PyTorch-Metrics Documentation

Release 0.3.0

PyTorchLightning et al.

Apr 20, 2021

USER GUIDE

TorchMetrics is a collection of Machine learning metrics for distributed, scalable PyTorch models and an easy-to-use API to create custom metrics. It offers the following benefits:

- Optimized for distributed-training
- 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 with in 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.

CHAPTER

ONE

USING TORCHMETRICS

1.1 Module metrics

```
import torch
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}")
```

Module metric usage remains the same when using multiple GPUs or multiple nodes.

1.2 Functional metrics

```
import torch
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)
```

1.3 Implementing a metric

```
class MyAccuracy(Metric):
    def __init__(self, dist_sync_on_step=False):
        # call `self.add_state`for every internal state that is needed for the␣
↪metrics computations
        # dist_reduce_fx indicates the function that should be used to reduce
        # state from multiple processes
        super().__init__(dist_sync_on_step=dist_sync_on_step)

        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):
        # update metric states
        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):
        # compute final result
        return self.correct.float() / self.total
```

CHAPTER

TWO

MORE READING

2.1 Quick Start

TorchMetrics is a collection of 25+ PyTorch metrics implementations and an easy-to-use API to create custom metrics. It offers:

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

You can use TorchMetrics in any PyTorch model, or with in 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.

2.1.1 Install

You can install TorchMetrics using pip or conda:

```
pip install torchmetrics
```

2.1.2 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
```

(continues on next page)

(continued from previous page)

```
# 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}")

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

2.1.3 Implementing your own metric

Implementing your own metric is as easy as subclassing an `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*.

2.2 Overview

The `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(compute_on_step=False)

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)

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

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.

2.2.1 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 be automatically move to the same device as the 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():
    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)
```

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 synchonized between the main process and the replicas before they are destroyed.

Metrics in Distributed Data Parallel (DDP) mode

When using metrics in Distributed Data Parallel (DPP) 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 sligtly bias 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.

2.2.2 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:
  - *PSNR* and *psnr [func]*
  - *SSIM* and *ssim [func]*

2.2.3 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`

2.2.4 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 theses 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

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

class MyModule():
    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)
        # ...
        output = self.valid_metrics(logits, y)
        # use log_dict instead of log
        # metrics are logged with keys: val_Accuracy, val_Precision and val_Recall
        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.

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

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

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
>>> 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)}
>>> metrics.persistent()
```

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

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`

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]

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.

2.2.5 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.

2.2.6 Metrics and differentiability

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

```
@property
def is_differentiable(self) -> bool:
    return True/False
```

However, note that the cached state is detached from the computational graph and cannot be backpropagated. 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:

```
metric = MyMetric()
val = metric(pred, target) # this value can be backpropagated
val = metric.compute() # this value cannot be backpropagated
```

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

2.3 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.

All you need to do is call `add_state` correctly to implement a custom metric with DDP. `reset()` is called on metric state variables added using `add_state()`.

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, dist_sync_on_step=False):
        super().__init__(dist_sync_on_step=dist_sync_on_step)

        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
```

2.3.1 Internal implementation details

This section briefly describes how metrics work internally. We encourage looking at the source code for more info. Internally, Lightning 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` and `compute` in the following way (assuming metric is initialized with `compute_on_step=True`):

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.

This procedure has the consequence of calling the user defined `update` **twice** during a single forward call (one to update global statistics and one for getting the batch statistics).

---

class torchmetrics.**Metric**(*compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

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 we 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

- **compute_on_step**¶ (`bool`) – Forward only calls `update()` and returns None if this is set to False. default: True
- **dist_sync_on_step**¶ (`bool`) – Synchronize metric state across processes at each `forward()` before returning the value at the step.
- **process_group**¶ (`Optional`[`Any`]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)
- **dist_sync_fn**¶ (`Optional`[`Callable`]) – Callback that performs the allgather operation on the metric state. When *None*, DDP will be used to perform the allgather.

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** – 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"`, or `"cat"`, we will use `torch.sum`, `torch.mean`, and `torch.cat` 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`.

clone()
Make a copy of the metric

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

forward(**args*, ***kwargs*)
Automatically calls `update()`. Returns the metric value over inputs if `compute_on_step` is True.

persistent(*mode=False*)
Method for post-init to change if metric states should be saved to its state_dict

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

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.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

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

abstract update()

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

Return type None

2.3.2 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 showcases 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 `tests/"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 :]

2.4 TorchMetrics in PyTorch Lightning

TorchMetrics was originaly created as part of PyTorch Lightning, a powerful deep learning research framework designed for scaling models without boilerplate.

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

- Module 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.
- 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:

```
def __init__(self):
    ...
    self.accuracy = pl.metrics.Accuracy()

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

(continues on next page)

(continued from previous page)

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

2.4.1 Logging TorchMetrics

Metric objects can also be directly logged 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.

```
def __init__(self):
    ...
    self.train_acc = pl.metrics.Accuracy()
    self.valid_acc = pl.metrics.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)
    ...
    self.valid_acc(logits, y)
    self.log('valid_acc', self.valid_acc, on_step=True, on_epoch=True)
```

Note: 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 due to metric states else being destroyed after each forward pass, leading 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)
```

For more details see Lightning Docs

2.5 Module metrics

2.5.1 Base class

The base `Metric` class is an abstract base class that are used as the building block for all other Module metrics.

class torchmetrics.**Metric**(*compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

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 we 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

- **compute_on_step** (`bool`) – Forward only calls `update()` and returns None if this is set to False. default: True
- **dist_sync_on_step** (`bool`) – Synchronize metric state across processes at each `forward()` before returning the value at the step.
- **process_group** (`Optional[Any]`) – Specify the process group on which synchronization is called. default: None (which selects the entire world)
- **dist_sync_fn** (`Optional[Callable]`) – Callback that performs the allgather operation on the metric state. When *None*, DDP will be used to perform the allgather.

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** – 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"`, or `"cat"`, we will use `torch.sum`, `torch.mean`, and `torch.cat` 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`.

clone()
Make a copy of the metric

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

forward(**args*, ***kwargs*)
Automatically calls `update()`. Returns the metric value over inputs if `compute_on_step` is True.

persistent(*mode=False*)
Method for post-init to change if metric states should be saved to its state_dict

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

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.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

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

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

Return type None

We also have an AverageMeter class that is helpful for defining ad-hoc metrics, when creating your own metric type might be too burdensome.

class torchmetrics.**AverageMeter**(*compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Computes the average of a stream of values.

Forward accepts

- value (float tensor): (...)
- weight (float tensor): (...)

Parameters

- **compute_on_step** (bool) – Forward only calls update() and returns None if this is set to False. default: True
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each forward() before returning the value at the step.
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)
- **dist_sync_fn** (Optional[Callable]) – Callback that performs the allgather operation on the metric state. When *None*, DDP will be used to perform the allgather.

Example::

