the angle between the spectra, where small angles between indicate high similarity and high angles indicate low similarity.

#### Parameters

- **reduction** (`Literal['elementwise_mean', 'sum', 'none']`) – a method to reduce metric score over labels.
  - 'elementwise\_mean': takes the mean (default)
  - 'sum': takes the sum
  - 'none' or None: no reduction will be applied
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Returns** Tensor with SpectralAngleMapper score

## Example

```
>>> import torch
>>> from torchmetrics import SpectralAngleMapper
>>> preds = torch.rand([16, 3, 16, 16], generator=torch.manual_seed(42))
>>> target = torch.rand([16, 3, 16, 16], generator=torch.manual_seed(123))
>>> sam = SpectralAngleMapper()
>>> sam(preds, target)
tensor(0.5943)
```

## References

[1] Roberta H. Yuhas, Alexander F. H. Goetz and Joe W. Boardman, “Discrimination among semi-arid landscape endmembers using the Spectral Angle Mapper (SAM) algorithm” in PL, Summaries of the Third Annual JPL Airborne Geoscience Workshop, vol. 1, June 1, 1992.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes spectra over state.

**Return type** `Tensor`

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

## 1.53.2 Functional Interface

`torchmetrics.functional.spectral_angle_mapper(preds, target, reduction='elementwise_mean')`

Universal Spectral Angle Mapper.

**Parameters**

- **preds** (`Tensor`) – estimated image
- **target** (`Tensor`) – ground truth image
- **reduction** (`Literal`[‘elementwise\_mean’, ‘sum’, ‘none’, `None`]) – a method to reduce metric score over labels.
  - ‘elementwise\_mean’: takes the mean (default)
  - ‘sum’: takes the sum
  - ‘none’ or `None`: no reduction will be applied

**Return type** `Tensor`

**Returns** Tensor with Spectral Angle Mapper score

**Raises**

- **`TypeError`** – If preds and target don't have the same data type.
- **`ValueError`** – If preds and target don't have BxCxHxW shape.

### Example

```
>>> from torchmetrics.functional import spectral_angle_mapper
>>> preds = torch.rand([16, 3, 16, 16], generator=torch.manual_seed(42))
>>> target = torch.rand([16, 3, 16, 16], generator=torch.manual_seed(123))
>>> spectral_angle_mapper(preds, target)
tensor(0.5943)
```

### References

[1] Roberta H. Yuhas, Alexander F. H. Goetz and Joe W. Boardman, “Discrimination among semi-arid landscape endmembers using the Spectral Angle Mapper (SAM) algorithm” in PL, Summaries of the Third Annual JPL Airborne Geoscience Workshop, vol. 1, June 1, 1992.

## 1.54 Spectral Distortion Index

### 1.54.1 Module Interface

**class** `torchmetrics.SpectralDistortionIndex`(*p=1*, *reduction='elementwise\_mean'*, *\*\*kwargs*)

Computes Spectral Distortion Index (`SpectralDistortionIndex`) also now as `D_lambda` is used to compare the spectral distortion between two images.

#### Parameters

- **`p`** (`int`) – Large spectral differences
- **`reduction`** (`Literal`[`'elementwise_mean'`, `'sum'`, `'none'`]) – a method to reduce metric score over labels.
  - `'elementwise_mean'`: takes the mean (default)
  - `'sum'`: takes the sum
  - `'none'`: no reduction will be applied
- **`kwargs`** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> import torch
>>> _ = torch.manual_seed(42)
>>> from torchmetrics import SpectralDistortionIndex
>>> preds = torch.rand([16, 3, 16, 16])
>>> target = torch.rand([16, 3, 16, 16])
>>> sdi = SpectralDistortionIndex()
>>> sdi(preds, target)
tensor(0.0234)
```

## References

[1] Alparone, Luciano & Aiazzi, Bruno & Baronti, Stefano & Garzelli, Andrea & Nencini, Filippo & Selva, Massimo. (2008). Multispectral and Panchromatic Data Fusion Assessment Without Reference. ASPRS Journal of Photogrammetric Engineering and Remote Sensing. 74. 193-200. 10.14358/PERS.74.2.193.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes and returns spectral distortion index.

**Return type** `Tensor`

**update(preds, target)**

Update state with preds and target.

**Parameters**

- **preds** (`Tensor`) – Low resolution multispectral image
- **target** (`Tensor`) – High resolution fused image

**Return type** `None`

## 1.54.2 Functional Interface

`torchmetrics.functional.spectral_distortion_index(preds, target, p=1, reduction='elementwise_mean')`

Spectral Distortion Index (`SpectralDistortionIndex`) also now as `D_lambda` is used to compare the spectral distortion between two images.

**Parameters**

- **preds** (`Tensor`) – Low resolution multispectral image
- **target** (`Tensor`) – High resolution fused image
- **p** (`int`) – Large spectral differences
- **reduction** (`Literal`['elementwise\_mean', 'sum', 'none']) – a method to reduce metric score over labels.
  - 'elementwise\_mean': takes the mean (default)
  - 'sum': takes the sum
  - 'none': no reduction will be applied

**Return type** `Tensor`

**Returns** Tensor with `SpectralDistortionIndex` score

**Raises**

- **TypeError** – If `preds` and `target` don't have the same data type.
- **ValueError** – If `preds` and `target` don't have `BxCxHxW` shape.
- **ValueError** – If `p` is not a positive integer.

### Example

```
>>> from torchmetrics.functional import spectral_distortion_index
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand([16, 3, 16, 16])
>>> target = torch.rand([16, 3, 16, 16])
>>> spectral_distortion_index(preds, target)
tensor(0.0234)
```

## 1.55 Structural Similarity Index Measure (SSIM)

### 1.55.1 Module Interface

```
class torchmetrics.StructuralSimilarityIndexMeasure(gaussian_kernel=True, sigma=1.5,
                                                    kernel_size=11, reduction='elementwise_mean',
                                                    data_range=None, k1=0.01, k2=0.03,
                                                    return_full_image=False,
                                                    return_contrast_sensitivity=False, **kwargs)
```

Computes Structural Similarity Index Measure (SSIM).

#### Parameters

- **preds** – estimated image
- **target** – ground truth image
- **gaussian\_kernel** (*bool*) – If True (default), a gaussian kernel is used, if False a uniform kernel is used
- **sigma** (*Union[float, Sequence[float]]*) – Standard deviation of the gaussian kernel, anisotropic kernels are possible. Ignored if a uniform kernel is used
- **kernel\_size** (*Union[int, Sequence[int]]*) – the size of the uniform kernel, anisotropic kernels are possible. Ignored if a Gaussian kernel is used
- **reduction** (*Literal['elementwise\_mean', 'sum', 'none', None]*) – a method to reduce metric score over individual batch scores
  - 'elementwise\_mean': takes the mean
  - 'sum': takes the sum
  - 'none' or None: no reduction will be applied
- **data\_range** (*Optional[float]*) – Range of the image. If None, it is determined from the image (max - min)
- **k1** (*float*) – Parameter of SSIM.
- **k2** (*float*) – Parameter of SSIM.
- **return\_full\_image** (*bool*) – If true, the full ssim image is returned as a second argument. Mutually exclusive with `return_contrast_sensitivity`
- **return\_contrast\_sensitivity** (*bool*) – If true, the constant term is returned as a second argument. The luminance term can be obtained with `luminance=ssim/contrast` Mutually exclusive with `return_full_image`
- **kwargs** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Returns** Tensor with SSIM score

### Example

```
>>> from torchmetrics import StructuralSimilarityIndexMeasure
>>> import torch
>>> preds = torch.rand([3, 3, 256, 256])
>>> target = preds * 0.75
>>> ssim = StructuralSimilarityIndexMeasure(data_range=1.0)
>>> ssim(preds, target)
tensor(0.9219)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes SSIM over state.

**Return type** Union[Tensor, Tuple[Tensor, Tensor]]

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (Tensor) – Predictions from model
- **target** (Tensor) – Ground truth values

**Return type** None

## 1.55.2 Functional Interface

`torchmetrics.functional.structural_similarity_index_measure(preds, target, gaussian_kernel=True, sigma=1.5, kernel_size=11, reduction='elementwise_mean', data_range=None, k1=0.01, k2=0.03, return_full_image=False, return_contrast_sensitivity=False)`

Computes Structural Similarity Index Measure.

**Parameters**

- **preds** (Tensor) – estimated image
- **target** (Tensor) – ground truth image
- **gaussian\_kernel** (bool) – If true (default), a gaussian kernel is used, if false a uniform kernel is used
- **sigma** (Union[float, Sequence[float]]) – Standard deviation of the gaussian kernel, anisotropic kernels are possible. Ignored if a uniform kernel is used
- **kernel\_size** (Union[int, Sequence[int]]) – the size of the uniform kernel, anisotropic kernels are possible. Ignored if a Gaussian kernel is used
- **reduction** (Literal['elementwise\_mean', 'sum', 'none', None]) – a method to reduce metric score over labels.
  - 'elementwise\_mean': takes the mean

- 'sum': takes the sum
- 'none' or None: no reduction will be applied
- **data\_range** (Optional[float]) – Range of the image. If None, it is determined from the image (max - min)
- **k1** (float) – Parameter of SSIM.
- **k2** (float) – Parameter of SSIM.
- **return\_full\_image** (bool) – If true, the full ssim image is returned as a second argument. Mutually exclusive with `return_contrast_sensitivity`
- **return\_contrast\_sensitivity** (bool) – If true, the constant term is returned as a second argument. The luminance term can be obtained with `luminance=ssim/contrast` Mutually exclusive with `return_full_image`

**Return type** Union[Tensor, Tuple[Tensor, Tensor]]

**Returns** Tensor with SSIM score

**Raises**

- **TypeError** – If preds and target don't have the same data type.
- **ValueError** – If preds and target don't have BxCxHxW shape.
- **ValueError** – If the length of kernel\_size or sigma is not 2.
- **ValueError** – If one of the elements of kernel\_size is not an odd positive number.
- **ValueError** – If one of the elements of sigma is not a positive number.

### Example

```
>>> from torchmetrics.functional import structural_similarity_index_measure
>>> preds = torch.rand([3, 3, 256, 256])
>>> target = preds * 0.75
>>> structural_similarity_index_measure(preds, target)
tensor(0.9219)
```

## 1.56 Universal Image Quality Index

### 1.56.1 Module Interface

```
class torchmetrics.UniversalImageQualityIndex(kernel_size=(11, 11), sigma=(1.5, 1.5),
                                              reduction='elementwise_mean', data_range=None,
                                              **kwargs)
```

Computes Universal Image Quality Index (UniversalImageQualityIndex).

**Parameters**

- **kernel\_size** (Sequence[int]) – size of the gaussian kernel
- **sigma** (Sequence[float]) – Standard deviation of the gaussian kernel
- **reduction** (Literal['elementwise\_mean', 'sum', 'none', None]) – a method to reduce metric score over labels.



- 'elementwise\_mean': takes the mean (default)
- 'sum': takes the sum
- 'none' or None: no reduction will be applied
- **data\_range** (Optional[float]) – Range of the image. If None, it is determined from the image (max - min)
- **kwargs** (Any) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Returns** Tensor with UniversalImageQualityIndex score

### Example

```
>>> import torch
>>> from torchmetrics import UniversalImageQualityIndex
>>> preds = torch.rand([16, 1, 16, 16])
>>> target = preds * 0.75
>>> uqi = UniversalImageQualityIndex()
>>> uqi(preds, target)
tensor(0.9216)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes explained variance over state.

**Return type** Tensor

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (Tensor) – Predictions from model
- **target** (Tensor) – Ground truth values

**Return type** None

## 1.56.2 Functional Interface

```
torchmetrics.functional.universal_image_quality_index(preds, target, kernel_size=(11, 11),
                                                       sigma=(1.5, 1.5),
                                                       reduction='elementwise_mean',
                                                       data_range=None)
```

Universal Image Quality Index.

**Parameters**

- **preds** (Tensor) – estimated image
- **target** (Tensor) – ground truth image
- **kernel\_size** (Sequence[int]) – size of the gaussian kernel
- **sigma** (Sequence[float]) – Standard deviation of the gaussian kernel

- **reduction** (`Optional[Literal['elementwise_mean', 'sum', 'none']]`) – a method to reduce metric score over labels.
  - 'elementwise\_mean': takes the mean (default)
  - 'sum': takes the sum
  - 'none' or `None`: no reduction will be applied
- **data\_range** (`Optional[float]`) – Range of the image. If `None`, it is determined from the image (max - min)

**Return type** `Tensor`

**Returns** Tensor with UniversalImageQualityIndex score

**Raises**

- **TypeError** – If preds and target don't have the same data type.
- **ValueError** – If preds and target don't have BxCxHxW shape.
- **ValueError** – If the length of kernel\_size or sigma is not 2.
- **ValueError** – If one of the elements of kernel\_size is not an odd positive number.
- **ValueError** – If one of the elements of sigma is not a positive number.

### Example

```
>>> from torchmetrics.functional import universal_image_quality_index
>>> preds = torch.rand([16, 1, 16, 16])
>>> target = preds * 0.75
>>> universal_image_quality_index(preds, target)
tensor(0.9216)
```

### References

- [1] Zhou Wang and A. C. Bovik, "A universal image quality index," in IEEE Signal Processing Letters, vol. 9, no. 3, pp. 81-84, March 2002, doi: 10.1109/97.995823.
- [2] Zhou Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," in IEEE Transactions on Image Processing, vol. 13, no. 4, pp. 600-612, April 2004, doi: 10.1109/TIP.2003.819861.

## 1.57 Mean-Average-Precision (mAP)

### 1.57.1 Module Interface

```
class torchmetrics.detection.mean_ap.MeanAveragePrecision(box_format='xyxy', iou_type='bbox',
                                                         iou_thresholds=None,
                                                         rec_thresholds=None,
                                                         max_detection_thresholds=None,
                                                         class_metrics=False, **kwargs)
```

Computes the [Mean-Average-Precision \(mAP\)](#) and [Mean-Average-Recall \(mAR\)](#) for object detection predictions. Optionally, the mAP and mAR values can be calculated per class.

Predicted boxes and targets have to be in Pascal VOC format (xmin-top left, ymin-top left, xmax-bottom right, ymax-bottom right). See the `update()` method for more information about the input format to this metric.

For an example on how to use this metric check the [torchmetrics examples](#)

---

**Note:** This metric is following the mAP implementation of [pycocotools](#), a standard implementation for the mAP metric for object detection.

---



---

**Note:** This metric requires you to have *torchvision* version 0.8.0 or newer installed (with corresponding version 1.7.0 of torch or newer). This metric requires *pycocotools* installed when `iou_type` is *segm*. Please install with `pip install torchvision` or `pip install torchmetrics[detection]`.

---

### Parameters

- **box\_format** (`str`) – Input format of given boxes. Supported formats are [``xyxy``, ``xywh``, ``cxcywh``].
- **iou\_type** (`str`) – Type of input (either masks or bounding-boxes) used for computing IOU. Supported IOU types are [`"bbox"`, `"segm"`]. If using `"segm"`, masks should be provided (see `update()`).
- **iou\_thresholds** (`Optional[List[float]]`) – IoU thresholds for evaluation. If set to `None` it corresponds to the stepped range `[0.5, ..., 0.95]` with step `0.05`. Else provide a list of floats.
- **rec\_thresholds** (`Optional[List[float]]`) – Recall thresholds for evaluation. If set to `None` it corresponds to the stepped range `[0, ..., 1]` with step `0.01`. Else provide a list of floats.
- **max\_detection\_thresholds** (`Optional[List[int]]`) – Thresholds on max detections per image. If set to `None` will use thresholds `[1, 10, 100]`. Else, please provide a list of ints.
- **class\_metrics** (`bool`) – Option to enable per-class metrics for mAP and mAR\_100. Has a performance impact.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> import torch
>>> from torchmetrics.detection.mean_ap import MeanAveragePrecision
>>> preds = [
...     dict(
...         boxes=torch.tensor([[258.0, 41.0, 606.0, 285.0]]),
...         scores=torch.tensor([0.536]),
...         labels=torch.tensor([0]),
...     )
... ]
>>> target = [
```

(continues on next page)

(continued from previous page)

```

...     dict(
...         boxes=torch.tensor([[214.0, 41.0, 562.0, 285.0]]),
...         labels=torch.tensor([0]),
...     )
... ]
>>> metric = MeanAveragePrecision()
>>> metric.update(preds, target)
>>> from pprint import pprint
>>> pprint(metric.compute())
{'map': tensor(0.6000),
 'map_50': tensor(1.),
 'map_75': tensor(1.),
 'map_large': tensor(0.6000),
 'map_medium': tensor(-1.),
 'map_per_class': tensor(-1.),
 'map_small': tensor(-1.),
 'mar_1': tensor(0.6000),
 'mar_10': tensor(0.6000),
 'mar_100': tensor(0.6000),
 'mar_100_per_class': tensor(-1.),
 'mar_large': tensor(0.6000),
 'mar_medium': tensor(-1.),
 'mar_small': tensor(-1.)}

```

**Raises**

- **ModuleNotFoundError** – If torchvision is not installed or version installed is lower than 0.8.0
- **ModuleNotFoundError** – If iou\_type is equal to seqm and pycocotools is not installed
- **ValueError** – If class\_metrics is not a boolean

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Compute the *Mean-Average-Precision (mAP)* and *Mean-Average-Recall (mAR)* scores.

---

**Note:** map score is calculated with @[ IoU=self.iou\_thresholds | area=all | max\_dets=max\_detection\_thresholds ]

---

Caution: If the initialization parameters are changed, dictionary keys for mAR can change as well. The default properties are also accessible via fields and will raise an **AttributeError** if not available.

---

**Return type** dict**Returns**

dict containing

- map: torch.Tensor
- map\_small: torch.Tensor
- map\_medium: torch.Tensor

- `map_large`: `torch.Tensor`
- `mar_1`: `torch.Tensor`
- `mar_10`: `torch.Tensor`
- `mar_100`: `torch.Tensor`
- `mar_small`: `torch.Tensor`
- `mar_medium`: `torch.Tensor`
- `mar_large`: `torch.Tensor`
- `map_50`: `torch.Tensor` (-1 if 0.5 not in the list of iou thresholds)
- `map_75`: `torch.Tensor` (-1 if 0.75 not in the list of iou thresholds)
- `map_per_class`: `torch.Tensor` (-1 if class metrics are disabled)
- `mar_100_per_class`: `torch.Tensor` (-1 if class metrics are disabled)

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

Add detections and ground truth to the metric.

#### Parameters

- **preds** (`List[Dict[str, Tensor]]`) – A list consisting of dictionaries each containing the key-values (each dictionary corresponds to a single image):
  - `boxes`: `torch.FloatTensor` of shape `[num_boxes, 4]` containing `num_boxes` detection boxes of the format specified in the constructor. By default, this method expects `[xmin, ymin, xmax, ymax]` in absolute image coordinates.
  - `scores`: `torch.FloatTensor` of shape `[num_boxes]` containing detection scores for the boxes.
  - `labels`: `torch.IntTensor` of shape `[num_boxes]` containing 0-indexed detection classes for the boxes.
  - `masks`: `torch.bool` of shape `[num_boxes, image_height, image_width]` containing boolean masks. Only required when `iou_type="segm"`.
- **target** (`List[Dict[str, Tensor]]`) – A list consisting of dictionaries each containing the key-values (each dictionary corresponds to a single image):
  - `boxes`: `torch.FloatTensor` of shape `[num_boxes, 4]` containing `num_boxes` ground truth boxes of the format specified in the constructor. By default, this method expects `[xmin, ymin, xmax, ymax]` in absolute image coordinates.
  - **`labels`: `torch.IntTensor` of shape `[num_boxes]` containing 0-indexed ground truth classes for the boxes.**
  - `masks`: `torch.bool` of shape `[num_boxes, image_height, image_width]` containing boolean masks. Only required when `iou_type="segm"`.

#### Raises

- **ValueError** – If `preds` is not of type `List[Dict[str, Tensor]]`
- **ValueError** – If `target` is not of type `List[Dict[str, Tensor]]`
- **ValueError** – If `preds` and `target` are not of the same length
- **ValueError** – If any of `preds.boxes`, `preds.scores` and `preds.labels` are not of the same length

- **ValueError** – If any of `target.bboxes` and `target.labels` are not of the same length
- **ValueError** – If any box is not type float and of length 4
- **ValueError** – If any class is not type int and of length 1
- **ValueError** – If any score is not type float and of length 1

Return type `None`

## 1.58 Cosine Similarity

### 1.58.1 Functional Interface

`torchmetrics.functional.pairwise_cosine_similarity(x, y=None, reduction=None, zero_diagonal=None)`

Calculates pairwise cosine similarity:

$$s_{cos}(x, y) = \frac{\langle x, y \rangle}{\|x\| \cdot \|y\|} = \frac{\sum_{d=1}^D x_d \cdot y_d}{\sqrt{\sum_{d=1}^D x_d^2} \cdot \sqrt{\sum_{d=1}^D y_d^2}}$$

If both  $x$  and  $y$  are passed in, the calculation will be performed pairwise between the rows of  $x$  and  $y$ . If only  $x$  is passed in, the calculation will be performed between the rows of  $x$ .

#### Parameters

- **x** (`Tensor`) – Tensor with shape  $[N, d]$
- **y** (`Optional[Tensor]`) – Tensor with shape  $[M, d]$ , optional
- **reduction** (`Optional[Literal['mean', 'sum', 'none', None]]`) – reduction to apply along the last dimension. Choose between `'mean'`, `'sum'` (applied along column dimension) or `'none'`, `None` for no reduction
- **zero\_diagonal** (`Optional[bool]`) – if the diagonal of the distance matrix should be set to 0. If only  $x$  is given this defaults to `True` else if  $y$  is also given it defaults to `False`

Return type `Tensor`

Returns A  $[N, N]$  matrix of distances if only  $x$  is given, else a  $[N, M]$  matrix

#### Example

```
>>> import torch
>>> from torchmetrics.functional import pairwise_cosine_similarity
>>> x = torch.tensor([[2, 3], [3, 5], [5, 8]], dtype=torch.float32)
>>> y = torch.tensor([[1, 0], [2, 1]], dtype=torch.float32)
>>> pairwise_cosine_similarity(x, y)
tensor([[0.5547, 0.8682],
        [0.5145, 0.8437],
        [0.5300, 0.8533]])
>>> pairwise_cosine_similarity(x)
tensor([[0.0000, 0.9989, 0.9996],
        [0.9989, 0.0000, 0.9998],
        [0.9996, 0.9998, 0.0000]])
```

## 1.59 Euclidean Distance

### 1.59.1 Functional Interface

`torchmetrics.functional.pairwise_euclidean_distance(x, y=None, reduction=None, zero_diagonal=None)`

Calculates pairwise euclidean distances:

$$d_{\text{euc}}(x, y) = \|x - y\|_2 = \sqrt{\sum_{d=1}^D (x_d - y_d)^2}$$

If both  $x$  and  $y$  are passed in, the calculation will be performed pairwise between the rows of  $x$  and  $y$ . If only  $x$  is passed in, the calculation will be performed between the rows of  $x$ .

#### Parameters

- **x** (`Tensor`) – Tensor with shape  $[N, d]$
- **y** (`Optional[Tensor]`) – Tensor with shape  $[M, d]$ , optional
- **reduction** (`Optional[Literal['mean', 'sum', 'none', None]]`) – reduction to apply along the last dimension. Choose between `'mean'`, `'sum'` (applied along column dimension) or `'none'`, `None` for no reduction
- **zero\_diagonal** (`Optional[bool]`) – if the diagonal of the distance matrix should be set to 0. If only  $x$  is given this defaults to `True` else if  $y$  is also given it defaults to `False`

**Return type** `Tensor`

**Returns** A  $[N, N]$  matrix of distances if only  $x$  is given, else a  $[N, M]$  matrix

#### Example

```
>>> import torch
>>> from torchmetrics.functional import pairwise_euclidean_distance
>>> x = torch.tensor([[2, 3], [3, 5], [5, 8]], dtype=torch.float32)
>>> y = torch.tensor([[1, 0], [2, 1]], dtype=torch.float32)
>>> pairwise_euclidean_distance(x, y)
tensor([[3.1623, 2.0000],
        [5.3852, 4.1231],
        [8.9443, 7.6158]])
>>> pairwise_euclidean_distance(x)
tensor([[0.0000, 2.2361, 5.8310],
        [2.2361, 0.0000, 3.6056],
        [5.8310, 3.6056, 0.0000]])
```

## 1.60 Linear Similarity

### 1.60.1 Functional Interface

`torchmetrics.functional.pairwise_linear_similarity(x, y=None, reduction=None, zero_diagonal=None)`

Calculates pairwise linear similarity:

$$s_{lin}(x, y) = \langle x, y \rangle = \sum_{d=1}^D x_d \cdot y_d$$

If both  $x$  and  $y$  are passed in, the calculation will be performed pairwise between the rows of  $x$  and  $y$ . If only  $x$  is passed in, the calculation will be performed between the rows of  $x$ .

#### Parameters

- **x** (`Tensor`) – Tensor with shape  $[N, d]$
- **y** (`Optional[Tensor]`) – Tensor with shape  $[M, d]$ , optional
- **reduction** (`Optional[Literal['mean', 'sum', 'none', None]]`) – reduction to apply along the last dimension. Choose between `'mean'`, `'sum'` (applied along column dimension) or `'none'`, `None` for no reduction
- **zero\_diagonal** (`Optional[bool]`) – if the diagonal of the distance matrix should be set to 0. If only  $x$  is given this defaults to `True` else if  $y$  is also given it defaults to `False`

**Return type** `Tensor`

**Returns** A  $[N, N]$  matrix of distances if only  $x$  is given, else a  $[N, M]$  matrix

#### Example

```
>>> import torch
>>> from torchmetrics.functional import pairwise_linear_similarity
>>> x = torch.tensor([[2, 3], [3, 5], [5, 8]], dtype=torch.float32)
>>> y = torch.tensor([[1, 0], [2, 1]], dtype=torch.float32)
>>> pairwise_linear_similarity(x, y)
tensor([[ 2.,  7.],
        [ 3., 11.],
        [ 5., 18.]])
>>> pairwise_linear_similarity(x)
tensor([[ 0., 21., 34.],
        [21.,  0., 55.],
        [34., 55.,  0.]])
```



## 1.61 Manhattan Distance

### 1.61.1 Functional Interface

`torchmetrics.functional.pairwise_manhattan_distance(x, y=None, reduction=None, zero_diagonal=None)`

Calculates pairwise manhattan distance:

$$d_{man}(x, y) = ||x - y||_1 = \sum_{d=1}^D |x_d - y_d|$$

If both  $x$  and  $y$  are passed in, the calculation will be performed pairwise between the rows of  $x$  and  $y$ . If only  $x$  is passed in, the calculation will be performed between the rows of  $x$ .

#### Parameters

- **x** (`Tensor`) – Tensor with shape  $[N, d]$
- **y** (`Optional[Tensor]`) – Tensor with shape  $[M, d]$ , optional
- **reduction** (`Optional[Literal['mean', 'sum', 'none', None]]`) – reduction to apply along the last dimension. Choose between `'mean'`, `'sum'` (applied along column dimension) or `'none'`, `None` for no reduction
- **zero\_diagonal** (`Optional[bool]`) – if the diagonal of the distance matrix should be set to 0. If only  $x$  is given this defaults to `True` else if  $y$  is also given it defaults to `False`

**Return type** `Tensor`

**Returns** A  $[N, N]$  matrix of distances if only  $x$  is given, else a  $[N, M]$  matrix

#### Example

```
>>> import torch
>>> from torchmetrics.functional import pairwise_manhattan_distance
>>> x = torch.tensor([[2, 3], [3, 5], [5, 8]], dtype=torch.float32)
>>> y = torch.tensor([[1, 0], [2, 1]], dtype=torch.float32)
>>> pairwise_manhattan_distance(x, y)
tensor([[ 4.,  2.],
        [ 7.,  5.],
        [12., 10.]])
>>> pairwise_manhattan_distance(x)
tensor([[0., 3., 8.],
        [3., 0., 5.],
        [8., 5., 0.]])
```

## 1.62 Concordance Corr. Coef.

### 1.62.1 Module Interface

**class** torchmetrics.ConcordanceCorrCoef(num\_outputs=1, \*\*kwargs)

Computes concordance correlation coefficient that measures the agreement between two variables. It is defined as

$$\rho_c = \frac{2\rho\sigma_x\sigma_y}{\sigma_x^2 + \sigma_y^2 + (\mu_x - \mu_y)^2}$$

where  $\mu_x, \mu_y$  is the means for the two variables,  $\sigma_x^2, \sigma_y^2$  are the corresponding variances and rho is the pearson correlation coefficient between the two variables.

Forward accepts - preds (float tensor): either single output tensor with shape (N,) or multioutput tensor of shape (N,d) - target (float tensor): either single output tensor with shape (N,) or multioutput tensor of shape (N,d)

#### Parameters

- **num\_outputs** (int) – Number of outputs in multioutput setting
- **kwargs** (Any) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (single output regression):**

```
>>> from torchmetrics import ConcordanceCorrCoef
>>> import torch
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> concordance = ConcordanceCorrCoef()
>>> concordance(preds, target)
tensor(0.9777)
```

**Example (multi output regression):**

```
>>> from torchmetrics import ConcordanceCorrCoef
>>> import torch
>>> target = torch.tensor([[3, -0.5], [2, 7]])
>>> preds = torch.tensor([[2.5, 0.0], [2, 8]])
>>> concordance = ConcordanceCorrCoef(num_outputs=2)
>>> concordance(preds, target)
tensor([0.7273, 0.9887])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes final concordance correlation coefficient over metric states.

**Return type** Tensor

## 1.62.2 Functional Interface

`torchmetrics.functional.concordance_corrcoef(preds, target)`

Computes concordance correlation coefficient that measures the agreement between two variables. It is defined as

$$\rho_c = \frac{2\rho\sigma_x\sigma_y}{\sigma_x^2 + \sigma_y^2 + (\mu_x - \mu_y)^2}$$

where  $\mu_x, \mu_y$  is the means for the two variables,  $\sigma_x^2, \sigma_y^2$  are the corresponding variances and rho is the pearson correlation coefficient between the two variables.

### Parameters

- **preds** (`Tensor`) – estimated scores
- **target** (`Tensor`) – ground truth scores

**Example (single output regression):**

```
>>> from torchmetrics.functional import concordance_corrcoef
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> concordance_corrcoef(preds, target)
tensor([0.9777])
```

**Example (multi output regression):**

```
>>> from torchmetrics.functional import concordance_corrcoef
>>> target = torch.tensor([[3, -0.5], [2, 7]])
>>> preds = torch.tensor([[2.5, 0.0], [2, 8]])
>>> concordance_corrcoef(preds, target)
tensor([0.7273, 0.9887])
```

**Return type** `Tensor`

## 1.63 Cosine Similarity

### 1.63.1 Module Interface

`class torchmetrics.CosineSimilarity(reduction='sum', **kwargs)`

Computes the *Cosine Similarity* between targets and predictions:

$$\text{cos}_{sim}(x, y) = \frac{x \cdot y}{\|x\| \cdot \|y\|} = \frac{\sum_{i=1}^n x_i y_i}{\sqrt{\sum_{i=1}^n x_i^2} \sqrt{\sum_{i=1}^n y_i^2}}$$

where  $y$  is a tensor of target values, and  $x$  is a tensor of predictions.

Forward accepts

- **preds** (float tensor): (N, d)
- **target** (float tensor): (N, d)

### Parameters

- **reduction** (`Literal`['mean', 'sum', 'none', None]) – how to reduce over the batch dimension using 'sum', 'mean' or 'none' (taking the individual scores)
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torchmetrics import CosineSimilarity
>>> target = torch.tensor([[0, 1], [1, 1]])
>>> preds = torch.tensor([[0, 1], [0, 1]])
>>> cosine_similarity = CosineSimilarity(reduction = 'mean')
>>> cosine_similarity(preds, target)
tensor(0.8536)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

#### `compute()`

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tensor`

#### `update(preds, target)`

Update metric states with predictions and targets.

##### Parameters

- **preds** (`Tensor`) – Predicted tensor with shape (N, d)
- **target** (`Tensor`) – Ground truth tensor with shape (N, d)

**Return type** `None`

## 1.63.2 Functional Interface

`torchmetrics.functional.cosine_similarity(preds, target, reduction='sum')`

Computes the *Cosine Similarity* between targets and predictions:

$$\text{cos}_{sim}(x, y) = \frac{x \cdot y}{\|x\| \cdot \|y\|} = \frac{\sum_{i=1}^n x_i y_i}{\sqrt{\sum_{i=1}^n x_i^2} \sqrt{\sum_{i=1}^n y_i^2}}$$

where  $y$  is a tensor of target values, and  $x$  is a tensor of predictions.

##### Parameters

- **preds** (`Tensor`) – Predicted tensor with shape (N, d)
- **target** (`Tensor`) – Ground truth tensor with shape (N, d)
- **reduction** (`Optional`[`str`]) – The method of reducing along the batch dimension using sum, mean or taking the individual scores

### Example

```
>>> from torchmetrics.functional.regression import cosine_similarity
>>> target = torch.tensor([[1, 2, 3, 4],
...                        [1, 2, 3, 4]])
>>> preds = torch.tensor([[1, 2, 3, 4],
...                       [-1, -2, -3, -4]])
>>> cosine_similarity(preds, target, 'none')
tensor([ 1.0000, -1.0000])
```

Return type `Tensor`

## 1.64 Explained Variance

### 1.64.1 Module Interface

**class** `torchmetrics.ExplainedVariance`(*multioutput*='uniform\_average', *\*\*kwargs*)

Computes *explained variance*:

$$\text{ExplainedVariance} = 1 - \frac{\text{Var}(y - \hat{y})}{\text{Var}(y)}$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

Forward accepts

- `preds` (float tensor): (N,) or (N, ...) (multioutput)
- `target` (long tensor): (N,) or (N, ...) (multioutput)

In the case of multioutput, as default the variances will be uniformly averaged over the additional dimensions. Please see argument `multioutput` for changing this behavior.

#### Parameters

- **`multioutput`** (`str`) – Defines aggregation in the case of multiple output scores. Can be one of the following strings (default is 'uniform\_average'.):
  - 'raw\_values' returns full set of scores
  - 'uniform\_average' scores are uniformly averaged
  - 'variance\_weighted' scores are weighted by their individual variances
- **`kwargs`** (`Any`) – Additional keyword arguments, see *Advanced metric settings* for more info.

**Raises** `ValueError` – If `multioutput` is not one of "raw\_values", "uniform\_average" or "variance\_weighted".

### Example

```
>>> from torchmetrics import ExplainedVariance
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> explained_variance = ExplainedVariance()
>>> explained_variance(preds, target)
tensor(0.9572)
```

```
>>> target = torch.tensor([[0.5, 1], [-1, 1], [7, -6]])
>>> preds = torch.tensor([[0, 2], [-1, 2], [8, -5]])
>>> explained_variance = ExplainedVariance(multioutput='raw_values')
>>> explained_variance(preds, target)
tensor([0.9677, 1.0000])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes explained variance over state.

**Return type** `Union[Tensor, Sequence[Tensor]]`

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

## 1.64.2 Functional Interface

`torchmetrics.functional.explained_variance(preds, target, multioutput='uniform_average')`

Computes explained variance.

**Parameters**

- **preds** (`Tensor`) – estimated labels
- **target** (`Tensor`) – ground truth labels
- **multioutput** (`str`) – Defines aggregation in the case of multiple output scores. Can be one of the following strings:
  - 'raw\_values' returns full set of scores
  - 'uniform\_average' scores are uniformly averaged
  - 'variance\_weighted' scores are weighted by their individual variances

### Example

```
>>> from torchmetrics.functional import explained_variance
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> explained_variance(preds, target)
tensor(0.9572)
```

```
>>> target = torch.tensor([[0.5, 1], [-1, 1], [7, -6]])
>>> preds = torch.tensor([[0, 2], [-1, 2], [8, -5]])
>>> explained_variance(preds, target, multioutput='raw_values')
tensor([0.9677, 1.0000])
```

Return type `Union[Tensor, Sequence[Tensor]]`

## 1.65 KL Divergence

### 1.65.1 Module Interface

**class** `torchmetrics.KLDivergence`(*log\_prob=False*, *reduction='mean'*, *\*\*kwargs*)

Computes the *KL divergence*:

$$D_{KL}(P||Q) = \sum_{x \in \mathcal{X}} P(x) \log \frac{P(x)}{Q(x)}$$

Where  $P$  and  $Q$  are probability distributions where  $P$  usually represents a distribution over data and  $Q$  is often a prior or approximation of  $P$ . It should be noted that the KL divergence is a non-symmetrical metric i.e.  $D_{KL}(P||Q) \neq D_{KL}(Q||P)$ .

#### Parameters

- **p** – data distribution with shape `[N, d]`
- **q** – prior or approximate distribution with shape `[N, d]`
- **log\_prob** (`bool`) – bool indicating if input is log-probabilities or probabilities. If given as probabilities, will normalize to make sure the distributes sum to 1.
- **reduction** (`Literal['mean', 'sum', 'none', None]`) – Determines how to reduce over the N/batch dimension:
  - 'mean' [default]: Averages score across samples
  - 'sum': Sum score across samples
  - 'none' or None: Returns score per sample
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

- **TypeError** – If `log_prob` is not an `bool`.
- **ValueError** – If `reduction` is not one of 'mean', 'sum', 'none' or None.

---

**Note:** Half precision is only support on GPU for this metric

---

### Example

```
>>> import torch
>>> from torchmetrics.functional import kl_divergence
>>> p = torch.tensor([[0.36, 0.48, 0.16]])
>>> q = torch.tensor([[1/3, 1/3, 1/3]])
>>> kl_divergence(p, q)
tensor(0.0853)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### **compute()**

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tensor`

#### **update(p, q)**

Override this method to update the state variables of your metric class.

**Return type** `None`

## 1.65.2 Functional Interface

`torchmetrics.functional.kl_divergence(p, q, log_prob=False, reduction='mean')`

Computes *KL divergence*

$$D_{KL}(P||Q) = \sum_{x \in \mathcal{X}} P(x) \log \frac{P(x)}{Qx}$$

Where  $P$  and  $Q$  are probability distributions where  $P$  usually represents a distribution over data and  $Q$  is often a prior or approximation of  $P$ . It should be noted that the KL divergence is a non-symmetrical metric i.e.  $D_{KL}(P||Q) \neq D_{KL}(Q||P)$ .

#### Parameters

- **p** (`Tensor`) – data distribution with shape `[N, d]`
- **q** (`Tensor`) – prior or approximate distribution with shape `[N, d]`
- **log\_prob** (`bool`) – bool indicating if input is log-probabilities or probabilities. If given as probabilities, will normalize to make sure the distributes sum to 1
- **reduction** (`Literal``['mean', 'sum', 'none', None]`) – Determines how to reduce over the N/batch dimension:
  - 'mean' [default]: Averages score across samples
  - 'sum': Sum score across samples
  - 'none' or None: Returns score per sample



### Example

```
>>> import torch
>>> p = torch.tensor([[0.36, 0.48, 0.16]])
>>> q = torch.tensor([[1/3, 1/3, 1/3]])
>>> kl_divergence(p, q)
tensor(0.0853)
```

Return type `Tensor`

## 1.66 Mean Absolute Error (MAE)

### 1.66.1 Module Interface

**class** `torchmetrics.MeanAbsoluteError(**kwargs)`

Computes Mean Absolute Error (MAE):

$$\text{MAE} = \frac{1}{N} \sum_i^N |y_i - \hat{y}_i|$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

**Parameters** `kwargs` (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

### Example

```
>>> from torchmetrics import MeanAbsoluteError
>>> target = torch.tensor([3.0, -0.5, 2.0, 7.0])
>>> preds = torch.tensor([2.5, 0.0, 2.0, 8.0])
>>> mean_absolute_error = MeanAbsoluteError()
>>> mean_absolute_error(preds, target)
tensor(0.5000)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

**compute()**

Computes mean absolute error over state.

Return type `Tensor`

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

Return type `None`

## 1.66.2 Functional Interface

`torchmetrics.functional.mean_absolute_error(preds, target)`

Computes mean absolute error.

**Parameters**

- **preds** (`Tensor`) – estimated labels
- **target** (`Tensor`) – ground truth labels

**Return type** `Tensor`

**Returns** Tensor with MAE

**Example**

```
>>> from torchmetrics.functional import mean_absolute_error
>>> x = torch.tensor([0., 1, 2, 3])
>>> y = torch.tensor([0., 1, 2, 2])
>>> mean_absolute_error(x, y)
tensor(0.2500)
```

## 1.67 Mean Absolute Percentage Error (MAPE)

### 1.67.1 Module Interface

`class torchmetrics.MeanAbsolutePercentageError(**kwargs)`

Computes Mean Absolute Percentage Error (MAPE):

$$\text{MAPE} = \frac{1}{n} \sum_{i=1}^n \frac{|y_i - \hat{y}_i|}{\max(\epsilon, |y_i|)}$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

**Parameters** **kwargs** (`Any`) – Additional keyword arguments, see *Advanced metric settings* for more info.

---

**Note:** The epsilon value is taken from `scikit-learn`'s implementation of MAPE.

---

---

**Note:** MAPE output is a non-negative floating point. Best result is `0.0`. But it is important to note that, bad predictions, can lead to arbitrarily large values. Especially when some `target` values are close to 0. This [MAPE implementation](#) returns a very large number instead of `inf`.

---

### Example

```
>>> from torchmetrics import MeanAbsolutePercentageError
>>> target = torch.tensor([1, 10, 1e6])
>>> preds = torch.tensor([0.9, 15, 1.2e6])
>>> mean_abs_percentage_error = MeanAbsolutePercentageError()
>>> mean_abs_percentage_error(preds, target)
tensor(0.2667)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes mean absolute percentage error over state.

**Return type** `Tensor`

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

## 1.67.2 Functional Interface

`torchmetrics.functional.mean_absolute_percentage_error(preds, target)`

Computes mean absolute percentage error.

**Parameters**

- **preds** (`Tensor`) – estimated labels
- **target** (`Tensor`) – ground truth labels

**Return type** `Tensor`

**Returns** Tensor with MAPE

---

**Note:** The epsilon value is taken from `scikit-learn`'s implementation of MAPE.

---

### Example

```
>>> from torchmetrics.functional import mean_absolute_percentage_error
>>> target = torch.tensor([1, 10, 1e6])
>>> preds = torch.tensor([0.9, 15, 1.2e6])
>>> mean_absolute_percentage_error(preds, target)
tensor(0.2667)
```

## 1.68 Mean Squared Error (MSE)

### 1.68.1 Module Interface

**class** `torchmetrics.MeanSquaredError`(*squared=True, \*\*kwargs*)

Computes [mean squared error](#) (MSE):

$$\text{MSE} = \frac{1}{N} \sum_i^N (y_i - \hat{y}_i)^2$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

#### Parameters

- **squared** (`bool`) – If True returns MSE value, if False returns RMSE value.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Example

```
>>> from torchmetrics import MeanSquaredError
>>> target = torch.tensor([2.5, 5.0, 4.0, 8.0])
>>> preds = torch.tensor([3.0, 5.0, 2.5, 7.0])
>>> mean_squared_error = MeanSquaredError()
>>> mean_squared_error(preds, target)
tensor(0.8750)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

**compute()**

Computes mean squared error over state.

**Return type** `Tensor`

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

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

### 1.68.2 Functional Interface

`torchmetrics.functional.mean_squared_error`(*preds, target, squared=True*)

Computes mean squared error.

**Parameters**

- **preds** (`Tensor`) – estimated labels
- **target** (`Tensor`) – ground truth labels

- **squared** (`bool`) – returns RMSE value if set to False

**Return type** `Tensor`

**Returns** Tensor with MSE

### Example

```
>>> from torchmetrics.functional import mean_squared_error
>>> x = torch.tensor([0., 1, 2, 3])
>>> y = torch.tensor([0., 1, 2, 2])
>>> mean_squared_error(x, y)
tensor(0.2500)
```

## 1.69 Mean Squared Log Error (MSLE)

### 1.69.1 Module Interface

**class** `torchmetrics.MeanSquaredLogError` (\*\*kwargs)

Computes mean squared logarithmic error (MSLE):

$$\text{MSLE} = \frac{1}{N} \sum_i^N (\log_e(1 + y_i) - \log_e(1 + \hat{y}_i))^2$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

**Parameters** **kwargs** (`Any`) – Additional keyword arguments, see *Advanced metric settings* for more info.

### Example

```
>>> from torchmetrics import MeanSquaredLogError
>>> target = torch.tensor([2.5, 5, 4, 8])
>>> preds = torch.tensor([3, 5, 2.5, 7])
>>> mean_squared_log_error = MeanSquaredLogError()
>>> mean_squared_log_error(preds, target)
tensor(0.0397)
```

---

**Note:** Half precision is only support on GPU for this metric

---

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

**compute()**

Compute mean squared logarithmic error over state.

**Return type** `Tensor`

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

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

## 1.69.2 Functional Interface

`torchmetrics.functional.mean_squared_log_error(preds, target)`

Computes mean squared log error.

**Parameters**

- **preds** (`Tensor`) – estimated labels
- **target** (`Tensor`) – ground truth labels

**Return type** `Tensor`

**Returns** Tensor with RMSLE

### Example

```
>>> from torchmetrics.functional import mean_squared_log_error
>>> x = torch.tensor([0., 1, 2, 3])
>>> y = torch.tensor([0., 1, 2, 2])
>>> mean_squared_log_error(x, y)
tensor(0.0207)
```

---

**Note:** Half precision is only support on GPU for this metric

---

## 1.70 Pearson Corr. Coef.

### 1.70.1 Module Interface

**class** `torchmetrics.PearsonCorrCoef(num_outputs=1, **kwargs)`

Computes `Pearson Correlation Coefficient`:

$$P_{corr}(x, y) = \frac{cov(x, y)}{\sigma_x \sigma_y}$$

Where  $y$  is a tensor of target values, and  $x$  is a tensor of predictions.

Forward accepts

- **preds** (float tensor): either single output tensor with shape  $(N,)$  or multioutput tensor of shape  $(N, d)$
- **target** (float tensor): either single output tensor with shape  $(N,)$  or multioutput tensor of shape  $(N, d)$

**Parameters**

- **num\_outputs** (`int`) – Number of outputs in multioutput setting

- **kwargs** (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

**Example (single output regression):**

```
>>> from torchmetrics import PearsonCorrCoef
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> pearson = PearsonCorrCoef()
>>> pearson(preds, target)
tensor(0.9849)
```

**Example (multi output regression):**

```
>>> from torchmetrics import PearsonCorrCoef
>>> target = torch.tensor([[3, -0.5], [2, 7]])
>>> preds = torch.tensor([[2.5, 0.0], [2, 8]])
>>> pearson = PearsonCorrCoef(num_outputs=2)
>>> pearson(preds, target)
tensor([1., 1.])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes pearson correlation coefficient over state.

**Return type** *Tensor*

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (*Tensor*) – Predictions from model
- **target** (*Tensor*) – Ground truth values

**Return type** *None*

## 1.70.2 Functional Interface

`torchmetrics.functional.pearson_corrcoef(preds, target)`

Computes pearson correlation coefficient.

**Parameters**

- **preds** (*Tensor*) – estimated scores
- **target** (*Tensor*) – ground truth scores

**Example (single output regression):**

```
>>> from torchmetrics.functional import pearson_corrcoef
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> pearson_corrcoef(preds, target)
tensor(0.9849)
```

Example (multi output regression):

```
>>> from torchmetrics.functional import pearson_corrcoef
>>> target = torch.tensor([[3, -0.5], [2, 7]])
>>> preds = torch.tensor([[2.5, 0.0], [2, 8]])
>>> pearson_corrcoef(preds, target)
tensor([1., 1.])
```

Return type `Tensor`

## 1.71 R2 Score

### 1.71.1 Module Interface

**class** `torchmetrics.R2Score`(*num\_outputs=1*, *adjusted=0*, *multioutput='uniform\_average'*, *\*\*kwargs*)

Computes r2 score also known as [R2 Score\\_Coefficient Determination](#):

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

where  $SS_{res} = \sum_i (y_i - f(x_i))^2$  is the sum of residual squares, and  $SS_{tot} = \sum_i (y_i - \bar{y})^2$  is total sum of squares. Can also calculate adjusted r2 score given by

$$R^2_{adj} = 1 - \frac{(1 - R^2)(n - 1)}{n - k - 1}$$

where the parameter  $k$  (the number of independent regressors) should be provided as the *adjusted* argument.

Forward accepts

- `preds` (float tensor): (N,) or (N, M) (multioutput)
- `target` (float tensor): (N,) or (N, M) (multioutput)

In the case of multioutput, as default the variances will be uniformly averaged over the additional dimensions. Please see argument `multioutput` for changing this behavior.

#### Parameters

- **`num_outputs`** (`int`) – Number of outputs in multioutput setting
- **`adjusted`** (`int`) – number of independent regressors for calculating adjusted r2 score.
- **`multioutput`** (`str`) – Defines aggregation in the case of multiple output scores. Can be one of the following strings:
  - `'raw_values'` returns full set of scores
  - `'uniform_average'` scores are uniformly averaged
  - `'variance_weighted'` scores are weighted by their individual variances
- **`kwargs`** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

- **`ValueError`** – If `adjusted` parameter is not an integer larger or equal to 0.
- **`ValueError`** – If `multioutput` is not one of `"raw_values"`, `"uniform_average"` or `"variance_weighted"`.



## Example

```
>>> from torchmetrics import R2Score
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> r2score = R2Score()
>>> r2score(preds, target)
tensor(0.9486)
```

```
>>> target = torch.tensor([[0.5, 1], [-1, 1], [7, -6]])
>>> preds = torch.tensor([[0, 2], [-1, 2], [8, -5]])
>>> r2score = R2Score(num_outputs=2, multioutput='raw_values')
>>> r2score(preds, target)
tensor([0.9654, 0.9082])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes r2 score over the metric states.

**Return type** `Tensor`

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

## 1.71.2 Functional Interface

`torchmetrics.functional.r2_score(preds, target, adjusted=0, multioutput='uniform_average')`

Computes r2 score also known as **R2 Score\_Coefficient Determination**:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

where  $SS_{res} = \sum_i (y_i - f(x_i))^2$  is the sum of residual squares, and  $SS_{tot} = \sum_i (y_i - \bar{y})^2$  is total sum of squares. Can also calculate adjusted r2 score given by

$$R_{adj}^2 = 1 - \frac{(1 - R^2)(n - 1)}{n - k - 1}$$

where the parameter  $k$  (the number of independent regressors) should be provided as the `adjusted` argument.

**Parameters**

- **preds** (`Tensor`) – estimated labels
- **target** (`Tensor`) – ground truth labels
- **adjusted** (`int`) – number of independent regressors for calculating adjusted r2 score.
- **multioutput** (`str`) – Defines aggregation in the case of multiple output scores. Can be one of the following strings:

- 'raw\_values' returns full set of scores
- 'uniform\_average' scores are uniformly averaged
- 'variance\_weighted' scores are weighted by their individual variances

#### Raises

- **ValueError** – If both preds and targets are not 1D or 2D tensors.
- **ValueError** – If len(preds) is less than 2 since at least 2 samples are needed to calculate r2 score.
- **ValueError** – If multioutput is not one of raw\_values, uniform\_average or variance\_weighted.
- **ValueError** – If adjusted is not an integer greater than 0.

#### Example

```
>>> from torchmetrics.functional import r2_score
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> r2_score(preds, target)
tensor(0.9486)
```

```
>>> target = torch.tensor([[0.5, 1], [-1, 1], [7, -6]])
>>> preds = torch.tensor([[0, 2], [-1, 2], [8, -5]])
>>> r2_score(preds, target, multioutput='raw_values')
tensor([0.9654, 0.9082])
```

Return type `Tensor`

## 1.72 Spearman Corr. Coef.

### 1.72.1 Module Interface

**class** torchmetrics.SpearmanCorrCoef(*num\_outputs=1, \*\*kwargs*)

Computes spearman's rank correlation coefficient.

where  $rg_x$  and  $rg_y$  are the rank associated to the variables  $x$  and  $y$ . Spearman's correlations coefficient corresponds to the standard pearson's correlation coefficient calculated on the rank variables.

Forward accepts

- preds (float tensor): (N,d)
- target (float tensor): (N,d)

#### Parameters

- **num\_outputs** (`int`) – Number of outputs in multioutput setting
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (single output regression):**

```
>>> from torchmetrics import SpearmanCorrCoef
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> spearman = SpearmanCorrCoef()
>>> spearman(preds, target)
tensor(1.0000)
```

**Example (multi output regression):**

```
>>> from torchmetrics import SpearmanCorrCoef
>>> target = torch.tensor([[3, -0.5], [2, 7]])
>>> preds = torch.tensor([[2.5, 0.0], [2, 8]])
>>> spearman = SpearmanCorrCoef(num_outputs=2)
>>> spearman(preds, target)
tensor([1.0000, 1.0000])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes Spearman’s correlation coefficient.

**Return type** `Tensor`

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

## 1.72.2 Functional Interface

`torchmetrics.functional.spearman_corrcoef(preds, target)`

Computes `spearman’s rank correlation coefficient`:

where  $rg_x$  and  $rg_y$  are the rank associated to the variables x and y. Spearman’s correlation coefficient corresponds to the standard Pearson’s correlation coefficient calculated on the rank variables.

**Parameters**

- **preds** (`Tensor`) – estimated scores
- **target** (`Tensor`) – ground truth scores

**Example (single output regression):**

```
>>> from torchmetrics.functional import spearman_corrcoef
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> spearman_corrcoef(preds, target)
tensor(1.0000)
```

Example (multi output regression):

```
>>> from torchmetrics.functional import spearman_corrcoef
>>> target = torch.tensor([[3, -0.5], [2, 7]])
>>> preds = torch.tensor([[2.5, 0.0], [2, 8]])
>>> spearman_corrcoef(preds, target)
tensor([1.0000, 1.0000])
```

Return type `Tensor`

## 1.73 Symmetric Mean Absolute Percentage Error (SMAPE)

### 1.73.1 Module Interface

**class** `torchmetrics.SymmetricMeanAbsolutePercentageError`(\*\**kwargs*)

Computes symmetric mean absolute percentage error (SMAPE).

$$\text{SMAPE} = \frac{2}{n} \sum_1^n \max\left(\frac{|y_i - \hat{y}_i|}{|y_i| + |\hat{y}_i|}, \epsilon\right)$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

**Parameters** *kwargs* (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

---

**Note:** The epsilon value is taken from `scikit-learn`'s implementation of SMAPE.

---

---

**Note:** SMAPE output is a non-negative floating point between 0 and 1. Best result is 0.0 .

---

### Example

```
>>> from torchmetrics import SymmetricMeanAbsolutePercentageError
>>> target = tensor([1, 10, 1e6])
>>> preds = tensor([0.9, 15, 1.2e6])
>>> smape = SymmetricMeanAbsolutePercentageError()
>>> smape(preds, target)
tensor(0.2290)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

**compute()**

Computes mean absolute percentage error over state.