```
>>> from torchmetrics import AverageMeter
>>> avg = AverageMeter()
>>> avg.update(3)
>>> avg.update(1)
>>> avg.compute()
tensor(2.)
```

```
>>> avg = AverageMeter()
>>> values = torch.tensor([1., 2., 3.])
>>> avg(values)
tensor(2.)
```

```
>>> avg = AverageMeter()
>>> values = torch.tensor([1., 2.])
>>> weights = torch.tensor([3., 1.])
>>> avg(values, weights)
tensor(1.2500)
```

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(*value*, *weight=1.0*)

Updates the average with.

Parameters

- **value** (Union[Tensor, float]) – A tensor of observations (can also be a scalar value)
- **weight** (Union[Tensor, float]) – The weight of each observation (automatically broadcasted to fit value)

Return type None

2.5.2 Classification Metrics

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 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 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
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]])
```

Accuracy

class torchmetrics.**Accuracy**(*threshold=0.5*, *num_classes=None*, *average='micro'*, *mdmc_average='global'*, *ignore_index=None*, *top_k=None*, *multiclass=None*, *subset_accuracy=False*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

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 predictions, the parameter `top_k` generalizes this metric to a Top-K accuracy metric: for each sample the top-K highest probability 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 probability value for transforming probability predictions to binary (0,1) predictions, in the case of binary or multi-label inputs.
- **average** (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`.

- **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 highest probability predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class inputs with probability predictions. 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 counter 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.
- **compute_on_step**¶ (`bool`) – Forward only calls `update()` and return `None` if this is set to `False`.
- **dist_sync_on_step**¶ (`bool`) – Synchronize metric state across processes at each `forward()` before returning the value at the step
- **process_group**¶ (`Optional[Any]`) – Specify the process group on which synchronization is called. default: `None` (which selects the entire world)
- **dist_sync_fn**¶ (`Optional[Callable]`) – Callback that performs the allgather operation on the metric state. When `None`, DDP will be used to perform the allgather

Raises

- **ValueError** – If `threshold` is not between `0` and `1`.
- **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 `mult-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)
```

(continues on next page)

(continued from previous page)

```
>>> 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 (probabilities, or labels)
- **target** (`Tensor`) – Ground truth labels

AveragePrecision

class torchmetrics.**AveragePrecision**(*num_classes=None*, *pos_label=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)
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 nessesary to provide 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]
- **compute_on_step** (`bool`) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step** (`bool`) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False
- **process_group** (`Optional[Any]`) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

Example (binary case):

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

(continues on next page)

(continued from previous page)

```
>>> 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_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 will 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

AUC

class torchmetrics.**AUC**(*reorder=False*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)
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

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
- **compute_on_step** (bool) – Forward only calls update() and return None if this is set to False.
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each forward() before returning the value at the step.
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)
- **dist_sync_fn** (Optional[Callable]) – Callback that performs the allgather operation on the metric state. When None, DDP will be used to perform the allgather.

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(*x*, *y*)
Update state with predictions and targets.

Parameters

- **x** (`Tensor`) – Predictions from model (probabilities, or labels)
- **y** (`Tensor`) – Ground truth labels

AUROC

class torchmetrics.**AUROC**(*num_classes=None*, *pos_label=None*, *average='macro'*, *max_fpr=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)
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 interpretated as multilabel and if `preds` have one dimension more than the `target` tensor the input will be interpretated as multiclass.

Parameters

- **num_classes** (`Optional[int]`) – integer with number of classes. Not nessesary to provide 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[str]`) –
  - `'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.
- **compute_on_step** (`bool`) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step** (`bool`) – Synchronize metric state across processes at each `forward()` before returning the value at the step.

- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)
- **dist_sync_fn** (Optional[Callable]) – Callback that performs the allgather operation on the metric state. When None, DDP will be used to perform the allgather

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

BinnedAveragePrecision

class torchmetrics.**BinnedAveragePrecision**(*num_classes*, *num_thresholds=100*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)

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 `num_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 nessesary to provide for binary problems.
- **num_thresholds** (`int`) – number of bins used for computation. More bins will lead to more detailed curve and accurate estimates, but will be slower and consume more memory. Default 100
- **compute_on_step** (`bool`) – Forward only calls `update()` and return None if this is set to False. default: True
- **process_group** (`Optional[Any]`) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

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, num_
↪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, num_
↪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]

BinnedPrecisionRecallCurve

class torchmetrics.**BinnedPrecisionRecallCurve**(*num_classes*, *num_thresholds=100*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)

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 `num_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.
- **num_thresholds** (int) – number of bins used for computation. More bins will lead to more detailed curve and accurate estimates, but will be slower and consume more memory. Default 100
- **compute_on_step** (bool) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

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, num_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])
```

(continues on next page)