**Return type** `Tensor`

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

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

## 1.73.2 Functional Interface

`torchmetrics.functional.symmetric_mean_absolute_percentage_error(preds, target)`

Computes symmetric mean absolute percentage error (SMAPE):

$$\text{SMAPE} = \frac{2}{n} \sum_1^n \frac{|y_i - \hat{y}_i|}{\max(|y_i| + |\hat{y}_i|, \epsilon)}$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

**Parameters**

- **preds** (`Tensor`) – estimated labels
- **target** (`Tensor`) – ground truth labels

**Return type** `Tensor`

**Returns** Tensor with SMAPE.

### Example

```
>>> from torchmetrics.functional import symmetric_mean_absolute_percentage_error
>>> target = torch.tensor([1, 10, 1e6])
>>> preds = torch.tensor([0.9, 15, 1.2e6])
>>> symmetric_mean_absolute_percentage_error(preds, target)
tensor(0.2290)
```

## 1.74 Tweedie Deviance Score

### 1.74.1 Module Interface

`class torchmetrics.TweedieDevianceScore(power=0.0, **kwargs)`

Computes the *Tweedie Deviance Score* between targets and predictions:

$$\text{deviance\_score}(\hat{y}, y) = \begin{cases} (\hat{y} - y)^2, & \text{for } p = 0 \\ 2 * (y * \log(\frac{y}{\hat{y}}) + \hat{y} - y), & \text{for } p = 1 \\ 2 * (\log(\frac{\hat{y}}{y}) + \frac{y}{\hat{y}} - 1), & \text{for } p = 2 \\ 2 * (\frac{(\max(y, 0))^{2-p}}{(1-p)(2-p)} - \frac{y(\hat{y})^{1-p}}{1-p} + \frac{(\hat{y})^{2-p}}{2-p}), & \text{otherwise} \end{cases}$$

where  $y$  is a tensor of targets values,  $\hat{y}$  is a tensor of predictions, and  $p$  is the *power*.

Forward accepts

- **preds** (float tensor): (N, ...)
- **targets** (float tensor): (N, ...)

### Parameters

- **power** (`float`) –
  - `power < 0` : Extreme stable distribution. (Requires: `preds > 0`.)
  - `power = 0` : Normal distribution. (Requires: targets and preds can be any real numbers.)
  - `power = 1` : Poisson distribution. (Requires: targets  $\geq 0$  and `y_pred > 0`.)
  - `1 < p < 2` : Compound Poisson distribution. (Requires: targets  $\geq 0$  and `preds > 0`.)
  - `power = 2` : Gamma distribution. (Requires: targets  $> 0$  and `preds > 0`.)
  - `power = 3` : Inverse Gaussian distribution. (Requires: targets  $> 0$  and `preds > 0`.)
  - otherwise : Positive stable distribution. (Requires: targets  $> 0$  and `preds > 0`.)
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torchmetrics import TweedieDevianceScore
>>> targets = torch.tensor([1.0, 2.0, 3.0, 4.0])
>>> preds = torch.tensor([4.0, 3.0, 2.0, 1.0])
>>> deviance_score = TweedieDevianceScore(power=2)
>>> deviance_score(preds, targets)
tensor(1.2083)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

#### `compute()`

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Tensor`

#### `update(preds, targets)`

Update metric states with predictions and targets.

#### Parameters

- **preds** (`Tensor`) – Predicted tensor with shape  $(N, d)$
- **targets** (`Tensor`) – Ground truth tensor with shape  $(N, d)$

**Return type** `None`

## 1.74.2 Functional Interface

`torchmetrics.functional.tweedie_deviance_score(preds, targets, power=0.0)`

Computes the *Tweedie Deviance Score* between targets and predictions:

$$\text{deviance\_score}(\hat{y}, y) = \begin{cases} (\hat{y} - y)^2, & \text{for } p = 0 \\ 2 * (y * \log(\frac{y}{\hat{y}}) + \hat{y} - y), & \text{for } p = 1 \\ 2 * (\log(\frac{\hat{y}}{y}) + \frac{\hat{y}}{y} - 1), & \text{for } p = 2 \\ 2 * (\frac{(\max(y, 0))^{2-p}}{(1-p)(2-p)} - \frac{y(\hat{y})^{1-p}}{1-p} + \frac{(\hat{y})^{2-p}}{2-p}), & \text{otherwise} \end{cases}$$

where  $y$  is a tensor of targets values,  $\hat{y}$  is a tensor of predictions, and  $p$  is the *power*.

#### Parameters

- **preds** ([Tensor](#)) – Predicted tensor with shape  $(N, \dots)$
- **targets** ([Tensor](#)) – Ground truth tensor with shape  $(N, \dots)$
- **power** ([float](#)) –
  - $power < 0$  : Extreme stable distribution. (Requires: preds  $> 0$ .)
  - $power = 0$  : Normal distribution. (Requires: targets and preds can be any real numbers.)
  - $power = 1$  : Poisson distribution. (Requires: targets  $\geq 0$  and y\_pred  $> 0$ .)
  - $1 < p < 2$  : Compound Poisson distribution. (Requires: targets  $\geq 0$  and preds  $> 0$ .)
  - $power = 2$  : Gamma distribution. (Requires: targets  $> 0$  and preds  $> 0$ .)
  - $power = 3$  : Inverse Gaussian distribution. (Requires: targets  $> 0$  and preds  $> 0$ .)
  - *otherwise* : Positive stable distribution. (Requires: targets  $> 0$  and preds  $> 0$ .)

#### Example

```
>>> from torchmetrics.functional import tweedie_deviance_score
>>> targets = torch.tensor([1.0, 2.0, 3.0, 4.0])
>>> preds = torch.tensor([4.0, 3.0, 2.0, 1.0])
>>> tweedie_deviance_score(preds, targets, power=2)
tensor(1.2083)
```

Return type [Tensor](#)

## 1.75 Weighted MAPE

### 1.75.1 Module Interface

**class** `torchmetrics.WeightedMeanAbsolutePercentageError`(\*\**kwargs*)

Computes weighted mean absolute percentage error (**WMAPE**). The output of WMAPE metric is a non-negative floating point, where the optimal value is 0. It computes as:

$$\text{WMAPE} = \frac{\sum_{t=1}^n |y_t - \hat{y}_t|}{\sum_{t=1}^n |y_t|}$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

**Parameters** **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> import torch
>>> _ = torch.manual_seed(42)
>>> preds = torch.randn(20,)
>>> target = torch.randn(20,)
>>> metric = WeightedMeanAbsolutePercentageError()
>>> metric(preds, target)
tensor(1.3967)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes weighted mean absolute percentage error over state.

**Return type** `Tensor`

**update(preds, target)**

Update state with predictions and targets.

**Parameters**

- **preds** (`Tensor`) – Predictions from model
- **target** (`Tensor`) – Ground truth values

**Return type** `None`

## 1.75.2 Functional Interface

`torchmetrics.functional.weighted_mean_absolute_percentage_error(preds, target)`

Computes weighted mean absolute percentage error (WMAPE).

The output of WMAPE metric is a non-negative floating point, where the optimal value is 0. It is computes as:

$$\text{WMAPE} = \frac{\sum_{t=1}^n |y_t - \hat{y}_t|}{\sum_{t=1}^n |y_t|}$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

**Parameters**

- **preds** (`Tensor`) – estimated labels
- **target** (`Tensor`) – ground truth labels

**Return type** `Tensor`

**Returns** Tensor with WMAPE.



### Example

```
>>> import torch
>>> _ = torch.manual_seed(42)
>>> preds = torch.randn(20,)
>>> target = torch.randn(20,)
>>> weighted_mean_absolute_percentage_error(preds, target)
tensor(1.3967)
```

## 1.76 Retrieval Fall-Out

### 1.76.1 Module Interface

**class** `torchmetrics.RetrievalFallOut`(*empty\_target\_action='pos', ignore\_index=None, k=None, \*\*kwargs*)

Computes [Fall-out](#).

Works with binary target data. Accepts float predictions from a model output.

Forward accepts:

- `preds` (float tensor): (N, ...)
- `target` (long or bool tensor): (N, ...)
- `indexes` (long tensor): (N, ...)

`indexes`, `preds` and `target` must have the same dimension. `indexes` indicate to which query a prediction belongs. Predictions will be first grouped by `indexes` and then *Fall-out* will be computed as the mean of the *Fall-out* over each query.

#### Parameters

- **`empty_target_action`** (`str`) – Specify what to do with queries that do not have at least a negative target. Choose from:
  - `'neg'`: those queries count as `0.0` (default)
  - `'pos'`: those queries count as `1.0`
  - `'skip'`: skip those queries; if all queries are skipped, `0.0` is returned
  - `'error'`: raise a `ValueError`
- **`ignore_index`** (`Optional[int]`) – Ignore predictions where the target is equal to this number.
- **`k`** (`Optional[int]`) – consider only the top `k` elements for each query (default: `None`, which considers them all)
- **`kwargs`** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

- **`ValueError`** – If `empty_target_action` is not one of `error`, `skip`, `neg` or `pos`.
- **`ValueError`** – If `ignore_index` is not `None` or an integer.
- **`ValueError`** – If `k` parameter is not `None` or an integer larger than 0.

### Example

```
>>> from torchmetrics import RetrievalFallOut
>>> indexes = tensor([0, 0, 0, 1, 1, 1, 1])
>>> preds = tensor([0.2, 0.3, 0.5, 0.1, 0.3, 0.5, 0.2])
>>> target = tensor([False, False, True, False, True, False, True])
>>> fo = RetrievalFallOut(k=2)
>>> fo(preds, target, indexes=indexes)
tensor(0.5000)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### **compute()**

First concat state `indexes`, `preds` and `target` since they were stored as lists.

After that, compute list of groups that will help in keeping together predictions about the same query. Finally, for each group compute the `_metric` if the number of negative targets is at least 1, otherwise behave as specified by `self.empty_target_action`.

**Return type** `Tensor`

## 1.76.2 Functional Interface

`torchmetrics.functional.retrieval_fall_out(preds, target, k=None)`

Computes the Fall-out (for information retrieval), as explained in [IR Fall-out](#). Fall-out is the fraction of non-relevant documents retrieved among all the non-relevant documents.

`preds` and `target` should be of the same shape and live on the same device. If no `target` is `True`, `0` is returned. `target` must be either *bool* or *integers* and `preds` must be *float*, otherwise an error is raised. If you want to measure Fall-out@K, `k` must be a positive integer.

#### **Parameters**

- **preds** (`Tensor`) – estimated probabilities of each document to be relevant.
- **target** (`Tensor`) – ground truth about each document being relevant or not.
- **k** (`Optional[int]`) – consider only the top k elements (default: `None`, which considers them all)

**Return type** `Tensor`

**Returns** a single-value tensor with the fall-out (at k) of the predictions `preds` w.r.t. the labels `target`.

**Raises** `ValueError` – If `k` parameter is not `None` or an integer larger than 0

### Example

```
>>> from torchmetrics.functional import retrieval_fall_out
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([True, False, True])
>>> retrieval_fall_out(preds, target, k=2)
tensor(1.)
```

## 1.77 Retrieval Hit Rate

### 1.77.1 Module Interface

**class** torchmetrics.RetrievalHitRate(empty\_target\_action='neg', ignore\_index=None, k=None, \*\*kwargs)

Computes *IR HitRate*.

Works with binary target data. Accepts float predictions from a model output.

Forward accepts:

- preds (float tensor): (N, ...)
- target (long or bool tensor): (N, ...)
- indexes (long tensor): (N, ...)

indexes, preds and target must have the same dimension. indexes indicate to which query a prediction belongs. Predictions will be first grouped by indexes and then the *Hit Rate* will be computed as the mean of the *Hit Rate* over each query.

#### Parameters

- **empty\_target\_action** (str) – Specify what to do with queries that do not have at least a positive target. Choose from:
  - 'neg': those queries count as 0.0 (default)
  - 'pos': those queries count as 1.0
  - 'skip': skip those queries; if all queries are skipped, 0.0 is returned
  - 'error': raise a ValueError
- **ignore\_index** (Optional[int]) – Ignore predictions where the target is equal to this number.
- **k** (Optional[int]) – consider only the top k elements for each query (default: None, which considers them all)
- **kwargs** (Any) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

- **ValueError** – If empty\_target\_action is not one of error, skip, neg or pos.
- **ValueError** – If ignore\_index is not None or an integer.
- **ValueError** – If k parameter is not None or an integer larger than 0.

#### Example

```
>>> from torchmetrics import RetrievalHitRate
>>> indexes = tensor([0, 0, 0, 1, 1, 1, 1])
>>> preds = tensor([0.2, 0.3, 0.5, 0.1, 0.3, 0.5, 0.2])
>>> target = tensor([True, False, False, False, True, False, True])
>>> hr2 = RetrievalHitRate(k=2)
>>> hr2(preds, target, indexes=indexes)
tensor(0.5000)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

## 1.77.2 Functional Interface

`torchmetrics.functional.retrieval_hit_rate(preds, target, k=None)`

Computes the hit rate (for information retrieval). The hit rate is 1.0 if there is at least one relevant document among all the top  $k$  retrieved documents.

`preds` and `target` should be of the same shape and live on the same device. If no `target` is `True`, `0` is returned. `target` must be either *bool* or *integers* and `preds` must be *float*, otherwise an error is raised. If you want to measure HitRate@K, `k` must be a positive integer.

### Parameters

- **preds** (`Tensor`) – estimated probabilities of each document to be relevant.
- **target** (`Tensor`) – ground truth about each document being relevant or not.
- **k** (`Optional[int]`) – consider only the top  $k$  elements (default: *None*, which considers them all)

**Return type** `Tensor`

**Returns** a single-value tensor with the hit rate (at  $k$ ) of the predictions `preds` w.r.t. the labels `target`.

**Raises** `ValueError` – If `k` parameter is not *None* or an integer larger than 0

### Example

```
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([True, False, True])
>>> retrieval_hit_rate(preds, target, k=2)
tensor(1.)
```

## 1.78 Retrieval Mean Average Precision (MAP)

### 1.78.1 Module Interface

**class** `torchmetrics.RetrievalMAP(empty_target_action='neg', ignore_index=None, **kwargs)`

Computes *Mean Average Precision*.

Works with binary target data. Accepts float predictions from a model output.

Forward accepts

- **preds** (float tensor): (N, ...)
- **target** (long or bool tensor): (N, ...)
- **indexes** (long tensor): (N, ...)

`indexes`, `preds` and `target` must have the same dimension. `indexes` indicate to which query a prediction belongs. Predictions will be first grouped by `indexes` and then *MAP* will be computed as the mean of the *Average Precisions* over each query.

### Parameters

- **empty\_target\_action** (`str`) – Specify what to do with queries that do not have at least a positive target. Choose from:
  - 'neg': those queries count as 0.0 (default)
  - 'pos': those queries count as 1.0
  - 'skip': skip those queries; if all queries are skipped, 0.0 is returned
  - 'error': raise a `ValueError`
- **ignore\_index** (`Optional[int]`) – Ignore predictions where the target is equal to this number.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Raises

- **ValueError** – If `empty_target_action` is not one of `error`, `skip`, `neg` or `pos`.
- **ValueError** – If `ignore_index` is not `None` or an integer.

### Example

```
>>> from torchmetrics import RetrievalMAP
>>> indexes = tensor([0, 0, 0, 1, 1, 1, 1])
>>> preds = tensor([0.2, 0.3, 0.5, 0.1, 0.3, 0.5, 0.2])
>>> target = tensor([False, False, True, False, True, False, True])
>>> rmap = RetrievalMAP()
>>> rmap(preds, target, indexes=indexes)
tensor(0.7917)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

## 1.78.2 Functional Interface

`torchmetrics.functional.retrieval_average_precision(preds, target)`

Computes average precision (for information retrieval), as explained in [IR Average precision](#).

`preds` and `target` should be of the same shape and live on the same device. If no `target` is `True`, 0 is returned. `target` must be either *bool* or *integers* and `preds` must be *float*, otherwise an error is raised.

### Parameters

- **preds** (`Tensor`) – estimated probabilities of each document to be relevant.
- **target** (`Tensor`) – ground truth about each document being relevant or not.

### Return type

`Tensor`

**Returns** a single-value tensor with the average precision (AP) of the predictions `preds` w.r.t. the labels `target`.

### Example

```
>>> from torchmetrics.functional import retrieval_average_precision
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([True, False, True])
>>> retrieval_average_precision(preds, target)
tensor(0.8333)
```

## 1.79 Retrieval Mean Reciprocal Rank (MRR)

### 1.79.1 Module Interface

**class** torchmetrics.RetrievalMRR(*empty\_target\_action='neg', ignore\_index=None, \*\*kwargs*)

Computes [Mean Reciprocal Rank](#).

Works with binary target data. Accepts float predictions from a model output.

Forward accepts

- **preds** (float tensor): (N, ...)
- **target** (long or bool tensor): (N, ...)
- **indexes** (long tensor): (N, ...)

**indexes**, **preds** and **target** must have the same dimension. **indexes** indicate to which query a prediction belongs. Predictions will be first grouped by **indexes** and then *MRR* will be computed as the mean of the *Reciprocal Rank* over each query.

#### Parameters

- **empty\_target\_action** (**str**) – Specify what to do with queries that do not have at least a positive target. Choose from:
  - 'neg': those queries count as 0.0 (default)
  - 'pos': those queries count as 1.0
  - 'skip': skip those queries; if all queries are skipped, 0.0 is returned
  - 'error': raise a `ValueError`
- **ignore\_index** ([Optional\[int\]](#)) – Ignore predictions where the target is equal to this number.
- **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

- **ValueError** – If **empty\_target\_action** is not one of `error`, `skip`, `neg` or `pos`.
- **ValueError** – If **ignore\_index** is not `None` or an integer.

### Example

```
>>> from torchmetrics import RetrievalMRR
>>> indexes = tensor([0, 0, 0, 1, 1, 1, 1])
>>> preds = tensor([0.2, 0.3, 0.5, 0.1, 0.3, 0.5, 0.2])
>>> target = tensor([False, False, True, False, True, False, True])
>>> mrr = RetrievalMRR()
>>> mrr(preds, target, indexes=indexes)
tensor(0.7500)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

## 1.79.2 Functional Interface

`torchmetrics.functional.retrieval_reciprocal_rank(preds, target)`

Computes reciprocal rank (for information retrieval). See [Mean Reciprocal Rank](#)

`preds` and `target` should be of the same shape and live on the same device. If no `target` is `True`, 0 is returned. `target` must be either *bool* or *integers* and `preds` must be *float*, otherwise an error is raised.

#### Parameters

- **preds** ([Tensor](#)) – estimated probabilities of each document to be relevant.
- **target** ([Tensor](#)) – ground truth about each document being relevant or not.

#### Return type [Tensor](#)

**Returns** a single-value tensor with the reciprocal rank (RR) of the predictions `preds` wrt the labels `target`.

### Example

```
>>> from torchmetrics.functional import retrieval_reciprocal_rank
>>> preds = torch.tensor([0.2, 0.3, 0.5])
>>> target = torch.tensor([False, True, False])
>>> retrieval_reciprocal_rank(preds, target)
tensor(0.5000)
```

## 1.80 Retrieval Normalized DCG

### 1.80.1 Module Interface

`class torchmetrics.RetrievalNormalizedDCG(empty_target_action='neg', ignore_index=None, k=None, **kwargs)`

Computes [Normalized Discounted Cumulative Gain](#).

Works with binary or positive integer target data. Accepts float predictions from a model output.

Forward accepts:

- `preds` (float tensor): (N, ...)

- `target` (long, int, bool or float tensor): (N, ...)
- `indexes` (long tensor): (N, ...)

`indexes`, `preds` and `target` must have the same dimension. `indexes` indicate to which query a prediction belongs. Predictions will be first grouped by `indexes` and then *Normalized Discounted Cumulative Gain* will be computed as the mean of the *Normalized Discounted Cumulative Gain* over each query.

#### Parameters

- **`empty_target_action`** (`str`) – Specify what to do with queries that do not have at least a positive target. Choose from:
  - `'neg'`: those queries count as 0.0 (default)
  - `'pos'`: those queries count as 1.0
  - `'skip'`: skip those queries; if all queries are skipped, 0.0 is returned
  - `'error'`: raise a `ValueError`
- **`ignore_index`** (`Optional[int]`) – Ignore predictions where the target is equal to this number.
- **`k`** (`Optional[int]`) – consider only the top k elements for each query (default: `None`, which considers them all)
- **`kwargs`** (`Any`) – Additional keyword arguments, see *Advanced metric settings* for more info.

#### Raises

- **`ValueError`** – If `empty_target_action` is not one of `error`, `skip`, `neg` or `pos`.
- **`ValueError`** – If `ignore_index` is not `None` or an integer.
- **`ValueError`** – If `k` parameter is not `None` or an integer larger than 0.

#### Example

```
>>> from torchmetrics import RetrievalNormalizedDCG
>>> indexes = tensor([0, 0, 0, 1, 1, 1, 1])
>>> preds = tensor([0.2, 0.3, 0.5, 0.1, 0.3, 0.5, 0.2])
>>> target = tensor([False, False, True, False, True, False, True])
>>> ndcg = RetrievalNormalizedDCG()
>>> ndcg(preds, target, indexes=indexes)
tensor(0.8467)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.



## 1.80.2 Functional Interface

`torchmetrics.functional.retrieval_normalized_dcg(preds, target, k=None)`

Computes [Normalized Discounted Cumulative Gain](#) (for information retrieval).

`preds` and `target` should be of the same shape and live on the same device. `target` must be either *bool* or *integers* and `preds` must be float, otherwise an error is raised.

### Parameters

- **preds** ([Tensor](#)) – estimated probabilities of each document to be relevant.
- **target** ([Tensor](#)) – ground truth about each document relevance.
- **k** ([Optional\[int\]](#)) – consider only the top k elements (default: `None`, which considers them all)

**Return type** [Tensor](#)

**Returns** a single-value tensor with the nDCG of the predictions `preds` w.r.t. the labels `target`.

**Raises** [ValueError](#) – If k parameter is not *None* or an integer larger than 0

### Example

```
>>> from torchmetrics.functional import retrieval_normalized_dcg
>>> preds = torch.tensor([.1, .2, .3, 4, 70])
>>> target = torch.tensor([10, 0, 0, 1, 5])
>>> retrieval_normalized_dcg(preds, target)
tensor(0.6957)
```

## 1.81 Retrieval Precision

### 1.81.1 Module Interface

`class torchmetrics.RetrievalPrecision(empty_target_action='neg', ignore_index=None, k=None, adaptive_k=False, **kwargs)`

Computes [IR Precision](#).

Works with binary target data. Accepts float predictions from a model output.

Forward accepts:

- `preds` (float tensor): (N, ...)
- `target` (long or bool tensor): (N, ...)
- `indexes` (long tensor): (N, ...)

`indexes`, `preds` and `target` must have the same dimension. `indexes` indicate to which query a prediction belongs. Predictions will be first grouped by `indexes` and then *Precision* will be computed as the mean of the *Precision* over each query.

### Parameters

- **empty\_target\_action** ([str](#)) – Specify what to do with queries that do not have at least a positive `target`. Choose from:

- 'neg': those queries count as 0.0 (default)
- 'pos': those queries count as 1.0
- 'skip': skip those queries; if all queries are skipped, 0.0 is returned
- 'error': raise a `ValueError`
- **ignore\_index** (`Optional[int]`) – Ignore predictions where the target is equal to this number.
- **k** (`Optional[int]`) – consider only the top k elements for each query (default: `None`, which considers them all)
- **adaptive\_k** (`bool`) – adjust k to  $\min(k, \text{number of documents})$  for each query
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

- **ValueError** – If `empty_target_action` is not one of `error`, `skip`, `neg` or `pos`.
- **ValueError** – If `ignore_index` is not `None` or an integer.
- **ValueError** – If k is not `None` or an integer larger than 0.
- **ValueError** – If `adaptive_k` is not boolean.

#### Example

```
>>> from torchmetrics import RetrievalPrecision
>>> indexes = tensor([0, 0, 0, 1, 1, 1, 1])
>>> preds = tensor([0.2, 0.3, 0.5, 0.1, 0.3, 0.5, 0.2])
>>> target = tensor([False, False, True, False, True, False, True])
>>> p2 = RetrievalPrecision(k=2)
>>> p2(preds, target, indexes=indexes)
tensor(0.5000)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

## 1.81.2 Functional Interface

`torchmetrics.functional.retrieval_precision(preds, target, k=None, adaptive_k=False)`

Computes the precision metric (for information retrieval). Precision is the fraction of relevant documents among all the retrieved documents.

`preds` and `target` should be of the same shape and live on the same device. If no `target` is `True`, 0 is returned. `target` must be either *bool* or *integers* and `preds` must be *float*, otherwise an error is raised. If you want to measure `Precision@K`, k must be a positive integer.

#### Parameters

- **preds** (`Tensor`) – estimated probabilities of each document to be relevant.
- **target** (`Tensor`) – ground truth about each document being relevant or not.
- **k** (`Optional[int]`) – consider only the top k elements (default: `None`, which considers them all)
- **adaptive\_k** (`bool`) – adjust k to  $\min(k, \text{number of documents})$  for each query

**Return type** `Tensor`

**Returns** a single-value tensor with the precision (at `k`) of the predictions `preds` w.r.t. the labels `target`.

**Raises**

- **ValueError** – If `k` is not `None` or an integer larger than 0.
- **ValueError** – If `adaptive_k` is not boolean.

### Example

```
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([True, False, True])
>>> retrieval_precision(preds, target, k=2)
tensor(0.5000)
```

## 1.82 Precision Recall Curve

### 1.82.1 Module Interface

```
class torchmetrics.RetrievalPrecisionRecallCurve(max_k=None, adaptive_k=False,
                                                empty_target_action='neg', ignore_index=None,
                                                **kwargs)
```

Computes precision-recall pairs for different `k` (from 1 to `max_k`).

In a ranked retrieval context, appropriate sets of retrieved documents are naturally given by the top `k` retrieved documents.

Recall is the fraction of relevant documents retrieved among all the relevant documents. Precision is the fraction of relevant documents among all the retrieved documents.

For each such set, precision and recall values can be plotted to give a recall-precision curve.

Forward accepts:

- `preds` (float tensor): (N, ...)
- `target` (long or bool tensor): (N, ...)
- `indexes` (long tensor): (N, ...)

`indexes`, `preds` and `target` must have the same dimension. `indexes` indicate to which query a prediction belongs. Predictions will be first grouped by `indexes` and then *RetrievalRecallAtFixedPrecision* will be computed as the mean of the *RetrievalRecallAtFixedPrecision* over each query.