(continued from previous page)

```
>>> pr_curve = BinnedPrecisionRecallCurve(num_classes=5, num_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 Tuple[Tensor, Tensor, Tensor]

update(*preds*, *targets*)

Args preds: (n_samples, n_classes) tensor targets: (n_samples, n_classes) tensor

Return type None

BinnedRecallAtFixedPrecision

class torchmetrics.**BinnedRecallAtFixedPrecision**(*num_classes*, *min_precision*, *num_thresholds=100*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)

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

Computation is performed in constant-memory by computing precision and recall for `num_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 for binary problems.

- **min_precision** (float) – float value specifying minimum precision threshold.
- **num_thresholds** (int) – number of bins used for computation. More bins will lead to more detailed curve and accurate estimates, but will be slower and consume more memory. Default 100
- **compute_on_step** (bool) – Forward only calls `update()` and return None if this is set to False. default: True
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

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, num_
↪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, num_
↪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]

CohenKappa

class torchmetrics.**CohenKappa**(*num_classes*, *weights=None*, *threshold=0.5*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)

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.

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 value for binary or multi-label probabilites. default: 0.5
- **compute_on_step** (`bool`) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step** (`bool`) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False
- **process_group** (`Optional[Any]`) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

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

ConfusionMatrix

class torchmetrics.**ConfusionMatrix**(*num_classes*, *normalize=None*, *threshold=0.5*, *multilabel=False*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)

Computes the confusion matrix. 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, 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.

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 value for binary or multi-label probabilites. default: 0.5
- **multilabel** (`bool`) – determines if data is multilabel or not.
- **compute_on_step** (`bool`) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step** (`bool`) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False
- **process_group** (`Optional[Any]`) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

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

F1

class torchmetrics.**F1**(*num_classes=None*, *threshold=0.5*, *average='micro'*, *mdmc_average=None*, *ignore_index=None*, *top_k=None*, *multiclass=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*, *is_multiclass=None*)
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 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 probability value for transforming probability predictions to binary (0,1) predictions, in the case of binary or multi-label inputs.
- **average** (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.

- **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 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.
- **compute_on_step** (bool) – Forward only calls update() and return None if this is set to False.

- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each `forward()` before returning the value at the step
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: `None` (which selects the entire world)
- **dist_sync_fn** (Optional[Callable]) – Callback that performs the allgather operation on the metric state. When `None`, DDP will be used to perform the allgather.

Example

```
>>> from torchmetrics import F1
>>> target = torch.tensor([0, 1, 2, 0, 1, 2])
>>> preds = torch.tensor([0, 2, 1, 0, 0, 1])
>>> f1 = F1(num_classes=3)
>>> f1(preds, target)
tensor(0.3333)
```

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

FBeta

class torchmetrics.**FBeta**(*num_classes=None*, *beta=1.0*, *threshold=0.5*, *average='micro'*, *mdmc_average=None*, *ignore_index=None*, *top_k=None*, *multiclass=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*, *is_multiclass=None*)

Computes F-score, specifically:

$$F_\beta = (1 + \beta^2) * \frac{\text{precision} * \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}}$$

Where β is some positive real factor. 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.

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 probability value for transforming probability predictions to binary (0,1) predictions, in the case of binary or multi-label inputs.
- **average** (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`.

- **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 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.
- **compute_on_step** (`bool`) – Forward only calls `update()` and return `None` if this is set to `False`.
- **dist_sync_on_step** (`bool`) – Synchronize metric state across processes at each `forward()` before returning the value at the step
- **process_group** (`Optional[Any]`) – Specify the process group on which synchronization is called. default: `None` (which selects the entire world)
- **dist_sync_fn** (`Optional[Callable]`) – Callback that performs the allgather operation on the metric state. When `None`, DDP will be used to perform the allgather.

Raises **ValueError** – If `average` is none of `"micro"`, `"macro"`, `"weighted"`, `"none"`, `None`.

Example

```
>>> from torchmetrics import FBeta
>>> target = torch.tensor([0, 1, 2, 0, 1, 2])
>>> preds = torch.tensor([0, 2, 1, 0, 0, 1])
>>> f_beta = FBeta(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 fbeta over state.

Return type Tensor

HammingDistance

class torchmetrics.**HammingDistance**(*threshold=0.5*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

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 probability value for transforming probability predictions to binary (0 or 1) predictions, in the case of binary or multi-label inputs.
- **compute_on_step** (bool) – Forward only calls `update()` and return None if this is set to False.
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each `forward()` before returning the value at the step.
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)
- **dist_sync_fn** (Optional[Callable]) – Callback that performs the allgather operation on the metric state. When None, DDP will be used to perform the all gather.

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, or labels)
- **target** (`Tensor`) – Ground truth labels

Hinge

class torchmetrics.**Hinge**(*squared=False*, *multiclass_mode=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)
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 $y \in 0, ..., \mathrm{C}$ is the target class (where C is the number of classes), and $\hat{y} \in \mathbb{R}^{\mathrm{C}}$ 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.

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 Hinge
>>> target = torch.tensor([0, 1, 1])
>>> preds = torch.tensor([-2.2, 2.4, 0.1])
>>> hinge = Hinge()
>>> 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 = Hinge()
>>> 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 = Hinge(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.

IoU

class torchmetrics.**IoU**(*num_classes*, *ignore_index=None*, *absent_score=0.0*, *threshold=0.5*, *reduction='elementwise_mean'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)

Computes Intersection over union, or Jaccard index calculation:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

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.
- **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]. 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 *pred* AND no instances of the class index were present in *target*. For example, if we have 3 classes, [0, 0] for *pred*, and [0, 2] for *target*, then class 1 would be assigned the *absent_score*.
- **threshold** (`float`) – Threshold value for binary or multi-label probabilities.
- **reduction** (`str`) – a method to reduce metric score over labels.
  - `'elementwise_mean'`: takes the mean (default)
  - `'sum'`: takes the sum
  - `'none'`: no reduction will be applied
- **compute_on_step** (`bool`) – Forward only calls `update()` and return None if this is set to False.
- **dist_sync_on_step** (`bool`) – Synchronize metric state across processes at each `forward()` before returning the value at the step.
- **process_group** (`Optional[Any]`) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