#### Parameters

- **max\_k** (`Optional[int]`) – Calculate recall and precision for all possible top `k` from 1 to `max_k` (default: `None`, which considers all possible top `k`)
- **adaptive\_k** (`bool`) – adjust `k` to  $\min(k, \text{number of documents})$  for each query
- **empty\_target\_action** (`str`) – Specify what to do with queries that do not have at least a positive `target`. Choose from:
  - `'neg'`: those queries count as `0.0` (default)

- 'pos': those queries count as 1.0
- 'skip': skip those queries; if all queries are skipped, 0.0 is returned
- 'error': raise a ValueError
- **ignore\_index** (Optional[int]) – Ignore predictions where the target is equal to this number.
- **kwargs** (Any) – Additional keyword arguments, see *Advanced metric settings* for more info.

#### Raises

- **ValueError** – If empty\_target\_action is not one of error, skip, neg or pos.
- **ValueError** – If ignore\_index is not None or an integer.
- **ValueError** – If max\_k parameter is not None or an integer larger than 0.

#### Example

```
>>> from torchmetrics import RetrievalPrecisionRecallCurve
>>> indexes = tensor([0, 0, 0, 0, 1, 1, 1])
>>> preds = tensor([0.4, 0.01, 0.5, 0.6, 0.2, 0.3, 0.5])
>>> target = tensor([True, False, False, True, True, False, True])
>>> r = RetrievalPrecisionRecallCurve(max_k=4)
>>> precisions, recalls, top_k = r(preds, target, indexes=indexes)
>>> precisions
tensor([1.0000, 0.5000, 0.6667, 0.5000])
>>> recalls
tensor([0.5000, 0.5000, 1.0000, 1.0000])
>>> top_k
tensor([1, 2, 3, 4])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### **compute()**

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** Tuple[Tensor, Tensor, Tensor]

#### **update(preds, target, indexes)**

Check shape, check and convert dtypes, flatten and add to accumulators.

**Return type** None

## 1.82.2 Functional Interface

`torchmetrics.functional.retrieval_precision_recall_curve(preds, target, max_k=None, adaptive_k=False)`

Computes precision-recall pairs for different k (from 1 to max\_k).

In a ranked retrieval context, appropriate sets of retrieved documents are naturally given by the top k retrieved documents.

Recall is the fraction of relevant documents retrieved among all the relevant documents. Precision is the fraction of relevant documents among all the retrieved documents.

For each such set, precision and recall values can be plotted to give a recall-precision curve.

`preds` and `target` should be of the same shape and live on the same device. If no `target` is `True`, `0` is returned. `target` must be either *bool* or *integers* and `preds` must be *float*, otherwise an error is raised.

#### Parameters

- **`preds`** (`Tensor`) – estimated probabilities of each document to be relevant.
- **`target`** (`Tensor`) – ground truth about each document being relevant or not.
- **`max_k`** (`Optional[int]`) – Calculate recall and precision for all possible top k from 1 to `max_k` (default: `None`, which considers all possible top k)
- **`adaptive_k`** (`bool`) – adjust `max_k` to `min(max_k, number of documents)` for each query

**Return type** `Tuple[Tensor, Tensor, Tensor]`

**Returns** tensor with the precision values for each k (at k) from 1 to `max_k` tensor with the recall values for each k (at k) from 1 to `max_k` tensor with all possibles k

#### Raises

- **`ValueError`** – If `max_k` is not `None` or an integer larger than 0.
- **`ValueError`** – If `adaptive_k` is not boolean.

#### Example

```
>>> from torchmetrics.functional import retrieval_precision_recall_curve
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([True, False, True])
>>> precisions, recalls, top_k = retrieval_precision_recall_curve(preds, target,
↳max_k=2)
>>> precisions
tensor([1.0000, 0.5000])
>>> recalls
tensor([0.5000, 0.5000])
>>> top_k
tensor([1, 2])
```

## 1.83 Retrieval R-Precision

### 1.83.1 Module Interface

**class** `torchmetrics.RetrievalRPrecision`(`empty_target_action='neg'`, `ignore_index=None`, `**kwargs`)

Computes IR R-Precision.

Works with binary target data. Accepts float predictions from a model output.

Forward accepts:

- `preds` (float tensor): (N, ...)
- `target` (long or bool tensor): (N, ...)

- `indexes` (long tensor): (N, ...)

`indexes`, `preds` and `target` must have the same dimension. `indexes` indicate to which query a prediction belongs. Predictions will be first grouped by `indexes` and then *R-Precision* will be computed as the mean of the *R-Precision* over each query.

#### Parameters

- `empty_target_action` (`str`) – Specify what to do with queries that do not have at least a positive target. Choose from:
  - `'neg'`: those queries count as `0.0` (default)
  - `'pos'`: those queries count as `1.0`
  - `'skip'`: skip those queries; if all queries are skipped, `0.0` is returned
  - `'error'`: raise a `ValueError`
- `ignore_index` (`Optional[int]`) – Ignore predictions where the target is equal to this number.
- `kwargs` (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

- `ValueError` – If `empty_target_action` is not one of `error`, `skip`, `neg` or `pos`.
- `ValueError` – If `ignore_index` is not `None` or an integer.

#### Example

```
>>> from torchmetrics import RetrievalRPrecision
>>> indexes = tensor([0, 0, 0, 1, 1, 1, 1])
>>> preds = tensor([0.2, 0.3, 0.5, 0.1, 0.3, 0.5, 0.2])
>>> target = tensor([False, False, True, False, True, False, True])
>>> p2 = RetrievalRPrecision()
>>> p2(preds, target, indexes=indexes)
tensor(0.7500)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

## 1.83.2 Functional Interface

`torchmetrics.functional.retrieval_r_precision(preds, target)`

Computes the r-precision metric (for information retrieval). R-Precision is the fraction of relevant documents among all the top `k` retrieved documents where `k` is equal to the total number of relevant documents.

`preds` and `target` should be of the same shape and live on the same device. If no `target` is `True`, `0` is returned. `target` must be either *bool* or *integers* and `preds` must be *float*, otherwise an error is raised. If you want to measure Precision@K, `k` must be a positive integer.

#### Parameters

- `preds` (`Tensor`) – estimated probabilities of each document to be relevant.
- `target` (`Tensor`) – ground truth about each document being relevant or not.

**Return type** `Tensor`

**Returns** a single-value tensor with the r-precision of the predictions `preds` w.r.t. the labels `target`.

### Example

```
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([True, False, True])
>>> retrieval_r_precision(preds, target)
tensor(0.5000)
```

## 1.84 Retrieval Recall

### 1.84.1 Module Interface

**class** `torchmetrics.RetrievalRecall`(*empty\_target\_action='neg', ignore\_index=None, k=None, \*\*kwargs*)

Computes [IR Recall](#).

Works with binary target data. Accepts float predictions from a model output.

Forward accepts:

- `preds` (float tensor): (N, ...)
- `target` (long or bool tensor): (N, ...)
- `indexes` (long tensor): (N, ...)

`indexes`, `preds` and `target` must have the same dimension. `indexes` indicate to which query a prediction belongs. Predictions will be first grouped by `indexes` and then *Recall* will be computed as the mean of the *Recall* over each query.

#### Parameters

- **`empty_target_action`** (`str`) – Specify what to do with queries that do not have at least a positive target. Choose from:
  - `'neg'`: those queries count as `0.0` (default)
  - `'pos'`: those queries count as `1.0`
  - `'skip'`: skip those queries; if all queries are skipped, `0.0` is returned
  - `'error'`: raise a `ValueError`
- **`ignore_index`** (`Optional[int]`) – Ignore predictions where the target is equal to this number.
- **`k`** (`Optional[int]`) – consider only the top `k` elements for each query (default: `None`, which considers them all)
- **`kwargs`** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

- **`ValueError`** – If `empty_target_action` is not one of `error`, `skip`, `neg` or `pos`.
- **`ValueError`** – If `ignore_index` is not `None` or an integer.
- **`ValueError`** – If `k` parameter is not `None` or an integer larger than 0.

### Example

```
>>> from torchmetrics import RetrievalRecall
>>> indexes = tensor([0, 0, 0, 1, 1, 1, 1])
>>> preds = tensor([0.2, 0.3, 0.5, 0.1, 0.3, 0.5, 0.2])
>>> target = tensor([False, False, True, False, True, False, True])
>>> r2 = RetrievalRecall(k=2)
>>> r2(preds, target, indexes=indexes)
tensor(0.7500)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

## 1.84.2 Functional Interface

`torchmetrics.functional.retrieval_recall(preds, target, k=None)`

Computes the recall metric (for information retrieval). Recall is the fraction of relevant documents retrieved among all the relevant documents.

`preds` and `target` should be of the same shape and live on the same device. If no `target` is `True`, `0` is returned. `target` must be either *bool* or *integers* and `preds` must be *float*, otherwise an error is raised. If you want to measure Recall@K, `k` must be a positive integer.

#### Parameters

- **preds** (`Tensor`) – estimated probabilities of each document to be relevant.
- **target** (`Tensor`) – ground truth about each document being relevant or not.
- **k** (`Optional[int]`) – consider only the top `k` elements (default: *None*, which considers them all)

**Return type** `Tensor`

**Returns** a single-value tensor with the recall (at `k`) of the predictions `preds` w.r.t. the labels `target`.

**Raises** `ValueError` – If `k` parameter is not *None* or an integer larger than 0

### Example

```
>>> from torchmetrics.functional import retrieval_recall
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([True, False, True])
>>> retrieval_recall(preds, target, k=2)
tensor(0.5000)
```



## 1.85 BERT Score

### 1.85.1 Module Interface

```
class torchmetrics.text.bert.BERTScore(model_name_or_path=None, num_layers=None,
                                       all_layers=False, model=None, user_tokenizer=None,
                                       user_forward_fn=None, verbose=False, idf=False, device=None,
                                       max_length=512, batch_size=64, num_threads=4,
                                       return_hash=False, lang='en', rescale_with_baseline=False,
                                       baseline_path=None, baseline_url=None, **kwargs)
```

**Bert\_score Evaluating Text Generation** leverages the pre-trained contextual embeddings from BERT and matches words in candidate and reference sentences by cosine similarity. It has been shown to correlate with human judgment on sentence-level and system-level evaluation. Moreover, BERTScore computes precision, recall, and F1 measure, which can be useful for evaluating different language generation tasks.

This implemenation follows the original implementation from [BERT\\_score](#).

#### Parameters

- **preds** – An iterable of predicted sentences.
- **target** – An iterable of target sentences.
- **model\_type** – A name or a model path used to load *transformers* pretrained model.
- **num\_layers** ([Optional\[int\]](#)) – A layer of representation to use.
- **all\_layers** ([bool](#)) – An indication of whether the representation from all model’s layers should be used. If *all\_layers = True*, the argument *num\_layers* is ignored.
- **model** ([Optional\[Module\]](#)) – A user’s own model. Must be of *torch.nn.Module* instance.
- **user\_tokenizer** ([Optional\[Any\]](#)) – A user’s own tokenizer used with the own model. This must be an instance with the `__call__` method. This method must take an iterable of sentences (*List[str]*) and must return a python dictionary containing “*input\_ids*” and “*attention\_mask*” represented by *torch.Tensor*. It is up to the user’s model of whether “*input\_ids*” is a *torch.Tensor* of input ids or embedding vectors. This tokenizer must prepend an equivalent of [CLS] token and append an equivalent of [SEP] token as *transformers* tokenizer does.
- **user\_forward\_fn** ([Optional\[Callable\[\[Module, Dict\[str, Tensor\]\], Tensor\]\]](#)) – A user’s own forward function used in a combination with *user\_model*. This function must take *user\_model* and a python dictionary of containing “*input\_ids*” and “*attention\_mask*” represented by *torch.Tensor* as an input and return the model’s output represented by the single *torch.Tensor*.
- **verbose** ([bool](#)) – An indication of whether a progress bar to be displayed during the embeddings’ calculation.
- **idf** ([bool](#)) – An indication whether normalization using inverse document frequencies should be used.
- **device** ([Union\[str, device, None\]](#)) – A device to be used for calculation.
- **max\_length** ([int](#)) – A maximum length of input sequences. Sequences longer than *max\_length* are to be trimmed.
- **batch\_size** ([int](#)) – A batch size used for model processing.
- **num\_threads** ([int](#)) – A number of threads to use for a dataloader.

- **return\_hash** (`bool`) – An indication of whether the corresponding *hash\_code* should be returned.
- **lang** (`str`) – A language of input sentences.
- **rescale\_with\_baseline** (`bool`) – An indication of whether bertscore should be rescaled with a pre-computed baseline. When a pretrained model from *transformers* model is used, the corresponding baseline is downloaded from the original *bert-score* package from [BERT\\_score](#) if available. In other cases, please specify a path to the baseline csv/tsv file, which must follow the formatting of the files from [BERT\\_score](#).
- **baseline\_path** (`Optional[str]`) – A path to the user's own local csv/tsv file with the baseline scale.
- **baseline\_url** (`Optional[str]`) – A url path to the user's own csv/tsv file with the baseline scale.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Returns** Python dictionary containing the keys *precision*, *recall* and *f1* with corresponding values.

### Example

```
>>> from torchmetrics.text.bert import BERTScore
>>> preds = ["hello there", "general kenobi"]
>>> target = ["hello there", "master kenobi"]
>>> bertscore = BERTScore()
>>> score = bertscore(preds, target)
>>> from pprint import pprint
>>> rounded_score = {k: [round(v, 3) for v in vv] for k, vv in score.items()}
>>> pprint(rounded_score)
{'f1': [1.0, 0.996], 'precision': [1.0, 0.996], 'recall': [1.0, 0.996]}
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### **compute()**

Calculate BERT scores.

**Return type** `Dict[str, Union[List[float], str]]`

**Returns** Python dictionary containing the keys *precision*, *recall* and *f1* with corresponding values.

#### **update(preds, target)**

Store predictions/references for computing BERT scores. It is necessary to store sentences in a tokenized form to ensure the DDP mode working.

##### **Parameters**

- **preds** (`List[str]`) – An iterable of predicted sentences.
- **target** (`List[str]`) – An iterable of reference sentences.

**Return type** `None`

## 1.85.2 Functional Interface

```
torchmetrics.functional.text.bert.bert_score(preds, target, model_name_or_path=None,
                                             num_layers=None, all_layers=False, model=None,
                                             user_tokenizer=None, user_forward_fn=None,
                                             verbose=False, idf=False, device=None,
                                             max_length=512, batch_size=64, num_threads=4,
                                             return_hash=False, lang='en',
                                             rescale_with_baseline=False, baseline_path=None,
                                             baseline_url=None)
```

**Bert\_score** Evaluating Text Generation leverages the pre-trained contextual embeddings from BERT and matches words in candidate and reference sentences by cosine similarity.

It has been shown to correlate with human judgment on sentence-level and system-level evaluation. Moreover, BERTScore computes precision, recall, and F1 measure, which can be useful for evaluating different language generation tasks.

This implementation follows the original implementation from [BERT\\_score](#).

### Parameters

- **preds** (`Union[List[str], Dict[str, Tensor]]`) – Either an iterable of predicted sentences or a Dict[input\_ids, attention\_mask].
- **target** (`Union[List[str], Dict[str, Tensor]]`) – Either an iterable of target sentences or a Dict[input\_ids, attention\_mask].
- **model\_name\_or\_path** (`Optional[str]`) – A name or a model path used to load transformers pretrained model.
- **num\_layers** (`Optional[int]`) – A layer of representation to use.
- **all\_layers** (`bool`) – An indication of whether the representation from all model's layers should be used. If `all_layers = True`, the argument `num_layers` is ignored.
- **model** (`Optional[Module]`) – A user's own model.
- **user\_tokenizer** (`Optional[Any]`) – A user's own tokenizer used with the own model. This must be an instance with the `__call__` method. This method must take an iterable of sentences (`List[str]`) and must return a python dictionary containing "input\_ids" and "attention\_mask" represented by `torch.Tensor`. It is up to the user's model of whether "input\_ids" is a `torch.Tensor` of input ids or embedding vectors. his tokenizer must prepend an equivalent of [CLS] token and append an equivalent of [SEP] token as *transformers* tokenizer does.
- **user\_forward\_fn** (`Optional[Callable[[Module, Dict[str, Tensor]], Tensor]]`) – A user's own forward function used in a combination with `user_model`. This function must take `user_model` and a python dictionary of containing "input\_ids" and "attention\_mask" represented by `torch.Tensor` as an input and return the model's output represented by the single `torch.Tensor`.
- **verbose** (`bool`) – An indication of whether a progress bar to be displayed during the embeddings' calculation.
- **idf** (`bool`) – An indication of whether normalization using inverse document frequencies should be used.
- **device** (`Union[str, device, None]`) – A device to be used for calculation.
- **max\_length** (`int`) – A maximum length of input sequences. Sequences longer than `max_length` are to be trimmed.

- **batch\_size** (`int`) – A batch size used for model processing.
- **num\_threads** (`int`) – A number of threads to use for a dataloader.
- **return\_hash** (`bool`) – An indication of whether the corresponding hash\_code should be returned.
- **lang** (`str`) – A language of input sentences. It is used when the scores are rescaled with a baseline.
- **rescale\_with\_baseline** (`bool`) – An indication of whether bertscore should be rescaled with a pre-computed baseline. When a pretrained model from `transformers` model is used, the corresponding baseline is downloaded from the original `bert-score` package from `BERT_score` if available. In other cases, please specify a path to the baseline csv/tsv file, which must follow the formatting of the files from `BERT_score`
- **baseline\_path** (`Optional[str]`) – A path to the user's own local csv/tsv file with the baseline scale.
- **baseline\_url** (`Optional[str]`) – A url path to the user's own csv/tsv file with the baseline scale.

**Return type** `Dict[str, Union[List[float], str]]`

**Returns** Python dictionary containing the keys `precision`, `recall` and `f1` with corresponding values.

**Raises**

- **ValueError** – If `len(preds) != len(target)`.
- **ModuleNotFoundError** – If `tqdm` package is required and not installed.
- **ModuleNotFoundError** – If `transformers` package is required and not installed.
- **ValueError** – If `num_layer` is larger than the number of the model layers.
- **ValueError** – If invalid input is provided.

### Example

```
>>> from torchmetrics.functional.text.bert import bert_score
>>> preds = ["hello there", "general kenobi"]
>>> target = ["hello there", "master kenobi"]
>>> score = bert_score(preds, target)
>>> from pprint import pprint
>>> rounded_score = {k: [round(v, 3) for v in vv] for k, vv in score.items()}
>>> pprint(rounded_score)
{'f1': [1.0, 0.996], 'precision': [1.0, 0.996], 'recall': [1.0, 0.996]}
```

## 1.86 BLEU Score

### 1.86.1 Module Interface

**class** `torchmetrics.BLEUScore`(*n\_gram=4, smooth=False, weights=None, \*\*kwargs*)

Calculate [BLEU score](#) of machine translated text with one or more references.

#### Parameters

- **n\_gram** (`int`) – Gram value ranged from 1 to 4
- **smooth** (`bool`) – Whether or not to apply smoothing, see [2]
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.
- **weights** (`Optional[Sequence[float]]`) – Weights used for unigrams, bigrams, etc. to calculate BLEU score. If not provided, uniform weights are used.

**Raises** [ValueError](#) – If a length of a list of weights is not None and not equal to n\_gram.

#### Example

```
>>> from torchmetrics import BLEUScore
>>> preds = ['the cat is on the mat']
>>> target = [['there is a cat on the mat', 'a cat is on the mat']]
>>> metric = BLEUScore()
>>> metric(preds, target)
tensor(0.7598)
```

#### References

[1] BLEU: a Method for Automatic Evaluation of Machine Translation by Papineni, Kishore, Salim Roukos, Todd Ward, and Wei-Jing Zhu [BLEU](#)

[2] Automatic Evaluation of Machine Translation Quality Using Longest Common Subsequence and Skip-Bigram Statistics by Chin-Yew Lin and Franz Josef Och [Machine Translation Evolution](#)

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

#### **compute()**

Calculate BLEU score.

**Return type** [Tensor](#)

**Returns** Tensor with BLEU Score

#### **update(preds, target)**

Compute Precision Scores.

#### Parameters

- **preds** (`Sequence[str]`) – An iterable of machine translated corpus
- **target** (`Sequence[Sequence[str]]`) – An iterable of iterables of reference corpus

**Return type** [None](#)

## 1.86.2 Functional Interface

`torchmetrics.functional.bleu_score(preds, target, n_gram=4, smooth=False, weights=None)`

Calculate BLEU score of machine translated text with one or more references.

### Parameters

- **preds** (`Union[str, Sequence[str]]`) – An iterable of machine translated corpus
- **target** (`Sequence[Union[str, Sequence[str]]]`) – An iterable of iterables of reference corpus
- **n\_gram** (`int`) – Gram value ranged from 1 to 4
- **smooth** (`bool`) – Whether to apply smoothing – see [2]
- **weights** (`Optional[Sequence[float]]`) – Weights used for unigrams, bigrams, etc. to calculate BLEU score. If not provided, uniform weights are used.

**Return type** `Tensor`

**Returns** Tensor with BLEU Score

### Raises

- **ValueError** – If preds and target corpus have different lengths.
- **ValueError** – If a length of a list of weights is not None and not equal to n\_gram.

### Example

```
>>> from torchmetrics.functional import bleu_score
>>> preds = ['the cat is on the mat']
>>> target = [['there is a cat on the mat', 'a cat is on the mat']]
>>> bleu_score(preds, target)
tensor(0.7598)
```

### References

[1] BLEU: a Method for Automatic Evaluation of Machine Translation by Papineni, Kishore, Salim Roukos, Todd Ward, and Wei-Jing Zhu [BLEU](#)

[2] Automatic Evaluation of Machine Translation Quality Using Longest Common Subsequence and Skip-Bigram Statistics by Chin-Yew Lin and Franz Josef Och [Machine Translation Evolution](#)

## 1.87 Char Error Rate

### 1.87.1 Module Interface

`class torchmetrics.CharErrorRate(**kwargs)`

Character Error Rate (**CER**) is a metric of the performance of an automatic speech recognition (ASR) system.

This value indicates the percentage of characters that were incorrectly predicted. The lower the value, the better the performance of the ASR system with a CharErrorRate of 0 being a perfect score. Character error rate can

then be computed as:

$$\text{CharErrorRate} = \frac{S + D + I}{N} = \frac{S + D + I}{S + D + C}$$

where:

- $S$  is the number of substitutions,
- $D$  is the number of deletions,
- $I$  is the number of insertions,
- $C$  is the number of correct characters,
- $N$  is the number of characters in the reference ( $N=S+D+C$ ).

Compute CharErrorRate score of transcribed segments against references.

**Parameters** **kwargs** (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

**Returns** Character error rate score

## Examples

```
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> metric = CharErrorRate()
>>> metric(preds, target)
tensor(0.3415)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Calculate the character error rate.

**Return type** *Tensor*

**Returns** Character error rate score

**update(preds, target)**

Store references/predictions for computing Character Error Rate scores.

**Parameters**

- **preds** (*Union[str, List[str]]*) – Transcription(s) to score as a string or list of strings
- **target** (*Union[str, List[str]]*) – Reference(s) for each speech input as a string or list of strings

**Return type** *None*

## 1.87.2 Functional Interface

`torchmetrics.functional.char_error_rate(preds, target)`

character error rate is a common metric of the performance of an automatic speech recognition system. This value indicates the percentage of characters that were incorrectly predicted. The lower the value, the better the performance of the ASR system with a CER of 0 being a perfect score.

### Parameters

- **preds** (`Union[str, List[str]]`) – Transcription(s) to score as a string or list of strings
- **target** (`Union[str, List[str]]`) – Reference(s) for each speech input as a string or list of strings

**Return type** `Tensor`

**Returns** Character error rate score

### Examples

```
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> char_error_rate(preds=preds, target=target)
tensor(0.3415)
```

## 1.88 ChrF Score

### 1.88.1 Module Interface

`class torchmetrics.CHRFScore(n_char_order=6, n_word_order=2, beta=2.0, lowercase=False, whitespace=False, return_sentence_level_score=False, **kwargs)`

Calculate `chrF score` of machine translated text with one or more references.

This implementation supports both ChrF score computation introduced in [1] and chrF++ score introduced in *chrF++ score*. This implementation follows the implimenetaions from <https://github.com/m-popovic/chrF> and <https://github.com/mjpost/sacrebleu/blob/master/sacrebleu/metrics/chrF.py>.

### Parameters

- **n\_char\_order** (`int`) – A character n-gram order. If `n_char_order=6`, the metrics refers to the official chrF/chrF++.
- **n\_word\_order** (`int`) – A word n-gram order. If `n_word_order=2`, the metric refers to the official chrF++. If `n_word_order=0`, the metric is equivalent to the original ChrF.
- **beta** (`float`) – parameter determining an importance of recall w.r.t. precision. If `beta=1`, their importance is equal.
- **lowercase** (`bool`) – An indication whether to enable case-insesitivity.
- **whitespace** (`bool`) – An indication whether keep whitespaces during n-gram extraction.
- **return\_sentence\_level\_score** (`bool`) – An indication whether a sentence-level chrF/chrF++ score to be returned.
- **kwargs** (`Any`) – Additional keyword arguments, see *Advanced metric settings* for more info.



**Raises**

- **ValueError** – If `n_char_order` is not an integer greater than or equal to 1.
- **ValueError** – If `n_word_order` is not an integer greater than or equal to 0.
- **ValueError** – If `beta` is smaller than 0.

**Example**

```
>>> from torchmetrics import CHRFScore
>>> preds = ['the cat is on the mat']
>>> target = [['there is a cat on the mat', 'a cat is on the mat']]
>>> metric = CHRFScore()
>>> metric(preds, target)
tensor(0.8640)
```

**References**

[1] chrF: character n-gram F-score for automatic MT evaluation by Maja Popović [chrF score](#)

[2] chrF++: words helping character n-grams by Maja Popović [chrF++ score](#)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Calculate chrF/chrF++ score.

**Return type** `Union[Tensor, Tuple[Tensor, Tensor]]`

**Returns** A corpus-level chrF/chrF++ score. (Optionally) A list of sentence-level chrF/chrF++ scores if `return_sentence_level_score=True`.

**update(preds, target)**

Compute Precision Scores.

**Parameters**

- **preds** (`Sequence[str]`) – An iterable of hypothesis corpus.
- **target** (`Sequence[Sequence[str]]`) – An iterable of iterables of reference corpus.

**Return type** `None`

**1.88.2 Functional Interface**

`torchmetrics.functional.chrf_score(preds, target, n_char_order=6, n_word_order=2, beta=2.0, lowercase=False, whitespace=False, return_sentence_level_score=False)`

Calculate [chrF score](#) of machine translated text with one or more references. This implementation supports both chrF score computation introduced in [1] and chrF++ score introduced in [chrF++ score](#). This implementation follows the implmenetaions from <https://github.com/m-popovic/chrF> and <https://github.com/mjpost/sacrebleu/blob/master/sacrebleu/metrics/chrf.py>.

**Parameters**

- **preds** (`Union[str, Sequence[str]]`) – An iterable of hypothesis corpus.

- **target** (`Sequence[Union[str, Sequence[str]]]`) – An iterable of iterables of reference corpus.
- **n\_char\_order** (`int`) – A character n-gram order. If `n_char_order=6`, the metrics refers to the official chrF/chrF++.
- **n\_word\_order** (`int`) – A word n-gram order. If `n_word_order=2`, the metric refers to the official chrF++. If `n_word_order=0`, the metric is equivalent to the original chrF.
- **beta** (`float`) – A parameter determining an importance of recall w.r.t. precision. If `beta=1`, their importance is equal.
- **lowercase** (`bool`) – An indication whether to enable case-insensitivity.
- **whitespace** (`bool`) – An indication whether to keep whitespaces during character n-gram extraction.
- **return\_sentence\_level\_score** (`bool`) – An indication whether a sentence-level chrF/chrF++ score to be returned.

**Return type** `Union[Tensor, Tuple[Tensor, Tensor]]`

**Returns** A corpus-level chrF/chrF++ score. (Optionally) A list of sentence-level chrF/chrF++ scores if `return_sentence_level_score=True`.

**Raises**

- **ValueError** – If `n_char_order` is not an integer greater than or equal to 1.
- **ValueError** – If `n_word_order` is not an integer greater than or equal to 0.
- **ValueError** – If `beta` is smaller than 0.

### Example