Example

```
>>> from torchmetrics import IoU
>>> 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]
>>> iou = IoU(num_classes=2)
>>> iou(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

MatthewsCorrcoef

class torchmetrics.**MatthewsCorrcoef**(*num_classes*, *threshold=0.5*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

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 postitives, 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 probabilites. default: 0.5
- **compute_on_step** (bool) – Forward only calls update() and return None if this is set to False. default: True
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each forward() before returning the value at the step. default: False
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

- **dist_sync_fn** (Optional[Callable]) – Callback that performs the allgather operation on the metric state. When None, DDP will be used to perform the allgather

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

Precision

class torchmetrics.**Precision**(*num_classes=None*, *threshold=0.5*, *average='micro'*, *mdmc_average=None*, *ignore_index=None*, *top_k=None*, *multiclass=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*, *is_multiclass=None*)
Computes Precision:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Where TP and FP represent the number of true positives and false positives respecitively. 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 probability value for transforming probability predictions to binary (0,1) predictions, in the case of binary or multi-label inputs.
- **average** (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`.

- **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 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.
- **compute_on_step** (`bool`) – Forward only calls `update()` and return `None` if this is set to `False`.
- **dist_sync_on_step** (`bool`) – Synchronize metric state across processes at each `forward()` before returning the value at the step
- **process_group** (`Optional[Any]`) – Specify the process group on which synchronization is called. default: `None` (which selects the entire world)
- **dist_sync_fn** (`Optional[Callable]`) – Callback that performs the allgather operation on the metric state. When `None`, DDP will be used to perform the allgather.

Raises **ValueError** – If `average` is none of `"micro"`, `"macro"`, `"weighted"`, `"samples"`, `"none"`, `None`.

Example

```
>>> 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.

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

PrecisionRecallCurve

class torchmetrics.**PrecisionRecallCurve**(*num_classes=None*, *pos_label=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)

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. Not nessesary to provide 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]
- **compute_on_step** (`bool`) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step** (`bool`) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False
- **process_group** (`Optional[Any]`) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

Example (binary case):

```
>>> from torchmetrics import PrecisionRecallCurve
>>> pred = torch.tensor([0, 1, 2, 3])
>>> 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, 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])
>>> 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

Recall

class torchmetrics.**Recall**(*num_classes=None*, *threshold=0.5*, *average='micro'*, *mdmc_average=None*, *ignore_index=None*, *top_k=None*, *multiclass=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*, *is_multiclass=None*)

Computes Recall:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where TP and FN represent the number of true positives and false negatives respecitively. 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 probability value for transforming probability predictions to binary (0,1) predictions, in the case of binary or multi-label inputs.
- **average** (`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`.

- **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 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.
- **compute_on_step** (bool) – Forward only calls `update()` and return `None` if this is set to `False`.
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each `forward()` before returning the value at the step
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: `None` (which selects the entire world)
- **dist_sync_fn** (Optional[Callable]) – Callback that performs the allgather operation on the metric state. When `None`, DDP will be used to perform the allgather.

Raises **ValueError** – If `average` is none of `"micro"`, `"macro"`, `"weighted"`, `"samples"`, `"none"`, `None`.

Example

```
>>> 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.

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

ROC

class torchmetrics.**ROC**(*num_classes=None*, *pos_label=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

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

Parameters

- **num_classes** (`Optional[int]`) – integer with number of classes. Not nessesary to provide 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]
- **compute_on_step** (`bool`) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step** (`bool`) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False
- **process_group** (`Optional[Any]`) – Specify the process group on which synchronization is called. default: None (which selects the entire world)
- **dist_sync_fn** (`Optional[Callable]`) – Callback that performs the allgather operation on the metric state. When `None`, DDP will be used to perform the allgather

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])]
```

(continues on next page)

(continued from previous page)

```
>>> 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 postive rates

update(*preds*, *target*)

Update state with predictions and targets.

Parameters

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

StatScores

class torchmetrics.**StatScores**(*threshold=0.5*, *top_k=None*, *reduce='micro'*, *num_classes=None*, *ignore_index=None*, *mdmc_reduce=None*, *multiclass=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*, *is_multiclass=None*)

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 probability value for transforming probability predictions to binary (0 or 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.
- **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: Wwhat 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 handeled. 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.
- **compute_on_step** (`bool`) – Forward only calls `update()` and return `None` if this is set to `False`.
- **dist_sync_on_step** (`bool`) – Synchronize metric state across processes at each `forward()` before returning the value at the step
- **process_group** (`Optional[Any]`) – Specify the process group on which synchronization is called. default: `None` (which selects the entire world)
- **dist_sync_fn** (`Optional[Callable]`) – Callback that performs the allgather operation on the metric state. When `None`, DDP will be used to perform the allgather.

Raises

- **ValueError** – If `threshold` is not a `float` between `0` and `1`.
- **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` <= `ignore_index` < `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 `(..., 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)`

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 or labels)
- **target** (`Tensor`) – Ground truth values

2.5.3 Regression Metrics

ExplainedVariance

class torchmetrics.**ExplainedVariance**(*multioutput='uniform_average'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

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

- **compute_on_step** (bool) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

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.

update(*preds*, *target*)
Update state with predictions and targets.

Parameters

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

MeanAbsoluteError

class torchmetrics.**MeanAbsoluteError**(*compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

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

- **compute_on_step**¶ (bool) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step**¶ (bool) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False
- **process_group**¶ (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

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.

update(*preds*, *target*)
Update state with predictions and targets.

Parameters

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

MeanSquaredError

class torchmetrics.**MeanSquaredError**(*compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

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

- **compute_on_step**¶ (bool) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step**¶ (bool) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False
- **process_group**¶ (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

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.

update(*preds*, *target*)
Update state with predictions and targets.

Parameters

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

MeanSquaredLogError

class torchmetrics.**MeanSquaredLogError**(*compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

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

- **compute_on_step** (bool) – Forward only calls update() and return None if this is set to False. default: True
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each forward() before returning the value at the step. default: False
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

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.

update(*preds*, *target*)
Update state with predictions and targets.

Parameters

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

PearsonCorrcoef

class torchmetrics.**PearsonCorrcoef**(*compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)

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): (N,)
- target``(float tensor): ``(N,)

Parameters

- **compute_on_step**¶ (bool) – Forward only calls update() and return None if this is set to False. default: True
- **dist_sync_on_step**¶ (bool) – Synchronize metric state across processes at each forward() before returning the value at the step. default: False
- **process_group**¶ (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

Example

```
>>> 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)
```

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

compute()
Computes pearson correlation coefficient over state.

update(*preds*, *target*)
Update state with predictions and targets.