```
>>> from torchmetrics.functional import chrf_score
>>> preds = ['the cat is on the mat']
>>> target = [['there is a cat on the mat', 'a cat is on the mat']]
>>> chrf_score(preds, target)
tensor(0.8640)
```

### References

[1] chrF: character n-gram F-score for automatic MT evaluation by Maja Popović [chrF score](#)

[2] chrF++: words helping character n-grams by Maja Popović [chrF++ score](#)

## 1.89 Extended Edit Distance

### 1.89.1 Module Interface

**class** `torchmetrics.ExtendedEditDistance`(`language='en'`, `return_sentence_level_score=False`, `alpha=2.0`, `rho=0.3`, `deletion=0.2`, `insertion=1.0`, `**kwargs`)

Computes extended edit distance score ([ExtendedEditDistance](#)) [1] for strings or list of strings.

The metric utilises the Levenshtein distance and extends it by adding a jump operation.

**Parameters**

- **language** (`Literal['en', 'ja']`) – Language used in sentences. Only supports English (en) and Japanese (ja) for now.
- **return\_sentence\_level\_score** (`bool`) – An indication of whether sentence-level EED score is to be returned
- **alpha** (`float`) – optimal jump penalty, penalty for jumps between characters
- **rho** (`float`) – coverage cost, penalty for repetition of characters
- **deletion** (`float`) – penalty for deletion of character
- **insertion** (`float`) – penalty for insertion or substitution of character
- **kwargs** (`Any`) – Additional keyword arguments, see *Advanced metric settings* for more info.

**Returns** Extended edit distance score as a tensor

**Example**

```
>>> from torchmetrics import ExtendedEditDistance
>>> preds = ["this is the prediction", "here is an other sample"]
>>> target = ["this is the reference", "here is another one"]
>>> metric = ExtendedEditDistance()
>>> metric(preds=preds, target=target)
tensor(0.3078)
```

**References**

[1] P. Stanchev, W. Wang, and H. Ney, “EED: Extended Edit Distance Measure for Machine Translation”, submitted to WMT 2019. [ExtendedEditDistance](#)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Calculate extended edit distance score.

**Return type** `Union[Tensor, Tuple[Tensor, Tensor]]`

**Returns** Extended edit distance score as tensor

**update(preds, target)**

Update ExtendedEditDistance statistics.

**Parameters**

- **preds** (`Union[str, Sequence[str]]`) – An iterable of hypothesis corpus
- **target** (`Sequence[Union[str, Sequence[str]]]`) – An iterable of iterables of reference corpus

**Return type** `None`

## 1.89.2 Functional Interface

```
torchmetrics.functional.extended_edit_distance(preds, target, language='en',
                                              return_sentence_level_score=False, alpha=2.0,
                                              rho=0.3, deletion=0.2, insertion=1.0)
```

Computes extended edit distance score ([ExtendedEditDistance](#)) [1] for strings or list of strings. The metric utilises the Levenshtein distance and extends it by adding a jump operation.

### Parameters

- **preds** ([Union](#)[[str](#), [Sequence](#)[[str](#)]]) – An iterable of hypothesis corpus.
- **target** ([Sequence](#)[[Union](#)[[str](#), [Sequence](#)[[str](#)]]]) – An iterable of iterables of reference corpus.
- **language** ([Literal](#)['en', 'ja']) – Language used in sentences. Only supports English (en) and Japanese (ja) for now. Defaults to en
- **return\_sentence\_level\_score** ([bool](#)) – An indication of whether sentence-level EED score is to be returned.
- **alpha** ([float](#)) – optimal jump penalty, penalty for jumps between characters
- **rho** ([float](#)) – coverage cost, penalty for repetition of characters
- **deletion** ([float](#)) – penalty for deletion of character
- **insertion** ([float](#)) – penalty for insertion or substitution of character

**Return type** [Union](#)[[Tensor](#), [Tuple](#)[[Tensor](#), [Tensor](#)]]

**Returns** Extended edit distance score as a tensor

### Example

```
>>> from torchmetrics.functional import extended_edit_distance
>>> preds = ["this is the prediction", "here is an other sample"]
>>> target = ["this is the reference", "here is another one"]
>>> extended_edit_distance(preds=preds, target=target)
tensor(0.3078)
```

### References

[1] P. Stanchev, W. Wang, and H. Ney, “EED: Extended Edit Distance Measure for Machine Translation”, submitted to WMT 2019. [ExtendedEditDistance](#)

## 1.90 InfoLM

### 1.90.1 Module Interface

```
class torchmetrics.text.infolm.InfoLM(model_name_or_path='bert-base-uncased', temperature=0.25,
                                     information_measure='kl_divergence', idf=True, alpha=None,
                                     beta=None, device=None, max_length=None, batch_size=64,
                                     num_threads=0, verbose=True,
                                     return_sentence_level_score=False, **kwargs)
```

Calculate [InfoLM](#) [1] - i.e. calculate a distance/divergence between predicted and reference sentence discrete distribution using one of the following information measures:

- [KL divergence](#)
- [alpha divergence](#)
- [beta divergence](#)
- [AB divergence](#)
- [Rényi divergence](#)
- [L1 distance](#)
- [L2 distance](#)
- [L-infinity distance](#)
- [Fisher-Rao distance](#)

[InfoLM](#) is a family of untrained embedding-based metrics which addresses some famous flaws of standard string-based metrics thanks to the usage of pre-trained masked language models. This family of metrics is mainly designed for summarization and data-to-text tasks.

The implementation of this metric is fully based HuggingFace *transformers*’ package.

#### Parameters

- **model\_name\_or\_path** ([Union](#)[[str](#), [PathLike](#)]) – A name or a model path used to load *transformers* pretrained model. By default the “*bert-base-uncased*” model is used.
- **temperature** ([float](#)) – A temperature for calibrating language modelling. For more information, please reference [InfoLM](#) paper.
- **information\_measure** ([Literal](#)['kl\_divergence', 'alpha\_divergence', 'beta\_divergence', 'ab\_divergence', 'renyi\_divergence', 'l1\_distance', 'l2\_distance', 'l\_infinity\_distance', 'fisher\_rao\_distance']) – A name of information measure to be used. Please use one of: ['kl\_divergence', 'alpha\_divergence', 'beta\_divergence', 'ab\_divergence', 'renyi\_divergence', 'l1\_distance', 'l2\_distance', 'l\_infinity\_distance', 'fisher\_rao\_distance']
- **idf** ([bool](#)) – An indication of whether normalization using inverse document frequencies should be used.
- **alpha** ([Optional](#)[[float](#)]) – Alpha parameter of the divergence used for alpha, AB and Rényi divergence measures.
- **beta** ([Optional](#)[[float](#)]) – Beta parameter of the divergence used for beta and AB divergence measures.
- **device** ([Union](#)[[str](#), [device](#), [None](#)]) – A device to be used for calculation.
- **max\_length** ([Optional](#)[[int](#)]) – A maximum length of input sequences. Sequences longer than *max\_length* are to be trimmed.
- **batch\_size** ([int](#)) – A batch size used for model processing.
- **num\_threads** ([int](#)) – A number of threads to use for a dataloader.
- **verbose** ([bool](#)) – An indication of whether a progress bar to be displayed during the embeddings calculation.
- **return\_sentence\_level\_score** ([bool](#)) – An indication whether a sentence-level InfoLM score to be returned.

### Example

```
>>> from torchmetrics.text.infolm import InfoLM
>>> preds = ['he read the book because he was interested in world history']
>>> target = ['he was interested in world history because he read the book']
>>> infolm = InfoLM('google/bert_uncased_L-2_H-128_A-2', idf=False)
>>> infolm(preds, target)
tensor(-0.1784)
```

### References

[1] InfoLM: A New Metric to Evaluate Summarization & Data2Text Generation by Pierre Colombo, Chloé Clavel and Pablo Piantanida [InfoLM](#)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### **compute()**

Calculate selected information measure using the pre-trained language model.

**Return type** `Union[Tensor, Tuple[Tensor, Tensor]]`

**Returns** A corpus-level InfoLM score.

#### **update(preds, target)**

Update the metric state by a tokenization of `preds` and `target` sentences.

##### **Parameters**

- **preds** (`Union[str, Sequence[str]]`) –
- **corpus.** (*An `_sphinx_paramlinks_torchmetrics.text.infolm.InfoLM.update.iterable of hypothesis`*) –

**target:** An iterable of reference corpus.

**Return type** `None`

## 1.90.2 Functional Interface

```
torchmetrics.functional.text.infolm.infolm(preds, target, model_name_or_path='bert-base-uncased',
                                             temperature=0.25, information_measure='kl_divergence',
                                             idf=True, alpha=None, beta=None, device=None,
                                             max_length=None, batch_size=64, num_threads=0,
                                             verbose=True, return_sentence_level_score=False)
```

Calculate [InfoLM](#) [1] - i.e. calculate a distance/divergence between predicted and reference sentence discrete distribution using one of the following information measures:

- KL divergence
- alpha divergence
- beta divergence
- AB divergence
- Rényi divergence
- L1 distance

- L2 distance
- L-infinity distance
- Fisher-Rao distance

**InfoLM** is a family of untrained embedding-based metrics which addresses some famous flaws of standard string-based metrics thanks to the usage of pre-trained masked language models. This family of metrics is mainly designed for summarization and data-to-text tasks.

If you want to use IDF scaling over the whole dataset, please use the class metric.

The implementation of this metric is fully based HuggingFace *transformers* package.

#### Parameters

- **preds** (`Union[str, Sequence[str]]`) – An iterable of hypothesis corpus.
- **target** (`Union[str, Sequence[str]]`) – An iterable of reference corpus.
- **model\_name\_or\_path** (`Union[str, PathLike]`) – A name or a model path used to load *transformers* pretrained model.
- **temperature** (`float`) – A temperature for calibrating language modelling. For more information, please reference **InfoLM** paper.
- **information\_measure** (`Literal['kl_divergence', 'alpha_divergence', 'beta_divergence', 'ab_divergence', 'renyi_divergence', 'l1_distance', 'l2_distance', 'l_infinity_distance', 'fisher_rao_distance']`) – A name of information measure to be used. Please use one of: ['kl\_divergence', 'alpha\_divergence', 'beta\_divergence', 'ab\_divergence', 'renyi\_divergence', 'l1\_distance', 'l2\_distance', 'l\_infinity\_distance', 'fisher\_rao\_distance']
- **idf** (`bool`) – An indication of whether normalization using inverse document frequencies should be used.
- **alpha** (`Optional[float]`) – Alpha parameter of the divergence used for alpha, AB and Rényi divergence measures.
- **beta** (`Optional[float]`) – Beta parameter of the divergence used for beta and AB divergence measures.
- **device** (`Union[str, device, None]`) – A device to be used for calculation.
- **max\_length** (`Optional[int]`) – A maximum length of input sequences. Sequences longer than *max\_length* are to be trimmed.
- **batch\_size** (`int`) – A batch size used for model processing.
- **num\_threads** (`int`) – A number of threads to use for a dataloader.
- **verbose** (`bool`) – An indication of whether a progress bar to be displayed during the embeddings calculation.
- **return\_sentence\_level\_score** (`bool`) – An indication whether a sentence-level InfoLM score to be returned.

**Return type** `Union[Tensor, Tuple[Tensor, Tensor]]`

**Returns** A corpus-level InfoLM score. (Optionally) A list of sentence-level InfoLM scores if *return\_sentence\_level\_score=True*.

### Example

```
>>> from torchmetrics.functional.text.infolm import infolm
>>> preds = ['he read the book because he was interested in world history']
>>> target = ['he was interested in world history because he read the book']
>>> infolm(preds, target, model_name_or_path='google/bert_uncased_L-2_H-128_A-2',
↳idf=False)
tensor(-0.1784)
```

### References

[1] InfoLM: A New Metric to Evaluate Summarization & Data2Text Generation by Pierre Colombo, Chloé Clavel and Pablo Piantanida [InfoLM](#)

## 1.91 Match Error Rate

### 1.91.1 Module Interface

**class** torchmetrics.**MatchErrorRate**(\*\*kwargs)

Match Error Rate (**MER**) is a common metric of the performance of an automatic speech recognition system.

This value indicates the percentage of words that were incorrectly predicted and inserted. The lower the value, the better the performance of the ASR system with a MatchErrorRate of 0 being a perfect score. Match error rate can then be computed as:

$$mer = \frac{S + D + I}{N + I} = \frac{S + D + I}{S + D + C + I}$$

where:

- $S$  is the number of substitutions,
- $D$  is the number of deletions,
- $I$  is the number of insertions,
- $C$  is the number of correct words,
- $N$  is the number of words in the reference ( $N = S + D + C$ ).

**Parameters** **kwargs** (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

**Returns** Match error rate score



## Examples

```
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> metric = MatchErrorRate()
>>> metric(preds, target)
tensor(0.4444)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

### **compute()**

Calculate the Match error rate.

**Return type** `Tensor`

**Returns** Match error rate

### **update(preds, target)**

Store references/predictions for computing Match Error Rate scores.

#### **Parameters**

- **preds** (`Union[str, List[str]]`) – Transcription(s) to score as a string or list of strings
- **target** (`Union[str, List[str]]`) – Reference(s) for each speech input as a string or list of strings

**Return type** `None`

## 1.91.2 Functional Interface

`torchmetrics.functional.match_error_rate(preds, target)`

Match error rate is a metric of the performance of an automatic speech recognition system. This value indicates the percentage of words that were incorrectly predicted and inserted. The lower the value, the better the performance of the ASR system with a MatchErrorRate of 0 being a perfect score.

#### **Parameters**

- **preds** (`Union[str, List[str]]`) – Transcription(s) to score as a string or list of strings
- **target** (`Union[str, List[str]]`) – Reference(s) for each speech input as a string or list of strings

**Return type** `Tensor`

**Returns** Match error rate score

## Examples

```
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> match_error_rate(preds=preds, target=target)
tensor(0.4444)
```

## 1.92 Perplexity

### 1.92.1 Module Interface

**class** torchmetrics.text.perplexity.**Perplexity**(*ignore\_index=None, \*\*kwargs*)

Perplexity measures how well a language model predicts a text sample. It's calculated as the average number of bits per word a model needs to represent the sample.

#### Parameters

- **ignore\_index** ([Optional\[int\]](#)) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score.
- **kwargs** ([Dict\[str, Any\]](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Examples

```
>>> import torch
>>> preds = torch.rand(2, 8, 5, generator=torch.manual_seed(22))
>>> target = torch.randint(5, (2, 8), generator=torch.manual_seed(22))
>>> target[0, 6:] = -100
>>> metric = Perplexity(ignore_index=-100)
>>> metric(preds, target)
tensor(5.2545)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Compute the Perplexity.

**Return type** [Tensor](#)

**Returns** Perplexity

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

Compute and store intermediate statistics for Perplexity.

#### Parameters

- **preds** ([Tensor](#)) – Probabilities assigned to each token in a sequence with shape [batch\_size, seq\_len, vocab\_size].
- **target** ([Tensor](#)) – Ground truth values with a shape [batch\_size, seq\_len].

**Return type** [None](#)

## 1.92.2 Functional Interface

`torchmetrics.functional.text.perplexity.perplexity(preds, target, ignore_index=None)`

Perplexity measures how well a language model predicts a text sample. It's calculated as the average number of bits per word a model needs to represent the sample.

### Parameters

- **preds** (`Tensor`) – Probabilities assigned to each token in a sequence with shape `[batch_size, seq_len, vocab_size]`.
- **target** (`Tensor`) – Ground truth values with a shape `[batch_size, seq_len]`.
- **ignore\_index** (`Optional[int]`) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score.

**Return type** `Tensor`

**Returns** Perplexity value

### Examples

```
>>> import torch
>>> preds = torch.rand(2, 8, 5, generator=torch.manual_seed(22))
>>> target = torch.randint(5, (2, 8), generator=torch.manual_seed(22))
>>> target[0, 6:] = -100
>>> perplexity(preds, target, ignore_index=-100)
tensor(5.2545)
```

## 1.93 ROUGE Score

### 1.93.1 Module Interface

`class torchmetrics.text.rouge.ROUGEScore(use_stemmer=False, normalizer=None, tokenizer=None, accumulate='best', rouge_keys=('rouge1', 'rouge2', 'rougeL', 'rougeLsum'), **kwargs)`

Calculate Rouge Score, used for automatic summarization.

This implementation should imitate the behaviour of the *rouge-score* package *Python ROUGE Implementation*

### Parameters

- **use\_stemmer** (`bool`) – Use Porter stemmer to strip word suffixes to improve matching.
- **normalizer** (`Optional[Callable[[str], str]]`) – A user's own normalizer function. If this is `None`, replacing any non-alpha-numeric characters with spaces is default. This function must take a `str` and return a `str`.
- **tokenizer** (`Optional[Callable[[str], Sequence[str]]]`) – A user's own tokenizer function. If this is `None`, splitting by spaces is default. This function must take a `str` and return `Sequence[str]`.
- **accumulate** (`Literal['avg', 'best']`) – Useful in case of multi-reference rouge score.
  - `avg` takes the avg of all references with respect to predictions

- `best` takes the best fmeasure score obtained between prediction and multiple corresponding references.
- **rouge\_keys** (`Union[str, Tuple[str, ...]]`) – A list of rouge types to calculate. Keys that are allowed are `rougeL`, `rougeLsum`, and `rouge1` through `rouge9`.
- **kwargs** (`Any`) – Additional keyword arguments, see *Advanced metric settings* for more info.

### Example

```
>>> from torchmetrics.text.rouge import ROUGEScore
>>> preds = "My name is John"
>>> target = "Is your name John"
>>> rouge = ROUGEScore()
>>> from pprint import pprint
>>> pprint(rouge(preds, target))
{'rouge1_fmeasure': tensor(0.7500),
 'rouge1_precision': tensor(0.7500),
 'rouge1_recall': tensor(0.7500),
 'rouge2_fmeasure': tensor(0.),
 'rouge2_precision': tensor(0.),
 'rouge2_recall': tensor(0.),
 'rougeL_fmeasure': tensor(0.5000),
 'rougeL_precision': tensor(0.5000),
 'rougeL_recall': tensor(0.5000),
 'rougeLsum_fmeasure': tensor(0.5000),
 'rougeLsum_precision': tensor(0.5000),
 'rougeLsum_recall': tensor(0.5000)}
```

### Raises

- **ValueError** – If the python packages `nltk` is not installed.
- **ValueError** – If any of the `rouge_keys` does not belong to the allowed set of keys.

### References

[1] ROUGE: A Package for Automatic Evaluation of Summaries by Chin-Yew Lin [Rouge Detail](#)

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

#### **compute()**

Calculate (Aggregate and provide confidence intervals) ROUGE score.

**Return type** `Dict[str, Tensor]`

**Returns** Python dictionary of rouge scores for each input rouge key.

#### **update(preds, target)**

Compute rouge scores.

#### **Parameters**

- **preds** (`Union[str, Sequence[str]]`) – An iterable of predicted sentences or a single predicted sentence.

- **target** (`Union[str, Sequence[str], Sequence[Sequence[str]]]`) – An iterable of target sentences or an iterable of target sentences or a single target sentence.

**Return type** `None`

### 1.93.2 Functional Interface

`torchmetrics.functional.text.rouge.rouge_score(preds, target, accumulate='best', use_stemmer=False, normalizer=None, tokenizer=None, rouge_keys=('rouge1', 'rouge2', 'rougeL', 'rougeLsum'))`

Calculate [Calculate Rouge Score](#) , used for automatic summarization.

#### Parameters

- **preds** (`Union[str, Sequence[str]]`) – An iterable of predicted sentences or a single predicted sentence.
- **target** (`Union[str, Sequence[str], Sequence[Sequence[str]]]`) – An iterable of iterables of target sentences or an iterable of target sentences or a single target sentence.
- **accumulate** (`Literal['avg', 'best']`) – Useful incase of multi-reference rouge score.
  - avg takes the avg of all references with respect to predictions
  - best takes the best fmeasure score obtained between prediction and multiple corresponding references.
- **use\_stemmer** (`bool`) – Use Porter stemmer to strip word suffixes to improve matching.
- **normalizer** (`Optional[Callable[[str], str]]`) – A user's own normalizer function. If this is `None`, replacing any non-alpha-numeric characters with spaces is default. This function must take a `str` and return a `str`.
- **tokenizer** (`Optional[Callable[[str], Sequence[str]]]`) – A user's own tokenizer function. If this is `None`, splitting by spaces is default This function must take a `str` and return `Sequence[str]`
- **rouge\_keys** (`Union[str, Tuple[str, ...]]`) – A list of rouge types to calculate. Keys that are allowed are `rougeL`, `rougeLsum`, and `rouge1` through `rouge9`.

**Return type** `Dict[str, Tensor]`

**Returns** Python dictionary of rouge scores for each input rouge key.

#### Example

```
>>> from torchmetrics.functional.text.rouge import rouge_score
>>> preds = "My name is John"
>>> target = "Is your name John"
>>> from pprint import pprint
>>> pprint(rouge_score(preds, target))
{'rouge1_fmeasure': tensor(0.7500),
 'rouge1_precision': tensor(0.7500),
 'rouge1_recall': tensor(0.7500),
 'rouge2_fmeasure': tensor(0.),
 'rouge2_precision': tensor(0.),
 'rouge2_recall': tensor(0.),
```

(continues on next page)

(continued from previous page)

```
'rougeL_fmeasure': tensor(0.5000),
'rougeL_precision': tensor(0.5000),
'rougeL_recall': tensor(0.5000),
'rougeLsum_fmeasure': tensor(0.5000),
'rougeLsum_precision': tensor(0.5000),
'rougeLsum_recall': tensor(0.5000)}
```

**Raises**

- **ModuleNotFoundError** – If the python package `nltk` is not installed.
- **ValueError** – If any of the `rouge_keys` does not belong to the allowed set of keys.

**References**

[1] ROUGE: A Package for Automatic Evaluation of Summaries by Chin-Yew Lin. <https://aclanthology.org/W04-1013/>

## 1.94 Sacre BLEU Score

### 1.94.1 Module Interface

```
class torchmetrics.SacreBLEUScore(n_gram=4, smooth=False, tokenize='13a', lowercase=False,
                                   weights=None, **kwargs)
```

Calculate **BLEU score** [1] of machine translated text with one or more references. This implementation follows the behaviour of SacreBLEU [2] implementation from <https://github.com/mjpost/sacrebleu>.

The SacreBLEU implementation differs from the NLTK BLEU implementation in tokenization techniques.

**Parameters**

- **n\_gram** (`int`) – Gram value ranged from 1 to 4
- **smooth** (`bool`) – Whether to apply smoothing, see [2]
- **tokenize** (`Literal`['none', '13a', 'zh', 'intl', 'char']) – Tokenization technique to be used. Supported tokenization: ['none', '13a', 'zh', 'intl', 'char']
- **lowercase** (`bool`) – If True, BLEU score over lowercased text is calculated.
- **kwargs** (`Any`) – Additional keyword arguments, see *Advanced metric settings* for more info.
- **weights** (`Optional`[`Sequence`[`float`]]) –

**Weights used for unigrams, bigrams, etc. to calculate BLEU score.** If not provided, uniform weights are used.

**Raises:**

**ValueError:** If `tokenize` not one of 'none', '13a', 'zh', 'intl' or 'char'

**ValueError:** If `tokenize` is set to 'intl' and `regex` is not installed

**ValueError:** If a length of a list of weights is not None and not equal to `n_gram`.

## Example

```
>>> from torchmetrics import SacreBLEUScore
>>> preds = ['the cat is on the mat']
>>> target = [['there is a cat on the mat', 'a cat is on the mat']]
>>> metric = SacreBLEUScore()
>>> metric(preds, target)
tensor(0.7598)
```

## References

[1] BLEU: a Method for Automatic Evaluation of Machine Translation by Papineni, Kishore, Salim Roukos, Todd Ward, and Wei-Jing Zhu [BLEU](#)

[2] A Call for Clarity in Reporting BLEU Scores by Matt Post.

[3] Automatic Evaluation of Machine Translation Quality Using Longest Common Subsequence and Skip-Bigram Statistics by Chin-Yew Lin and Franz Josef Och [Machine Translation Evolution](#)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

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

Compute Precision Scores.

### Parameters

- **preds** ([Sequence](#)[[str](#)]) – An iterable of machine translated corpus
- **target** ([Sequence](#)[[Sequence](#)[[str](#)]]) – An iterable of iterables of reference corpus

**Return type** [None](#)

## 1.94.2 Functional Interface

`torchmetrics.functional.sacre_bleu_score(preds, target, n_gram=4, smooth=False, tokenize='l3a', lowercase=False, weights=None)`

Calculate [BLEU score](#) [1] of machine translated text with one or more references. This implementation follows the behaviour of SacreBLEU [2] implementation from <https://github.com/mjpost/sacrebleu>.

### Parameters

- **preds** ([Sequence](#)[[str](#)]) – An iterable of machine translated corpus
- **target** ([Sequence](#)[[Sequence](#)[[str](#)]]) – An iterable of iterables of reference corpus
- **n\_gram** ([int](#)) – Gram value ranged from 1 to 4
- **smooth** ([bool](#)) – Whether to apply smoothing – see [2]
- **tokenize** ([Literal](#)['none', 'l3a', 'zh', 'intl', 'char']) – Tokenization technique to be used. Supported tokenization: ['none', 'l3a', 'zh', 'intl', 'char']
- **lowercase** ([bool](#)) – If [True](#), BLEU score over lowercased text is calculated.
- **weights** ([Optional](#)[[Sequence](#)[[float](#)]]) – Weights used for unigrams, bigrams, etc. to calculate BLEU score. If not provided, uniform weights are used.

**Return type** [Tensor](#)

**Returns** Tensor with BLEU Score

**Raises**

- **ValueError** – If preds and target corpus have different lengths.
- **ValueError** – If a length of a list of weights is not None and not equal to n\_gram.

**Example**

```
>>> from torchmetrics.functional import sacre_bleu_score
>>> preds = ['the cat is on the mat']
>>> target = [['there is a cat on the mat', 'a cat is on the mat']]
>>> sacre_bleu_score(preds, target)
tensor(0.7598)
```

**References**

- [1] BLEU: a Method for Automatic Evaluation of Machine Translation by Papineni, Kishore, Salim Roukos, Todd Ward, and Wei-Jing Zhu [BLEU](#)
- [2] A Call for Clarity in Reporting BLEU Scores by Matt Post.
- [3] Automatic Evaluation of Machine Translation Quality Using Longest Common Subsequence and Skip-Bigram Statistics by Chin-Yew Lin and Franz Josef Och [Machine Translation Evolution](#)

## 1.95 SQuAD

### 1.95.1 Module Interface

**class** torchmetrics.SQuAD(\*\*kwargs)

Calculate [SQuAD Metric](#) which corresponds to the scoring script for version 1 of the Stanford Question Answering Dataset (SQuAD).

**Parameters** **kwargs** ([Any](#)) – Additional keyword arguments, see *Advanced metric settings* for more info.

**Example**

```
>>> from torchmetrics import SQuAD
>>> preds = [{"prediction_text": "1976", "id": "56e10a3be3433e1400422b22"}]
>>> target = [{"answers": {"answer_start": [97], "text": ["1976"]}, "id":
↪ "56e10a3be3433e1400422b22"}]
>>> squad = SQuAD()
>>> squad(preds, target)
{'exact_match': tensor(100.), 'f1': tensor(100.)}
```



## References

[1] SQuAD: 100,000+ Questions for Machine Comprehension of Text by Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, Percy Liang [SQuAD Metric](#) .

Initializes internal Module state, shared by both nn.Module and ScriptModule.

### `compute()`

Aggregate the F1 Score and Exact match for the batch.

**Return type** `Dict[str, Tensor]`

**Returns** Dictionary containing the F1 score, Exact match score for the batch.

### `update(preds, target)`

Compute F1 Score and Exact Match for a collection of predictions and references.

#### Parameters

- **preds** (`Union[Dict[str, str], List[Dict[str, str]]]`) – A Dictionary or List of Dictionary-s that map `id` and `prediction_text` to the respective values. Example prediction:

```
{"prediction_text": "TorchMetrics is awesome", "id": "123"}
```

- **target** (`Union[Dict[str, Union[str, Dict[str, Union[List[str], List[int]]]]], List[Dict[str, Union[str, Dict[str, Union[List[str], List[int]]]]]]]`) – A Dictionary or List of Dictionary-s that contain the `answers` and `id` in the SQuAD Format. Example target:

```
{
  'answers': [{'answer_start': [1], 'text': ['This is a test_
↪answer']}],
  'id': '1',
}
```

Reference SQuAD Format:

```
{
  'answers': {'answer_start': [1], 'text': ['This is a test_
↪text']},
  'context': 'This is a test context.',
  'id': '1',
  'question': 'Is this a test?',
  'title': 'train test'
}
```

**Raises** `KeyError` – If the required keys are missing in either predictions or targets.

**Return type** `None`

## 1.95.2 Functional Interface

`torchmetrics.functional.squad(preds, target)`

Calculate SQuAD Metric .

### Parameters

- **preds** (`Union[Dict[str, str], List[Dict[str, str]]]`) – A Dictionary or List of Dictionary-s that map *id* and *prediction\_text* to the respective values.

Example prediction:

```
{"prediction_text": "TorchMetrics is awesome", "id": "123"}
```

- **target** (`Union[Dict[str, Union[str, Dict[str, Union[List[str], List[int]]]]], List[Dict[str, Union[str, Dict[str, Union[List[str], List[int]]]]]]]`) – A Dictionary or List of Dictionary-s that contain the *answers* and *id* in the SQuAD Format.

Example target:

```
{
  'answers': [{'answer_start': [1], 'text': ['This is a test_
↪answer']}]},
  'id': '1',
}
```

Reference SQuAD Format:

```
{
  'answers': {'answer_start': [1], 'text': ['This is a test text
↪']},
  'context': 'This is a test context.',
  'id': '1',
  'question': 'Is this a test?',
  'title': 'train test'
}
```

**Return type** `Dict[str, Tensor]`

**Returns** Dictionary containing the F1 score, Exact match score for the batch.

### Example

```
>>> from torchmetrics.functional.text.squad import squad
>>> preds = [{"prediction_text": "1976", "id": "56e10a3be3433e1400422b22"}]
>>> target = [{"answers": {"answer_start": [97], "text": ["1976"]}, "id":
↪ "56e10a3be3433e1400422b22"}]
>>> squad(preds, target)
{'exact_match': tensor(100.), 'f1': tensor(100.)}
```

**Raises** `KeyError` – If the required keys are missing in either predictions or targets.

## References

[1] SQuAD: 100,000+ Questions for Machine Comprehension of Text by Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, Percy Liang [SQuAD Metric](#) .

## 1.96 Translation Edit Rate (TER)

### 1.96.1 Module Interface

```
class torchmetrics.TranslationEditRate(normalize=False, no_punctuation=False, lowercase=True,
                                       asian_support=False, return_sentence_level_score=False,
                                       **kwargs)
```

Calculate Translation edit rate (TER) of machine translated text with one or more references.

This implementation follows the implmenetaions from <https://github.com/mjpost/sacrebleu/blob/master/sacrebleu/metrics/ter.py>. The *sacrebleu* implmenetation is a near-exact reimplementaion of the Tercom algorithm, produces identical results on all “sane” outputs.

#### Parameters

- **normalize** (*bool*) – An indication whether a general tokenization to be applied.
- **no\_punctuation** (*bool*) – An indication whteher a punctuation to be removed from the sentences.
- **lowercase** (*bool*) – An indication whether to enable case-insesitivity.
- **asian\_support** (*bool*) – An indication whether asian characters to be processed.
- **return\_sentence\_level\_score** (*bool*) – An indication whether a sentence-level TER to be returned.
- **kwargs** (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

#### Example

```
>>> preds = ['the cat is on the mat']
>>> target = [['there is a cat on the mat', 'a cat is on the mat']]
>>> metric = TranslationEditRate()
>>> metric(preds, target)
tensor(0.1538)
```

## References

[1] A Study of Translation Edit Rate with Targeted Human Annotation by Mathew Snover, Bonnie Dorr, Richard Schwartz, Linnea Micciulla and John Makhoul [TER](#)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Calculate the translate error rate (TER).

**Return type** `Union[Tensor, Tuple[Tensor, Tensor]]`

**Returns** A corpus-level translation edit rate (TER). (Optionally) A list of sentence-level translation\_edit\_rate (TER) if `return_sentence_level_score=True`.

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

Update TER statistics.

**Parameters**

- **preds** (`Union[str, Sequence[str]]`) – An iterable of hypothesis corpus.
- **target** (`Sequence[Union[str, Sequence[str]]]`) – An iterable of iterables of reference corpus.

**Return type** `None`

## 1.96.2 Functional Interface

`torchmetrics.functional.translation_edit_rate`(*preds*, *target*, *normalize=False*, *no\_punctuation=False*, *lowercase=True*, *asian\_support=False*, *return\_sentence\_level\_score=False*)

Calculate Translation edit rate (TER) of machine translated text with one or more references. This implementation follows the implmenetaions from <https://github.com/mjpost/sacrebleu/blob/master/sacrebleu/metrics/ter.py>. The *sacrebleu* implmenetation is a near-exact reimplementaion of the Tercom algorithm, produces identical results on all “sane” outputs.

**Parameters**

- **preds** (`Union[str, Sequence[str]]`) – An iterable of hypothesis corpus.
- **target** (`Sequence[Union[str, Sequence[str]]]`) – An iterable of iterables of reference corpus.
- **normalize** (`bool`) – An indication whether a general tokenization to be applied.
- **no\_punctuation** (`bool`) – An indication whteher a punctuation to be removed from the sentences.
- **lowercase** (`bool`) – An indication whether to enable case-insesitivity.
- **asian\_support** (`bool`) – An indication whether asian characters to be processed.
- **return\_sentence\_level\_score** (`bool`) – An indication whether a sentence-level TER to be returned.

**Return type** `Union[Tensor, Tuple[Tensor, List[Tensor]]]`

**Returns** A corpus-level translation edit rate (TER). (Optionally) A list of sentence-level translation\_edit\_rate (TER) if `return_sentence_level_score=True`.

### Example

```
>>> preds = ['the cat is on the mat']
>>> target = [['there is a cat on the mat', 'a cat is on the mat']]
>>> translation_edit_rate(preds, target)
tensor(0.1538)
```

## References

[1] A Study of Translation Edit Rate with Targeted Human Annotation by Mathew Snover, Bonnie Dorr, Richard Schwartz, Linnea Micciulla and John Makhoul [TER](#)

## 1.97 Word Error Rate

### 1.97.1 Module Interface

**class** torchmetrics.**WordErrorRate**(\*\*kwargs)

Word error rate ([WordErrorRate](#)) is a common metric of the performance of an automatic speech recognition system. This value indicates the percentage of words that were incorrectly predicted. The lower the value, the better the performance of the ASR system with a WER of 0 being a perfect score. Word error rate can then be computed as:

$$WER = \frac{S + D + I}{N} = \frac{S + D + I}{S + D + C}$$

where:

- $S$  is the number of substitutions,
- $D$  is the number of deletions,
- $I$  is the number of insertions,
- $C$  is the number of correct words,
- $N$  is the number of words in the reference ( $N = S + D + C$ ).

Compute WER score of transcribed segments against references.

**Parameters** **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Returns** Word error rate score

### Examples

```
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> metric = WordErrorRate()
>>> metric(preds, target)
tensor(0.5000)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Calculate the word error rate.

**Return type** [Tensor](#)

**Returns** Word error rate score

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

Store references/predictions for computing Word Error Rate scores.

**Parameters**

- **preds** (`Union[str, List[str]]`) – Transcription(s) to score as a string or list of strings
- **target** (`Union[str, List[str]]`) – Reference(s) for each speech input as a string or list of strings

**Return type** `None`

## 1.97.2 Functional Interface

`torchmetrics.functional.word_error_rate(preds, target)`

Word error rate ([WordErrorRate](#)) is a common metric of the performance of an automatic speech recognition system. This value indicates the percentage of words that were incorrectly predicted. The lower the value, the better the performance of the ASR system with a WER of 0 being a perfect score.

**Parameters**

- **preds** (`Union[str, List[str]]`) – Transcription(s) to score as a string or list of strings
- **target** (`Union[str, List[str]]`) – Reference(s) for each speech input as a string or list of strings

**Return type** `Tensor`

**Returns** Word error rate score

### Examples

```
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> word_error_rate(preds=preds, target=target)
tensor(0.5000)
```

## 1.98 Word Info. Lost

### 1.98.1 Module Interface

**class** `torchmetrics.WordInfoLost(**kwargs)`

Word Information Lost ([WIL](#)) is a metric of the performance of an automatic speech recognition system. This value indicates the percentage of words that were incorrectly predicted between a set of ground-truth sentences and a set of hypothesis sentences. The lower the value, the better the performance of the ASR system with a WordInfoLost of 0 being a perfect score. Word Information Lost rate can then be computed as:

$$wil = 1 - \frac{C}{N} + \frac{C}{P}$$

where:

- *C* is the number of correct words,
- *N* is the number of words in the reference

- $P$  is the number of words in the prediction

**Parameters** **kwargs** (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

## Examples

```
>>> from torchmetrics import WordInfoLost
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> metric = WordInfoLost()
>>> metric(preds, target)
tensor(0.6528)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Calculate the Word Information Lost.

**Return type** `Tensor`

**Returns** Word Information Lost score

**update(preds, target)**

Store predictions/references for computing Word Information Lost scores.

**Parameters**

- **preds** (`Union[str, List[str]]`) – Transcription(s) to score as a string or list of strings
- **target** (`Union[str, List[str]]`) – Reference(s) for each speech input as a string or list of strings

**Return type** `None`

## 1.98.2 Functional Interface

`torchmetrics.functional.word_information_lost(preds, target)`

Word Information Lost rate is a metric of the performance of an automatic speech recognition system. This value indicates the percentage of characters that were incorrectly predicted. The lower the value, the better the performance of the ASR system with a Word Information Lost rate of 0 being a perfect score.

**Parameters**

- **preds** (`Union[str, List[str]]`) – Transcription(s) to score as a string or list of strings
- **target** (`Union[str, List[str]]`) – Reference(s) for each speech input as a string or list of strings

**Return type** `Tensor`

**Returns** Word Information Lost rate

## Examples

```
>>> from torchmetrics.functional import word_information_lost
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> word_information_lost(preds, target)
tensor(0.6528)
```

## 1.99 Word Info. Preserved

### 1.99.1 Module Interface

**class** torchmetrics.**WordInfoPreserved**(\*\*kwargs)

Word Information Preserved (WIP) is a metric of the performance of an automatic speech recognition system. This value indicates the percentage of words that were correctly predicted between a set of ground-truth sentences and a set of hypothesis sentences. The higher the value, the better the performance of the ASR system with a WordInfoPreserved of 0 being a perfect score. Word Information Preserved rate can then be computed as:

$$wip = \frac{C}{N} + \frac{C}{P}$$

where:

- $C$  is the number of correct words,
- $N$  is the number of words in the reference
- $P$  is the number of words in the prediction

**Parameters** **kwargs** (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

## Examples

```
>>> from torchmetrics import WordInfoPreserved
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> metric = WordInfoPreserved()
>>> metric(preds, target)
tensor(0.3472)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute**()

Calculate the word Information Preserved.

**Return type** *Tensor*

**Returns** word Information Preserved score

**update**(preds, target)

Store predictions/references for computing word Information Preserved scores.

**Parameters**



- **preds** (`Union[str, List[str]]`) – Transcription(s) to score as a string or list of strings
- **target** (`Union[str, List[str]]`) – Reference(s) for each speech input as a string or list of strings

**Return type** `None`

## 1.99.2 Functional Interface

`torchmetrics.functional.word_information_preserved(preds, target)`

Word Information Preserved rate is a metric of the performance of an automatic speech recognition system. This value indicates the percentage of characters that were incorrectly predicted. The lower the value, the better the performance of the ASR system with a Word Information preserved rate of 0 being a perfect score.

### Parameters

- **preds** (`Union[str, List[str]]`) – Transcription(s) to score as a string or list of strings
- **target** (`Union[str, List[str]]`) – Reference(s) for each speech input as a string or list of strings

**Return type** `Tensor`

**Returns** Word Information preserved rate

### Examples

```
>>> from torchmetrics.functional import word_information_preserved
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> word_information_preserved(preds, target)
tensor(0.3472)
```

## 1.100 Concatenation

### 1.100.1 Module Interface

`class torchmetrics.CatMetric(nan_strategy='warn', **kwargs)`

Concatenate a stream of values.

### Parameters

- **nan\_strategy** (`Union[str, float]`) – options: - 'error': if any *nan* values are encountered will give a `RuntimeError` - 'warn': if any *nan* values are encountered will give a warning and continue - 'ignore': all *nan* values are silently removed - a float: if a float is provided will impute any *nan* values with this value
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises** `ValueError` – If `nan_strategy` is not one of `error`, `warn`, `ignore` or a float

### Example

```
>>> from torchmetrics import CatMetric
>>> metric = CatMetric()
>>> metric.update(1)
>>> metric.update(torch.tensor([2, 3]))
>>> metric.compute()
tensor([1., 2., 3.]
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### **compute()**

Compute the aggregated value.

**Return type** `Tensor`

#### **update(value)**

Update state with data.

**Parameters** **value** (`Union[float, Tensor]`) – Either a float or tensor containing data. Additional tensor dimensions will be flattened

**Return type** `None`

## 1.101 Maximum

### 1.101.1 Module Interface

**class** `torchmetrics.MaxMetric(nan_strategy='warn', **kwargs)`

Aggregate a stream of value into their maximum value.

#### **Parameters**

- **nan\_strategy** (`Union[str, float]`) – options: - 'error': if any *nan* values are encountered will give a `RuntimeError` - 'warn': if any *nan* values are encountered will give a warning and continue - 'ignore': all *nan* values are silently removed - a float: if a float is provided will impude any *nan* values with this value
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises** `ValueError` – If `nan_strategy` is not one of `error`, `warn`, `ignore` or a float

### Example

```
>>> from torchmetrics import MaxMetric
>>> metric = MaxMetric()
>>> metric.update(1)
>>> metric.update(torch.tensor([2, 3]))
>>> metric.compute()
tensor(3.)
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**update**(*value*)

Update state with data.

**Parameters** **value** (`Union[float, Tensor]`) – Either a float or tensor containing data. Additional tensor dimensions will be flattened

**Return type** `None`

## 1.102 Mean

### 1.102.1 Module Interface

**class** `torchmetrics.MeanMetric`(*nan\_strategy='warn', \*\*kwargs*)

Aggregate a stream of value into their mean value.

**Parameters** **nan\_strategy** (`Union[str, float]`) –

**options:**

- 'error': if any *nan* values are encountered will give a `RuntimeError`
- 'warn': if any *nan* values are encountered will give a warning and continue
- 'ignore': all *nan* values are silently removed
- a float: if a float is provided will impute any *nan* values with this value

**kwargs:** Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises** `ValueError` – If *nan\_strategy* is not one of `error`, `warn`, `ignore` or a float

#### Example

```
>>> from torchmetrics import MeanMetric
>>> metric = MeanMetric()
>>> metric.update(1)
>>> metric.update(torch.tensor([2, 3]))
>>> metric.compute()
tensor(2.)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

**compute**()

Compute the aggregated value.

**Return type** `Tensor`

**update**(*value, weight=1.0*)

Update state with data.

**Parameters**

- **value** (`Union[float, Tensor]`) – Either a float or tensor containing data. Additional tensor dimensions will be flattened
- **weight** (`Union[float, Tensor]`) – Either a float or tensor containing weights for calculating the average. Shape of weight should be able to broadcast with the shape of *value*. Default to *1.0* corresponding to simple harmonic average.

Return type `None`

## 1.103 Minimum

### 1.103.1 Module Interface

**class** `torchmetrics.MinMetric`(*nan\_strategy*='warn', *\*\*kwargs*)

Aggregate a stream of value into their minimum value.

#### Parameters

- **nan\_strategy** (`Union[str, float]`) – options: - 'error': if any *nan* values are encountered will give a `RuntimeError` - 'warn': if any *nan* values are encountered will give a warning and continue - 'ignore': all *nan* values are silently removed - a float: if a float is provided will impude any *nan* values with this value
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises** `ValueError` – If *nan\_strategy* is not one of error, warn, ignore or a float

#### Example

```
>>> from torchmetrics import MinMetric
>>> metric = MinMetric()
>>> metric.update(1)
>>> metric.update(torch.tensor([2, 3]))
>>> metric.compute()
tensor(1.)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

**update**(*value*)

Update state with data.

**Parameters** **value** (`Union[float, Tensor]`) – Either a float or tensor containing data. Additional tensor dimensions will be flattened

**Return type** `None`

## 1.104 Sum

### 1.104.1 Module Interface

**class** `torchmetrics.SumMetric`(*nan\_strategy*='warn', *\*\*kwargs*)

Aggregate a stream of value into their sum.

#### Parameters

- **nan\_strategy** (`Union[str, float]`) – options: - 'error': if any *nan* values are encountered will give a `RuntimeError` - 'warn': if any *nan* values are encountered will give a warning and continue - 'ignore': all *nan* values are silently removed - a float: if a float is provided will impude any *nan* values with this value

- **kwargs** (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

Raises **ValueError** – If `nan_strategy` is not one of `error`, `warn`, `ignore` or a float

### Example

```
>>> from torchmetrics import SumMetric
>>> metric = SumMetric()
>>> metric.update(1)
>>> metric.update(torch.tensor([2, 3]))
>>> metric.compute()
tensor(6.)
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

**update**(*value*)

Update state with data.

**Parameters** **value** (`Union[float, Tensor]`) – Either a float or tensor containing data. Additional tensor dimensions will be flattened

**Return type** `None`

## 1.105 Bootstrapper

### 1.105.1 Module Interface

**class** `torchmetrics.BootStrapper`(*base\_metric*, *num\_bootstraps=10*, *mean=True*, *std=True*, *quantile=None*, *raw=False*, *sampling\_strategy='poisson'*, *\*\*kwargs*)

Using [Turn a Metric into a Bootstrapped](#)

That can automate the process of getting confidence intervals for metric values. This wrapper class basically keeps multiple copies of the same base metric in memory and whenever `update` or `forward` is called, all input tensors are resampled (with replacement) along the first dimension.

#### Parameters

- **base\_metric** (*Metric*) – base metric class to wrap
- **num\_bootstraps** (*int*) – number of copies to make of the base metric for bootstrapping
- **mean** (*bool*) – if `True` return the mean of the bootstraps
- **std** (*bool*) – if `True` return the standard deviation of the bootstraps
- **quantile** (`Union[float, Tensor, None]`) – if given, returns the quantile of the bootstraps. Can only be used with pytorch version 1.6 or higher
- **raw** (*bool*) – if `True`, return all bootstrapped values
- **sampling\_strategy** (*str*) – Determines how to produce bootstrapped samplings. Either `'poisson'` or `'multinomial'`. If `'poisson'` is chosen, the number of times each sample will be included in the bootstrap will be given by  $n \sim \text{Poisson}(\lambda = 1)$ , which approximates the true bootstrap distribution when the number of samples is large. If `'multinomial'` is chosen, we will apply true bootstrapping at the batch level to approximate bootstrapping over the whole dataset.

- **kwargs** (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

Example::

```
>>> from pprint import pprint
>>> from torchmetrics import Accuracy, BootStrapper
>>> _ = torch.manual_seed(123)
>>> base_metric = Accuracy()
>>> bootstrap = BootStrapper(base_metric, num_bootstraps=20)
>>> bootstrap.update(torch.randint(5, (20,)), torch.randint(5, (20,)))
>>> output = bootstrap.compute()
>>> pprint(output)
{'mean': tensor(0.2205), 'std': tensor(0.0859)}
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**compute()**

Computes the bootstrapped metric values.

Always returns a dict of tensors, which can contain the following keys: mean, std, quantile and raw depending on how the class was initialized.

**Return type** `Dict[str, Tensor]`

**update(\*args, \*\*kwargs)**

Updates the state of the base metric.

Any tensor passed in will be bootstrapped along dimension 0.

**Return type** `None`

## 1.106 Classwise Wrapper

### 1.106.1 Module Interface

**class** `torchmetrics.ClasswiseWrapper(metric, labels=None)`

Wrapper class for altering the output of classification metrics that returns multiple values to include label information.

**Parameters**

- **metric** (*Metric*) – base metric that should be wrapped. It is assumed that the metric outputs a single tensor that is split along the first dimension.
- **labels** (*Optional[List[str]]*) – list of strings indicating the different classes.

### Example

```
>>> import torch
>>> _ = torch.manual_seed(42)
>>> from torchmetrics import Accuracy, ClasswiseWrapper
>>> metric = ClasswiseWrapper(Accuracy(num_classes=3, average=None))
>>> preds = torch.randn(10, 3).softmax(dim=-1)
>>> target = torch.randint(3, (10,))
>>> metric(preds, target)
{'accuracy_0': tensor(0.5000), 'accuracy_1': tensor(0.7500), 'accuracy_2': tensor(0.
  ↪ )}
```

### Example (labels as list of strings):

```
>>> import torch
>>> from torchmetrics import Accuracy, ClasswiseWrapper
>>> metric = ClasswiseWrapper(
...     Accuracy(num_classes=3, average=None),
...     labels=["horse", "fish", "dog"]
... )
>>> preds = torch.randn(10, 3).softmax(dim=-1)
>>> target = torch.randint(3, (10,))
>>> metric(preds, target)
{'accuracy_horse': tensor(0.3333), 'accuracy_fish': tensor(0.6667), 'accuracy_
  ↪ dog': tensor(0.)}
```

### Example (in metric collection):

```
>>> import torch
>>> from torchmetrics import Accuracy, ClasswiseWrapper, MetricCollection,
  ↪ Recall
>>> labels = ["horse", "fish", "dog"]
>>> metric = MetricCollection(
...     {'accuracy': ClasswiseWrapper(Accuracy(num_classes=3, average=None),
  ↪ labels),
...     'recall': ClasswiseWrapper(Recall(num_classes=3, average=None),
  ↪ labels)}
... )
>>> preds = torch.randn(10, 3).softmax(dim=-1)
>>> target = torch.randint(3, (10,))
>>> metric(preds, target)
{'accuracy_horse': tensor(0.), 'accuracy_fish': tensor(0.3333), 'accuracy_dog':
  ↪ tensor(0.4000),
 'recall_horse': tensor(0.), 'recall_fish': tensor(0.3333), 'recall_dog':
  ↪ tensor(0.4000)}
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

#### compute()

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** `Dict[str, Tensor]`

**forward**(\*args, \*\*kwargs)

forward serves the dual purpose of both computing the metric on the current batch of inputs but also add the batch statistics to the overall accumulating metric state.

Input arguments are the exact same as corresponding update method. The returned output is the exact same as the output of compute.

**Return type** Any

**reset**()

This method automatically resets the metric state variables to their default value.

**Return type** None

**update**(\*args, \*\*kwargs)

Override this method to update the state variables of your metric class.

**Return type** None

## 1.107 Metric Tracker

### 1.107.1 Module Interface

**class** torchmetrics.MetricTracker(metric, maximize=True)

A wrapper class that can help keeping track of a metric or metric collection over time and implement useful methods. The wrapper implements the standard .update(), .compute(), .reset() methods that just calls corresponding method of the currently tracked metric. However, the following additional methods are provided:

-MetricTracker.n\_steps: number of metrics being tracked  
 -MetricTracker.increment(): initialize a new metric for being tracked  
 -MetricTracker.compute\_all(): get the metric value for all steps  
 -MetricTracker.best\_metric(): returns the best value

#### Parameters

- **metric** (Union[Metric, MetricCollection]) – instance of a torchmetrics.Metric or torchmetrics.MetricCollection to keep track of at each timestep.
- **maximize** (Union[bool, List[bool]]) – either single bool or list of bool indicating if higher metric values are better (True) or lower is better (False).

#### Example (single metric):

```
>>> from torchmetrics import Accuracy, MetricTracker
>>> _ = torch.manual_seed(42)
>>> tracker = MetricTracker(Accuracy(num_classes=10))
>>> for epoch in range(5):
...     tracker.increment()
...     for batch_idx in range(5):
...         preds, target = torch.randint(10, (100,)), torch.randint(10, (100,
↪))
...         tracker.update(preds, target)
...         print(f"current acc={tracker.compute()}")
current acc=0.1120000034570694
current acc=0.08799999952316284
current acc=0.12600000202655792
```

(continues on next page)



(continued from previous page)

```

current acc=0.07999999821186066
current acc=0.10199999809265137
>>> best_acc, which_epoch = tracker.best_metric(return_step=True)
>>> best_acc
0.1260...
>>> which_epoch
2
>>> tracker.compute_all()
tensor([0.1120, 0.0880, 0.1260, 0.0800, 0.1020])

```

**Example (multiple metrics using MetricCollection):**

```

>>> from torchmetrics import MetricTracker, MetricCollection, MeanSquaredError,
    ↳ ExplainedVariance
>>> _ = torch.manual_seed(42)
>>> tracker = MetricTracker(MetricCollection([MeanSquaredError(),
    ↳ ExplainedVariance()]), maximize=[False, True])
>>> for epoch in range(5):
...     tracker.increment()
...     for batch_idx in range(5):
...         preds, target = torch.randn(100), torch.randn(100)
...         tracker.update(preds, target)
...         print(f"current stats={tracker.compute()}")
current stats={'MeanSquaredError': tensor(1.8218), 'ExplainedVariance':
    ↳ tensor(-0.8969)}
current stats={'MeanSquaredError': tensor(2.0268), 'ExplainedVariance':
    ↳ tensor(-1.0206)}
current stats={'MeanSquaredError': tensor(1.9491), 'ExplainedVariance':
    ↳ tensor(-0.8298)}
current stats={'MeanSquaredError': tensor(1.9800), 'ExplainedVariance':
    ↳ tensor(-0.9199)}
current stats={'MeanSquaredError': tensor(2.2481), 'ExplainedVariance':
    ↳ tensor(-1.1622)}
>>> from pprint import pprint
>>> best_res, which_epoch = tracker.best_metric(return_step=True)
>>> pprint(best_res)
{'ExplainedVariance': -0.829...,
 'MeanSquaredError': 1.821...}
>>> which_epoch
{'MeanSquaredError': 0, 'ExplainedVariance': 2}
>>> pprint(tracker.compute_all())
{'ExplainedVariance': tensor([-0.8969, -1.0206, -0.8298, -0.9199, -1.1622]),
 'MeanSquaredError': tensor([1.8218, 2.0268, 1.9491, 1.9800, 2.2481])}

```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**best\_metric**(return\_step=False)

Returns the highest metric out of all tracked.

**Parameters** **return\_step** (bool) – If True will also return the step with the highest metric value.

**Return type** Union[None, float, Tuple[int, float], Tuple[None, None], Dict[str, Optional[float]], Tuple[Dict[str, Optional[int]], Dict[str,

`Optional[float]]]`

**Returns** The best metric value, and optionally the time-step.

**compute()**

Call compute of the current metric being tracked.

**Return type** `Any`

**compute\_all()**

Compute the metric value for all tracked metrics.

**Return type** `Tensor`

**forward(\*args, \*\*kwargs)**

Calls forward of the current metric being tracked.

**Return type** `None`

**increment()**

Creates a new instance of the input metric that will be updated next.

**Return type** `None`

**reset()**

Resets the current metric being tracked.

**Return type** `None`

**reset\_all()**

Resets all metrics being tracked.

**Return type** `None`

**update(\*args, \*\*kwargs)**

Updates the current metric being tracked.

**Return type** `None`

**property n\_steps: int**

Returns the number of times the tracker has been incremented.

**Return type** `int`

## 1.108 Min / Max

### 1.108.1 Module Interface

**class** `torchmetrics.MinMaxMetric`(*base\_metric*, *\*\*kwargs*)

Wrapper Metric that tracks both the minimum and maximum of a scalar/tensor across an experiment. The min/max value will be updated each time `.compute` is called.

**Parameters**

- **base\_metric** (*Metric*) – The metric of which you want to keep track of its maximum and minimum values.
- **kwargs** (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

Raises **ValueError** – If `base_metric`` argument is not a subclasses instance of `torchmetrics.Metric`

Example::

```
>>> import torch
>>> from torchmetrics import Accuracy
>>> from pprint import pprint
>>> base_metric = Accuracy()
>>> minmax_metric = MinMaxMetric(base_metric)
>>> preds_1 = torch.Tensor([[0.1, 0.9], [0.2, 0.8]])
>>> preds_2 = torch.Tensor([[0.9, 0.1], [0.2, 0.8]])
>>> labels = torch.Tensor([[0, 1], [0, 1]]).long()
>>> pprint(minmax_metric(preds_1, labels))
{'max': tensor(1.), 'min': tensor(1.), 'raw': tensor(1.)}
>>> pprint(minmax_metric.compute())
{'max': tensor(1.), 'min': tensor(1.), 'raw': tensor(1.)}
>>> minmax_metric.update(preds_2, labels)
>>> pprint(minmax_metric.compute())
{'max': tensor(1.), 'min': tensor(0.7500), 'raw': tensor(0.7500)}
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

**compute()**

Computes the underlying metric as well as max and min values for this metric.