Parameters

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

PSNR

class torchmetrics.**PSNR**(*data_range=None*, *base=10.0*, *reduction='elementwise_mean'*, *dim=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)

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 (default: 10)
- **reduction** (str) – a method to reduce metric score over labels.
  - 'elementwise_mean': takes the mean (default)
  - 'sum': takes the sum
  - '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.
- **compute_on_step** (bool) – Forward only calls update() and return None if this is set to False. default: True
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each forward() before returning the value at the step. default: False
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

Raises **ValueError** – If dim is not None and data_range is not given.

Example

```
>>> from torchmetrics import PSNR
>>> psnr = PSNR()
>>> 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.

update(*preds*, *target*)
Update state with predictions and targets.

Parameters

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

R2Score

class torchmetrics.**R2Score**(*num_outputs=1*, *adjusted=0*, *multioutput='uniform_average'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Computes r2 score also known as coefficient of 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 (default is 1)
- **adjusted** (int) – number of independent regressors for calculating adjusted r2 score. Default 0 (standard r2 score).
- **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
- **compute_on_step** (bool) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

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

SpearmanCorrcoef

class torchmetrics.**SpearmanCorrcoef**(*compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)
Computes spearmans rank correlation coefficient.

where rg_x and rg_y are the rank associated to the variables x and y. Spearmans correlations coefficient corresponds to the standard pearsons correlation coefficient calculated on the rank variables.

Parameters

- **compute_on_step** (bool) – Forward only calls update() and return None if this is set to False. default: True
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each forward() before returning the value at the step. default: False
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)
- **dist_sync_fn** (Optional[Callable]) – Callback that performs the allgather operation on the metric state. When None, DDP will be used to perform the allgather

Example

```
>>> 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)
```

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

compute()
Computes spearmans correlation coefficient

update(*preds*, *target*)
Update state with predictions and targets.

Parameters

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

SSIM

class torchmetrics.**SSIM**(*kernel_size=(11, 11)*, *sigma=(1.5, 1.5)*, *reduction='elementwise_mean'*, *data_range=None*, *k1=0.01*, *k2=0.03*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)
Computes Structual Similarity Index Measure (SSIM).

Parameters

- **kernel_size** (Sequence[int]) – size of the gaussian kernel (default: (11, 11))
- **sigma** (Sequence[float]) – Standard deviation of the gaussian kernel (default: (1.5, 1.5))
- **reduction** (str) – a method to reduce metric score over labels.
  - 'elementwise_mean': takes the mean (default)
  - 'sum': takes the sum
  - '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. Default: 0.01
- **k2** (float) – Parameter of SSIM. Default: 0.03

Returns Tensor with SSIM score

Example

```
>>> from torchmetrics import SSIM
>>> preds = torch.rand([16, 1, 16, 16])
>>> target = preds * 0.75
>>> ssim = SSIM()
>>> ssim(preds, target)
tensor(0.9219)
```

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

compute()
Computes explained variance over state.

update(*preds*, *target*)
Update state with predictions and targets.

Parameters

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

2.5.4 Retrieval

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
`long`	(N,...)	`float`	(N,...)	`long` or `bool`	(N,...)

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_i, D_j)`, 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_i, D_j)` having the same query `Q_i`.

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])

>>> map = RetrievalMAP() # or some other retrieval metric
>>> map(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)
```

RetrievalMAP

class torchmetrics.**RetrievalMAP**(*empty_target_action='neg'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

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`
- **compute_on_step** (`bool`) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step** (`bool`) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False

- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)
- **dist_sync_fn** (Optional[Callable]) – Callback that performs the allgather operation on the metric state. When *None*, DDP will be used to perform the allgather. default: None

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])
>>> map = RetrievalMAP()
>>> map(preds, target, indexes=indexes)
tensor(0.7917)
```

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

RetrievalMRR

class torchmetrics.**RetrievalMRR**(*empty_target_action='neg'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

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`
- **compute_on_step** (bool) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

- **dist_sync_fn** (Optional[Callable]) – Callback that performs the allgather operation on the metric state. When *None*, DDP will be used to perform the allgather. default: None

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.

RetrievalPrecision

class torchmetrics.**RetrievalPrecision**(*empty_target_action='neg'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*, *k=None*)

Computes 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`
- **compute_on_step** (bool) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

- **dist_sync_fn** (Optional[Callable]) – Callback that performs the allgather operation on the metric state. When *None*, DDP will be used to perform the allgather. default: None
- **k** (Optional[int]) – consider only the top k elements for each query. default: None

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.

RetrievalRecall

class torchmetrics.**RetrievalRecall**(*empty_target_action='neg'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*, *k=None*)

Computes 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`
- **compute_on_step** (bool) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

- **dist_sync_fn** (Optional[Callable]) – Callback that performs the allgather operation on the metric state. When *None*, DDP will be used to perform the allgather. default: None
- **k** (Optional[int]) – consider only the top k elements for each query. default: None

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.

RetrievalFallOut

class torchmetrics.**RetrievalFallOut**(*empty_target_action='pos'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*, *k=None*)

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`
- **compute_on_step** (bool) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

- **dist_sync_fn** (Optional[Callable]) – Callback that performs the allgather operation on the metric state. When *None*, DDP will be used to perform the allgather. default: None
- **k** (Optional[int]) – consider only the top k elements for each query. default: None

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

RetrievalNormalizedDCG

class torchmetrics.**RetrievalNormalizedDCG**(*empty_target_action='neg'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*, *k=None*)

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 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 *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`

- **compute_on_step** (bool) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)
- **dist_sync_fn** (Optional[Callable]) – Callback that performs the allgather operation on the metric state. When *None*, DDP will be used to perform the allgather. default: None
- **k** (Optional[int]) – consider only the top k elements for each query. default: None

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.

2.5.5 Wrappers

Modular wrapper metrics are not metrics in themself, but instead take a metric and alter the internal logic of the base metric.

class torchmetrics.**BootStrapper**(*base_metric*, *num_bootstraps=10*, *mean=True*, *std=True*, *quantile=None*, *raw=False*, *sampling_strategy='poisson'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Use to turn a metric into a bootstrapped metric 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 diviation 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 `'possion'` is chosen, the number of times

each sample will be included in the bootstrap will be given by $n \sim Poisson(\lambda = 1)$, 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 hole dataset.

- **compute_on_step** (bool) – Forward only calls `update()` and return `None` if this is set to `False`.
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each `forward()` before returning the value at the step
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: `None` (which selects the entire world)
- **dist_sync_fn** (Optional[Callable]) – Callback that performs the allgather operation on the metric state. When `None`, DDP will be used to perform the allgather.

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)}
```

compute()
Computes the bootstrapped metric values. Allways 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