Returns a dictionary that consists of the computed value (`raw`), as well as the minimum (`min`) and maximum (`max`) values.

**Return type** `Dict[str, Tensor]`

**reset()**

Sets `max_val` and `min_val` to the initialization bounds and resets the base metric.

**Return type** `None`

**update(\*args, \*\*kwargs)**

Updates the underlying metric.

**Return type** `None`

## 1.109 Multi-output Wrapper

### 1.109.1 Module Interface

```
class torchmetrics.MultioutputWrapper(base_metric, num_outputs, output_dim=-1, remove_nans=True,  
                                     squeeze_outputs=True)
```

Wrap a base metric to enable it to support multiple outputs.

Several `torchmetrics` metrics, such as `torchmetrics.regression.spearman.SpearmanCorrcoef` lack support for multioutput mode. This class wraps such metrics to support computing one metric per output. Unlike specific `torchmetric` metrics, it doesn't support any aggregation across outputs. This means if you set `num_outputs` to 2, `.compute()` will return a Tensor of dimension (2, ...) where ... represents the dimensions the metric returns when not wrapped.

In addition to enabling multioutput support for metrics that lack it, this class also supports, albeit in a crude fashion, dealing with missing labels (or other data). When `remove_nans` is passed, the class will remove the intersection of NaN containing “rows” upon each update for each output. For example, suppose a user uses *MultioutputWrapper* to wrap `torchmetrics.regression.r2.R2Score` with 2 outputs, one of which occasionally has missing labels for classes like `R2Score` is that this class supports removing NaN values (parameter `remove_nans`) on a per-output basis. When `remove_nans` is passed the wrapper will remove all rows

### Parameters

- **base\_metric** (*Metric*) – Metric being wrapped.
- **num\_outputs** (*int*) – Expected dimensionality of the output dimension. This parameter is used to determine the number of distinct metrics we need to track.
- **output\_dim** (*int*) – Dimension on which output is expected. Note that while this provides some flexibility, the output dimension must be the same for all inputs to update. This applies even for metrics such as *Accuracy* where the labels can have a different number of dimensions than the predictions. This can be worked around if the output dimension can be set to -1 for both, even if -1 corresponds to different dimensions in different inputs.
- **remove\_nans** (*bool*) – Whether to remove the intersection of rows containing NaNs from the values passed through to each underlying metric. Proper operation requires all tensors passed to update to have dimension (N, ...) where N represents the length of the batch or dataset being passed in.
- **squeeze\_outputs** (*bool*) – If True, will squeeze the 1-item dimensions left after `index_select` is applied. This is sometimes unnecessary but harmless for metrics such as *R2Score* but useful for certain classification metrics that can’t handle additional 1-item dimensions.

### Example

```
>>> # Mimic R2Score in `multioutput`, `raw_values` mode:
>>> import torch
>>> from torchmetrics import MultioutputWrapper, R2Score
>>> target = torch.tensor([[0.5, 1], [-1, 1], [7, -6]])
>>> preds = torch.tensor([[0, 2], [-1, 2], [8, -5]])
>>> r2score = MultioutputWrapper(R2Score(), 2)
>>> r2score(preds, target)
[tensor(0.9654), tensor(0.9082)]
>>> # Classification metric where prediction and label tensors have different
↳ shapes.
>>> from torchmetrics import BinnedAveragePrecision
>>> target = torch.tensor([[1, 2], [2, 0], [1, 2]])
>>> preds = torch.tensor([
...     [[.1, .8], [.8, .05], [.1, .15]],
...     [[.1, .1], [.2, .3], [.7, .6]],
...     [[.002, .4], [.95, .45], [.048, .15]]
... ])
>>> binned_avg_precision = MultioutputWrapper(BinnedAveragePrecision(3,
↳ thresholds=5), 2)
>>> binned_avg_precision(preds, target)
[[tensor(-0.), tensor(1.0000), tensor(1.0000)], [tensor(0.3333), tensor(-0.),
↳ tensor(0.6667)]]
```

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

**compute()**

Compute metrics.

**Return type** `List[Tensor]`

**forward(\*args, \*\*kwargs)**

Call underlying forward methods and aggregate the results if they're non-null.

We override this method to ensure that state variables get copied over on the underlying metrics.

**Return type** `Any`

**reset()**

Reset all underlying metrics.

**Return type** `None`

**update(\*args, \*\*kwargs)**

Update each underlying metric with the corresponding output.

**Return type** `None`

## 1.110 torchmetrics.Metric

The base `Metric` class is an abstract base class that are used as the building block for all other Module metrics.

**class** `torchmetrics.Metric(**kwargs)`

Base class for all metrics present in the Metrics API.

Implements `add_state()`, `forward()`, `reset()` and a few other things to handle distributed synchronization and per-step metric computation.

Override `update()` and `compute()` functions to implement your own metric. Use `add_state()` to register metric state variables which keep track of state on each call of `update()` and are synchronized across processes when `compute()` is called.

---

**Note:** Metric state variables can either be `torch.Tensors` or an empty list which can be used to store `torch.Tensors`.

---



---

**Note:** Different metrics only override `update()` and not `forward()`. A call to `update()` is valid, but it won't return the metric value at the current step. A call to `forward()` automatically calls `update()` and also returns the metric value at the current step.

---

**Parameters** `kwargs` (`Any`) – additional keyword arguments, see [Advanced metric settings](#) for more info.

- **compute\_on\_cpu:** If metric state should be stored on CPU during computations. Only works for list states.
- **dist\_sync\_on\_step:** If metric state should synchronize on `forward()`. Default is `False`
- **process\_group:** The process group on which the synchronization is called. Default is the world.
- **dist\_sync\_fn:** function that performs the allgather option on the metric state. Default is an custom implementation that calls `torch.distributed.all_gather` internally.

- `sync_on_compute`: If metric state should synchronize when `compute` is called. Default is `True`-

Initializes internal Module state, shared by both `nn.Module` and `ScriptModule`.

**add\_state**(*name*, *default*, *dist\_reduce\_fx=None*, *persistent=False*)

Adds metric state variable. Only used by subclasses.

#### Parameters

- **name** (`str`) – The name of the state variable. The variable will then be accessible at `self.name`.
- **default** (`Union[list, Tensor]`) – Default value of the state; can either be a `torch.Tensor` or an empty list. The state will be reset to this value when `self.reset()` is called.
- **dist\_reduce\_fx** (*Optional*) – Function to reduce state across multiple processes in distributed mode. If value is `"sum"`, `"mean"`, `"cat"`, `"min"` or `"max"` we will use `torch.sum`, `torch.mean`, `torch.cat`, `torch.min` and `torch.max` respectively, each with argument `dim=0`. Note that the `"cat"` reduction only makes sense if the state is a list, and not a tensor. The user can also pass a custom function in this parameter.
- **persistent** (*Optional*) – whether the state will be saved as part of the modules `state_dict`. Default is `False`.

---

**Note:** Setting `dist_reduce_fx` to `None` will return the metric state synchronized across different processes. However, there won't be any reduction function applied to the synchronized metric state.

The metric states would be synced as follows

- If the metric state is `torch.Tensor`, the synced value will be a stacked `torch.Tensor` across the process dimension if the metric state was a `torch.Tensor`. The original `torch.Tensor` metric state retains dimension and hence the synchronized output will be of shape `(num_process, ...)`.
- If the metric state is a `list`, the synced value will be a `list` containing the combined elements from all processes.

---

**Note:** When passing a custom function to `dist_reduce_fx`, expect the synchronized metric state to follow the format discussed in the above note.

---

#### Raises

- **ValueError** – If `default` is not a tensor or an empty `list`.
- **ValueError** – If `dist_reduce_fx` is not callable or one of `"mean"`, `"sum"`, `"cat"`, `None`.

**Return type** `None`

**clone()**

Make a copy of the metric.

**Return type** `Metric`

**abstract compute()**

Override this method to compute the final metric value from state variables synchronized across the distributed backend.

**Return type** *Any*

**double()**

Method override default and prevent dtype casting.

Please use `metric.set_dtype(dtype)` instead.

**Return type** *Metric*

**float()**

Method override default and prevent dtype casting.

Please use `metric.set_dtype(dtype)` instead.

**Return type** *Metric*

**forward(\*args, \*\*kwargs)**

`forward` serves the dual purpose of both computing the metric on the current batch of inputs but also add the batch statistics to the overall accumulating metric state.

Input arguments are the exact same as corresponding `update` method. The returned output is the exact same as the output of `compute`.

**Return type** *Any*

**half()**

Method override default and prevent dtype casting.

Please use `metric.set_dtype(dtype)` instead.

**Return type** *Metric*

**persistent(mode=False)**

Method for post-init to change if metric states should be saved to its `state_dict`.

**Return type** *None*

**reset()**

This method automatically resets the metric state variables to their default value.

**Return type** *None*

**set\_dtype(dst\_type)**

Special version of `type` for transferring all metric states to specific dtype :type  
`_sphinx_paramlinks_torchmetrics.Metric.set_dtype.dst_type:` `Union[str, dtype]` :param  
`_sphinx_paramlinks_torchmetrics.Metric.set_dtype.dst_type:` the desired type :type  
`_sphinx_paramlinks_torchmetrics.Metric.set_dtype.dst_type:` type or string

**Return type** *Metric*

**state\_dict(destination=None, prefix="", keep\_vars=False)**

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

**Warning:** Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument `destination` as it is not designed for end-users.

### Parameters

- **destination** (*dict*, *optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str*, *optional*) – a prefix added to parameter and buffer names to compose the keys in `state_dict`. Default: `''`.
- **keep\_vars** (*bool*, *optional*) – by default the `Tensor`s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

**Returns** a dictionary containing a whole state of the module

**Return type** `dict`

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

**sync** (*dist\_sync\_fn=None*, *process\_group=None*, *should\_sync=True*, *distributed\_available=<function jit\_distributed\_available>*)

Sync function for manually controlling when metrics states should be synced across processes.

### Parameters

- **dist\_sync\_fn** (*Optional[Callable]*) – Function to be used to perform states synchronization
- **process\_group** (*Optional[Any]*) – Specify the process group on which synchronization is called. default: `None` (which selects the entire world)
- **should\_sync** (*bool*) – Whether to apply to state synchronization. This will have an impact only when running in a distributed setting.
- **distributed\_available** (*Optional[Callable]*) – Function to determine if we are running inside a distributed setting

**Return type** `None`

**sync\_context** (*dist\_sync\_fn=None*, *process\_group=None*, *should\_sync=True*, *should\_unsync=True*, *distributed\_available=<function jit\_distributed\_available>*)

Context manager to synchronize the states between processes when running in a distributed setting and restore the local cache states after yielding.

### Parameters

- **dist\_sync\_fn** (*Optional[Callable]*) – Function to be used to perform states synchronization
- **process\_group** (*Optional[Any]*) – Specify the process group on which synchronization is called. default: `None` (which selects the entire world)
- **should\_sync** (*bool*) – Whether to apply to state synchronization. This will have an impact only when running in a distributed setting.



- **should\_unsync** (`bool`) – Whether to restore the cache state so that the metrics can continue to be accumulated.
- **distributed\_available** (`Optional[Callable]`) – Function to determine if we are running inside a distributed setting

**Return type** `Generator`

**type**(*dst\_type*)

Method override default and prevent dtype casting.

Please use `metric.set_dtype(dtype)` instead.

**Return type** `Metric`

**unsync**(*should\_unsync=True*)

Unsync function for manually controlling when metrics states should be reverted back to their local states.

**Parameters** **should\_unsync** (`bool`) – Whether to perform unsync

**Return type** `None`

**abstract update**(*\*, \*\*\_\_*)

Override this method to update the state variables of your metric class.

**Return type** `None`

**property device:** `torch.device`

Return the device of the metric.

**Return type** `device`

## 1.111 torchmetrics.utilities.data

### 1.111.1 select\_topk

`torchmetrics.utilities.data.select_topk(prob_tensor, topk=1, dim=1)`

Convert a probability tensor to binary by selecting top-k the highest entries.

**Parameters**

- **prob\_tensor** (`Tensor`) – dense tensor of shape `[..., C, ...]`, where C is in the position defined by the `dim` argument
- **topk** (`int`) – number of the highest entries to turn into 1s
- **dim** (`int`) – dimension on which to compare entries

**Return type** `Tensor`

**Returns** A binary tensor of the same shape as the input tensor of type `torch.int32`

### Example

```
>>> x = torch.tensor([[1.1, 2.0, 3.0], [2.0, 1.0, 0.5]])
>>> select_topk(x, topk=2)
tensor([[0, 1, 1],
        [1, 1, 0]], dtype=torch.int32)
```

## 1.111.2 to\_categorical

`torchmetrics.utilities.data.to_categorical(x, argmax_dim=1)`

Converts a tensor of probabilities to a dense label tensor.

#### Parameters

- **x** (`Tensor`) – probabilities to get the categorical label [N, d1, d2, ...]
- **argmax\_dim** (`int`) – dimension to apply

**Return type** `Tensor`

**Returns** A tensor with categorical labels [N, d2, ...]

### Example

```
>>> x = torch.tensor([[0.2, 0.5], [0.9, 0.1]])
>>> to_categorical(x)
tensor([1, 0])
```

## 1.111.3 to\_onehot

`torchmetrics.utilities.data.to_onehot(label_tensor, num_classes=None)`

Converts a dense label tensor to one-hot format.

#### Parameters

- **label\_tensor** (`Tensor`) – dense label tensor, with shape [N, d1, d2, ...]
- **num\_classes** (`Optional[int]`) – number of classes C

**Return type** `Tensor`

**Returns** A sparse label tensor with shape [N, C, d1, d2, ...]

### Example

```
>>> x = torch.tensor([1, 2, 3])
>>> to_onehot(x)
tensor([[0, 1, 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 1]])
```

## 1.112 TorchMetrics Governance

This document describes governance processes we follow in developing TorchMetrics.

### 1.112.1 Persons of Interest

#### Leads

- Nicki Skafte ([skaftenicki](#))
- Jirka Borovec ([Borda](#))
- Justus Schock ([justusschock](#))

#### Core Maintainers

- Luca Di Liello ([lucadiliello](#))
- Daniel Stancil ([stancld](#))
- Maxim Grechkin ([maximsch2](#))
- Changsheng Quan ([quancs](#))

#### Alumni

- Ananya Harsh Jha ([ananyahjha93](#))
- Teddy Koker ([teddykoker](#))

### 1.112.2 Releases

We release a new minor version (e.g., 0.5.0) every few months and bugfix releases if needed. The minor versions contain new features, API changes, deprecations, removals, potential backward-incompatible changes and also all previous bugfixes included in any bugfix release. With every release, we publish a changelog where we list additions, removals, changed functionality and fixes.

### 1.112.3 Project Management and Decision Making

The decision what goes into a release is governed by the *staff contributors and leaders* of TorchMetrics development. Whenever possible, discussion happens publicly on GitHub and includes the whole community. When a consensus is reached, staff and core contributors assign milestones and labels to the issue and/or pull request and start tracking the development. It is possible that priorities change over time.

Commits to the project are exclusively to be added by pull requests on GitHub and anyone in the community is welcome to review them. However, reviews submitted by *code owners* have higher weight and it is necessary to get the approval of code owners before a pull request can be merged. Additional requirements may apply case by case.

### 1.112.4 API Evolution

TorchMetrics development is driven by research and best practices in a rapidly developing field of AI and machine learning. Change is inevitable and when it happens, the Torchmetric team is committed to minimizing user friction and maximizing ease of transition from one version to the next. We take backward compatibility and reproducibility very seriously.

For API removal, renaming or other forms of backward-incompatible changes, the procedure is:

1. A deprecation process is initiated at version X, producing warning messages at runtime and in the documentation.
2. Calls to the deprecated API remain unchanged in their function during the deprecation phase.
3. One minor versions in the future at version X+1 the breaking change takes effect.

The “X+1” rule is a recommendation and not a strict requirement. Longer deprecation cycles may apply for some cases.

## 1.113 Contributor Covenant Code of Conduct

### 1.113.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

### 1.113.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others’ private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

### 1.113.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

### 1.113.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

### 1.113.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at [waf2107@columbia.edu](mailto:waf2107@columbia.edu). All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

### 1.113.6 Attribution

This Code of Conduct is adapted from the [Contributor Covenant](https://www.contributor-covenant.org/version/1/4/code-of-conduct.html), version 1.4, available at <https://www.contributor-covenant.org/version/1/4/code-of-conduct.html>

For answers to common questions about this code of conduct, see <https://www.contributor-covenant.org/faq>

## 1.114 Contributing

Welcome to the Torchmetrics community! We're building largest collection of native pytorch metrics, with the goal of reducing boilerplate and increasing reproducibility.

### 1.114.1 Contribution Types

We are always looking for help implementing new features or fixing bugs.

## Bug Fixes:

1. If you find a bug please submit a github issue.
  - Make sure the title explains the issue.
  - Describe your setup, what you are trying to do, expected vs. actual behaviour. Please add configs and code samples.
  - Add details on how to reproduce the issue - a minimal test case is always best, colab is also great. Note, that the sample code shall be minimal and if needed with publicly available data.
2. Try to fix it or recommend a solution. We highly recommend to use test-driven approach:
  - Convert your minimal code example to a unit/integration test with assert on expected results.
  - Start by debugging the issue... You can run just this particular test in your IDE and draft a fix.
  - Verify that your test case fails on the master branch and only passes with the fix applied.
3. Submit a PR!

*Note, even if you do not find the solution, sending a PR with a test covering the issue is a valid contribution and we can help you or finish it with you :]*

## New Features:

1. Submit a github issue - describe what is the motivation of such feature (adding the use case or an example is helpful).
2. Let's discuss to determine the feature scope.
3. Submit a PR! We recommend test driven approach to adding new features as well:
  - Write a test for the functionality you want to add.
  - Write the functional code until the test passes.
4. Add/update the relevant tests!
  - [This PR](#) is a good example for adding a new metric

## Test cases:

Want to keep Torchmetrics healthy? Love seeing those green tests? So do we! How to we keep it that way? We write tests! We value tests contribution even more than new features. One of the core values of torchmetrics is that our users can trust our metric implementation. We can only guarantee this if our metrics are well tested.

---

## 1.114.2 Guidelines

### Developments scripts

To build the documentation locally, simply execute the following commands from project root (only for Unix):

- `make clean` cleans repo from temp/generated files
- `make docs` builds documentation under `docs/build/html`
- `make test` runs all project's tests with coverage

## Original code

All added or edited code shall be the own original work of the particular contributor. If you use some third-party implementation, all such blocks/functions/modules shall be properly referred and if possible also agreed by code's author. For example - This code is inspired from <http://...>. In case you adding new dependencies, make sure that they are compatible with the actual Torchmetrics license (ie. dependencies should be *at least* as permissive as the Torchmetrics license).

## Coding Style

1. Use f-strings for output formation (except logging when we stay with lazy `logging.info("Hello %s!", name)`).
2. You can use `pre-commit` to make sure your code style is correct.

## Documentation

We are using Sphinx with Napoleon extension. Moreover, we set Google style to follow with type convention.

- Napoleon formatting with Google style
- ReStructured Text (reST)
- Paragraph-level markup

See following short example of a sample function taking one position string and optional

```
from typing import Optional

def my_func(param_a: int, param_b: Optional[float] = None) -> str:
    """Sample function.

    Args:
        param_a: first parameter
        param_b: second parameter

    Return:
        sum of both numbers

    Example:
        Sample doctest example...
        >>> my_func(1, 2)
        3

    .. note:: If you want to add something.
    """
    p = param_b if param_b else 0
    return str(param_a + p)
```

When updating the docs make sure to build them first locally and visually inspect the html files (in the browser) for formatting errors. In certain cases, a missing blank line or a wrong indent can lead to a broken layout. Run these commands

```
make docs
```

and open docs/build/html/index.html in your browser.

Notes:

- You need to have LaTeX installed for rendering math equations. You can for example install TeXLive by doing one of the following:
  - on Ubuntu (Linux) run `apt-get install texlive` or otherwise follow the instructions on the TeXLive website
  - use the [RTD docker image](#)
- with PL used class meta you need to use python 3.7 or higher

When you send a PR the continuous integration will run tests and build the docs.

## Testing

**Local:** Testing your work locally will help you speed up the process since it allows you to focus on particular (failing) test-cases. To setup a local development environment, install both local and test dependencies:

```
python -m pip install -r requirements/test.txt
python -m pip install pre-commit
```

You can run the full test-case in your terminal via this make script:

```
make test
# or natively
python -m pytest torchmetrics tests
```

Note: if your computer does not have multi-GPU nor TPU these tests are skipped.

**GitHub Actions:** For convenience, you can also use your own GHActions building which will be triggered with each commit. This is useful if you do not test against all required dependency versions.