2.6 Functional metrics

2.6.1 Classification Metrics

accuracy [func]

torchmetrics.functional.**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*)

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 predictions, the parameter `top_k` generalizes this metric to a Top-K accuracy metric: for each sample the top-K highest probability 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, or labels)
- **target** (`Tensor`) – Ground truth labels
- **average** (`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`.

- **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 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 predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class inputs with probability predictions. 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 counter 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 `threshold` is not a `float` between `0` and `1`.
- **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

auc [func]

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

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

Parameters

- **x** (Tensor) – x-coordinates
- **y** (Tensor) – y-coordinates
- **reorder** (bool) – if True, will reorder the arrays

Return type Tensor

Returns Tensor containing AUC score (float)

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 or 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.)
```

auroc [func]

torchmetrics.functional.**auroc**(*preds*, *target*, *num_classes=None*, *pos_label=None*, *average='macro'*, *max_fpr=None*, *sample_weights=None*)

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

Parameters

- **preds** (Tensor) – predictions from model (logits or probabilities)
- **target** (Tensor) – Ground truth labels
- **num_classes** (Optional[int]) – integer with number of classes. Not nessesary to provide 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[str]) –
  - '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.
- **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 Tensor

average_precision [func]

torchmetrics.functional.**average_precision**(*preds*, *target*, *num_classes=None*, *pos_label=None*, *sample_weights=None*)

Computes the average precision score.

Parameters

- **preds** (Tensor) – predictions from model (logits or probabilities)
- **target** (Tensor) – ground truth values
- **num_classes** (Optional[int]) – integer with number of classes. Not nessesary to provide 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]
- **sample_weights** (Optional[Sequence]) – sample weights for each data point

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)
[tensor(1.), tensor(1.), tensor(0.2500), tensor(0.2500), tensor(nan)]
```

cohen_kappa [func]

torchmetrics.functional.**cohen_kappa**(*preds*, *target*, *num_classes*, *weights=None*, *threshold=0.5*)

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 isg 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** (Optional[str]) – 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. default: 0.5

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

confusion_matrix [func]

torchmetrics.functional.**confusion_matrix**(*preds*, *target*, *num_classes*, *normalize=None*, *threshold=0.5*, *multilabel=False*)

Computes the confusion matrix. 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, 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.

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/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[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 value for binary or multi-label probabilities. default: 0.5
- **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)
```

(continues on next page)

(continued from previous page)

```
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

dice_score [func]

torchmetrics.functional.**dice_score**(*pred*, *target*, *bg=False*, *nan_score=0.0*, *no_fg_score=0.0*, *reduction='elementwise_mean'*)

Compute dice score from prediction scores

Parameters

- **pred** (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) – score to return, if no foreground pixel was found in target
- **reduction** (str) – 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 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)
```

f1 [func]

torchmetrics.functional.**f1**(*preds*, *target*, *beta=1.0*, *average='micro'*, *mdmc_average=None*, *ignore_index=None*, *num_classes=None*, *threshold=0.5*, *top_k=None*, *multiclass=None*, *is_multiclass=None*)

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 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.

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 or labels)
- **target** (`Tensor`) – Ground truth values
- **average** (`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`.

- **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

Example

```
>>> from torchmetrics.functional import f1
>>> target = torch.tensor([0, 1, 2, 0, 1, 2])
>>> preds = torch.tensor([0, 2, 1, 0, 0, 1])
>>> f1(preds, target, num_classes=3)
tensor(0.3333)
```

fbeta [func]

torchmetrics.functional.**fbeta**(*preds*, *target*, *beta=1.0*, *average='micro'*, *mdmc_average=None*, *ignore_index=None*, *num_classes=None*, *threshold=0.5*, *top_k=None*, *multiclass=None*, *is_multiclass=None*)

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 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.

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 or labels)
- **target** (`Tensor`) – Ground truth values
- **average** (`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`.

- **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

Example

```
>>> from torchmetrics.functional import fbeta
>>> target = torch.tensor([0, 1, 2, 0, 1, 2])
>>> preds = torch.tensor([0, 2, 1, 0, 0, 1])
>>> fbeta(preds, target, num_classes=3, beta=0.5)
tensor(0.3333)
```

hamming_distance [func]

torchmetrics.functional.**hamming_distance**(*preds*, *target*, *threshold=0.5*)

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
- **target** (Tensor) – Ground truth
- **threshold** (float) – Threshold probability value for transforming probability predictions to binary (0 or 1) predictions, in the case of binary or multi-label inputs.

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

hinge [func]

torchmetrics.functional.**hinge**(*preds*, *target*, *squared=False*, *multiclass_mode=None*)

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 $y \in 0, ..., \text{C}$ is the target class (where C is the number of classes), and $\hat{y} \in \mathbb{R}^{\text{C}}$ 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** (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.

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
>>> target = torch.tensor([0, 1, 1])
>>> preds = torch.tensor([-2.2, 2.4, 0.1])
>>> 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(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(preds, target, multiclass_mode="one-vs-all")
tensor([2.2333, 1.5000, 1.2333])
```

Return type `Tensor`

iou [func]

`torchmetrics.functional.`**iou** (*preds*, *target*, *ignore_index=None*, *absent_score=0.0*, *threshold=0.5*, *num_classes=None*, *reduction='elementwise_mean'*)

Computes Intersection over union, or Jaccard index calculation:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

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, ...]`
- **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 *pred* AND no instances of the class index were present in *target*. For example, if we have 3 classes, [0, 0] for *pred*, and [0, 2] for *target*, then class 1 would be assigned the *absent_score*.
- **threshold** (float) – Threshold value for binary or multi-label probabilities. default: 0.5
- **num_classes** (Optional[int]) – Optionally specify the number of classes
- **reduction** (str) – a method to reduce metric score over labels.
  - `'elementwise_mean'`: takes the mean (default)
  - `'sum'`: takes the sum
  - `'none'`: no reduction will be applied

Returns Tensor containing single value if reduction is 'elementwise_mean', or number of classes if reduction is 'none'

Return type IoU score

Example

```
>>> from torchmetrics.functional import iou
>>> 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]
>>> iou(pred, target)
tensor(0.9660)
```

matthews_corrcoef [func]

torchmetrics.functional.**matthews_corrcoef**(*preds*, *target*, *num_classes*, *threshold=0.5*)

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 postitives, 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. default: 0.5

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

roc [func]

torchmetrics.functional.**roc**(*preds*, *target*, *num_classes=None*, *pos_label=None*, *sample_weights=None*)

Computes the Receiver Operating Characteristic (ROC). Works with both binary, multiclass and multilabel input.

Parameters

- **preds** (Tensor) – predictions from model (logits or probabilities)
- **target** (Tensor) – ground truth values
- **num_classes** (Optional[int]) – integer with number of classes. Not nessesary to provide 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]
- **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.])]
>>> 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])]
```

precision [func]

torchmetrics.functional.**precision**(*preds*, *target*, *average='micro'*, *mdmc_average=None*, *ignore_index=None*, *num_classes=None*, *threshold=0.5*, *top_k=None*, *multiclass=None*, *is_multiclass=None*)

Computes Precision:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Where TP and FP represent the number of true positives and false positives respecitively. 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 or labels)
- **target** (`Tensor`) – Ground truth values
- **average** (`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`.

- **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 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)
```

precision_recall [func]

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*, *is_multiclass=None*)

Computes Precision and Recall:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where TPtext{FN}` and FP represent the number of true positives, false negatives and false positives respecitively. 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, or labels)
- **target** (`Tensor`) – Ground truth values
- **average** (`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`.

- **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.

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))
```

precision_recall_curve [func]

torchmetrics.functional.**precision_recall_curve**(*preds*, *target*, *num_classes=None*, *pos_label=None*, *sample_weights=None*)

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. Not nessesary to provide 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]
- **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.])]
>>> thresholds
[tensor([0.7500]), tensor([0.7500]), tensor([0.0500, 0.7500]), tensor([0.0500,
↪ 0.7500]), tensor([0.0500])]
```

recall [func]

torchmetrics.functional.**recall**(*preds*, *target*, *average='micro'*, *mdmc_average=None*, *ignore_index=None*, *num_classes=None*, *threshold=0.5*, *top_k=None*, *multiclass=None*, *is_multiclass=None*)

Computes Recall:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where TP and FN represent the number of true positives and false negatives respecitively. 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

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

- **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 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)
```

select_topk [func]

torchmetrics.utilities.data.**select_topk**(*prob_tensor*, *topk=1*, *dim=1*)

Convert a probability tensor to binary by selecting top-k 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 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)
```

stat_scores [func]

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*, *is_multiclass=None*)

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 or labels)
- **target** (Tensor) – Ground truth values
- **threshold** (float) – Threshold probability value for transforming probability predictions to binary (0 or 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.

- **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 handeled. 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])
```

to_categorical [func]

torchmetrics.utilities.data.**to_categorical**(*tensor*, *argmax_dim=1*)

Converts a tensor of probabilities to a dense label tensor

Parameters

- **tensor** (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])
```

to_onehot [func]

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]])
```

2.6.2 Regression Metrics

explained_variance [func]

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 (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

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]]

image_gradients [func]

torchmetrics.functional.**image_gradients**(*img*)

Computes the gradients 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

mean_absolute_error [func]

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)
```

mean_squared_error [func]

torchmetrics.functional.**mean_squared_error**(*preds*, *target*)

Computes mean squared error

Parameters

- **preds** (Tensor) – estimated labels
- **target** (Tensor) – ground truth labels

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)
```

mean_squared_log_error [func]

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

pearson_corrcoef [func]

torchmetrics.functional.**pearson_corrcoef**(*preds*, *target*)

Computes pearson correlation coefficient.

Parameters

- **preds** (Tensor) – estimated scores
- **target** (Tensor) – ground truth scores

Example

```
>>> 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)
```

Return type Tensor

psnr [func]

torchmetrics.functional.**psnr**(*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) – groun 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 (default: 10)
- **reduction** (str) – a method to reduce metric score over labels.
  - 'elementwise_mean': takes the mean (default)
  - 'sum': takes the sum
  - '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 psnr
>>> pred = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
>>> target = torch.tensor([[3.0, 2.0], [1.0, 0.0]])
>>> psnr(pred, target)
tensor(2.5527)
```

Note: Half precision is only support on GPU for this metric

r2score [func]

torchmetrics.functional.**r2score**(*preds*, *target*, *adjusted=0*, *multioutput='uniform_average'*)

Computes r2 score also known as coefficient of 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.