## 1.115 Changelog

All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

**Note: we move fast, but still we preserve 0.1 version (one feature release) back compatibility.**

### 1.115.1 [0.10.0] - 2022-10-04

#### [0.10.0] - Added

- Added a new NLP metric InfoLM ([#915](#))
- Added Perplexity metric ([#922](#))
- Added ConcordanceCorrCoef metric to regression package ([#1201](#))
- Added argument `normalize` to LPIPS metric ([#1216](#))



- Added support for multiprocessing of batches in PESQ metric (#1227)
- Added support for multioutput in PearsonCorrCoef and SpearmanCorrCoef (#1200)

#### [0.10.0] - Changed

- Classification refactor ( #1054, #1143, #1145, #1151, #1159, #1163, #1167, #1175, #1189, #1197, #1215, #1195 )
- Changed update in FID metric to be done in online fashion to save memory (#1199)
- Improved performance of retrieval metrics (#1242)
- Changed SSIM and MSSSIM update to be online to reduce memory usage (#1231)

#### [0.10.0] - Fixed

- Fixed a bug in ssim when return\_full\_image=True where the score was still reduced (#1204)
- Fixed MPS support for:
  - MAE metric (#1210)
  - Jaccard index (#1205)
- Fixed bug in ClasswiseWrapper such that compute gave wrong result (#1225)
- Fixed synchronization of empty list states (#1219)

### 1.115.2 [0.9.3] - 2022-08-22

#### [0.9.3] - Added

- Added global option sync\_on\_compute to disable automatic synchronization when compute is called (#1107)

#### [0.9.3] - Fixed

- Fixed missing reset in ClasswiseWrapper (#1129)
- Fixed JaccardIndex multi-label compute (#1125)
- Fix SSIM propagate device if gaussian\_kernel is False, add test (#1149)

### 1.115.3 [0.9.2] - 2022-06-29

#### [0.9.2] - Fixed

- Fixed mAP calculation for areas with 0 predictions (#1080)
- Fixed bug where avg precision state and auroc state was not merge when using MetricCollections (#1086)
- Skip box conversion if no boxes are present in MeanAveragePrecision (#1097)
- Fixed inconsistency in docs and code when setting average="none" in AvaragePrecision metric (#1116)

#### 1.115.4 [0.9.1] - 2022-06-08

##### [0.9.1] - Added

- Added specific `RuntimeError` when metric object is on the wrong device (#1056)
- Added an option to specify own n-gram weights for `BLEUScore` and `SacreBLEUScore` instead of using uniform weights only. (#1075)

##### [0.9.1] - Fixed

- Fixed aggregation metrics when input only contains zero (#1070)
- Fixed `TypeError` when providing superclass arguments as `kwargs` (#1069)
- Fixed bug related to state reference in metric collection when using compute groups (#1076)

#### 1.115.5 [0.9.0] - 2022-05-30

##### [0.9.0] - Added

- Added `RetrievalPrecisionRecallCurve` and `RetrievalRecallAtFixedPrecision` to retrieval package (#951)
- Added class property `full_state_update` that determines `forward` should call `update` once or twice ( #984, #1033)
- Added support for nested metric collections (#1003)
- Added `Dice` to classification package (#1021)
- Added support to segmentation type `segm` as IOU for mean average precision (#822)

##### [0.9.0] - Changed

- Renamed `reduction` argument to `average` in Jaccard score and added additional options (#874)

##### [0.9.0] - Removed

- Removed deprecated `compute_on_step` argument ( #962, #967, #979, #990, #991, #993, #1005, #1004, #1007 )

##### [0.9.0] - Fixed

- Fixed non-empty state dict for a few metrics (#1012)
- Fixed bug when comparing states while finding compute groups (#1022)
- Fixed `torch.double` support in stat score metrics (#1023)
- Fixed FID calculation for non-equal size real and fake input (#1028)
- Fixed case where `KLDivergence` could output `Nan` (#1030)
- Fixed deterministic for `PyTorch<1.8` (#1035)

- Fixed default value for `mdmc_average` in `Accuracy` (#1036)
- Fixed missing copy of property when using compute groups in `MetricCollection` (#1052)

### 1.115.6 [0.8.2] - 2022-05-06

#### [0.8.2] - Fixed

- Fixed multi device aggregation in `PearsonCorrCoef` (#998)
- Fixed MAP metric when using custom list of thresholds (#995)
- Fixed compatibility between compute groups in `MetricCollection` and `prefix/postfix` arg (#1007)
- Fixed compatibility with future Pytorch 1.12 in `safe_matmul` (#1011, #1014)

### 1.115.7 [0.8.1] - 2022-04-27

#### [0.8.1] - Changed

- Reimplemented the `signal_distortion_ratio` metric, which removed the absolute requirement of `fast-bss-eval` (#964)

#### [0.8.1] - Fixed

- Fixed “Sort currently does not support bool dtype on CUDA” error in MAP for empty preds (#983)
- Fixed `BinnedPrecisionRecallCurve` when `thresholds` argument is not provided (#968)
- Fixed `CalibrationError` to work on logit input (#985)

### 1.115.8 [0.8.0] - 2022-04-14

#### [0.8.0] - Added

- Added `WeightedMeanAbsolutePercentageError` to regression package (#948)
- Added new classification metrics:
  - `CoverageError` (#787)
  - `LabelRankingAveragePrecision` and `LabelRankingLoss` (#787)
- Added new image metric:
  - `SpectralAngleMapper` (#885)
  - `ErrorRelativeGlobalDimensionlessSynthesis` (#894)
  - `UniversalImageQualityIndex` (#824)
  - `SpectralDistortionIndex` (#873)
- Added support for `MetricCollection` in `MetricTracker` (#718)
- Added support for 3D image and uniform kernel in `StructuralSimilarityIndexMeasure` (#818)
- Added smart update of `MetricCollection` (#709)

- Added `ClasswiseWrapper` for better logging of classification metrics with multiple output values (#832)
- Added `**kwargs` argument for passing additional arguments to base class (#833)
- Added negative `ignore_index` for the Accuracy metric (#362)
- Added `adaptive_k` for the RetrievalPrecision metric (#910)
- Added `reset_real_features` argument image quality assessment metrics (#722)
- Added new keyword argument `compute_on_cpu` to all metrics (#867)

### [0.8.0] - Changed

- Made `num_classes` in `jaccard_index` a required argument (#853, #914)
- Added `normalizer`, `tokenizer` to ROUGE metric (#838)
- Improved shape checking of `permutation_invariant_training` (#864)
- Allowed reduction `None` (#891)
- `MetricTracker.best_metric` will now give a warning when computing on metric that do not have a best (#913)

### [0.8.0] - Deprecated

- Deprecated argument `compute_on_step` (#792)
- Deprecated passing in `dist_sync_on_step`, `process_group`, `dist_sync_fn` direct argument (#833)

### [0.8.0] - Removed

- Removed support for versions of `Pytorch-Lightning` lower than v1.5 (#788)
- Removed deprecated functions, and warnings in `Text` (#773)
  - `WER` and `functional.wer`
- Removed deprecated functions and warnings in `Image` (#796)
  - `SSIM` and `functional.ssim`
  - `PSNR` and `functional.psnr`
- Removed deprecated functions, and warnings in classification and regression (#806)
  - `FBeta` and `functional.fbeta`
  - `F1` and `functional.f1`
  - `Hinge` and `functional.hinge`
  - `IoU` and `functional.iou`
  - `MatthewsCorrcoef`
  - `PearsonCorrcoef`
  - `SpearmanCorrcoef`
- Removed deprecated functions, and warnings in detection and pairwise (#804)
  - `MAP` and `functional.pairwise.manhattan`

- Removed deprecated functions, and warnings in Audio (#805)
  - PESQ and `functional.audio.pesq`
  - PIT and `functional.audio.pit`
  - SDR and `functional.audio.sdr` and `functional.audio.si_sdr`
  - SNR and `functional.audio.snr` and `functional.audio.si_snr`
  - STOI and `functional.audio.stoi`
- Removed unused `get_num_classes` from `torchmetrics.utilities.data` (#914)

#### [0.8.0] - Fixed

- Fixed device mismatch for MAP metric in specific cases (#950)
- Improved testing speed (#820)
- Fixed compatibility of `ClasswiseWrapper` with the `prefix` argument of `MetricCollection` (#843)
- Fixed `BestScore` on GPU (#912)
- Fixed `Lsum` computation for `ROUGEScore` (#944)

### 1.115.9 [0.7.3] - 2022-03-23

#### [0.7.3] - Fixed

- Fixed unsafe log operation in `TweedieDeviace` for `power=1` (#847)
- Fixed bug in MAP metric related to either no ground truth or no predictions (#884)
- Fixed `ConfusionMatrix`, `AUROC` and `AveragePrecision` on GPU when running in deterministic mode (#900)
- Fixed NaN or Inf results returned by `signal_distortion_ratio` (#899)
- Fixed memory leak when using `update` method with tensor where `requires_grad=True` (#902)

### 1.115.10 [0.7.2] - 2022-02-10

#### [0.7.2] - Fixed

- Minor patches in JOSS paper.

### 1.115.11 [0.7.1] - 2022-02-03

#### [0.7.1] - Changed

- Used `torch.bucketize` in calibration error when `torch>1.8` for faster computations (#769)
- Improve mAP performance (#742)

### [0.7.1] - Fixed

- Fixed check for available modules (#772)
- Fixed Matthews correlation coefficient when the denominator is 0 (#781)

## 1.115.12 [0.7.0] - 2022-01-17

### [0.7.0] - Added

- Added NLP metrics:
  - MatchErrorRate (#619)
  - WordInfoLost and WordInfoPreserved (#630)
  - SQuAD (#623)
  - CHRFScore (#641)
  - TranslationEditRate (#646)
  - ExtendedEditDistance (#668)
- Added MultiScaleSSIM into image metrics (#679)
- Added Signal to Distortion Ratio (SDR) to audio package (#565)
- Added MinMaxMetric to wrappers (#556)
- Added ignore\_index to retrieval metrics (#676)
- Added support for multi references in ROUGEScore (#680)
- Added a default VSCode devcontainer configuration (#621)

### [0.7.0] - Changed

- Scalar metrics will now consistently have additional dimensions squeezed (#622)
- Metrics having third party dependencies removed from global import (#463)
- Untokenized for BLEUScore input stay consistent with all the other text metrics (#640)
- Arguments reordered for TER, BLEUScore, SacreBLEUScore, CHRFScore now expect input order as predictions first and target second (#696)
- Changed dtype of metric state from torch.float to torch.long in ConfusionMatrix to accommodate larger values (#715)
- Unify preds, target input argument's naming across all text metrics (#723, #727)
  - bert, bleu, chrfs, sacre\_bleu, wip, wil, cer, ter, wer, mer, rouge, squad

**[0.7.0] - Deprecated**

- Renamed IoU -> Jaccard Index (#662)
- Renamed text WER metric (#714)
  - `functional.wer` -> `functional.word_error_rate`
  - WER -> WordErrorRate
- Renamed correlation coefficient classes: (#710)
  - `MatthewsCorrcoef` -> `MatthewsCorrCoef`
  - `PearsonCorrcoef` -> `PearsonCorrCoef`
  - `SpearmanCorrcoef` -> `SpearmanCorrCoef`
- Renamed audio STOI metric: (#753, #758)
  - `audio.STOI` to `audio.ShortTimeObjectiveIntelligibility`
  - `functional.audio.stoi` to `functional.audio.short_time_objective_intelligibility`
- Renamed audio PESQ metrics: (#751)
  - `functional.audio.pesq` -> `functional.audio.perceptual_evaluation_speech_quality`
  - `audio.PESQ` -> `audio.PerceptualEvaluationSpeechQuality`
- Renamed audio SDR metrics: (#711)
  - `functional.sdr` -> `functional.signal_distortion_ratio`
  - `functional.si_sdr` -> `functional.scale_invariant_signal_distortion_ratio`
  - SDR -> SignalDistortionRatio
  - SI\_SDR -> ScaleInvariantSignalDistortionRatio
- Renamed audio SNR metrics: (#712)
  - `functional.snr` -> `functional.signal_distortion_ratio`
  - `functional.si_snr` -> `functional.scale_invariant_signal_noise_ratio`
  - SNR -> SignalNoiseRatio
  - SI\_SNR -> ScaleInvariantSignalNoiseRatio
- Renamed F-score metrics: (#731, #740)
  - `functional.f1` -> `functional.f1_score`
  - F1 -> F1Score
  - `functional.fbeta` -> `functional.fbeta_score`
  - FBeta -> FBetaScore
- Renamed Hinge metric: (#734)
  - `functional.hinge` -> `functional.hinge_loss`
  - Hinge -> HingeLoss
- Renamed image PSNR metrics (#732)
  - `functional.psnr` -> `functional.peak_signal_noise_ratio`
  - PSNR -> PeakSignalNoiseRatio

- Renamed image PIT metric: (#737)
  - `functional.pit` -> `functional.permutation_invariant_training`
  - PIT -> `PermutationInvariantTraining`
- Renamed image SSIM metric: (#747)
  - `functional.ssim` -> `functional.scale_invariant_signal_noise_ratio`
  - SSIM -> `StructuralSimilarityIndexMeasure`
- Renamed detection MAP to MeanAveragePrecision metric (#754)
- Renamed Fidelity & LPIPS image metric: (#752)
  - `image.FID` -> `image.FrechetInceptionDistance`
  - `image.KID` -> `image.KernelInceptionDistance`
  - `image.LPIPS` -> `image.LearnedPerceptualImagePatchSimilarity`

#### [0.7.0] - Removed

- Removed `embedding_similarity` metric (#638)
- Removed argument `concatenate_texts` from `wer` metric (#638)
- Removed arguments `newline_sep` and `decimal_places` from `rouge` metric (#638)

#### [0.7.0] - Fixed

- Fixed `MetricCollection` kwargs filtering when no kwargs are present in update signature (#707)

### 1.115.13 [0.6.2] - 2021-12-15

#### [0.6.2] - Fixed

- Fixed `torch.sort` currently does not support `bool` dtype on CUDA (#665)
- Fixed `mAP` properly checks if ground truths are empty (#684)
- Fixed initialization of tensors to be on correct device for `MAP` metric (#673)

### 1.115.14 [0.6.1] - 2021-12-06

#### [0.6.1] - Changed

- Migrate `MAP` metrics from `pycocotools` to `PyTorch` (#632)
- Use `torch.topk` instead of `torch.argsort` in retrieval precision for speedup (#627)



### [0.6.1] - Fixed

- Fix empty predictions in MAP metric (#594, #610, #624)
- Fix edge case of AUROC with average=weighted on GPU (#606)
- Fixed forward in compositional metrics (#645)

## 1.115.15 [0.6.0] - 2021-10-28

### [0.6.0] - Added

- Added audio metrics:
  - Perceptual Evaluation of Speech Quality (PESQ) (#353)
  - Short-Time Objective Intelligibility (STOI) (#353)
- Added Information retrieval metrics:
  - RetrievalRPrecision (#577)
  - RetrievalHitRate (#576)
- Added NLP metrics:
  - SacreBLEUScore (#546)
  - CharErrorRate (#575)
- Added other metrics:
  - Tweedie Deviance Score (#499)
  - Learned Perceptual Image Patch Similarity (LPIPS) (#431)
- Added MAP (mean average precision) metric to new detection package (#467)
- Added support for float targets in nDCG metric (#437)
- Added average argument to AveragePrecision metric for reducing multi-label and multi-class problems (#477)
- Added MultioutputWrapper (#510)
- Added metric sweeping:
  - higher\_is\_better as constant attribute (#544)
  - higher\_is\_better to rest of codebase (#584)
- Added simple aggregation metrics: SumMetric, MeanMetric, CatMetric, MinMetric, MaxMetric (#506)
- Added pairwise submodule with metrics (#553)
  - pairwise\_cosine\_similarity
  - pairwise\_euclidean\_distance
  - pairwise\_linear\_similarity
  - pairwise\_manhattan\_distance

### [0.6.0] - Changed

- `AveragePrecision` will now as default output the macro average for multilabel and multiclass problems (#477)
- `half`, `double`, `float` will no longer change the dtype of the metric states. Use `metric.set_dtype` instead (#493)
- Renamed `AverageMeter` to `MeanMetric` (#506)
- Changed `is_differentiable` from property to a constant attribute (#551)
- ROC and AUROC will no longer throw an error when either the positive or negative class is missing. Instead return 0 score and give a warning

### [0.6.0] - Deprecated

- Deprecated `functional.self_supervised.embedding_similarity` in favour of new pairwise submodule

### [0.6.0] - Removed

- Removed `dtype` property (#493)

### [0.6.0] - Fixed

- Fixed bug in F1 with `average='macro'` and `ignore_index!=None` (#495)
- Fixed bug in `pit` by using the returned first result to initialize device and type (#533)
- Fixed SSIM metric using too much memory (#539)
- Fixed bug where device property was not properly update when metric was a child of a module (#542)

## 1.115.16 [0.5.1] - 2021-08-30

### [0.5.1] - Added

- Added device and dtype properties (#462)
- Added `TextTester` class for robustly testing text metrics (#450)

### [0.5.1] - Changed

- Added support for float targets in `nDCG` metric (#437)

**[0.5.1] - Removed**

- Removed rouge-score as dependency for text package (#443)
- Removed jiwer as dependency for text package (#446)
- Removed bert-score as dependency for text package (#473)

**[0.5.1] - Fixed**

- Fixed ranking of samples in SpearmanCorrCoef metric (#448)
- Fixed bug where compositional metrics where unable to sync because of type mismatch (#454)
- Fixed metric hashing (#478)
- Fixed BootStrapper metrics not working on GPU (#462)
- Fixed the semantic ordering of kernel height and width in SSIM metric (#474)

**1.115.17 [0.5.0] - 2021-08-09****[0.5.0] - Added**

- Added **Text-related (NLP) metrics**:
  - Word Error Rate (WER) (#383)
  - ROUGE (#399)
  - BERT score (#424)
  - BLUE score (#360)
- Added **MetricTracker** wrapper metric for keeping track of the same metric over multiple epochs (#238)
- Added other metrics:
  - Symmetric Mean Absolute Percentage error (SMAPE) (#375)
  - Calibration error (#394)
  - Permutation Invariant Training (PIT) (#384)
- Added support in nDCG metric for target with values larger than 1 (#349)
- Added support for negative targets in nDCG metric (#378)
- Added **None** as reduction option in CosineSimilarity metric (#400)
- Allowed passing labels in (n\_samples, n\_classes) to AveragePrecision (#386)

### [0.5.0] - Changed

- Moved `psnr` and `ssim` from `functional.regression.*` to `functional.image.*` (#382)
- Moved `image_gradient` from `functional.image_gradients` to `functional.image.gradients` (#381)
- Moved `R2Score` from `regression.r2score` to `regression.r2` (#371)
- Pearson metric now only store 6 statistics instead of all predictions and targets (#380)
- Use `torch.argmax` instead of `torch.topk` when `k=1` for better performance (#419)
- Moved check for number of samples in `R2` score to support single sample updating (#426)

### [0.5.0] - Deprecated

- Rename `r2score >> r2_score` and `kldivergence >> kl_divergence` in `functional` (#371)
- Moved `bleu_score` from `functional.nlp` to `functional.text.bleu` (#360)

### [0.5.0] - Removed

- Removed restriction that `threshold` has to be in (0,1) range to support logit input ( #351 #401)
- Removed restriction that `preds` could not be bigger than `num_classes` to support logit input (#357)
- Removed module `regression.psnr` and `regression.ssim` (#382):
- Removed (#379):
  - function `functional.mean_relative_error`
  - `num_thresholds` argument in `BinnedPrecisionRecallCurve`

### [0.5.0] - Fixed

- Fixed bug where classification metrics with `average='macro'` would lead to wrong result if a class was missing (#303)
- Fixed `weighted, multi-class` AUROC computation to allow for 0 observations of some class, as contribution to final AUROC is 0 (#376)
- Fixed that `_forward_cache` and `_computed` attributes are also moved to the correct device if metric is moved (#413)
- Fixed calculation in IoU metric when using `ignore_index` argument (#328)

## 1.115.18 [0.4.1] - 2021-07-05

### [0.4.1] - Changed

- Extend typing (#330, #332, #333, #335, #314)

#### [0.4.1] - Fixed

- Fixed DDP by `is_sync` logic to `Metric` (#339)

### 1.115.19 [0.4.0] - 2021-06-29

#### [0.4.0] - Added

- Added **Image-related metrics**:
  - Fréchet inception distance (FID) (#213)
  - Kernel Inception Distance (KID) (#301)
  - Inception Score (#299)
  - KL divergence (#247)
- Added **Audio metrics**: SNR, SI\_SDR, SI\_SNR (#292)
- Added other metrics:
  - Cosine Similarity (#305)
  - Specificity (#210)
  - Mean Absolute Percentage error (MAPE) (#248)
- Added `add_metrics` method to `MetricCollection` for adding additional metrics after initialization (#221)
- Added pre-gather reduction in the case of `dist_reduce_fx="cat"` to reduce communication cost (#217)
- Added better error message for AUROC when `num_classes` is not provided for multiclass input (#244)
- Added support for unnormalized scores (e.g. logits) in `Accuracy`, `Precision`, `Recall`, `FBeta`, `F1`, `StatScore`, `Hamming`, `ConfusionMatrix` metrics (#200)
- Added `squared` argument to `MeanSquaredError` for computing RMSE (#249)
- Added `is_differentiable` property to `ConfusionMatrix`, `F1`, `FBeta`, `Hamming`, `Hinge`, `IOU`, `MatthewsCorrcoef`, `Precision`, `Recall`, `PrecisionRecallCurve`, `ROC`, `StatScores` (#253)
- Added `sync` and `sync_context` methods for manually controlling when metric states are synced (#302)

#### [0.4.0] - Changed

- Forward cache is reset when `reset` method is called (#260)
- Improved per-class metric handling for imbalanced datasets for `precision`, `recall`, `precision_recall`, `fbeta`, `f1`, `accuracy`, and `specificity` (#204)
- Decorated `torch.jit.unused` to `MetricCollection` forward (#307)
- Renamed `thresholds` argument to binned metrics for manually controlling the thresholds (#322)
- Extend typing (#324, #326, #327)

#### [0.4.0] - Deprecated

- Deprecated `functional.mean_relative_error`, use `functional.mean_absolute_percentage_error` (#248)
- Deprecated `num_thresholds` argument in `BinnedPrecisionRecallCurve` (#322)

#### [0.4.0] - Removed

- Removed argument `is_multiclass` (#319)

#### [0.4.0] - Fixed

- AUC can also support more dimensional inputs when all but one dimension are of size 1 (#242)
- Fixed `dtype` of modular metrics after reset has been called (#243)
- Fixed calculation in `matthews_corrcoef` to correctly match formula (#321)

### 1.115.20 [0.3.2] - 2021-05-10

#### [0.3.2] - Added

- Added `is_differentiable` property:
  - To AUC, AUROC, CohenKappa and AveragePrecision (#178)
  - To PearsonCorrCoef, SpearmanCorrcoef, R2Score and ExplainedVariance (#225)

#### [0.3.2] - Changed

- `MetricCollection` should return metrics with prefix on `items()`, `keys()` (#209)
- Calling `compute` before `update` will now give warning (#164)

#### [0.3.2] - Removed

- Removed `numpy` as direct dependency (#212)

#### [0.3.2] - Fixed

- Fixed auc calculation and add tests (#197)
- Fixed loading persisted metric states using `load_state_dict()` (#202)
- Fixed PSNR not working with DDP (#214)
- Fixed metric calculation with unequal batch sizes (#220)
- Fixed metric concatenation for list states for zero-dim input (#229)
- Fixed numerical instability in AUROC metric for large input (#230)

### 1.115.21 [0.3.1] - 2021-04-21

- Cleaning remaining inconsistency and fix PL develop integration ( [#191](#), [#192](#), [#193](#), [#194](#) )

### 1.115.22 [0.3.0] - 2021-04-20

#### [0.3.0] - Added

- Added BootStrapper to easily calculate confidence intervals for metrics ([#101](#))
- Added Binned metrics ([#128](#))
- Added metrics for Information Retrieval (([PL^5032](#))):
  - RetrievalMAP ([PL^5032](#))
  - RetrievalMRR ([#119](#))
  - RetrievalPrecision ([#139](#))
  - RetrievalRecall ([#146](#))
  - RetrievalNormalizedDCG ([#160](#))
  - RetrievalFallOut ([#161](#))
- Added other metrics:
  - CohenKappa ([#69](#))
  - MatthewsCorrcoef ([#98](#))
  - PearsonCorrcoef ([#157](#))
  - SpearmanCorrcoef ([#158](#))
  - Hinge ([#120](#))
- Added average='micro' as an option in AUROC for multilabel problems ([#110](#))
- Added multilabel support to ROC metric ([#114](#))
- Added testing for half precision ([#77](#), [#135](#) )
- Added AverageMeter for ad-hoc averages of values ([#138](#))
- Added prefix argument to MetricCollection ([#70](#))
- Added \_\_getitem\_\_ as metric arithmetic operation ([#142](#))
- Added property is\_differentiable to metrics and test for differentiability ([#154](#))
- Added support for average, ignore\_index and mdmc\_average in Accuracy metric ([#166](#))
- Added postfix arg to MetricCollection ([#188](#))

### [0.3.0] - Changed

- Changed ExplainedVariance from storing all preds/targets to tracking 5 statistics (#68)
- Changed behaviour of confusionmatrix for multilabel data to better match multilabel\_confusion\_matrix from sklearn (#134)
- Updated FBeta arguments (#111)
- Changed reset method to use detach.clone() instead of deepcopy when resetting to default (#163)
- Metrics passed as dict to MetricCollection will now always be in deterministic order (#173)
- Allowed MetricCollection pass metrics as arguments (#176)

### [0.3.0] - Deprecated

- Rename argument is\_multiclass -> multiclass (#162)

### [0.3.0] - Removed

- Prune remaining deprecated (#92)

### [0.3.0] - Fixed

- Fixed when \_stable\_1d\_sort to work when  $n \geq N$  (PL#6177)
- Fixed \_computed attribute not being correctly reset (#147)
- Fixed to Blau score (#165)
- Fixed backwards compatibility for logging with older version of pytorch-lightning (#182)

## 1.115.23 [0.2.0] - 2021-03-12

### [0.2.0] - Changed

- Decoupled PL dependency (#13)
- Refactored functional - mimic the module-like structure: classification, regression, etc. (#16)
- Refactored utilities - split to topics/submodules (#14)
- Refactored MetricCollection (#19)

### [0.2.0] - Removed

- Removed deprecated metrics from PL base (#12, #15)



## 1.115.24 [0.1.0] - 2021-02-22

- Added Accuracy metric now generalizes to Top-k accuracy for (multi-dimensional) multi-class inputs using the `top_k` parameter (PL^4838)
- Added Accuracy metric now enables the computation of subset accuracy for multi-label or multi-dimensional multi-class inputs with the `subset_accuracy` parameter (PL^4838)
- Added HammingDistance metric to compute the hamming distance (loss) (PL^4838)
- Added StatScores metric to compute the number of true positives, false positives, true negatives and false negatives (PL^4839)
- Added R2Score metric (PL^5241)
- Added MetricCollection (PL^4318)
- Added `.clone()` method to metrics (PL^4318)
- Added IoU class interface (PL^4704)
- The Recall and Precision metrics (and their functional counterparts `recall` and `precision`) can now be generalized to Recall@K and Precision@K with the use of `top_k` parameter (PL^4842)
- Added compositional metrics (PL^5464)
- Added AUC/AUROC class interface (PL^5479)
- Added QuantizationAwareTraining callback (PL^5706)
- Added ConfusionMatrix class interface (PL^4348)
- Added multiclass AUROC metric (PL^4236)
- Added PrecisionRecallCurve, ROC, AveragePrecision class metric (PL^4549)
- Classification metrics overhaul (PL^4837)
- Added F1 class metric (PL^4656)
- Added metrics aggregation in Horovod and fixed early stopping (PL^3775)
- Added `persistent(mode)` method to metrics, to enable and disable metric states being added to `state_dict` (PL^4482)
- Added unification of regression metrics (PL^4166)
- Added persistent flag to `Metric.add_state` (PL^4195)
- Added classification metrics (PL^4043)
- Added new Metrics API. (PL^3868, PL^3921)
- Added EMB similarity (PL^3349)
- Added SSIM metrics (PL^2671)
- Added BLEU metrics (PL^2535)



## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## INDEX

### A

`add_state()` (*torchmetrics.Metric* method), 21

### B

`bert_score()` (in module *torchmetrics.functional.text.bert*), 381

### C

`clone()` (*torchmetrics.Metric* method), 22

`compute()` (*torchmetrics.audio.pesq.PerceptualEvaluationSpeechQuality* method), 34

`compute()` (*torchmetrics.Metric* method), 22

### D

`device` (*torchmetrics.Metric* property), 25

`double()` (*torchmetrics.Metric* method), 22

### F

`float()` (*torchmetrics.Metric* method), 22

`forward()` (*torchmetrics.Metric* method), 22

### H

`half()` (*torchmetrics.Metric* method), 23

### I

`infolm()` (in module *torchmetrics.functional.text.infolm*), 392

### M

`Metric` (class in *torchmetrics*), 21

### P

`perceptual_evaluation_speech_quality()` (in module *torchmetrics.functional.audio.pesq*), 34

`PerceptualEvaluationSpeechQuality` (class in *torchmetrics.audio.pesq*), 33

`persistent()` (*torchmetrics.Metric* method), 23

### R

`reset()` (*torchmetrics.Metric* method), 23

### S

`set_dtype()` (*torchmetrics.Metric* method), 23

`spectral_distortion_index()` (in module *torchmetrics.functional*), 327

`state_dict()` (*torchmetrics.Metric* method), 23

`sync()` (*torchmetrics.Metric* method), 24

`sync_context()` (*torchmetrics.Metric* method), 24

### T

`type()` (*torchmetrics.Metric* method), 24

### U

`unsync()` (*torchmetrics.Metric* method), 24

`update()` (*torchmetrics.audio.pesq.PerceptualEvaluationSpeechQuality* method), 34

`update()` (*torchmetrics.Metric* method), 25