Parameters

- **preds** (Tensor) – estimated labels
- **target** (Tensor) – ground truth labels
- **adjusted** (int) – number of independent regressors for calculating adjusted r2 score. Default 0 (standard r2 score).
- **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

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 sampels 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 r2score
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> 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(preds, target, multioutput='raw_values')
tensor([0.9654, 0.9082])
```

Return type Tensor

spearman_corrcoef [func]

torchmetrics.functional.**spearman_corrcoef**(*preds*, *target*)

Computes spearmans rank correlation coefficient:

where rg_x and rg_y are the rank associated to the variables x and y. Spearmans correlations coefficient corresponds to the standard pearsons correlation coefficient calculated on the rank variables.

Parameters

- **preds** (Tensor) – estimated scores
- **target** (Tensor) – ground truth scores

Example

```
>>> 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)
```

Return type Tensor

ssim [func]

torchmetrics.functional.**ssim**(*preds*, *target*, *kernel_size=(11, 11)*, *sigma=(1.5, 1.5)*, *reduction='elementwise_mean'*, *data_range=None*, *k1=0.01*, *k2=0.03*)

Computes Structual Similarity Index Measure

Parameters

- **preds** (Tensor) – estimated image
- **target** (Tensor) – ground truth image
- **kernel_size** (Sequence[int]) – size of the gaussian kernel (default: (11, 11))
- **sigma** (Sequence[float]) – Standard deviation of the gaussian kernel (default: (1.5, 1.5))
- **reduction** (str) – a method to reduce metric score over labels.
  - 'elementwise_mean': takes the mean (default)
  - 'sum': takes the sum
  - '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. Default: 0.01
- **k2** (float) – Parameter of SSIM. Default: 0.03

Return type 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 ssim
>>> preds = torch.rand([16, 1, 16, 16])
>>> target = preds * 0.75
>>> ssim(preds, target)
tensor(0.9219)
```

2.6.3 NLP

bleu_score [func]

torchmetrics.functional.**bleu_score**(*translate_corpus*, *reference_corpus*, *n_gram=4*, *smooth=False*)

Calculate BLEU score of machine translated text with one or more references

Parameters

- **translate_corpus** (Sequence[str]) – An iterable of machine translated corpus
- **reference_corpus** (Sequence[str]) – An iterable of iterables of reference corpus
- **n_gram** (int) – Gram value ranged from 1 to 4 (Default 4)
- **smooth** (bool) – Whether or not to apply smoothing – Lin et al. 2004

Return type Tensor

Returns Tensor with BLEU Score

Example

```
>>> from torchmetrics.functional import bleu_score
>>> translate_corpus = ['the cat is on the mat'.split()]
>>> reference_corpus = [['there is a cat on the mat'.split(), 'a cat is on the mat
↪'.split()]]
>>> bleu_score(translate_corpus, reference_corpus)
tensor(0.7598)
```

2.6.4 Pairwise

embedding_similarity [func]

torchmetrics.functional.**embedding_similarity**(*batch*, *similarity='cosine'*, *reduction='none'*, *zero_diagonal=True*)

Computes representation similarity

Example

```
>>> from torchmetrics.functional import embedding_similarity
>>> embeddings = torch.tensor([[1., 2., 3., 4.], [1., 2., 3., 4.], [4., 5., 6., 7.
↪]])
>>> embedding_similarity(embeddings)
tensor([[0.0000, 1.0000, 0.9759],
        [1.0000, 0.0000, 0.9759],
        [0.9759, 0.9759, 0.0000]])
```

Parameters

- **batch** (Tensor) – (batch, dim)
- **similarity** (str) – 'dot' or 'cosine'
- **reduction** (str) – 'none', 'sum', 'mean' (all along dim -1)
- **zero_diagonal** (bool) – if True, the diagonals are set to zero

Return type Tensor

Returns A square matrix (batch, batch) with the similarity scores between all elements If sum or mean are used, then returns (b, 1) with the reduced value for each row

2.6.5 Retrieval

retrieval_average_precision [func]

torchmetrics.functional.**retrieval_average_precision**(*preds*, *target*)

Computes average precision (for information retrieval), as explained here.

`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)
```

retrieval_reciprocal_rank [func]

torchmetrics.functional.**retrieval_reciprocal_rank**(*preds*, *target*)

Computes reciprocal rank (for information retrieval), as explained here.

`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)
```

retrieval_precision [func]

torchmetrics.functional.**retrieval_precision**(*preds*, *target*, *k=None*)

Computes the precision metric (for information retrieval), as explained here. 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)

Return type `Tensor`

Returns a single-value tensor with the precision (at `k`) 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_precision(preds, target, k=2)
tensor(0.5000)
```

retrieval_recall [func]

torchmetrics.functional.**retrieval_recall**(*preds*, *target*, *k=None*)

Computes the recall metric (for information retrieval), as explained here. 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)

Return type `Tensor`

Returns a single-value tensor with the recall (at `k`) of the predictions `preds` w.r.t. the labels `target`.

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)
```

retrieval_fall_out [func]

torchmetrics.functional.**retrieval_fall_out**(*preds*, *target*, *k=None*)

Computes the Fall-out (for information retrieval), as explained here. 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)

Return type `Tensor`

Returns a single-value tensor with the fall-out (at `k`) of the predictions `preds` w.r.t. the labels `target`.

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.)
```

retrieval_normalized_dcg [func]

`torchmetrics.functional.`**`retrieval_normalized_dcg`**(*preds*, *target*, *k=None*)

Computes Normalized Discounted Cumulative Gain (for information retrieval), as explained here.

`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)

Return type `Tensor`

Returns a single-value tensor with the nDCG of the predictions `preds` w.r.t. the labels `target`.

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)
```

2.7 Contributor Covenant Code of Conduct

2.7.1 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.

2.7.2 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

2.7.3 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.

2.7.4 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.

2.7.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at 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.

2.7.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, 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

2.8 Contributing

Welcome to the Torchmetrics community! We're building largest collection of native pytorch metrics, with the goal of reducing boilerplate and increasing reproducibility.

2.8.1 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.

2.8.2 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
```

(continues on next page)

(continued from previous page)

```
    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.

2.9 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.

2.9.1 [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)):
  - Added `RetrievalMAP` (#5032)
  - Added `RetrievalMRR` (#119)
  - Added `RetrievalPrecision` (#139)
  - Added `RetrievalRecall` (#146)
  - Added `RetrievalNormalizedDCG` (#160)
  - Added `RetrievalFallOut` (#161)
- Added other metrics:
  - Added `CohenKappa` (#69)
  - Added `MatthewsCorrcoef` (#98)
  - Added `PearsonCorrcoef` (#157)
  - Added `SpearmanCorrcoef` (#158)
  - Added `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>=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)

2.9.2 [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)

2.9.3 [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)

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX