PyTorch-Metrics Documentation

Release 0.7.1

PyTorchLightning et al.

Feb 03, 2022

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.

USING TORCHMETRICS

1.1 Module metrics

```python
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

```python
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

```python
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
```

TWO

MORE READING

2.1 Quick Start

TorchMetrics is a collection of 60+ PyTorch metrics implementations and an easy-to-use API to create custom metrics. It offers:

- A standardized interface to increase reproducibility

- Reduces Boilerplate

- Distributed-training compatible

- Rigorously tested

- Automatic accumulation over batches

- Automatic synchronization between multiple devices

You can use TorchMetrics in any PyTorch model, or within PyTorch Lightning to enjoy additional features:

- This means that your data will always be placed on the same device as your metrics.

- Native support for logging metrics in Lightning to reduce even more boilerplate.

2.1.1 Install

You can install TorchMetrics using pip or conda:

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

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

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

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:

```python
import torch
# import our library
import torchmetrics

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

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

Module metrics

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

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

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

```python
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 a `torch.nn.Module`. Simply, subclass *Metric* and do the following:

1. Implement `__init__` where you call `self.add_state` for every internal state that is needed for the metrics computations

2. Implement `update` method, where all logic that is necessary for updating metric states go

3. Implement `compute` method, where the final metric computations happens

For practical examples and more info about implementing a metric, please see this *page*.

Development Environment

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

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:

```python
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)
```

```python
# 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:

```python
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.

```python
from torchmetrics import Accuracy, MetricCollection

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

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

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

Metrics in Dataparallel (DP) mode

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

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

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

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

Metrics in Distributed Data Parallel (DDP) mode

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

    - *PeakSignalNoiseRatio* and *peak_signal_noise_ratio [func]*

    - *StructuralSimilarityIndexMeasure* and *structural_similarity_index_measure [func]*

    - *KLDivergence* and *kl_divergence [func]*

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

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

```
        self.valid_metrics = metrics.clone(prefix='val_')

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

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

add_metrics(*metrics*, **additional_metrics*)
    Add new metrics to Metric Collection.

        Return type None

clone(*prefix=None*, *postfix=None*)
    Make a copy of the metric collection :type _sphinx_paramlinks_torchmetrics.MetricCollection.clone.prefix: Optional[str] :param _sphinx_paramlinks_torchmetrics.MetricCollection.clone.prefix: a string to append in front of the metric keys :type _sphinx_paramlinks_torchmetrics.MetricCollection.clone.postfix: Optional[str] :param _sphinx_paramlinks_torchmetrics.MetricCollection.clone.postfix: a string to append after the keys of the output dict

        Return type MetricCollection

forward(**args*, ***kwargs*)
    Iteratively call forward for each metric.

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

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

> > **items**(*keep_base=False*)
> > > Return an iterable of the ModuleDict key/value pairs. :type _sphinx_paramlinks_torchmetrics.MetricCollection.items.keep_b
> > > `bool` :param _sphinx_paramlinks_torchmetrics.MetricCollection.items.keep_base: Whether to add prefix/postfix on the items collection.

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

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

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

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

> > > > **Return type** `None`

> > **reset**()
> > > Iteratively call reset for each metric.

> > > > **Return type** `None`

> > **update**(**args*, ***kwargs*)
> > > Iteratively call update for each metric.

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

> > > > **Return type** `None`

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 differentiable or not.

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

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

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

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.

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

Example implementation:

```python
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.

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

- **dist_reduce_fx** (*Optional*) – Function to reduce state across multiple processes in distributed mode. If value is `"sum"`, `"mean"`, `"cat"`, `"min"` or `"max"` we will use `torch.sum`, `torch.mean`, `torch.cat`, `torch.min` and `torch.max`` respectively, each with argument `dim=0`. Note that the `"cat"` reduction only makes sense if the state is a list, and not a tensor. The user can also pass a custom function in this parameter.

- **persistent** (*Optional*) – whether the state will be saved as part of the modules `state_dict`. Default is `False`.

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

The metric states would be synced as follows

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

- If the metric state is a `list`, the synced value will be a `list` containing the combined elements from all processes.

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

Raises

- **ValueError** – If `default` is not a `tensor` or an `empty list`.

- **ValueError** – If `dist_reduce_fx` is not callable or one of `"mean"`, `"sum"`, `"cat"`, None.

Return type None

clone()
Make a copy of the metric.

Return type *Metric*

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

Return type Any

double()
Method override default and prevent dtype casting.

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

Return type *Metric*

float()
> Method override default and prevent dtype casting.
>
> Please use *metric.set_dtype(dtype)* instead.
>
> > **Return type** `Metric`

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

half()
> Method override default and prevent dtype casting.
>
> Please use *metric.set_dtype(dtype)* instead.
>
> > **Return type** `Metric`

persistent(*mode=False*)
> Method for post-init to change if metric states should be saved to its state_dict.
>
> > **Return type** `None`

reset()
> This method automatically resets the metric state variables to their default value.
>
> > **Return type** `None`

set_dtype(*dst_type*)
> Special version of *type* for transferring all metric states to specific dtype :type
> _sphinx_paramlinks_torchmetrics.Metric.set_dtype.dst_type: `Union[str, dtype]` :param
> _sphinx_paramlinks_torchmetrics.Metric.set_dtype.dst_type: the desired type :type
> _sphinx_paramlinks_torchmetrics.Metric.set_dtype.dst_type: type or string
>
> > **Return type** `None`

state_dict(*destination=None*, *prefix=''*, *keep_vars=False*)
> Returns a dictionary containing a whole state of the module.
>
> Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.
>
> > **Returns** a dictionary containing a whole state of the module
> >
> > **Return type** `dict`
>
> Example:

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

sync(*dist_sync_fn=None*, *process_group=None*, *should_sync=True*, *distributed_available=<function jit_distributed_available>*)
> Sync function for manually controlling when metrics states should be synced across processes.
>
> > **Parameters**
> >
> > - **dist_sync_fn** (`Optional[Callable]`) – Function to be used to perform states synchronization
> >
> > - **process_group** (`Optional[Any]`) – Specify the process group on which synchronization is called. default: *None* (which selects the entire world)

- **should_sync**⫝ (`bool`) – Whether to apply to state synchronization. This will have an impact only when running in a distributed setting.

- **distributed_available**⫝ (`Optional[Callable]`) – Function to determine if we are running inside a distributed setting

   Return type `None`

sync_context(*dist_sync_fn=None*, *process_group=None*, *should_sync=True*, *should_unsync=True*, *distributed_available=<function jit_distributed_available>*)

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

   Parameters

- **dist_sync_fn**⫝ (`Optional[Callable]`) – Function to be used to perform states synchronization

- **process_group**⫝ (`Optional[Any]`) – Specify the process group on which synchronization is called. default: *None* (which selects the entire world)

- **should_sync**⫝ (`bool`) – Whether to apply to state synchronization. This will have an impact only when running in a distributed setting.

- **should_unsync**⫝ (`bool`) – Whether to restore the cache state so that the metrics can continue to be accumulated.

- **distributed_available**⫝ (`Optional[Callable]`) – Function to determine if we are running inside a distributed setting

   Return type `Generator`

type(*dst_type*)

Method override default and prevent dtype casting.

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

   Return type `Metric`

unsync(*should_unsync=True*)

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

   Parameters should_unsync⫝ (`bool`) – Whether to perform unsync

   Return type `None`

abstract update(**_*, ***__*)

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

   Return type `None`

property device: torch.device

Return the device of the metric.

   Return type `device`

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:

```python
class MyModel(LightningModule):

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

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

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

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

2.4.1 Logging TorchMetrics

When *Metric* objects, which return a scalar tensor, are logged directly in Lightning using the LightningModule self.log method. Lightning will log the metric based on on_step and on_epoch flags present in self.log(...). If on_epoch is True, the logger automatically logs the end of epoch metric value by calling .compute().

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

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

```python
class MyModule(LightningModule):

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

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

    def validation_step(self, batch, batch_idx):
        logits = self(x)
        ...
        self.valid_acc(logits, y)
        self.log('valid_acc', self.valid_acc, on_step=True, on_epoch=True)
```

Note: the .reset() method of the metric will automatically be called at the end of an epoch within lightning only if you pass the metric instance inside self.log. Also if you are calling .compute by yourself, you need to call the .reset() too.

```python
class MyModule(LightningModule):

    def __init__(self):
        ...
        self.train_acc = torchmetrics.Accuracy()
        self.train_precision = torchmetrics.Precision()

    def training_step(self, batch, batch_idx):
        x, y = batch
        preds = self(x)
        ...

        # this will reset the metric automatically at the epoch end
        self.train_acc(preds, y)
        self.log('train_acc', self.train_acc, on_step=True, on_epoch=False)
```

(continues on next page)

```
        # this will not reset the metric automatically at the epoch end
        precision = self.train_precision(preds, y)
        self.log('train_precision', precision, on_step=True, on_epoch=False)

    def training_epoch_end(self, outputs):
        # this will compute and reset the metric automatically at the epoch end
        self.log('train_epoch_accuracy', self.training_acc)

        # this will not reset the metric automatically at the epoch end so you
        # need to call it yourself
        mean_precision = self.precision.compute()
        self.log('train_epoch_precision', mean_precision)
        self.precision.reset()
```

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:

```
class MyModule(LightningModule):

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

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

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

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

The metric states would be synced as follows

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

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

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

clone()

> Make a copy of the metric.
>
> **Return type** *Metric*

abstract compute()

> Override this method to compute the final metric value from state variables synchronized across the distributed backend.
>
> **Return type** Any

double()

> Method override default and prevent dtype casting.
>
> Please use *metric.set_dtype(dtype)* instead.
>
> **Return type** *Metric*

float()

> Method override default and prevent dtype casting.
>
> Please use *metric.set_dtype(dtype)* instead.
>
> **Return type** *Metric*

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

> Automatically calls `update()`.
>
> Returns the metric value over inputs if `compute_on_step` is True.
>
> **Return type** Any

half()

> Method override default and prevent dtype casting.
>
> Please use *metric.set_dtype(dtype)* instead.
>
> **Return type** *Metric*

persistent(*mode=False*)

> Method for post-init to change if metric states should be saved to its state_dict.
>
> **Return type** None

reset()

> This method automatically resets the metric state variables to their default value.
>
> **Return type** None

---

set_dtype(*dst_type*)

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

>**Return type** None

state_dict(*destination=None*, *prefix=''*, *keep_vars=False*)

Returns a dictionary containing a whole state of the module.

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

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

>**Return type** dict

Example:

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

sync(*dist_sync_fn=None*, *process_group=None*, *should_sync=True*, *distributed_available=<function jit_distributed_available>*)

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

>**Parameters**
>
>* **dist_sync_fn** (Optional[Callable]) – Function to be used to perform states synchronization
>* **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: *None* (which selects the entire world)
>* **should_sync** (bool) – Whether to apply to state synchronization. This will have an impact only when running in a distributed setting.
>* **distributed_available** (Optional[Callable]) – Function to determine if we are running inside a distributed setting

>**Return type** None

sync_context(*dist_sync_fn=None*, *process_group=None*, *should_sync=True*, *should_unsync=True*, *distributed_available=<function jit_distributed_available>*)

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

>**Parameters**
>
>* **dist_sync_fn** (Optional[Callable]) – Function to be used to perform states synchronization
>* **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: *None* (which selects the entire world)
>* **should_sync** (bool) – Whether to apply to state synchronization. This will have an impact only when running in a distributed setting.
>* **should_unsync** (bool) – Whether to restore the cache state so that the metrics can continue to be accumulated.
>* **distributed_available** (Optional[Callable]) – Function to determine if we are running inside a distributed setting

> **Return type** `Generator`

type(*dst_type*)

> Method override default and prevent dtype casting.
>
> Please use *metric.set_dtype(dtype)* instead.
>
> > **Return type** `Metric`

unsync(*should_unsync=True*)

> Unsync function for manually controlling when metrics states should be reverted back to their local states.
>
> > **Parameters** **should_unsync**⫯ (`bool`) – Whether to perform unsync
> >
> > **Return type** `None`

abstract update(**_, **__*)

> Override this method to update the state variables of your metric class.
>
> > **Return type** `None`

property device: torch.device

> Return the device of the metric.
>
> > **Return type** `device`

2.5.2 Basic Aggregation

Torchmetrics comes with a number of metrics for aggregation of basic statistics: mean, max, min etc. of either tensors or native python floats.

CatMetric

class torchmetrics.**CatMetric**(*nan_strategy='warn'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

> Concatenate a stream of values.
>
> > **Parameters**
> >
> > - **nan_strategy**⫯ (`Union`[`str`, `float`]) – options: - `'error'`: if any *nan* values are encounted will give a RuntimeError - `'warn'`: if any *nan* values are encounted will give a warning and continue - `'ignore'`: all *nan* values are silently removed - a float: if a float is provided will impude any *nan* values with this value
> >
> > - **compute_on_step**⫯ (`bool`) – Forward only calls `update()` and returns 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.
> >
> > - **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 nan_strategy is not one of `error`, `warn`, `ignore` or a float

Example

```
>>> from torchmetrics import CatMetric
>>> metric = CatMetric()
>>> metric.update(1)
>>> metric.update(torch.tensor([2, 3]))
>>> metric.compute()
tensor([1., 2., 3.])
```

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

compute()
>   Compute the aggregated value.

>   >   **Return type** `Tensor`

update(*value*)
>   Update state with data.

>   >   **Parameters value** (`Union[float, Tensor]`) – Either a float or tensor containing data. Additional tensor dimensions will be flattened

>   >   **Return type** `None`

MaxMetric

class torchmetrics.**MaxMetric**(*nan_strategy='warn'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)
>   Aggregate a stream of value into their maximum value.

>   **Parameters**

>   - **nan_strategy** (`Union[str, float]`) – options: - `'error'`: if any *nan* values are encounted will give a RuntimeError - `'warn'`: if any *nan* values are encounted will give a warning and continue - `'ignore'`: all *nan* values are silently removed - a float: if a float is provided will impude any *nan* values with this value

>   - **compute_on_step** (`bool`) – Forward only calls `update()` and returns 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.

>   - **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 nan_strategy is not one of `error`, `warn`, `ignore` or a float

Example

```
>>> from torchmetrics import MaxMetric
>>> metric = MaxMetric()
>>> metric.update(1)
>>> metric.update(torch.tensor([2, 3]))
>>> metric.compute()
tensor(3.)
```

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

update(*value*)
>   Update state with data.

>>   **Parameters value** (Union[float, Tensor]) – Either a float or tensor containing data. Additional tensor dimensions will be flattened

>>   **Return type** None

MeanMetric

class torchmetrics.**MeanMetric**(*nan_strategy='warn'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

> Aggregate a stream of value into their mean value.

>>   **Parameters nan_strategy** (Union[str, float]) –

>>>   **options:**

>>>>   • 'error': if any *nan* values are encounted will give a RuntimeError

>>>>   • 'warn': if any *nan* values are encounted will give a warning and continue

>>>>   • 'ignore': all *nan* values are silently removed

>>>>   • a float: if a float is provided will impude any *nan* values with this value

>>>   **compute_on_step:** Forward only calls update() and returns None if this is set to False.

>>>   **dist_sync_on_step:** Synchronize metric state across processes at each forward() before returning the value at the step.

>>>   **process_group:** Specify the process group on which synchronization is called.

>>>   **dist_sync_fn:** Callback that performs the allgather operation on the metric state. When *None*, DDP will be used to perform the allgather.

>>   **Raises ValueError** – If nan_strategy is not one of error, warn, ignore or a float

Example

```
>>> from torchmetrics import MeanMetric
>>> metric = MeanMetric()
>>> metric.update(1)
>>> metric.update(torch.tensor([2, 3]))
>>> metric.compute()
tensor(2.)
```

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

compute()
>   Compute the aggregated value.

>> **Return type** Tensor

update(*value*, *weight=1.0*)
>   Update state with data.

>> **Parameters**

>>> - **value**⫽ (Union[float, Tensor]) – Either a float or tensor containing data. Additional tensor dimensions will be flattened

>>> - **weight**⫽ (Union[float, Tensor]) – Either a float or tensor containing weights for calculating the average. Shape of weight should be able to broadcast with the shape of *value*. Default to *1.0* corresponding to simple harmonic average.

>> **Return type** None

MinMetric

class torchmetrics.**MinMetric**(*nan_strategy='warn'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

>   Aggregate a stream of value into their minimum value.

>> **Parameters**

>>> - **nan_strategy**⫽ (Union[str, float]) – options: - 'error': if any *nan* values are encounted will give a RuntimeError - 'warn': if any *nan* values are encounted will give a warning and continue - 'ignore': all *nan* values are silently removed - a float: if a float is provided will impude any *nan* values with this value

>>> - **compute_on_step**⫽ (bool) – Forward only calls update() and returns 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.

>>> - **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 nan_strategy is not one of error, warn, ignore or a float

Example

```
>>> from torchmetrics import MinMetric
>>> metric = MinMetric()
>>> metric.update(1)
>>> metric.update(torch.tensor([2, 3]))
>>> metric.compute()
tensor(1.)
```

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

update(*value*)
>   Update state with data.

> **Parameters value** (Union[float, Tensor]) – Either a float or tensor containing data. Additional tensor dimensions will be flattened
>
> **Return type** None

SumMetric

class torchmetrics.**SumMetric**(*nan_strategy='warn'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

> Aggregate a stream of value into their sum.
>
> **Parameters**
>
> - **nan_strategy** (Union[str, float]) – options: - 'error': if any *nan* values are encounted will give a RuntimeError - 'warn': if any *nan* values are encounted will give a warning and continue - 'ignore': all *nan* values are silently removed - a float: if a float is provided will impude any *nan* values with this value
>
> - **compute_on_step** (bool) – Forward only calls update() and returns 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.
>
> - **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 nan_strategy is not one of error, warn, ignore or a float

Example

```
>>> from torchmetrics import SumMetric
>>> metric = SumMetric()
>>> metric.update(1)
>>> metric.update(torch.tensor([2, 3]))
>>> metric.compute()
tensor(6.)
```

> Initializes internal Module state, shared by both nn.Module and ScriptModule.
>
> **update**(*value*)
>
> > Update state with data.
> >
> > **Parameters value** (Union[float, Tensor]) – Either a float or tensor containing data. Additional tensor dimensions will be flattened
> >
> > **Return type** None

2.5.3 Audio

For the purposes of audio metrics, inputs (predictions, targets) must have the same size. If the input is 1D tensors the output will be a scalar. If the input is multi-dimensional with shape `[...,time]` the metric will be computed over the `time` dimension.

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

PerceptualEvaluationSpeechQuality

class torchmetrics.audio.pesq.**PerceptualEvaluationSpeechQuality**(*fs*, *mode*,
compute_on_step=True,
dist_sync_on_step=False,
process_group=None,
dist_sync_fn=None)

Perceptual Evaluation of Speech Quality (PESQ)

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

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

Forward accepts

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

    Parameters

- **fs** (int) – sampling frequency, should be 16000 or 8000 (Hz)
- **mode** (str) – 'wb' (wide-band) or 'nb' (narrow-band)
- **keep_same_device** – whether to move the pesq value to the device of preds
- **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[[Tensor], Tensor]]) – Callback that performs the allgather operation on the metric state. When None, DDP will be used to perform the all-gather

Raises

- **ModuleNotFoundError** – If peqs package is not installed

- **ValueError** – If fs is not either 8000 or 16000

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

Example

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

References

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

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

compute()
> Computes average PESQ.
>
> > **Return type** Tensor

update(*preds*, *target*)
> Update state with predictions and targets.
>
> > **Parameters**
> >
> > - **preds** (Tensor) – Predictions from model
> >
> > - **target** (Tensor) – Ground truth values
> >
> > **Return type** None

PermutationInvariantTraining

class torchmetrics.**PermutationInvariantTraining**(*metric_func*, *eval_func='max'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*, ***kwargs*)

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

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

Forward accepts

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

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

Parameters

- **metric_func** (Callable) – a metric function accept a batch of target and estimate, i.e. metric_func(preds[:, i, . . .], target[:, j, . . .]), and returns a batch of metric tensors [batch]

- **eval_func** (str) – the function to find the best permutation, can be 'min' or 'max', i.e. the smaller the better or the larger the better.

- **compute_on_step** (bool) – Forward only calls update() and returns 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.

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

- **kwargs** (Dict[str, Any]) – additional args for metric_func

Returns average PermutationInvariantTraining metric

Example

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

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

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

compute()

> Computes average PermutationInvariantTraining metric.
>
> > **Return type** `Tensor`

update(*preds*, *target*)

> Update state with predictions and targets.
>
> > **Parameters**
> >
> > - **preds**¶ (`Tensor`) – Predictions from model
> >
> > - **target**¶ (`Tensor`) – Ground truth values
> >
> > **Return type** `None`

SignalDistortionRatio

class torchmetrics.**SignalDistortionRatio**(*use_cg_iter=None*, *filter_length=512*, *zero_mean=False*, *load_diag=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Signal to Distortion Ratio (SDR) [1,2,3]

Forward accepts

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

> **Parameters**
>
> - **use_cg_iter**¶ (`Optional`[`int`]) – If provided, an iterative method is used to solve for the distortion filter coefficients instead of direct Gaussian elimination. This can speed up the computation of the metrics in case the filters are long. Using a value of 10 here has been shown to provide good accuracy in most cases and is sufficient when using this loss to train neural separation networks.
>
> - **filter_length**¶ (`int`) – The length of the distortion filter allowed
>
> - **zero_mean**¶ (`bool`) – When set to True, the mean of all signals is subtracted prior to computation of the metrics
>
> - **load_diag**¶ (`Optional`[`float`]) – If provided, this small value is added to the diagonal coefficients of the system metrics when solving for the filter coefficients. This can help stabilize the metric in the case where some of the reference signals may sometimes be zero
>
> - **compute_on_step**¶ (`bool`) – Forward only calls `update()` and returns 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.
>
> - **dist_sync_fn**¶ (`Optional`[`Callable`[[`Tensor`], `Tensor`]]) – Callback that performs the allgather operation on the metric state. When *None*, DDP will be used to perform the all-gather.
>
> **Raises** `ModuleNotFoundError` – If `fast-bss-eval` package is not installed

Example

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

Note:

1. **when pytorch<1.8.0, numpy will be used to calculate this metric, which causes `sdr` to be** non-differentiable and slower to calculate

2. using this metrics requires you to have `fast-bss-eval` install. Either install as `pip install torchmetrics[audio]` or `pip install fast-bss-eval`

3. preds and target need to have the same dtype, otherwise target will be converted to preds' dtype

References

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

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

[3] https://github.com/fakufaku/fast_bss_eval

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

compute()
    Computes average SDR.

        Return type Tensor

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

        Parameters

            • **preds**¶ (Tensor) – Predictions from model

            • **target**¶ (Tensor) – Ground truth values

        Return type None

ScaleInvariantSignalDistortionRatio

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

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

Forward accepts

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

Parameters

- **zero_mean**⫲ (bool) – if to zero mean target and preds or not

- **compute_on_step**⫲ (bool) – Forward only calls update() and returns 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.

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

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

Returns average si-sdr value

Example

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

References

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

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

compute()
    Computes average SI-SDR.

        Return type Tensor

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

> **Parameters**
>
> - **preds**⫽ (Tensor) – Predictions from model
> - **target**⫽ (Tensor) – Ground truth values
>
> **Return type** None

ScaleInvariantSignalNoiseRatio

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

> Scale-invariant signal-to-noise ratio (SI-SNR).
>
> Forward accepts
>
> - preds: shape `[...,time]`
> - target: shape `[...,time]`
>
> > **Parameters**
> >
> > - **compute_on_step**⫽ (bool) – Forward only calls `update()` and returns 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.
> > - **dist_sync_fn**⫽ (Optional[Callable[[Tensor], Tensor]]) – Callback that performs the allgather operation on the metric state. When *None*, DDP will be used to perform the all-gather.
>
> **Raises** TypeError – if target and preds have a different shape
>
> **Returns** average si-snr value

Example

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

References

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

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

`compute()`
> Computes average SI-SNR.
>
> > **Return type** `Tensor`

`update(preds, target)`
> Update state with predictions and targets.
>
> > **Parameters**
> >
> > - **preds** (`Tensor`) – Predictions from model
> >
> > - **target** (`Tensor`) – Ground truth values
> >
> > **Return type** `None`

SignalNoiseRatio

`class torchmetrics.SignalNoiseRatio(zero_mean=False, compute_on_step=True, dist_sync_on_step=False, process_group=None, dist_sync_fn=None)`

Signal-to-noise ratio (SNR):

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

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

Forward accepts

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

> **Parameters**
>
> - **zero_mean** (`bool`) – if to zero mean target and preds or not
>
> - **compute_on_step** (`bool`) – Forward only calls `update()` and returns 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.
>
> - **dist_sync_fn** (`Optional[Callable[[Tensor], Tensor]]`) – Callback that performs the allgather operation on the metric state. When *None*, DDP will be used to perform the all-gather.
>
> **Raises** `TypeError` – if target and preds have a different shape
>
> **Returns** average snr value

Example

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

References

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

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

compute()
  Computes average SNR.

> **Return type** Tensor

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

> **Parameters**
>
> - **preds**⫽ (Tensor) – Predictions from model
>
> - **target**⫽ (Tensor) – Ground truth values
>
> **Return type** None

ShortTimeObjectiveIntelligibility

class torchmetrics.audio.stoi.**ShortTimeObjectiveIntelligibility**(*fs*, *extended=False*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

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

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

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

Forward accepts

- preds: shape [...,time]

- target: shape [...,time]

Parameters

- **fs**⚸ (int) – sampling frequency (Hz)

- **extended**⚸ (bool) – whether to use the extended STOI described in [4]

- **compute_on_step**⚸ (bool) – Forward only calls update() and returns 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.

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

Returns average STOI value

Raises `ModuleNotFoundError` – If pystoi package is not installed

Example

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

References

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

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

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

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

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

compute()
    Computes average STOI.

        Return type Tensor

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

        Parameters

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

Return type None

2.5.4 Classification

Input types

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

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

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

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

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

```python
# 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.

```python
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.

```python
>>> 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.

```python
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.

```python
>>> 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 or logits predictions, the parameter `top_k` generalizes this metric to a Top-K accuracy metric: for each sample the top-K highest probability or logit score items are considered to find the correct label.

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

Accepts all input types listed in *Input types*.

> **Parameters**
>
> - **num_classes**⫲ (`Optional[int]`) – Number of classes. Necessary for `'macro'`, `'weighted'` and `None` average methods.
>
> - **threshold**⫲ (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
>
> - **average**⫲ (`str`) – Defines the reduction that is applied. Should be one of the following:
>
>   - `'micro'` [default]: Calculate the metric globally, across all samples and classes.
>
>   - `'macro'`: Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
>
>   - `'weighted'`: Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (`tp + fn`).
>
>   - `'none'` or `None`: Calculate the metric for each class separately, and return the metric for every class.
>
>   - `'samples'`: Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).
>
> ---
>
> **Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_average`.
>
> ---
>
> ---
>
> **Note:** If `'none'` and a given class doesn't occur in the *preds* or *target*, the value for the class will be `nan`.
>
> ---
>
> - **mdmc_average**⫲ (`Optional[str]`) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
>
>   - `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
>
>   - `'samplewise'`: In this case, the statistics are computed separately for each sample on the `N` axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes `...` (see *Input types*) as the `N` dimension within the sample, and computing the metric for the sample based on that.
>
>   - `'global'`: In this case the `N` and `...` dimensions of the inputs (see *Input types*) are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were (`N_X, C`). From here on the `average` parameter applies as usual.
>
> - **ignore_index**⫲ (`Optional[int]`) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or `'none'`, the score for the ignored class will be returned as `nan`.

- **top_k** *(Optional[int])* – Number of highest probability or logit score predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (`None`) will be interpreted as 1 for these inputs.

  Should be left at default (`None`) for all other types of inputs.

- **multiclass** *(Optional[bool])* – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.

- **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 `top_k` is not an `integer` larger than `0`.

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

- **ValueError** – If two different input modes are provided, eg. using `multi-label` with `multi-class`.

- **ValueError** – If `top_k` parameter is set for `multi-label` inputs.

Example

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

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

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

compute()

> Computes accuracy based on inputs passed in to `update` previously.
>
> > **Return type** `Tensor`

update(*preds*, *target*)

> Update state with predictions and targets. See *Input types* for more information on input types.
>
> > **Parameters**
> >
> > * **preds** ¶ (`Tensor`) – Predictions from model (logits, probabilities, or labels)
> >
> > * **target** ¶ (`Tensor`) – Ground truth labels
> >
> > **Return type** `None`

AveragePrecision

class torchmetrics.**AveragePrecision**(*num_classes=None*, *pos_label=None*, *average='macro'*,
                                 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]
>
> * **average** ¶ (`Optional[str]`) – defines the reduction that is applied in the case of multiclass and multilabel input. Should be one of the following:
>
>   – `'macro'` [default]: Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
>
>   – `'micro'`: Calculate the metric globally, across all samples and classes. Cannot be used with multiclass input.
>
>   – `'weighted'`: Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support.

- – `'none'` or `None`: Calculate the metric for each class separately, and return the metric for every class.

  - **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.

Example (binary case):

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

Example (multiclass case):

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

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

compute()

  Compute the average precision score.

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

  Returns tensor with average precision. If multiclass 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

  Return type `None`

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.
> >
> > - **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**(*preds*, *target*)
>
> > Update state with predictions and targets.
> >
> > > **Parameters**
> > >
> > > - **preds**⧚ ([Tensor](#)) – Predictions from model (probabilities, or labels)
> > >
> > > - **target**⧚ ([Tensor](#)) – Ground truth labels
> > >
> > > **Return type** [None](#)

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 interpreted as multilabel and if `preds` have one dimension more than the `target` tensor the input will be interpreted as multiclass.

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

Parameters

- **num_classes** ⫲ (Optional[int]) – integer with number of classes for multi-label and multiclass problems. Should be set to None for binary problems

- **pos_label** ⫲ (Optional[int]) – integer determining the positive class. Default is None which for binary problem is translate to 1. For multiclass problems this argument should not be set as we iteratively change it in the range [0,num_classes-1]

- **average** ⫲ (Optional[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.

- **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.

- **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
> >
> > **Return type** `None`

BinnedAveragePrecision

class torchmetrics.**BinnedAveragePrecision**(*num_classes*, *thresholds=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.

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

Forward accepts

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

- `target` (long tensor): `(N, ...)` with integer labels

> **Parameters**
>
> - **num_classes**⫮ (`int`) – integer with number of classes. Not nessesary to provide for binary problems.
>
> - **thresholds**⫮ (`Union`[`int`, `Tensor`, `List`[`float`], `None`]) – list or tensor with specific thresholds or a number of bins from linear sampling. It is used for computation will lead to more detailed curve and accurate estimates, but will be slower and consume more memory
>
> - **compute_on_step**⫮ (`bool`) – Forward only calls `update()` and return None if this is set to False.

- **process_group** (`Optional[Any]`) – Specify the process group on which synchronization is called.

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

Example (binary case):

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

Example (multiclass case):

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

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

compute()

Returns float tensor of size n_classes.

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

BinnedPrecisionRecallCurve

class torchmetrics.**BinnedPrecisionRecallCurve**(*num_classes*, *thresholds=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.

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

Forward accepts

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

- `target` (long tensor): `(N, ...)` or `(N, C, ...)` with integer labels

Parameters

- **num_classes** (`int`) – integer with number of classes. For binary, set to 1.

- **thresholds** (`Union[int, Tensor, List[float], None]`) – list or tensor with specific thresholds or a number of bins from linear sampling. It is used for computation will lead to more detailed curve and accurate estimates, but will be slower and consume more memory.

- **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.

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

Example (binary case):

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

Example (multiclass case):

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

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

compute()

    Returns float tensor of size n_classes.

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

update(*preds*, *target*)

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

        Return type None

BinnedRecallAtFixedPrecision

class torchmetrics.**BinnedRecallAtFixedPrecision**(*num_classes*, *min_precision*, *thresholds=None*, *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 `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.

- **thresholds** (Union[int, Tensor, List[float], None]) – list or tensor with specific thresholds or a number of bins from linear sampling. It is used for computation will lead to more detailed curve and accurate estimates, but will be slower and consume more memory

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

- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called.

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

Example (binary case):

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

Example (multiclass case):

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

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

compute()

 Returns float tensor of size n_classes.

  Return type Tuple[Tensor, Tensor]

CalibrationError

class torchmetrics.**CalibrationError**(*n_bins=15*, *norm='l1'*, *compute_on_step=False*, *dist_sync_on_step=False*, *process_group=None*)

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

L1 norm (Expected Calibration Error)

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

Infinity norm (Maximum Calibration Error)

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

L2 norm (Root Mean Square Calibration Error)

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

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

Note: L2-norm debiasing is not yet supported.

 Parameters

- **n_bins** (int) – Number of bins to use when computing probabilites and accuracies.
- **norm** (str) – Norm used to compare empirical and expected probability bins. Defaults to "l1", or Expected Calibration Error.
- **debias** – Applies debiasing term, only implemented for l2 norm. Defaults to True.

- **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.

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

compute()
    Computes calibration error across all confidences and accuracies.

        Returns Calibration error across previously collected examples.

        Return type Tensor

update(*preds*, *target*)
    Computes top-level confidences and accuracies for the input probabilites and appends them to internal state.

        Parameters

- **preds**∥ (`Tensor`) – Model output probabilities.

- **target**∥ (`Tensor`) – Ground-truth target class labels.

        Return type None

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 or logits.

If preds has an extra dimension as in the case of multi-class scores we perform an argmax on `dim=1`.

    Parameters

- **num_classes**∥ (`int`) – Number of classes in the dataset.

- **weights**∥ (`Optional[str]`) – Weighting type to calculate the score. Choose from - None or 'none': no weighting - 'linear': linear weighting - 'quadratic': quadratic weighting

- **threshold**∥ (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.

- **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.

Example

```
>>> from torchmetrics import CohenKappa
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0, 1, 0, 0])
>>> cohenkappa = CohenKappa(num_classes=2)
>>> cohenkappa(preds, target)
tensor(0.5000)
```

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

compute()
Computes cohen kappa score.

> **Return type** Tensor

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

> **Parameters**
>
> - **preds**¶ (Tensor) – Predictions from model
>
> - **target**¶ (Tensor) – Ground truth values
>
> **Return type** None

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 or logits from a model output or integer class values in prediction. Works with multi-dimensional preds and target, but it should be noted that additional dimensions will be flattened.

Forward accepts

- preds (float or long tensor): (N, ...) or (N, C, ...) where C is the number of classes

- target (long tensor): (N, ...)

If preds and target are the same shape and preds is a float tensor, we use the `self.threshold` argument to convert into integer labels. This is the case for binary and multi-label probabilities or logits.

If preds has an extra dimension as in the case of multi-class scores we perform an argmax on `dim=1`.

If working with multilabel data, setting the *is_multilabel* argument to *True* will make sure that a confusion matrix gets calculated per label.

> **Parameters**

- **num_classes** (int) – Number of classes in the dataset.
- **normalize** (Optional[str]) – Normalization mode for confusion matrix. Choose from
  - None or 'none': no normalization (default)
  - 'true': normalization over the targets (most commonly used)
  - 'pred': normalization over the predictions
  - 'all': normalization over the whole matrix
- **threshold** (float) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
- **multilabel** (bool) – determines if data is multilabel or not.
- **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.

Example (binary data):

```
>>> from torchmetrics import ConfusionMatrix
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0, 1, 0, 0])
>>> confmat = ConfusionMatrix(num_classes=2)
>>> confmat(preds, target)
tensor([[2, 0],
        [1, 1]])
```

Example (multiclass data):

```
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> confmat = ConfusionMatrix(num_classes=3)
>>> confmat(preds, target)
tensor([[1, 1, 0],
        [0, 1, 0],
        [0, 0, 1]])
```

Example (multilabel data):

```
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> confmat = ConfusionMatrix(num_classes=3, multilabel=True)
>>> confmat(preds, target)
tensor([[[1, 0], [0, 1]],
        [[1, 0], [1, 0]],
        [[0, 1], [0, 1]]])
```

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

compute()
    Computes confusion matrix.

        Return type `Tensor`

        Returns If *multilabel=False* this will be a *[n_classes, n_classes]* tensor and if *multilabel=True* this will be a *[n_classes, 2, 2]* tensor

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

        Parameters

            • **preds**⫽ (`Tensor`) – Predictions from model

            • **target**⫽ (`Tensor`) – Ground truth values

        Return type `None`

F1Score

class torchmetrics.**F1Score**(*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*)
Computes F1 metric. F1 metrics correspond to a harmonic mean of the precision and recall scores.

Works with binary, multiclass, and multilabel data. Accepts logits or probabilities from a model output or integer class values in prediction. Works with multi-dimensional preds and target.

Forward accepts

    • preds (float or long tensor): `(N, ...)` or `(N, C, ...)` where C is the number of classes

    • target (long tensor): `(N, ...)`

If preds and target are the same shape and preds is a float tensor, we use the `self.threshold` argument. This is the case for binary and multi-label logits.

If preds has an extra dimension as in the case of multi-class scores we perform an argmax on `dim=1`.

    Parameters

        • **num_classes**⫽ (`Optional`[`int`]) – Number of classes. Necessary for `'macro'`, `'weighted'` and `None` average methods.

        • **threshold**⫽ (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.

        • **average**⫽ (`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 or logit score predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (`None`) will be interpreted as 1 for these inputs.

  Should be left at default (`None`) for all other types of inputs.

- **multiclass** ‖ (`Optional[bool]`) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.

- **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 F1Score
>>> target = torch.tensor([0, 1, 2, 0, 1, 2])
>>> preds = torch.tensor([0, 2, 1, 0, 0, 1])
>>> f1 = F1Score(num_classes=3)
>>> f1(preds, target)
tensor(0.3333)
```

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

FBetaScore

class torchmetrics.**FBetaScore**(*num_classes=None, beta=1.0, threshold=0.5, average='micro',
mdmc_average=None, ignore_index=None, top_k=None, multiclass=None,
compute_on_step=True, dist_sync_on_step=False, process_group=None,
dist_sync_fn=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 logit scores or probabilities from a model output or integer class values in prediction. Works with multi-dimensional preds and target.

Forward accepts

- preds (float or long tensor): (N, ...) or (N, C, ...) where C is the number of classes

- target (long tensor): (N, ...)

If preds and target are the same shape and preds is a float tensor, we use the self.threshold argument to convert into integer labels. This is the case for binary and multi-label logits and probabilities.

If preds has an extra dimension as in the case of multi-class scores we perform an argmax on dim=1.

Parameters

- **num_classes** (Optional[int]) – Number of classes. Necessary for 'macro', 'weighted' and None average methods.

- **beta** (float) – Beta coefficient in the F measure.

- **threshold** (float) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.

- **average** (str) – Defines the reduction that is applied. Should be one of the following:

  - 'micro' [default]: Calculate the metric globally, across all samples and classes.

  - 'macro': Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).

  - 'weighted': Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (tp + fn).

  - 'none' or None: Calculate the metric for each class separately, and return the metric for every class.

  - 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

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

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

- **mdmc_average**⁄ (Optional[str]) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:

  - `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class.

  - `'samplewise'`: In this case, the statistics are computed separately for each sample on the `N` axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes `...` (see *Input types*) as the `N` dimension within the sample, and computing the metric for the sample based on that.

  - `'global'`: In this case the `N` and `...` dimensions of the inputs (see *Input types*) are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were (`N_X, C`). From here on the `average` parameter applies as usual.

- **ignore_index**⁄ (Optional[int]) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or `'none'`, the score for the ignored class will be returned as `nan`.

- **top_k**⁄ (Optional[int]) – Number of highest probability or logit score predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (`None`) will be interpreted as 1 for these inputs.

  Should be left at default (`None`) for all other types of inputs.

- **multiclass**⁄ (Optional[bool]) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.

- **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 FBetaScore
>>> target = torch.tensor([0, 1, 2, 0, 1, 2])
>>> preds = torch.tensor([0, 2, 1, 0, 0, 1])
>>> f_beta = FBetaScore(num_classes=3, beta=0.5)
>>> f_beta(preds, target)
tensor(0.3333)
```

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

compute()
    Computes 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 for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
>
> - **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, logits or labels)

- **target** (Tensor) – Ground truth labels

Return type None

HingeLoss

class torchmetrics.**HingeLoss**(*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 HingeLoss
>>> target = torch.tensor([0, 1, 1])
>>> preds = torch.tensor([-2.2, 2.4, 0.1])
>>> hinge = HingeLoss()
>>> hinge(preds, target)
tensor(0.3000)
```

Example (default / multiclass case):

```
>>> target = torch.tensor([0, 1, 2])
>>> preds = torch.tensor([[-1.0, 0.9, 0.2], [0.5, -1.1, 0.8], [2.2, -0.5, 0.3]])
>>> hinge = HingeLoss()
>>> hinge(preds, target)
tensor(2.9000)
```

Example (multiclass example, one vs all mode):

```
>>> target = torch.tensor([0, 1, 2])
>>> preds = torch.tensor([[-1.0, 0.9, 0.2], [0.5, -1.1, 0.8], [2.2, -0.5, 0.3]])
>>> hinge = HingeLoss(multiclass_mode="one-vs-all")
>>> hinge(preds, target)
tensor([2.2333, 1.5000, 1.2333])
```

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

compute()
> Override this method to compute the final metric value from state variables synchronized across the distributed backend.
>
> > **Return type** Tensor

update(*preds*, *target*)
> Override this method to update the state variables of your metric class.
>
> > **Return type** None

JaccardIndex

class torchmetrics.**JaccardIndex**(*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:

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

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.

Example

```
>>> from torchmetrics import JaccardIndex
>>> target = torch.randint(0, 2, (10, 25, 25))
>>> pred = torch.tensor(target)
>>> pred[2:5, 7:13, 9:15] = 1 - pred[2:5, 7:13, 9:15]
>>> jaccard = JaccardIndex(num_classes=2)
>>> jaccard(pred, target)
tensor(0.9660)
```

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

compute()
> Computes intersection over union (IoU)
>
> > **Return type** Tensor

KLDivergence

class torchmetrics.**KLDivergence**(*log_prob=False*, *reduction='mean'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Computes the KL divergence:

$$D_{KL}(P||Q) = \sum_{x \in \mathcal{X}} P(x) \log \frac{P(x)}{Qx}$$

Where P and Q are probability distributions where P usually represents a distribution over data and Q is often a prior or approximation of P. It should be noted that the KL divergence is a non-symetrical metric i.e. $D_{KL}(P||Q) \neq D_{KL}(Q||P)$.

> **Parameters**
>
> - **p** – data distribution with shape [N, d]
>
> - **q** – prior or approximate distribution with shape [N, d]
>
> - **log_prob** (bool) – bool indicating if input is log-probabilities or probabilities. If given as probabilities, will normalize to make sure the distributes sum to 1
>
> - **reduction** (Optional[str]) – Determines how to reduce over the N/batch dimension:
>
>   - 'mean' [default]: Averages score across samples
>
>   - 'sum': Sum score across samples
>
>   - 'none' or None: Returns score per sample
>
> **Raises**
>
> - **TypeError** – If log_prob is not an bool
>
> - **ValueError** – If reduction is not one of 'mean', 'sum', 'none' or None

---

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

Example

```
>>> import torch
>>> from torchmetrics.functional import kl_divergence
>>> p = torch.tensor([[0.36, 0.48, 0.16]])
>>> q = torch.tensor([[1/3, 1/3, 1/3]])
>>> kl_divergence(p, q)
tensor(0.0853)
```

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

compute()

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

> **Return type** Tensor

update(*p, q*)

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

> **Return type** None

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.
> - **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.
> - **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
>   >
>   > **Return type** `None`

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*)

Computes Precision:

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

Where TP and FP represent the number of true positives and false positives respectively. With the use of `top_k` parameter, this metric can generalize to Precision@K.

The reduction method (how the precision scores are aggregated) is controlled by the `average` parameter, and additionally by the `mdmc_average` parameter in the multi-dimensional multi-class case. Accepts all inputs listed in *Input types*.

> **Parameters**
>
> - **num_classes**⫽ (`Optional`[`int`]) – Number of classes. Necessary for `'macro'`, `'weighted'` and `None` average methods.
>
> - **threshold**⫽ (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
>
> - **average**⫽ (`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 or logit score predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (None) will be interpreted as 1 for these inputs.

  Should be left at default (None) for all other types of inputs.

- **multiclass**⫽ (Optional[bool]) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.

- **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 for multi-label and multiclass problems. Should be set to None for binary problems

- **pos_label** (Optional[int]) – integer determining the positive class. Default is None which for binary problem is translate to 1. For multiclass problems this argument should not be set as we iteratively change it in the range [0,num_classes-1]

- **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.

Example (binary case):

```
>>> from torchmetrics import PrecisionRecallCurve
>>> pred = torch.tensor([0, 0.1, 0.8, 0.4])
>>> target = torch.tensor([0, 1, 1, 0])
>>> pr_curve = PrecisionRecallCurve(pos_label=1)
>>> precision, recall, thresholds = pr_curve(pred, target)
>>> precision
tensor([0.6667, 0.5000, 1.0000, 1.0000])
>>> recall
tensor([1.0000, 0.5000, 0.5000, 0.0000])
>>> thresholds
tensor([0.1000, 0.4000, 0.8000])
```

Example (multiclass case):

```
>>> pred = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                      [0.05, 0.75, 0.05, 0.05, 0.05],
...                      [0.05, 0.05, 0.75, 0.05, 0.05],
...                      [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> pr_curve = PrecisionRecallCurve(num_classes=5)
>>> precision, recall, thresholds = pr_curve(pred, target)
>>> precision
[tensor([1., 1.]), tensor([1., 1.]), tensor([0.2500, 0.0000, 1.0000]),
```

(continues on next page)

```
 tensor([0.2500, 0.0000, 1.0000]), tensor([0., 1.])]
>>> recall
[tensor([1., 0.]), tensor([1., 0.]), tensor([1., 0., 0.]), tensor([1., 0., 0.]),
↪ tensor([nan, 0.])]
>>> thresholds
[tensor(0.7500), tensor(0.7500), tensor([0.0500, 0.7500]), tensor([0.0500, 0.
↪7500]), tensor(0.0500)]
```

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

compute()

Compute the precision-recall curve.

> **Return type** Union[Tuple[Tensor, Tensor, Tensor], Tuple[List[Tensor], List[Tensor], List[Tensor]]]
>
> **Returns**
>
> > 3-element tuple containing
> >
> > **precision:** tensor where element i is the precision of predictions with score >= thresholds[i] and the last element is 1. If multiclass, this is a list of such tensors, one for each class.
> >
> > **recall:** tensor where element i is the recall of predictions with score >= thresholds[i] and the last element is 0. If multiclass, this is a list of such tensors, one for each class.
> >
> > **thresholds:** Thresholds used for computing precision/recall scores

update(*preds*, *target*)

Update state with predictions and targets.

> **Parameters**
>
> - **preds** (Tensor) – Predictions from model
> - **target** (Tensor) – Ground truth values
>
> **Return type** None

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*)

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 for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.

- **average**⁄ (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 or logit score predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class. The default value (None) will be interpreted as 1 for these inputs.

  Should be left at default (None) for all other types of inputs.

- **multiclass**⁄ (Optional[bool]) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.

- **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

Note: If either the positive class or negative class is completly missing in the target tensor, the roc values are not well defined in this case and a tensor of zeros will be returned (either fpr or tpr depending on what class is missing) together with an warning.

Parameters

- **num_classes**¶ (Optional[int]) – integer with number of classes for multi-label and multiclass problems. Should be set to None for binary problems

- **pos_label**¶ (Optional[int]) – integer determining the positive class. Default is None which for binary problem is translate to 1. For multiclass problems this argument should not be set as we iteratively change it in the range [0,num_classes-1]

- **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.

- **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])]
>>> 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
> >
> > **Return type** None

Specificity

class torchmetrics.**Specificity**(*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)

Computes Specificity:

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Where TN and FP represent the number of true negatives and false positives respectively. With the use of `top_k` parameter, this metric can generalize to Specificity@K.

The reduction method (how the specificity scores are aggregated) is controlled by the `average` parameter, and additionally by the `mdmc_average` parameter in the multi-dimensional multi-class case. Accepts all inputs listed in *Input types*.

Parameters

- **num_classes** (Optional[int]) – Number of classes. Necessary for `'macro'`, `'weighted'` and None average methods.

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

- **average** (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 (`tn + fp`).

  - `'none'` or None: Calculate the metric for each class separately, and return the metric for every class.

  - `'samples'`: Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

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

- **mdmc_average** (Optional[str]) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:

  - None [default]: Should be left unchanged if your data is not multi-dimensional multi-class.

  - `'samplewise'`: In this case, the statistics are computed separately for each sample on the N axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes `...` (see *Input types*) as the N dimension within the sample, and computing the metric for the sample based on that.

  - `'global'`: In this case the N and `...` dimensions of the inputs (see *Input types*) are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were (`N_X, C`). From here on the `average` parameter applies as usual.

- **ignore_index** (Optional[int]) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and average=None or 'none', the score for the ignored class will be returned as nan.

- **top_k** (Optional[int]) – Number of 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 Specificity
>>> preds  = torch.tensor([2, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> specificity = Specificity(average='macro', num_classes=3)
>>> specificity(preds, target)
tensor(0.6111)
>>> specificity = Specificity(average='micro')
>>> specificity(preds, target)
tensor(0.6250)
```

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

compute()

Computes the specificity score based on inputs passed in to update previously.

> **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

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*)

Computes the number of true positives, false positives, true negatives, false negatives. Related to Type I and Type II errors and the confusion matrix.

The reduction method (how the statistics are aggregated) is controlled by the `reduce` parameter, and additionally by the `mdmc_reduce` parameter in the multi-dimensional multi-class case.

Accepts all inputs listed in *Input types*.

> **Parameters**
>
> - **threshold** (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
>
> - **top_k** (`Optional[int]`) – Number of highest probability or logit score predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (`None`) will be interpreted as 1 for these inputs.
>
>   Should be left at default (`None`) for all other types of inputs.
>
> - **reduce** (`str`) – Defines the reduction that is applied. Should be one of the following:
>
>   - `'micro'` [default]: Counts the statistics by summing over all [sample, class] combinations (globally). Each statistic is represented by a single integer.
>
>   - `'macro'`: Counts the statistics for each class separately (over all samples). Each statistic is represented by a `(C,)` tensor. Requires `num_classes` to be set.
>
>   - `'samples'`: Counts the statistics for each sample separately (over all classes). Each statistic is represented by a `(N, )` 1d tensor.
>
> ---
>
> **Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_reduce`.
>
> ---
>
> - **num_classes** (`Optional[int]`) – Number of classes. Necessary for (multi-dimensional) multi-class or multi-label data.
>
> - **ignore_index** (`Optional[int]`) – Specify a class (label) to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `reduce='macro'`, the class statistics for the ignored class will all be returned as `-1`.
>
> - **mdmc_reduce** (`Optional[str]`) – Defines how the multi-dimensional multi-class inputs are 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 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, logits or labels)

- **target**¶ (Tensor) – Ground truth values

        Return type None

2.5.5 Image

Image quality metrics can be used to access the quality of synthetic generated images from machine learning algorithms such as Generative Adverserial Networks (GANs).

FrechetInceptionDistance

class torchmetrics.image.fid.**FrechetInceptionDistance**(*feature=2048*, *compute_on_step=False*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

    Calculates Fréchet inception distance (FID) which is used to access the quality of generated images. Given by

$$FID = |\mu - \mu_w| + tr(\Sigma + \Sigma_w - 2(\Sigma\Sigma_w)^{\frac{1}{2}})$$

where $\mathcal{N}(\mu, \Sigma)$ is the multivariate normal distribution estimated from Inception v3 [1] features calculated on real life images and $\mathcal{N}(\mu_w, \Sigma_w)$ is the multivariate normal distribution estimated from Inception v3 features calculated on generated (fake) images. The metric was originally proposed in [1].

Using the default feature extraction (Inception v3 using the original weights from [2]), the input is expected to be mini-batches of 3-channel RGB images of shape (3 x H x W) with dtype uint8. All images will be resized to 299 x 299 which is the size of the original training data. The boolian flag real determines if the images should update the statistics of the real distribution or the fake distribution.

Note: using this metrics requires you to have scipy install. Either install as pip install torchmetrics[image] or pip install scipy

---

Note: using this metric with the default feature extractor requires that `torch-fidelity` is installed. Either install as `pip install torchmetrics[image]` or `pip install torch-fidelity`

Note: the `forward` method can be used but `compute_on_step` is disabled by default (oppesit of all other metrics) as this metric does not really make sense to calculate on a single batch. This means that by default `forward` will just call `update` underneat.

> **Parameters**
>
> - **feature** _//_ (`Union`[`int`, `Module`]) – Either an integer or `nn.Module`:
>
>   - an integer will indicate the inceptionv3 feature layer to choose. Can be one of the following: 64, 192, 768, 2048
>
>   - an `nn.Module` for using a custom feature extractor. Expects that its forward method returns an `[N,d]` matrix where `N` is the batch size and `d` is the feature size.
>
> - **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`[[`Tensor`], `List`[`Tensor`]]]) – Callback that performs the allgather operation on the metric state. When `None`, DDP will be used to perform the allgather

References

[1] Rethinking the Inception Architecture for Computer Vision Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, Zbigniew Wojna https://arxiv.org/abs/1512.00567

[2] GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium, Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, Sepp Hochreiter https://arxiv.org/abs/1706.08500

> **Raises**
>
> - `ValueError` – If `feature` is set to an `int` (default settings) and `torch-fidelity` is not installed
>
> - `ValueError` – If `feature` is set to an `int` not in [64, 192, 768, 2048]
>
> - `TypeError` – If `feature` is not an `str`, `int` or `torch.nn.Module`

Example

```
>>> import torch
>>> _ = torch.manual_seed(123)
>>> from torchmetrics.image.fid import FrechetInceptionDistance
>>> fid = FrechetInceptionDistance(feature=64)
>>> # generate two slightly overlapping image intensity distributions
>>> imgs_dist1 = torch.randint(0, 200, (100, 3, 299, 299), dtype=torch.uint8)
>>> imgs_dist2 = torch.randint(100, 255, (100, 3, 299, 299), dtype=torch.uint8)
>>> fid.update(imgs_dist1, real=True)
>>> fid.update(imgs_dist2, real=False)
>>> fid.compute()
tensor(12.7202)
```

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

compute()
>   Calculate FID score based on accumulated extracted features from the two distributions.

>   **Return type** Tensor

update(*imgs*, *real*)
>   Update the state with extracted features.

>   **Parameters**

>   - **imgs**¶ (Tensor) – tensor with images feed to the feature extractor

>   - **real**¶ (bool) – bool indicating if imgs belong to the real or the fake distribution

>   **Return type** None

InceptionScore

class torchmetrics.image.inception.**InceptionScore**(*feature='logits_unbiased'*, *splits=10*, *compute_on_step=False*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Calculates the Inception Score (IS) which is used to access how realistic generated images are. It is defined as

$$IS = exp(\mathbb{E}_x KL(p(y|x)||p(y)))$$

where $KL(p(y|x)||p(y))$ is the KL divergence between the conditional distribution $p(y|x)$ and the margianl distribution $p(y)$. Both the conditional and marginal distribution is calculated from features extracted from the images. The score is calculated on random splits of the images such that both a mean and standard deviation of the score are returned. The metric was originally proposed in [1].

Using the default feature extraction (Inception v3 using the original weights from [2]), the input is expected to be mini-batches of 3-channel RGB images of shape (3 x H x W) with dtype uint8. All images will be resized to 299 x 299 which is the size of the original training data.

Note: using this metric with the default feature extractor requires that `torch-fidelity` is installed. Either install as `pip install torchmetrics[image]` or `pip install torch-fidelity`

Note: the `forward` method can be used but `compute_on_step` is disabled by default (oppesit of all other metrics) as this metric does not really make sense to calculate on a single batch. This means that by default

---

`forward` will just call `update` underneat.

>   **Parameters**
>
>   - **feature**∬ (`Union`[`str`, `int`, `Module`]) – Either an str, integer or `nn.Module`:
>
>   - an str or integer will indicate the inceptionv3 feature layer to choose. Can be one of the following: 'logits_unbiased', 64, 192, 768, 2048
>
>   - an `nn.Module` for using a custom feature extractor. Expects that its forward method returns an `[N,d]` matrix where `N` is the batch size and `d` is the feature size.
>
>   - **splits**∬ (`int`) – integer determining how many splits the inception score calculation should be split among
>
>   - **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`[[`Tensor`], `List`[`Tensor`]]]) – Callback that performs the allgather operation on the metric state. When `None`, DDP will be used to perform the allgather

References

[1] Improved Techniques for Training GANs Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, Xi Chen https://arxiv.org/abs/1606.03498

[2] GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium, Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, Sepp Hochreiter https://arxiv.org/abs/1706.08500

>   **Raises**
>
>   - **ValueError** – If `feature` is set to an `str` or `int` and `torch-fidelity` is not installed
>
>   - **ValueError** – If `feature` is set to an `str` or `int` and not one of ['logits_unbiased', 64, 192, 768, 2048]
>
>   - **TypeError** – If `feature` is not an `str`, `int` or `torch.nn.Module`

Example

```
>>> import torch
>>> _ = torch.manual_seed(123)
>>> from torchmetrics.image.inception import InceptionScore
>>> inception = InceptionScore()
>>> # generate some images
>>> imgs = torch.randint(0, 255, (100, 3, 299, 299), dtype=torch.uint8)
>>> inception.update(imgs)
>>> inception.compute()
(tensor(1.0544), tensor(0.0117))
```

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

`compute()`

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

        Return type `Tuple[Tensor, Tensor]`

`update(`*imgs*`)`

    Update the state with extracted features.

        Parameters imgs¶ (`Tensor`) – tensor with images feed to the feature extractor

        Return type `None`

KernelInceptionDistance

`class torchmetrics.image.kid.`**`KernelInceptionDistance`**`(`*feature=2048, subsets=100, subset_size=1000, degree=3, gamma=None, coef=1.0, compute_on_step=False, dist_sync_on_step=False, process_group=None, dist_sync_fn=None*`)`

Calculates Kernel Inception Distance (KID) which is used to access the quality of generated images. Given by

$$KID = MMD(f_{real}, f_{fake})^2$$

where MMD is the maximum mean discrepancy and I_{real}, I_{fake} are extracted features from real and fake images, see [1] for more details. In particular, calculating the MMD requires the evaluation of a polynomial kernel function k

$$k(x, y) = (\gamma * x^T y + coef)^{degree}$$

which controls the distance between two features. In practise the MMD is calculated over a number of subsets to be able to both get the mean and standard deviation of KID.

Using the default feature extraction (Inception v3 using the original weights from [2]), the input is expected to be mini-batches of 3-channel RGB images of shape (3 x H x W) with dtype uint8. All images will be resized to 299 x 299 which is the size of the original training data.

Note: using this metric with the default feature extractor requires that `torch-fidelity` is installed. Either install as `pip install torchmetrics[image]` or `pip install torch-fidelity`

Note: the `forward` method can be used but `compute_on_step` is disabled by default (oppesit of all other metrics) as this metric does not really make sense to calculate on a single batch. This means that by default `forward` will just call `update` underneat.

    Parameters

- **feature**¶ (`Union[str, int, Module]`) – Either an str, integer or `nn.Module`:
    - an str or integer will indicate the inceptionv3 feature layer to choose. Can be one of the following: 'logits_unbiased', 64, 192, 768, 2048
    - an `nn.Module` for using a custom feature extractor. Expects that its forward method returns an `[N,d]` matrix where `N` is the batch size and `d` is the feature size.
- **subsets**¶ (`int`) – Number of subsets to calculate the mean and standard deviation scores over

- **subset_size**⫲ (`int`) – Number of randomly picked samples in each subset

- **degree**⫲ (`int`) – Degree of the polynomial kernel function

- **gamma**⫲ (`Optional`[`float`]) – Scale-length of polynomial kernel. If set to `None` will be automatically set to the feature size

- **coef**⫲ (`float`) – Bias term in the polynomial kernel.

- **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

References

[1] Demystifying MMD GANs Mikołaj Bińkowski, Danica J. Sutherland, Michael Arbel, Arthur Gretton https://arxiv.org/abs/1801.01401

[2] GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium, Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, Sepp Hochreiter https://arxiv.org/abs/1706.08500

Raises

- **ValueError** – If `feature` is set to an `int` (default settings) and `torch-fidelity` is not installed

- **ValueError** – If `feature` is set to an `int` not in [64, 192, 768, 2048]

- **ValueError** – If `subsets` is not an integer larger than 0

- **ValueError** – If `subset_size` is not an integer larger than 0

- **ValueError** – If `degree` is not an integer larger than 0

- **ValueError** – If `gamma` is niether `None` or a float larger than 0

- **ValueError** – If `coef` is not an float larger than 0

Example

```
>>> import torch
>>> _ = torch.manual_seed(123)
>>> from torchmetrics.image.kid import KernelInceptionDistance
>>> kid = KernelInceptionDistance(subset_size=50)
>>> # generate two slightly overlapping image intensity distributions
>>> imgs_dist1 = torch.randint(0, 200, (100, 3, 299, 299), dtype=torch.uint8)
>>> imgs_dist2 = torch.randint(100, 255, (100, 3, 299, 299), dtype=torch.uint8)
>>> kid.update(imgs_dist1, real=True)
>>> kid.update(imgs_dist2, real=False)
>>> kid_mean, kid_std = kid.compute()
>>> print((kid_mean, kid_std))
(tensor(0.0337), tensor(0.0023))
```

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

compute()

> Calculate KID score based on accumulated extracted features from the two distributions. Returns a tuple of mean and standard deviation of KID scores calculated on subsets of extracted features.
>
> Implementation inspired by Fid Score
>
> > **Return type** Tuple[Tensor, Tensor]

update(*imgs*, *real*)

> Update the state with extracted features.
>
> > **Parameters**
> >
> > - **imgs** (Tensor) – tensor with images feed to the feature extractor
> >
> > - **real** (bool) – bool indicating if imgs belong to the real or the fake distribution
> >
> > **Return type** None

LearnedPerceptualImagePatchSimilarity

class torchmetrics.image.lpip.**LearnedPerceptualImagePatchSimilarity**(*net_type='alex'*, *reduction='mean'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

The Learned Perceptual Image Patch Similarity (*LPIPS_*) is used to judge the perceptual similarity between two images. LPIPS essentially computes the similarity between the activations of two image patches for some pre-defined network. This measure have been shown to match human perseption well. A low LPIPS score means that image patches are perceptual similar.

Both input image patches are expected to have shape *[N, 3, H, W]* and be normalized to the [-1,1] range. The minimum size of *H, W* depends on the chosen backbone (see *net_type* arg).

Note: using this metrics requires you to have `lpips` package installed. Either install as `pip install torchmetrics[image]` or `pip install lpips`

Note: this metric is not scriptable when using `torch<1.8`. Please update your pytorch installation if this is a issue.

> **Parameters**
>
> - **net_type** (str) – str indicating backbone network type to use. Choose between *'alex'*, *'vgg'* or *'squeeze'*
>
> - **reduction** (str) – str indicating how to reduce over the batch dimension. Choose between *'sum'* or *'mean'*.
>
> - **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[[Tensor], List[Tensor]]]) – Callback that performs the allgather operation on the metric state. When None, DDP will be used to perform the allgather

Raises

- **ModuleNotFoundError** – If lpips package is not installed

- **ValueError** – If net_type is not one of "vgg", "alex" or "squeeze"

- **ValueError** – If reduction is not one of "mean" or "sum"

Example

```
>>> import torch
>>> _ = torch.manual_seed(123)
>>> from torchmetrics.image.lpip import LearnedPerceptualImagePatchSimilarity
>>> lpips = LearnedPerceptualImagePatchSimilarity(net_type='vgg')
>>> img1 = torch.rand(10, 3, 100, 100)
>>> img2 = torch.rand(10, 3, 100, 100)
>>> lpips(img1, img2)
tensor(0.3566, grad_fn=<SqueezeBackward0>)
```

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

compute()
> Compute final perceptual similarity metric.
>
> > **Return type** Tensor

update(*img1*, *img2*)
> Update internal states with lpips score.
>
> > **Parameters**
> >
> > - **img1**¶ (Tensor) – tensor with images of shape [N, 3, H, W]
> >
> > - **img2**¶ (Tensor) – tensor with images of shape [N, 3, H, W]
> >
> > **Return type** None

MultiScaleStructuralSimilarityIndexMeasure

class torchmetrics.**MultiScaleStructuralSimilarityIndexMeasure**(*kernel_size=(11, 11)*, *sigma=(1.5, 1.5)*, *reduction='elementwise_mean'*, *data_range=None*, *k1=0.01*, *k2=0.03*, *betas=(0.0448, 0.2856, 0.3001, 0.2363, 0.1333)*, *normalize=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)

Computes MultiScaleSSIM, Multi-scale Structural Similarity Index Measure, which is a generalization of Structural Similarity Index Measure by incorporating image details at different resolution scores.

Parameters

- **kernel_size**¶ (Sequence[int]) – size of the gaussian kernel

- **sigma**¶ (Sequence[float]) – Standard deviation of the gaussian kernel

- **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 structural similarity index measure.

- **k2**¶ (float) – Parameter of structural similarity index measure.

- **betas**¶ (Tuple[float, ...]) – Exponent parameters for individual similarities and contrastive sensitivies returned by different image resolutions.

- **normalize**¶ (Optional[Literal['relu', 'simple']]) – When MultiScaleStructuralSimilarityIndexMeasure loss is used for training, it is desirable to use normalizes to improve the training stability. This *normalize* argument is out of scope of the original implementation [1], and it is adapted from https://github.com/jorge-pessoa/pytorch-msssim instead.

Returns Tensor with Multi-Scale SSIM score

Example

```
>>> from torchmetrics import MultiScaleStructuralSimilarityIndexMeasure
>>> preds = torch.rand([1, 1, 256, 256], generator=torch.manual_seed(42))
>>> target = preds * 0.75
>>> ms_ssim = MultiScaleStructuralSimilarityIndexMeasure()
>>> ms_ssim(preds, target)
tensor(0.9558)
```

References: [1] Multi-Scale Structural Similarity For Image Quality Assessment by Zhou Wang, Eero P. Simoncelli and Alan C. Bovik MultiScaleSSIM

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

compute()
> Computes explained variance over state.

> **Return type** Tensor

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

> **Parameters**

> - **preds**¶ (Tensor) – Predictions from model of shape *[N, C, H, W]*

> - **target**¶ (Tensor) – Ground truth values of shape *[N, C, H, W]*

> **Return type** None

PeakSignalNoiseRatio

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

Computes Computes Peak Signal-to-Noise Ratio (PSNR):

$$\text{PSNR}(I, J) = 10 * \log_{10}\left(\frac{\max(I)^2}{\text{MSE}(I, J)}\right)$$

Where MSE denotes the mean-squared-error function.

> **Parameters**
>
> - **data_range**⫽ (Optional[float]) – the range of the data. If None, it is determined from the data (max - min). The data_range must be given when dim is not None.
>
> - **base**⫽ (float) – a base of a logarithm to use.
>
> - **reduction**⫽ (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.
>
> - **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.
>
> **Raises** ValueError – If dim is not None and data_range is not given.

Example

```
>>> from torchmetrics import PeakSignalNoiseRatio
>>> psnr = PeakSignalNoiseRatio()
>>> preds = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
>>> target = torch.tensor([[3.0, 2.0], [1.0, 0.0]])
>>> psnr(preds, target)
tensor(2.5527)
```

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

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

compute()

> Compute peak signal-to-noise ratio over state.
>
> > **Return type** Tensor

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

> Update state with predictions and targets.

> > **Parameters**

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

> > **Return type** None

StructuralSimilarityIndexMeasure

`class` torchmetrics.`StructuralSimilarityIndexMeasure`(*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
> > >
> > > - **sigma**⫻ (Sequence[float]) – Standard deviation of the gaussian kernel
> > >
> > > - **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.
> > >
> > > - **k2**⫻ (float) – Parameter of SSIM.

> > **Returns** Tensor with SSIM score

Example

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

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

`compute`()

> Computes explained variance over state.

> > **Return type** Tensor

update(*preds*, *target*)

Update state with predictions and targets.

> **Parameters**
>
> - **preds**⫿ (Tensor) – Predictions from model
> - **target**⫿ (Tensor) – Ground truth values
>
> **Return type** None

2.5.6 Detection

Object detection metrics can be used to evaluate the predicted detections with given groundtruth detections on images.

MeanAveragePrecision

class torchmetrics.detection.map.**MeanAveragePrecision**(*box_format='xyxy'*, *iou_thresholds=None*, *rec_thresholds=None*, *max_detection_thresholds=None*, *class_metrics=False*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Computes the Mean-Average-Precision (mAP) and Mean-Average-Recall (mAR) for object detection predictions. Optionally, the mAP and mAR values can be calculated per class.

Predicted boxes and targets have to be in Pascal VOC format (xmin-top left, ymin-top left, xmax-bottom right, ymax-bottom right). See the update() method for more information about the input format to this metric.

For an example on how to use this metric check the torchmetrics examples

---

Note: This metric is following the mAP implementation of pycocotools, , a standard implementation for the mAP metric for object detection.

Note: This metric requires you to have *torchvision* version 0.8.0 or newer installed (with corresponding version 1.7.0 of torch or newer). Please install with pip install torchvision or pip install torchmetrics[detection].

> **Parameters**
>
> - **box_format**⫿ (str) – Input format of given boxes. Supported formats are [*xyxy*, *xywh*, *cxcywh*].
> - **iou_thresholds**⫿ (Optional[List[float]]) – IoU thresholds for evaluation. If set to *None* it corresponds to the stepped range *[0.5,...,0.95]* with step *0.05*. Else provide a list of floats.
> - **rec_thresholds**⫿ (Optional[List[float]]) – Recall thresholds for evaluation. If set to *None* it corresponds to the stepped range *[0,...,1]* with step *0.01*. Else provide a list of floats.
> - **max_detection_thresholds**⫿ (Optional[List[int]]) – Thresholds on max detections per image. If set to *None* will use thresholds *[1, 10, 100]*. Else please provide a list of ints.
> - **class_metrics**⫿ (bool) – Option to enable per-class metrics for mAP and mAR_100. Has a performance impact.

- **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

```python
>>> import torch
>>> from torchmetrics.detection.map import MeanAveragePrecision
>>> preds = [
...    dict(
...       boxes=torch.Tensor([[258.0, 41.0, 606.0, 285.0]]),
...       scores=torch.Tensor([0.536]),
...       labels=torch.IntTensor([0]),
...    )
... ]
>>> target = [
...    dict(
...       boxes=torch.Tensor([[214.0, 41.0, 562.0, 285.0]]),
...       labels=torch.IntTensor([0]),
...    )
... ]
>>> metric = MeanAveragePrecision()
>>> metric.update(preds, target)
>>> from pprint import pprint
>>> pprint(metric.compute())
{'map': tensor(0.6000),
 'map_50': tensor(1.),
 'map_75': tensor(1.),
 'map_large': tensor(0.6000),
 'map_medium': tensor(-1.),
 'map_per_class': tensor(-1.),
 'map_small': tensor(-1.),
 'mar_1': tensor(0.6000),
 'mar_10': tensor(0.6000),
 'mar_100': tensor(0.6000),
 'mar_100_per_class': tensor(-1.),
 'mar_large': tensor(0.6000),
 'mar_medium': tensor(-1.),
 'mar_small': tensor(-1.)}
```

Raises

- **ModuleNotFoundError** – If torchvision is not installed or version installed is lower than 0.8.0

- **ValueError** – If class_metrics is not a boolean

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

compute()

Compute the *Mean-Average-Precision (mAP) and Mean-Average-Recall (mAR)* scores.

> **Note:** *map* score is calculated with @[IoU=self.iou_thresholds | area=all | max_dets=max_detection_thresholds]
>
> Caution: If the initialization parameters are changed, dictionary keys for mAR can change as well. The default properties are also accessible via fields and will raise an `AttributeError` if not available.

> **Return type** `dict`
>
> **Returns**
>
> > dict containing
> >
> > - map: `torch.Tensor`
> > - map_50: `torch.Tensor`
> > - map_75: `torch.Tensor`
> > - map_small: `torch.Tensor`
> > - map_medium: `torch.Tensor`
> > - map_large: `torch.Tensor`
> > - mar_1: `torch.Tensor`
> > - mar_10: `torch.Tensor`
> > - mar_100: `torch.Tensor`
> > - mar_small: `torch.Tensor`
> > - mar_medium: `torch.Tensor`
> > - mar_large: `torch.Tensor`
> > - map_per_class: `torch.Tensor` (-1 if class metrics are disabled)
> > - mar_100_per_class: `torch.Tensor` (-1 if class metrics are disabled)

update(*preds*, *target*)

Add detections and ground truth to the metric.

> **Parameters**
>
> - **preds**¶ (`List[Dict[str, Tensor]]`) – A list consisting of dictionaries each containing the key-values
> - **image)** ((each *_sphinx_paramlinks_torchmetrics.detection.map.MeanAveragePrecision.update.dictionary corresponds to a single*) –
> - **``boxes``**¶ (–) – `torch.FloatTensor` of shape [num_boxes, 4] containing *num_boxes* detection boxes of the format specified in the contructor. By default, this method expects [xmin, ymin, xmax, ymax] in absolute image coordinates.
> - **``scores``**¶ (–) – `torch.FloatTensor` of shape [num_boxes] containing detection scores for the boxes.

- ``labels``⫘ (-) – torch.IntTensor of shape [num_boxes] containing 0-indexed detection classes for the boxes.

- **target**⫘ (List[Dict[str, Tensor]]) – A list consisting of dictionaries each containing the key-values

- **image)** –

- ``boxes``⫘ – torch.FloatTensor of shape [num_boxes, 4] containing *num_boxes* ground truth boxes of the format specified in the contructor. By default, this method expects [xmin, ymin, xmax, ymax] in absolute image coordinates.

- ``labels``⫘ – torch.IntTensor of shape [num_boxes] containing 1-indexed ground truth classes for the boxes.

Raises

- **ValueError** – If preds is not of type List[Dict[str, Tensor]]

- **ValueError** – If target is not of type List[Dict[str, Tensor]]

- **ValueError** – If preds and target are not of the same length

- **ValueError** – If any of preds.boxes, preds.scores and preds.labels are not of the same length

- **ValueError** – If any of target.boxes and target.labels are not of the same length

- **ValueError** – If any box is not type float and of length 4

- **ValueError** – If any class is not type int and of length 1

- **ValueError** – If any score is not type float and of length 1

Return type None

2.5.7 Regression

CosineSimilarity

class torchmetrics.**CosineSimilarity**(*reduction='sum'*, *compute_on_step=True*, *dist_sync_on_step=False*,
process_group=None, *dist_sync_fn=None*)

Computes the Cosine Similarity between targets and predictions:

$$cos_{sim}(x, y) = \frac{x \cdot y}{||x|| \cdot ||y||} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2} \sqrt{\sum_{i=1}^{n} y_i^2}}$$

where y is a tensor of target values, and x is a tensor of predictions.

Forward accepts

- preds (float tensor): (N,d)

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

Parameters

- **reduction**⫘ (str) – how to reduce over the batch dimension using 'sum', 'mean' or 'none' (taking the individual scores)

- **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.

Example

```
>>> from torchmetrics import CosineSimilarity
>>> target = torch.tensor([[0, 1], [1, 1]])
>>> preds = torch.tensor([[0, 1], [0, 1]])
>>> cosine_similarity = CosineSimilarity(reduction = 'mean')
>>> cosine_similarity(preds, target)
tensor(0.8536)
```

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

compute()
> Override this method to compute the final metric value from state variables synchronized across the distributed backend.
>
> > **Return type** `Tensor`

update(*preds*, *target*)
> Update metric states with predictions and targets.
>
> > **Parameters**
> >
> > - **preds** (`Tensor`) – Predicted tensor with shape (`N,d`)
> > - **target** (`Tensor`) – Ground truth tensor with shape (`N,d`)
> >
> > **Return type** `None`

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.

- **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.

  Raises `ValueError` – If `multioutput` is not one of `"raw_values"`, `"uniform_average"` or `"variance_weighted"`.

Example

```
>>> from torchmetrics import ExplainedVariance
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> explained_variance = ExplainedVariance()
>>> explained_variance(preds, target)
tensor(0.9572)
```

```
>>> target = torch.tensor([[0.5, 1], [-1, 1], [7, -6]])
>>> preds = torch.tensor([[0, 2], [-1, 2], [8, -5]])
>>> explained_variance = ExplainedVariance(multioutput='raw_values')
>>> explained_variance(preds, target)
tensor([0.9677, 1.0000])
```

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

compute()

Computes explained variance over state.

> **Return type** `Union[Tensor, Sequence[Tensor]]`

update(*preds*, *target*)

Update state with predictions and targets.

> **Parameters**
>
> - **preds** (`Tensor`) – Predictions from model
>
> - **target** (`Tensor`) – Ground truth values
>
> **Return type** `None`

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.
>
> - **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.

Example

```
>>> from torchmetrics import MeanAbsoluteError
>>> target = torch.tensor([3.0, -0.5, 2.0, 7.0])
>>> preds = torch.tensor([2.5, 0.0, 2.0, 8.0])
>>> mean_absolute_error = MeanAbsoluteError()
>>> mean_absolute_error(preds, target)
tensor(0.5000)
```

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

compute()
> Computes mean absolute error over state.
>
> > **Return type** `Tensor`

update(*preds*, *target*)
> Update state with predictions and targets.
>
> > **Parameters**
> >
> > - **preds** (`Tensor`) – Predictions from model
> >
> > - **target** (`Tensor`) – Ground truth values
> >
> > **Return type** `None`

MeanAbsolutePercentageError

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

Computes Mean Absolute Percentage Error (MAPE):

$$\text{MAPE} = \frac{1}{n} \sum_1^n \frac{|y_i - \hat{y}_i|}{\max(\epsilon, y_i)}$$

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

> **Parameters**
>
> - **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.

Note: The epsilon value is taken from scikit-learn's implementation of MAPE.

Note: MAPE output is a non-negative floating point. Best result is 0.0 . But it is important to note that, bad predictions, can lead to arbitarily large values. Especially when some target values are close to 0. This MAPE implementation returns a very large number instead of inf.

Example

```
>>> from torchmetrics import MeanAbsolutePercentageError
>>> target = torch.tensor([1, 10, 1e6])
>>> preds = torch.tensor([0.9, 15, 1.2e6])
>>> mean_abs_percentage_error = MeanAbsolutePercentageError()
>>> mean_abs_percentage_error(preds, target)
tensor(0.2667)
```

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

compute()
> Computes mean absolute percentage error over state.
>
> > **Return type** Tensor

update(*preds*, *target*)
> Update state with predictions and targets.
>
> > **Parameters**
> >
> > - **preds**⫲ (Tensor) – Predictions from model
> > - **target**⫲ (Tensor) – Ground truth values
> >
> > **Return type** None

MeanSquaredError

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

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.
>
> - **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.
>
> - **squared** (bool) – If True returns MSE value, if False returns RMSE value.

Example

```
>>> from torchmetrics import MeanSquaredError
>>> target = torch.tensor([2.5, 5.0, 4.0, 8.0])
>>> preds = torch.tensor([3.0, 5.0, 2.5, 7.0])
>>> mean_squared_error = MeanSquaredError()
>>> mean_squared_error(preds, target)
tensor(0.8750)
```

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

compute()

> Computes mean squared error over state.
>
> > **Return type** Tensor

update(*preds*, *target*)

> Update state with predictions and targets.
>
> > **Parameters**
> >
> > - **preds** (Tensor) – Predictions from model
> >
> > - **target** (Tensor) – Ground truth values
> >
> > **Return type** None

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.

   - **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.

   Example

   ```
   >>> from torchmetrics import MeanSquaredLogError
   >>> target = torch.tensor([2.5, 5, 4, 8])
   >>> preds = torch.tensor([3, 5, 2.5, 7])
   >>> mean_squared_log_error = MeanSquaredLogError()
   >>> mean_squared_log_error(preds, target)
   tensor(0.0397)
   ```

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

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

   compute()
      Compute mean squared logarithmic error over state.

      Return type Tensor

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

      Parameters

      - **preds** (Tensor) – Predictions from model

      - **target** (Tensor) – Ground truth values

      Return type None

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.
>
> - **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.

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.
>
> > **Return type** Tensor

update(*preds*, *target*)

> Update state with predictions and targets.
>
> > **Parameters**
> >
> > - **preds** ∥ (Tensor) – Predictions from model
> >
> > - **target** ∥ (Tensor) – Ground truth values
> >
> > **Return type** None

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 R2 Score_Coefficient Determination:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

where $SS_{res} = \sum_i (y_i - f(x_i))^2$ is the sum of residual squares, and $SS_{tot} = \sum_i (y_i - \bar{y})^2$ is total sum of squares. Can also calculate adjusted r2 score given by

$$R^2_{adj} = 1 - \frac{(1 - R^2)(n - 1)}{n - k - 1}$$

where the parameter k (the number of independent regressors) should be provided as the *adjusted* argument.

Forward accepts

- preds (float tensor): (N,) or (N, M) (multioutput)

- target (float tensor): (N,) or (N, M) (multioutput)

In the case of multioutput, as default the variances will be uniformly averaged over the additional dimensions. Please see argument *multioutput* for changing this behavior.

> **Parameters**
>
> - **num_outputs** (int) – Number of outputs in multioutput setting (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.
>
> - **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.
>
> **Raises**
>
> - **ValueError** – If adjusted parameter is not an integer larger or equal to 0.
>
> - **ValueError** – If multioutput is not one of "raw_values", "uniform_average" or "variance_weighted".

Example

```
>>> from torchmetrics import R2Score
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> r2score = R2Score()
>>> r2score(preds, target)
tensor(0.9486)
```

```
>>> target = torch.tensor([[0.5, 1], [-1, 1], [7, -6]])
>>> preds = torch.tensor([[0, 2], [-1, 2], [8, -5]])
>>> r2score = R2Score(num_outputs=2, multioutput='raw_values')
>>> r2score(preds, target)
tensor([0.9654, 0.9082])
```

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

compute()
> Computes r2 score over the metric states.
>
> > **Return type** Tensor

update(*preds*, *target*)
> Update state with predictions and targets.
>
> > **Parameters**
> >
> > * **preds** (Tensor) – Predictions from model
> >
> > * **target** (Tensor) – Ground truth values
> >
> > **Return type** None

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.
> >
> > * **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.
> >
> > * **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.

> **Return type** `Tensor`

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

> **Parameters**
>
> - **preds**⫽ (`Tensor`) – Predictions from model
>
> - **target**⫽ (`Tensor`) – Ground truth values
>
> **Return type** `None`

SymmetricMeanAbsolutePercentageError

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

Computes symmetric mean absolute percentage error (SMAPE).

$$\text{SMAPE} = \frac{2}{n} \sum_{1}^{n} max(\frac{|y_i - \hat{y_i}|}{|y_i| + |\hat{y_i}|, \epsilon})$$

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.
>
> - **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.

Note: The epsilon value is taken from scikit-learn's implementation of SMAPE.

Note: SMAPE output is a non-negative floating point between 0 and 1. Best result is 0.0 .

Example

```
>>> from torchmetrics import SymmetricMeanAbsolutePercentageError
>>> target = torch.tensor([1, 10, 1e6])
>>> preds = torch.tensor([0.9, 15, 1.2e6])
>>> smape = SymmetricMeanAbsolutePercentageError()
>>> smape(preds, target)
tensor(0.2290)
```

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

compute()
> Computes mean absolute percentage error over state.

> > **Return type** `Tensor`

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

> > **Parameters**

> > > • **preds**⁑ (`Tensor`) – Predictions from model

> > > • **target**⁑ (`Tensor`) – Ground truth values

> > **Return type** `None`

TweedieDevianceScore

class torchmetrics.**TweedieDevianceScore**(*power=0.0*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)
> Computes the Tweedie Deviance Score between targets and predictions:

$$deviance\_score(\hat{y}, y) = \begin{cases} (\hat{y} - y)^2, & \text{for } power = 0 \\ 2 * (y * log(\frac{y}{\hat{y}}) + \hat{y} - y), & \text{for } power = 1 \\ 2 * (log(\frac{\hat{y}}{y}) + \frac{y}{\hat{y}} - 1), & \text{for } power = 2 \\ 2 * (\frac{(max(y,0))^2}{(1-power)(2-power)} - \frac{y(\hat{y})^{1-power}}{1-power} + \frac{(\hat{y})^{2-power}}{2-power}), & \text{otherwise} \end{cases}$$

> where y is a tensor of targets values, and $\hat{y}$ is a tensor of predictions.

> Forward accepts

> > • preds (float tensor): (N,...)

> > • targets (float tensor): (N,...)

> > **Parameters**

> > > • **power**⁑ (`float`) –

> > > > – power < 0 : Extreme stable distribution. (Requires: preds > 0.)

> > > > – power = 0 : Normal distribution. (Requires: targets and preds can be any real numbers.)

> > > > – power = 1 : Poisson distribution. (Requires: targets >= 0 and y_pred > 0.)

> > > > – 1 < p < 2 : Compound Poisson distribution. (Requires: targets >= 0 and preds > 0.)

> > > > – power = 2 : Gamma distribution. (Requires: targets > 0 and preds > 0.)

> > > > – power = 3 : Inverse Gaussian distribution. (Requires: targets > 0 and preds > 0.)

- – otherwise : Positive stable distribution. (Requires: targets > 0 and preds > 0.)
  - **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.

Example

```
>>> from torchmetrics import TweedieDevianceScore
>>> targets = torch.tensor([1.0, 2.0, 3.0, 4.0])
>>> preds = torch.tensor([4.0, 3.0, 2.0, 1.0])
>>> deviance_score = TweedieDevianceScore(power=2)
>>> deviance_score(preds, targets)
tensor(1.2083)
```

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

compute()

> Override this method to compute the final metric value from state variables synchronized across the distributed backend.
>
> > **Return type** `Tensor`

update(*preds*, *targets*)

> Update metric states with predictions and targets.
>
> > **Parameters**
> >
> > - **preds**⫽ (`Tensor`) – Predicted tensor with shape `(N,d)`
> > - **targets**⫽ (`Tensor`) – Ground truth tensor with shape `(N,d)`
> >
> > **Return type** `None`

2.5.8 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
(N,...)	`long`	(N,...)	`float`	(N,...)	`long` or `bool`

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

In Information Retrieval you have a query that is compared with a variable number of documents. For each pair (Q_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'*, *ignore_index=None*, *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`

- **ignore_index**⫻ (Optional[int]) – Ignore predictions where the target is equal to this number.

- **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.

- **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 `empty_target_action` is not one of `error`, `skip`, `neg` or `pos`.

- **ValueError** – If `ignore_index` is not *None* or an integer.

Example

```
>>> from torchmetrics import RetrievalMAP
>>> indexes = tensor([0, 0, 0, 1, 1, 1, 1])
>>> preds = tensor([0.2, 0.3, 0.5, 0.1, 0.3, 0.5, 0.2])
>>> target = tensor([False, False, True, False, True, False, True])
>>> rmap = RetrievalMAP()
>>> rmap(preds, target, indexes=indexes)
tensor(0.7917)
```

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

RetrievalMRR

class torchmetrics.**RetrievalMRR**(*empty_target_action='neg'*, *ignore_index=None*, *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

- **ignore_index**⚏ (Optional[int]) – Ignore predictions where the target is equal to this number.

- **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.

- **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 empty_target_action is not one of error, skip, neg or pos.

- **ValueError** – If ignore_index is not *None* or an integer.

Example

```
>>> from torchmetrics import RetrievalMRR
>>> indexes = tensor([0, 0, 0, 1, 1, 1, 1])
>>> preds = tensor([0.2, 0.3, 0.5, 0.1, 0.3, 0.5, 0.2])
>>> target = tensor([False, False, True, False, True, False, True])
>>> mrr = RetrievalMRR()
>>> mrr(preds, target, indexes=indexes)
tensor(0.7500)
```

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

RetrievalPrecision

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

Computes IR Precision.

Works with binary target data. Accepts float predictions from a model output.

Forward accepts:

- preds (float tensor): (N, ...)
- target (long or bool tensor): (N, ...)
- indexes (long tensor): (N, ...)

indexes, preds and target must have the same dimension. indexes indicate to which query a prediction belongs. Predictions will be first grouped by indexes and then *Precision* will be computed as the mean of the *Precision* over each query.

Parameters

- **empty_target_action** (str) – Specify what to do with queries that do not have at least a positive target. Choose from:

  - 'neg': those queries count as 0.0 (default)

  - 'pos': those queries count as 1.0

  - 'skip': skip those queries; if all queries are skipped, 0.0 is returned

  - 'error': raise a ValueError

- **ignore_index** (Optional[int]) – Ignore predictions where the target is equal to this number.

- **k** (Optional[int]) – consider only the top k elements for each query (default: *None*, which considers them all)

- **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.

- **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 empty_target_action is not one of error, skip, neg or pos.

- **ValueError** – If ignore_index is not *None* or an integer.

- **ValueError** – If k parameter is not *None* or an integer larger than 0.

Example

```
>>> from torchmetrics import 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.

RetrievalRPrecision

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

Computes IR R-Precision.

Works with binary target data. Accepts float predictions from a model output.

Forward accepts:

- `preds` (float tensor): `(N, ...)`
- `target` (long or bool tensor): `(N, ...)`
- `indexes` (long tensor): `(N, ...)`

`indexes`, `preds` and `target` must have the same dimension. `indexes` indicate to which query a prediction belongs. Predictions will be first grouped by `indexes` and then *R-Precision* will be computed as the mean of the *R-Precision* over each query.

> **Parameters**
>
> - **empty_target_action** (str) – Specify what to do with queries that do not have at least a positive `target`. Choose from:
>
>   - `'neg'`: those queries count as `0.0` (default)
>
>   - `'pos'`: those queries count as `1.0`
>
>   - `'skip'`: skip those queries; if all queries are skipped, `0.0` is returned
>
>   - `'error'`: raise a `ValueError`
>
> - **ignore_index** (Optional[int]) – Ignore predictions where the target is equal to this number.
>
> - **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.
>
> - **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 empty_target_action is not one of error, skip, neg or pos.

- **ValueError** – If ignore_index is not *None* or an integer.

Example

```
>>> from torchmetrics import RetrievalRPrecision
>>> indexes = tensor([0, 0, 0, 1, 1, 1, 1])
>>> preds = tensor([0.2, 0.3, 0.5, 0.1, 0.3, 0.5, 0.2])
>>> target = tensor([False, False, True, False, True, False, True])
>>> p2 = RetrievalRPrecision()
>>> p2(preds, target, indexes=indexes)
tensor(0.7500)
```

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

RetrievalRecall

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

Computes IR Recall.

Works with binary target data. Accepts float predictions from a model output.

Forward accepts:

- preds (float tensor): (N, ...)

- target (long or bool tensor): (N, ...)

- indexes (long tensor): (N, ...)

indexes, preds and target must have the same dimension. indexes indicate to which query a prediction belongs. Predictions will be first grouped by indexes and then *Recall* will be computed as the mean of the *Recall* over each query.

   Parameters

- **empty_target_action** (str) – Specify what to do with queries that do not have at least a positive target. Choose from:

  - 'neg': those queries count as 0.0 (default)

  - 'pos': those queries count as 1.0

  - 'skip': skip those queries; if all queries are skipped, 0.0 is returned

  - 'error': raise a ValueError

- **ignore_index** (Optional[int]) – Ignore predictions where the target is equal to this number.

- **k** (Optional[int]) – consider only the top k elements for each query (default: *None*, which considers them all)

- **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.

- **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 `empty_target_action` is not one of `error`, `skip`, `neg` or `pos`.

- **ValueError** – If `ignore_index` is not *None* or an integer.

- **ValueError** – If `k` parameter is not *None* or an integer larger than 0.

Example

```
>>> from torchmetrics import RetrievalRecall
>>> indexes = tensor([0, 0, 0, 1, 1, 1, 1])
>>> preds = tensor([0.2, 0.3, 0.5, 0.1, 0.3, 0.5, 0.2])
>>> target = tensor([False, False, True, False, True, False, True])
>>> r2 = RetrievalRecall(k=2)
>>> r2(preds, target, indexes=indexes)
tensor(0.7500)
```

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

RetrievalFallOut

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

- **ignore_index**⍦ (Optional[int]) – Ignore predictions where the target is equal to this number.

- **k**⍦ (Optional[int]) – consider only the top k elements for each query (default: *None*, which considers them all)

- **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.

- **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 empty_target_action is not one of error, skip, neg or pos.

- **ValueError** – If ignore_index is not *None* or an integer.

- **ValueError** – If k parameter is not *None* or an integer larger than 0.

Example

```
>>> from torchmetrics import RetrievalFallOut
>>> indexes = tensor([0, 0, 0, 1, 1, 1, 1])
>>> preds = tensor([0.2, 0.3, 0.5, 0.1, 0.3, 0.5, 0.2])
>>> target = tensor([False, False, True, False, True, False, True])
>>> fo = RetrievalFallOut(k=2)
>>> fo(preds, target, indexes=indexes)
tensor(0.5000)
```

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

compute()

First concat state *indexes*, *preds* and *target* since they were stored as lists.

After that, compute list of groups that will help in keeping together predictions about the same query. Finally, for each group compute the *_metric* if the number of negative targets is at least 1, otherwise behave as specified by *self.empty_target_action*.

> **Return type** Tensor

RetrievalNormalizedDCG

class torchmetrics.**RetrievalNormalizedDCG**(*empty_target_action='neg'*, *ignore_index=None*, *k=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=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, int, bool or float tensor): `(N, ...)`

- `indexes` (long tensor): `(N, ...)`

`indexes`, `preds` and `target` must have the same dimension. `indexes` indicate to which query a prediction belongs. Predictions will be first grouped by `indexes` and then *Normalized Discounted Cumulative Gain* will be computed as the mean of the *Normalized Discounted Cumulative Gain* over each query.

> **Parameters**
>
> - **empty_target_action**⏲ (`str`) – Specify what to do with queries that do not have at least a positive `target`. Choose from:
>
>   - `'neg'`: those queries count as `0.0` (default)
>
>   - `'pos'`: those queries count as `1.0`
>
>   - `'skip'`: skip those queries; if all queries are skipped, `0.0` is returned
>
>   - `'error'`: raise a `ValueError`
>
> - **ignore_index**⏲ (`Optional[int]`) – Ignore predictions where the target is equal to this number.
>
> - **k**⏲ (`Optional[int]`) – consider only the top k elements for each query (default: *None*, which considers them all)
>
> - **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.
>
> - **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 `empty_target_action` is not one of `error`, `skip`, `neg` or `pos`.
>
> - `ValueError` – If `ignore_index` is not *None* or an integer.
>
> - `ValueError` – If k parameter is not *None* or an integer larger than 0.

Example

```
>>> from torchmetrics import RetrievalNormalizedDCG
>>> indexes = tensor([0, 0, 0, 1, 1, 1, 1])
>>> preds = tensor([0.2, 0.3, 0.5, 0.1, 0.3, 0.5, 0.2])
>>> target = tensor([False, False, True, False, True, False, True])
>>> ndcg = RetrievalNormalizedDCG()
>>> ndcg(preds, target, indexes=indexes)
tensor(0.8467)
```

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

RetrievalHitRate

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

> Computes *IR HitRate*.
>
> Works with binary target data. Accepts float predictions from a model output.
>
> Forward accepts:
>
> - preds (float tensor): (N, ...)
> - target (long or bool tensor): (N, ...)
> - indexes (long tensor): (N, ...)
>
> indexes, preds and target must have the same dimension. indexes indicate to which query a prediction belongs. Predictions will be first grouped by indexes and then the *Hit Rate* will be computed as the mean of the *Hit Rate* over each query.
>
> > **Parameters**
> >
> > - **empty_target_action** (str) – Specify what to do with queries that do not have at least a positive target. Choose from:
> >   - 'neg': those queries count as 0.0 (default)
> >   - 'pos': those queries count as 1.0
> >   - 'skip': skip those queries; if all queries are skipped, 0.0 is returned
> >   - 'error': raise a ValueError
> > - **ignore_index** (Optional[int]) – Ignore predictions where the target is equal to this number.
> > - **k** (Optional[int]) – consider only the top k elements for each query (default: *None*, which considers them all)
> > - **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.
> > - **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 empty_target_action is not one of error, skip, neg or pos.
> > - **ValueError** – If ignore_index is not *None* or an integer.
> > - **ValueError** – If k parameter is not *None* or an integer larger than 0.

Example

```
>>> from torchmetrics import RetrievalHitRate
>>> indexes = tensor([0, 0, 0, 1, 1, 1, 1])
>>> preds = tensor([0.2, 0.3, 0.5, 0.1, 0.3, 0.5, 0.2])
>>> target = tensor([True, False, False, False, True, False, True])
>>> hr2 = RetrievalHitRate(k=2)
>>> hr2(preds, target, indexes=indexes)
tensor(0.5000)
```

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

2.5.9 Text

BERTScore

class torchmetrics.text.bert.**BERTScore**(*model_name_or_path=None*, *num_layers=None*, *all_layers=False*, *model=None*, *user_tokenizer=None*, *user_forward_fn=None*, *verbose=False*, *idf=False*, *device=None*, *max_length=512*, *batch_size=64*, *num_threads=4*, *return_hash=False*, *lang='en'*, *rescale_with_baseline=False*, *baseline_path=None*, *baseline_url=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Bert_score Evaluating Text Generation leverages the pre-trained contextual embeddings from BERT and matches words in candidate and reference sentences by cosine similarity. It has been shown to correlate with human judgment on sentence-level and system-level evaluation. Moreover, BERTScore computes precision, recall, and F1 measure, which can be useful for evaluating different language generation tasks.

This implemenation follows the original implementation from BERT_score.

> **Parameters**
>
> - **preds**⫽ – An iterable of predicted sentences.
>
> - **target**⫽ – An iterable of target sentences.
>
> - **model_type**⫽ – A name or a model path used to load *transformers* pretrained model.
>
> - **num_layers**⫽ (Optional[int]) – A layer of representation to use.
>
> - **all_layers**⫽ (bool) – An indication of whether the representation from all model's layers should be used. If *all_layers = True*, the argument *num_layers* is ignored.
>
> - **model**⫽ (Optional[Module]) – A user's own model. Must be of *torch.nn.Module* instance.
>
> - **user_tokenizer**⫽ (Optional[Any]) – A user's own tokenizer used with the own model. This must be an instance with the *__call__* method. This method must take an iterable of sentences (*List[str]*) and must return a python dictionary containing *"input_ids"* and *"attention_mask"* represented by *torch.Tensor*. It is up to the user's model of whether *"input_ids"* is a *torch.Tensor* of input ids or embedding vectors. This tokenizer must prepend an equivalent of *[CLS]* token and append an equivalent of *[SEP]* token as *transformers* tokenizer does.
>
> - **user_forward_fn**⫽ (Optional[Callable[[Module, Dict[str, Tensor]], Tensor]]) – A user's own forward function used in a combination with *user_model*. This function must take *user_model* and a python dictionary of containing *"input_ids"* and *"attention_mask"*

represented by *torch.Tensor* as an input and return the model's output represented by the single *torch.Tensor*.

- **verbose** (bool) – An indication of whether a progress bar to be displayed during the embeddings calculation.

- **idf** (bool) – An indication whether normalization using inverse document frequencies should be used.

- **device** (Union[str, device, None]) – A device to be used for calculation.

- **max_length** (int) – A maximum length of input sequences. Sequences longer than *max_length* are to be trimmed.

- **batch_size** (int) – A batch size used for model processing.

- **num_threads** (int) – A number of threads to use for a dataloader.

- **return_hash** (bool) – An indication of whether the correspodning *hash_code* should be returned.

- **lang** (str) – A language of input sentences.

- **rescale_with_baseline** (bool) – An indication of whether bertscore should be rescaled with a pre-computed baseline. When a pretrained model from *transformers* model is used, the corresponding baseline is downloaded from the original *bert-score* package from BERT_score if available. In other cases, please specify a path to the baseline csv/tsv file, which must follow the formatting of the files from BERT_score.

- **baseline_path** (Optional[str]) – A path to the user's own local csv/tsv file with the baseline scale.

- **baseline_url** (Optional[str]) – A url path to the user's own csv/tsv file with the baseline scale.

- **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.

- **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

Returns Python dictionary containing the keys *precision*, *recall* and *f1* with corresponding values.

Example

```
>>> from torchmetrics.text.bert import BERTScore
>>> preds = ["hello there", "general kenobi"]
>>> target = ["hello there", "master kenobi"]
>>> bertscore = BERTScore()
>>> from pprint import pprint
>>> pprint(bertscore(preds, target))
{'f1': [0.999..., 0.996...],
 'precision': [0.999..., 0.996...],
 'recall': [0.999..., 0.996...]}
```

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

compute()
> Calculate BERT scores.

>> **Return type** Dict[str, Union[List[float], str]]

>> **Returns** Python dictionary containing the keys *precision*, *recall* and *f1* with corresponding values.

update(*preds*, *target*)
> Store predictions/references for computing BERT scores. It is necessary to store sentences in a tokenized form to ensure the DDP mode working.

>> **Parameters**

>>> • **preds**⫽ (List[str]) – An iterable of predicted sentences.

>>> • **target**⫽ (List[str]) – An iterable of reference sentences.

> Deprecated since version v0.7: Args: predictions: This argument is deprecated in favor of *preds* and will be removed in v0.8. references: This argument is deprecated in favor of *target* and will be removed in v0.8.

>> **Return type** None

BLEUScore

class torchmetrics.**BLEUScore**(*n_gram=4*, *smooth=False*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)
> Calculate BLEU score of machine translated text with one or more references.

>> **Parameters**

>>> • **n_gram**⫽ (int) – Gram value ranged from 1 to 4 (Default 4)

>>> • **smooth**⫽ (bool) – Whether or not to apply smoothing – see [2]

>>> • **compute_on_step**⫽ (bool) – Forward only calls update() and returns 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.

>>> • **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 BLEUScore
>>> preds = ['the cat is on the mat']
>>> target = [['there is a cat on the mat', 'a cat is on the mat']]
>>> metric = BLEUScore()
>>> metric(preds, target)
tensor(0.7598)
```

References

[1] BLEU: a Method for Automatic Evaluation of Machine Translation by Papineni, Kishore, Salim Roukos, Todd Ward, and Wei-Jing Zhu BLEU

[2] Automatic Evaluation of Machine Translation Quality Using Longest Common Subsequence and Skip-Bigram Statistics by Chin-Yew Lin and Franz Josef Och Machine Translation Evolution

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

`compute()`
> Calculate BLEU score.
>
> > **Return type** `Tensor`
> >
> > **Returns** Tensor with BLEU Score

`update`(*preds*, *target*)
> Compute Precision Scores.
>
> > **Parameters**
> >
> > - **preds**⫽ (`Sequence`[`str`]) – An iterable of machine translated corpus
> >
> > - **target**⫽ (`Sequence`[`Sequence`[`str`]]) – An iterable of iterables of reference corpus
>
> Deprecated since version v0.7: Args: translate_corpus: This argument is deprecated in favor of *preds* and will be removed in v0.8. reference_corpus: This argument is deprecated in favor of *target* and will be removed in v0.8.
>
> > **Return type** `None`

CharErrorRate

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

Character error rate (*CharErrorRate*) is a metric of the performance of an automatic speech recognition (ASR) system. This value indicates the percentage of characters that were incorrectly predicted. The lower the value, the better the performance of the ASR system with a CharErrorRate of 0 being a perfect score. Character error rate can then be computed as:

$$CharErrorRate = \frac{S + D + I}{N} = \frac{S + D + I}{S + D + C}$$

where:

> - S is the number of substitutions,
>
> - D is the number of deletions,
>
> - I is the number of insertions,
>
> - C is the number of correct characters,
>
> - N is the number of characters in the reference (N=S+D+C).

Compute CharErrorRate score of transcribed segments against references.

> **Parameters**
>
> - **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.

- **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

Returns Character error rate score

Examples

```
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> metric = CharErrorRate()
>>> metric(preds, target)
tensor(0.3415)
```

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

compute()
    Calculate the character error rate.

> **Return type** `Tensor`

> **Returns** Character error rate score

update(*preds*, *target*)
    Store references/predictions for computing Character Error Rate scores.

> **Parameters**
>
> - **preds**¶ (`Union`[`str`, `List`[`str`]]) – Transcription(s) to score as a string or list of strings
>
> - **target**¶ (`Union`[`str`, `List`[`str`]]) – Reference(s) for each speech input as a string or list of strings

Deprecated since version v0.7: Args: predictions: This argument is deprecated in favor of *preds* and will be removed in v0.8. references: This argument is deprecated in favor of *target* and will be removed in v0.8.

> **Return type** `None`

CHRFScore

`class` torchmetrics.**CHRFScore**(*n_char_order=6*, *n_word_order=2*, *beta=2.0*, *lowercase=False*, *whitespace=False*, *return_sentence_level_score=False*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Calculate chrf score of machine translated text with one or more references. This implementation supports both ChrF score computation introduced in [1] and chrF++ score introduced in *chrF++ score_*. This implementation follows the implmenetaions from https://github.com/m-popovic/chrF and https://github.com/mjpost/sacrebleu/blob/master/sacrebleu/metrics/chrf.py.

> **Parameters**
>
> - **n_char_order**¶ (`int`) – A character n-gram order. If *n_char_order=6*, the metrics refers to the official chrF/chrF++.

- **n_word_order**⁋ (int) – A word n-gram order. If *n_word_order=2*, the metric refers to the official chrF++. If *n_word_order=0*, the metric is equivalent to the original ChrF.

- **beta**⁋ (float) – A parameter determining an importance of recall w.r.t. precision. If *beta=1*, their importance is equal.

- **lowercase**⁋ (bool) – An indication whether to enable case-insesitivity.

- **whitespace**⁋ (bool) – An indication whether keep whitespaces during n-gram extraction.

- **return_sentence_level_score**⁋ (bool) – An indication whether a sentence-level chrF/chrF++ score to be returned.

- **compute_on_step**⁋ (bool) – Forward only calls `update()` and returns 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.

- **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 `n_char_order` is not an integer greater than or equal to 1.

- **ValueError** – If `n_word_order` is not an integer greater than or equal to 0.

- **ValueError** – If `beta` is smaller than 0.

Example

```
>>> from torchmetrics import CHRFScore
>>> preds = ['the cat is on the mat']
>>> target = [['there is a cat on the mat', 'a cat is on the mat']]
>>> metric = CHRFScore()
>>> metric(preds, target)
tensor(0.8640)
```

References

[1] chrF: character n-gram F-score for automatic MT evaluation by Maja Popović chrF score [2] chrF++: words helping character n-grams by Maja Popović chrF++ score

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

compute()
    Calculate chrF/chrF++ score.

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

        Returns A corpus-level chrF/chrF++ score. (Optionally) A list of sentence-level chrF/chrF++ scores if *return_sentence_level_score=True*.

update(*preds*, *target*)
    Compute Precision Scores.

        Parameters

- **preds**⫟ (Sequence[str]) – An iterable of hypothesis corpus.

- **target**⫟ (Sequence[Sequence[str]]) – An iterable of iterables of reference corpus.

    Return type None

ExtendedEditDistance

class torchmetrics.**ExtendedEditDistance**(*language='en'*, *return_sentence_level_score=False*, *alpha=2.0*, *rho=0.3*, *deletion=0.2*, *insertion=1.0*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Computes extended edit distance score (ExtendedEditDistance) [1] for strings or list of strings. The metric utilises the Levenshtein distance and extends it by adding an additional jump operation.

   Parameters

- **language**⫟ (Literal['en', 'ja']) – Language used in sentences. Only supports English (en) and Japanese (ja) for now. Defaults to en

- **return_sentence_level_score**⫟ (bool) – An indication of whether sentence-level EED score is to be returned

- **alpha**⫟ (float) – optimal jump penalty, penalty for jumps between characters

- **rho**⫟ (float) – coverage cost, penalty for repetition of characters

- **deletion**⫟ (float) – penalty for deletion of character

- **insertion**⫟ (float) – penalty for insertion or substitution of character

- **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

   Returns Extended edit distance score as a tensor

Example

```
>>> from torchmetrics import ExtendedEditDistance
>>> preds = ["this is the prediction", "here is an other sample"]
>>> target = ["this is the reference", "here is another one"]
>>> metric = ExtendedEditDistance()
>>> metric(preds=preds, target=target)
tensor(0.3078)
```

PyTorch-Metrics Documentation, Release 0.7.1

References

[1] P. Stanchev, W. Wang, and H. Ney, "EED: Extended Edit Distance Measure for Machine Translation", submitted to WMT 2019. ExtendedEditDistance

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

compute()
   Calculate extended edit distance score.

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

>   **Returns** Extended edit distance score as tensor

update(*preds*, *target*)
   Update ExtendedEditDistance statistics.

>   **Parameters**
>
>   - **preds**⫽ (Union[str, Sequence[str]]) – An iterable of hypothesis corpus
>
>   - **target**⫽ (Sequence[Union[str, Sequence[str]]]) – An iterable of iterables of reference corpus
>
>   **Return type** None

MatchErrorRate

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

Match error rate (*MatchErrorRate*) is a common metric of the performance of an automatic speech recognition system. This value indicates the percentage of words that were incorrectly predicted and inserted. The lower the value, the better the performance of the ASR system with a MatchErrorRate of 0 being a perfect score. Match error rate can then be computed as:

$$mer = \frac{S + D + I}{N + I} = \frac{S + D + I}{S + D + C + I}$$

where:

> - S is the number of substitutions,
>
> - D is the number of deletions,
>
> - I is the number of insertions,
>
> - C is the number of correct words,
>
> - N is the number of words in the reference (N=S+D+C).

>   **Parameters**
>
>   - **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.
>
>   - **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

124 | **Chapter 2. More reading**

> **Returns** Match error rate score

Examples

```
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> metric = MatchErrorRate()
>>> metric(preds, target)
tensor(0.4444)
```

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

compute()
> Calculate the Match error rate.

> > **Return type** Tensor

> > **Returns** Match error rate

update(*preds*, *target*)
> Store references/predictions for computing Match Error Rate scores.

> > **Parameters**

> > > • **preds** (Union[str, List[str]]) – Transcription(s) to score as a string or list of strings

> > > • **target** (Union[str, List[str]]) – Reference(s) for each speech input as a string or list of strings

> > **Return type** None

ROUGEScore

class torchmetrics.text.rouge.**ROUGEScore**(*use_stemmer=False*, *accumulate='best'*, *rouge_keys=('rouge1', 'rouge2', 'rougeL', 'rougeLsum')*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Calculate Rouge Score, used for automatic summarization. This implementation should imitate the behaviour of the *rouge-score* package *Python ROUGE Implementation*

> **Parameters**

> > • **use_stemmer** (bool) – Use Porter stemmer to strip word suffixes to improve matching.

> > • **accumulate** (Literal['avg', 'best']) – Useful incase of multi-reference rouge score. - avg takes the avg of all references with respect to predictions - best takes the best fmeasure score obtained between prediction and multiple corresponding references.

> > • **rouge_keys** (Union[str, Tuple[str, ...]]) – A list of rouge types to calculate. Keys that are allowed are rougeL, rougeLsum, and rouge1 through rouge9.

> > • **compute_on_step** (bool) – Forward only calls update() and returns 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.

- **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.text.rouge import ROUGEScore
>>> preds = "My name is John"
>>> target = "Is your name John"
>>> rouge = ROUGEScore()
>>> from pprint import pprint
>>> pprint(rouge(preds, target))
{'rouge1_fmeasure': tensor(0.7500),
 'rouge1_precision': tensor(0.7500),
 'rouge1_recall': tensor(0.7500),
 'rouge2_fmeasure': tensor(0.),
 'rouge2_precision': tensor(0.),
 'rouge2_recall': tensor(0.),
 'rougeL_fmeasure': tensor(0.5000),
 'rougeL_precision': tensor(0.5000),
 'rougeL_recall': tensor(0.5000),
 'rougeLsum_fmeasure': tensor(0.5000),
 'rougeLsum_precision': tensor(0.5000),
 'rougeLsum_recall': tensor(0.5000)}
```

Raises

- **ValueError** – If the python packages `nltk` is not installed.

- **ValueError** – If any of the `rouge_keys` does not belong to the allowed set of keys.

References

[1] ROUGE: A Package for Automatic Evaluation of Summaries by Chin-Yew Lin Rouge Detail

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

compute()
    Calculate (Aggregate and provide confidence intervals) ROUGE score.

        Return type Dict[str, Tensor]

        Returns Python dictionary of rouge scores for each input rouge key.

update(*preds*, *target*)
    Compute rouge scores.

        Parameters

        - **preds** *(Union[str, Sequence[str]])* – An iterable of predicted sentences or a single predicted sentence.

        - **target** *(Union[str, Sequence[str], Sequence[Sequence[str]]])* – An iterable of iterable of target sentences or an iterable of target sentences or a single target sentence.

        Return type None

SacreBLEUScore

class torchmetrics.**SacreBLEUScore**(*n_gram=4*, *smooth=False*, *tokenize='13a'*, *lowercase=False*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Calculate BLEU score [1] of machine translated text with one or more references. This implementation follows the behaviour of SacreBLEU [2] implementation from https://github.com/mjpost/sacrebleu.

The SacreBLEU implementation differs from the NLTK BLEU implementation in tokenization techniques.

> **Parameters**
>
> - **n_gram**⁗ (int) – Gram value ranged from 1 to 4 (Default 4)
>
> - **smooth**⁗ (bool) – Whether or not to apply smoothing – see [2]
>
> - **tokenize**⁗ (Literal['none', '13a', 'zh', 'intl', 'char']) – Tokenization technique to be used. (Default '13a') Supported tokenization: ['none', '13a', 'zh', 'intl', 'char']
>
> - **lowercase**⁗ (bool) – If True, BLEU score over lowercased text is calculated.
>
> - **compute_on_step**⁗ (bool) – Forward only calls update() and returns 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.
>
> - **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 tokenize not one of 'none', '13a', 'zh', 'intl' or 'char'
> >
> > **ValueError:** If tokenize is set to 'intl' and *regex* is not installed

Example

```
>>> from torchmetrics import SacreBLEUScore
>>> preds = ['the cat is on the mat']
>>> target = [['there is a cat on the mat', 'a cat is on the mat']]
>>> metric = SacreBLEUScore()
>>> metric(preds, target)
tensor(0.7598)
```

References

[1] BLEU: a Method for Automatic Evaluation of Machine Translation by Papineni, Kishore, Salim Roukos, Todd Ward, and Wei-Jing Zhu BLEU

[2] A Call for Clarity in Reporting BLEU Scores by Matt Post.

[3] Automatic Evaluation of Machine Translation Quality Using Longest Common Subsequence and Skip-Bigram Statistics by Chin-Yew Lin and Franz Josef Och Machine Translation Evolution

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

update(*preds*, *target*)
> Compute Precision Scores.
>
> > **Parameters**
> >
> > - **preds**⫽ (Sequence[str]) – An iterable of machine translated corpus
> >
> > - **target**⫽ (Sequence[Sequence[str]]) – An iterable of iterables of reference corpus
>
> Deprecated since version v0.7: Args: translate_corpus: This argument is deprecated in favor of *preds* and will be removed in v0.8. reference_corpus: This argument is deprecated in favor of *target* and will be removed in v0.8.
>
> > **Return type** None

SQuAD

class torchmetrics.**SQuAD**(*compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)
> Calculate SQuAD Metric which corresponds to the scoring script for version 1 of the Stanford Question Answering Dataset (SQuAD).
>
> > **Parameters**
> >
> > - **compute_on_step**⫽ (bool) – Forward only calls update() and returns 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.
> >
> > - **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 SQuAD
>>> preds = [{"prediction_text": "1976", "id": "56e10a3be3433e1400422b22"}]
>>> target = [{"answers": {"answer_start": [97], "text": ["1976"]}, "id":
↪"56e10a3be3433e1400422b22"}]
>>> sqaud = SQuAD()
>>> sqaud(preds, target)
{'exact_match': tensor(100.), 'f1': tensor(100.)}
```

References

[1] SQuAD: 100,000+ Questions for Machine Comprehension of Text by Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, Percy Liang SQuAD Metric .

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

`compute()`
> Aggregate the F1 Score and Exact match for the batch.
>
> > **Return type** `Dict[str, Tensor]`
> >
> > **Returns** Dictionary containing the F1 score, Exact match score for the batch.

`update`(*preds*, *target*)
> Compute F1 Score and Exact Match for a collection of predictions and references.
>
> > **Parameters**
> >
> > - **preds**⫽ (`Union[Dict[str, str], List[Dict[str, str]]]`) – A Dictionary or List of Dictionary-s that map *id* and *prediction_text* to the respective values.
> >
> >   Example prediction:
> >
> >   ```
> >   {"prediction_text": "TorchMetrics is awesome", "id": "123"}
> >   ```
> >
> > - **target**⫽ (`Union[Dict[str, Union[str, Dict[str, Union[List[str], List[int]]]]], List[Dict[str, Union[str, Dict[str, Union[List[str], List[int]]]]]]]`) – A Dictionary or List of Dictionary-s that contain the *answers* and *id* in the SQuAD Format.
> >
> >   Example target:
> >
> >   ```
> >   {
> >       'answers': [{'answer_start': [1], 'text': ['This is a test␣
> >   ↪answer']}],
> >       'id': '1',
> >   }
> >   ```
> >
> >   Reference SQuAD Format:
> >
> >   ```
> >   {
> >       'answers': {'answer_start': [1], 'text': ['This is a test text
> >   ↪']},
> >       'context': 'This is a test context.',
> >       'id': '1',
> >       'question': 'Is this a test?',
> >       'title': 'train test'
> >   }
> >   ```
> >
> > **Raises** `KeyError` – If the required keys are missing in either predictions or targets.
> >
> > **Return type** `None`

TranslationEditRate

class torchmetrics.**TranslationEditRate**(*normalize=False*, *no_punctuation=False*, *lowercase=True*, *asian_support=False*, *return_sentence_level_score=False*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Calculate Translation edit rate (TER) of machine translated text with one or more references. This implementation follows the implmenetaions from https://github.com/mjpost/sacrebleu/blob/master/sacrebleu/metrics/ter.py. The *sacrebleu* implmenetation is a near-exact reimplementation of the Tercom algorithm, produces identical results on all "sane" outputs.

> **Parameters**
>
> - **normalize**⫽ (bool) – An indication whether a general tokenization to be applied.
>
> - **no_punctuation**⫽ (bool) – An indication whteher a punctuation to be removed from the sentences.
>
> - **lowercase**⫽ (bool) – An indication whether to enable case-insesitivity.
>
> - **asian_support**⫽ (bool) – An indication whether asian characters to be processed.
>
> - **return_sentence_level_score**⫽ (bool) – An indication whether a sentence-level TER to be returned.
>
> - **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.
>
> - **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

```
>>> preds = ['the cat is on the mat']
>>> target = [['there is a cat on the mat', 'a cat is on the mat']]
>>> metric = TranslationEditRate()
>>> metric(preds, target)
tensor(0.1538)
```

References: [1] A Study of Translation Edit Rate with Targeted Human Annotation by Mathew Snover, Bonnie Dorr, Richard Schwartz, Linnea Micciulla and John Makhoul TER

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

compute()
> Calculate the translate error rate (TER).
>
> > **Return type** Union[Tensor, Tuple[Tensor, Tensor]]
> >
> > **Returns** A corpus-level translation edit rate (TER). (Optionally) A list of sentence-level translation_edit_rate (TER) if *return_sentence_level_score=True*.

update(*preds*, *target*)
> Update TER statistics.

Parameters

- **preds**⫮ (Union[str, Sequence[str]]) – An iterable of hypothesis corpus.

- **target**⫮ (Sequence[Union[str, Sequence[str]]]) – An iterable of iterables of reference corpus.

Return type None

WER

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

Word error rate (WER) is a common metric of the performance of an automatic speech recognition system.

Deprecated since version v0.7: Use `torchmetrics.WordErrorRate`. Will be removed in v0.8.

Examples

```
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> metric = WER()
>>> metric(preds, target)
tensor(0.5000)
```

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

WordInfoLost

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

Word Information Lost (*WordInfoLost*) is a metric of the performance of an automatic speech recognition system. This value indicates the percentage of words that were incorrectly predicted between a set of ground-truth sentences and a set of hypothesis sentences. The lower the value, the better the performance of the ASR system with a WordInfoLost of 0 being a perfect score. Word Information Lost rate can then be computed as:

$$wil = 1 - \frac{C}{N} + \frac{C}{P}$$

where:

- C is the number of correct words,

- N is the number of words in the reference

- P is the number of words in the prediction

Parameters

- **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.

- **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

Examples

```
>>> from torchmetrics import WordInfoLost
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> metric = WordInfoLost()
>>> metric(preds, target)
tensor(0.6528)
```

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

compute()
    Calculate the Word Information Lost.

> **Return type** Tensor

> **Returns** Word Information Lost score

update(*preds*, *target*)
    Store predictions/references for computing Word Information Lost scores.

> **Parameters**
>
> - **preds**⫻ (Union[str, List[str]]) – Transcription(s) to score as a string or list of strings
>
> - **target**⫻ (Union[str, List[str]]) – Reference(s) for each speech input as a string or list of strings

> **Return type** None

WordInfoPreserved

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

word Information Preserved (*WordInfoPreserved*) is a metric of the performance of an automatic speech recognition system. This value indicates the percentage of words that were correctly predicted between a set of ground-truth sentences and a set of hypothesis sentences. The higher the value, the better the performance of the ASR system with a WordInfoPreserved of 0 being a perfect score. word Information Preserved rate can then be computed as:

$$wip = \frac{C}{N} + \frac{C}{P}$$

where:

- C is the number of correct words,

- N is the number of words in the reference

- P is the number of words in the prediction

> **Parameters**
>
> - **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.

- **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

Examples

```
>>> from torchmetrics import WordInfoPreserved
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> metric = WordInfoPreserved()
>>> metric(preds, target)
tensor(0.3472)
```

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

compute()
> Calculate the word Information Preserved.
>
> > **Return type** `Tensor`
> >
> > **Returns** word Information Preserved score

update(*preds*, *target*)
> Store predictions/references for computing word Information Preserved scores.
>
> > **Parameters**
> >
> > - **preds** (`Union[str, List[str]]`) – Transcription(s) to score as a string or list of strings
> >
> > - **target** (`Union[str, List[str]]`) – Reference(s) for each speech input as a string or list of strings
> >
> > **Return type** `None`

2.5.10 Wrappers

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

BootStrapper

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*)
> Using Turn a Metric into a Bootstrapped That can automate the process of getting confidence intervals for metric values. This wrapper class basically keeps multiple copies of the same base metric in memory and whenever `update` or `forward` is called, all input tensors are resampled (with replacement) along the first dimension.
>
> > **Parameters**
> >
> > - **base_metric** (`Metric`) – base metric class to wrap
> >
> > - **num_bootstraps** (`int`) – number of copies to make of the base metric for bootstrapping

- **mean**⏐ (bool) – if `True` return the mean of the bootstraps

- **std**⏐ (bool) – if `True` return the standard 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

MetricTracker

class torchmetrics.**MetricTracker**(*metric*, *maximize=True*)

A wrapper class that can help keeping track of a metric over time and implement useful methods. The wrapper implements the standard *update*, *compute*, *reset* methods that just calls corresponding method of the currently tracked metric. However, the following additional methods are provided:

-MetricTracker.n_steps: number of metrics being tracked

-MetricTracker.increment(): initialize a new metric for being tracked

-MetricTracker.compute_all(): get the metric value for all steps

-MetricTracker.best_metric(): returns the best value

> **Parameters**
>
> - **metric**⫢ (*Metric*) – instance of a torchmetric modular to keep track of at each timestep.
>
> - **maximize**⫢ (*bool*) – bool indicating if higher metric values are better (*True*) or lower is better (*False*)

Example

```
>>> from torchmetrics import Accuracy, MetricTracker
>>> _ = torch.manual_seed(42)
>>> tracker = MetricTracker(Accuracy(num_classes=10))
>>> for epoch in range(5):
...     tracker.increment()
...     for batch_idx in range(5):
...         preds, target = torch.randint(10, (100,)), torch.randint(10, (100,))
...         tracker.update(preds, target)
...     print(f"current acc={tracker.compute()}")
current acc=0.1120000034570694
current acc=0.08799999952316284
current acc=0.12600000202655792
current acc=0.07999999821186066
current acc=0.10199999809265137
>>> best_acc, which_epoch = tracker.best_metric(return_step=True)
>>> tracker.compute_all()
tensor([0.1120, 0.0880, 0.1260, 0.0800, 0.1020])
```

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

best_metric(*return_step=False*)

Returns the highest metric out of all tracked.

> **Parameters return_step**⫢ (*bool*) – If *True* will also return the step with the highest metric value.
>
> **Return type** Union[float, Tuple[int, float]]
>
> **Returns** The best metric value, and optionally the timestep.

compute()

Call compute of the current metric being tracked.

> **Return type** Any

compute_all()
> Compute the metric value for all tracked metrics.
>
>> **Return type** Tensor

forward(**args*, ***kwargs*)
> Calls forward of the current metric being tracked.
>
>> **Return type** None

increment()
> Creates a new instance of the input metric that will be updated next.
>
>> **Return type** None

reset()
> Resets the current metric being tracked.
>
>> **Return type** None

reset_all()
> Resets all metrics being tracked.
>
>> **Return type** None

update(**args*, ***kwargs*)
> Updates the current metric being tracked.
>
>> **Return type** None

property n_steps: int
> Returns the number of times the tracker has been incremented.
>
>> **Return type** int

MinMaxMetric

class torchmetrics.**MinMaxMetric**(*base_metric*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)
> Wrapper Metric that tracks both the minimum and maximum of a scalar/tensor across an experiment. The min/max value will be updated each time *.compute* is called.
>
> **Parameters**
>
> - **base_metric** (*Metric*) – The metric of which you want to keep track of its maximum and minimum values.
>
> - **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.
>
> - **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 base_metric` argument is not an subclasses instance of ``torchmetrics.Metric
>
> **Example::**

```
>>> import torch
>>> from torchmetrics import Accuracy
>>> from pprint import pprint
>>> base_metric = Accuracy()
>>> minmax_metric = MinMaxMetric(base_metric)
>>> preds_1 = torch.Tensor([[0.1, 0.9], [0.2, 0.8]])
>>> preds_2 = torch.Tensor([[0.9, 0.1], [0.2, 0.8]])
>>> labels = torch.Tensor([[0, 1], [0, 1]]).long()
>>> pprint(minmax_metric(preds_1, labels))
{'max': tensor(1.), 'min': tensor(1.), 'raw': tensor(1.)}
>>> pprint(minmax_metric.compute())
{'max': tensor(1.), 'min': tensor(1.), 'raw': tensor(1.)}
>>> minmax_metric.update(preds_2, labels)
>>> pprint(minmax_metric.compute())
{'max': tensor(1.), 'min': tensor(0.7500), 'raw': tensor(0.7500)}
```

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

compute()
> Computes the underlying metric as well as max and min values for this metric.
>
> Returns a dictionary that consists of the computed value (`raw`), as well as the minimum (`min`) and maximum (`max`) values.
>
>> **Return type** Dict[str, Tensor]

reset()
> Sets `max_val` and `min_val` to the initialization bounds and resets the base metric.
>
>> **Return type** None

update(**args*, ***kwargs*)
> Updates the underlying metric.
>
>> **Return type** None

MultioutputWrapper

class torchmetrics.**MultioutputWrapper**(*base_metric*, *num_outputs*, *output_dim=- 1*, *remove_nans=True*, *squeeze_outputs=True*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)
Wrap a base metric to enable it to support multiple outputs.

Several torchmetrics metrics, such as `torchmetrics.regression.spearman.SpearmanCorrcoef` lack support for multioutput mode. This class wraps such metrics to support computing one metric per output. Unlike specific torchmetric metrics, it doesn't support any aggregation across outputs. This means if you set *num_outputs* to 2, *compute()* will return a Tensor of dimension (2, . . .) where . . . represents the dimensions the metric returns when not wrapped.

In addition to enabling multioutput support for metrics that lack it, this class also supports, albeit in a crude fashion, dealing with missing labels (or other data). When `remove_nans` is passed, the class will remove the intersection of NaN containing "rows" upon each update for each output. For example, suppose a user uses *MultioutputWrapper* to wrap `torchmetrics.regression.r2.R2Score` with 2 outputs, one of which occasionally has missing labels for classes like `R2Score` is that this class supports removing NaN values (parameter `remove_nans`) on a per-output basis. When `remove_nans` is passed the wrapper will remove all rows

Parameters

- **base_metric**¶ (*Metric*) – Metric being wrapped.

- **num_outputs**¶ (int) – Expected dimensionality of the output dimension. This parameter is used to determine the number of distinct metrics we need to track.

- **output_dim**¶ (int) – Dimension on which output is expected. Note that while this provides some flexibility, the output dimension must be the same for all inputs to update. This applies even for metrics such as *Accuracy* where the labels can have a different number of dimensions than the predictions. This can be worked around if the output dimension can be set to -1 for both, even if -1 corresponds to different dimensions in different inputs.

- **remove_nans**¶ (bool) – Whether to remove the intersection of rows containing NaNs from the values passed through to each underlying metric. Proper operation requires all tensors passed to update to have dimension *(N, . . .)* where N represents the length of the batch or dataset being passed in.

- **squeeze_outputs**¶ (bool) – If true, will squeeze the 1-item dimensions left after *index_select* is applied. This is sometimes unnecessary but harmless for metrics such as *R2Score* but useful for certain classification metrics that can't handle additional 1-item dimensions.

- **compute_on_step**¶ (bool) – Whether to recompute the metric value on each update step.

- **dist_sync_on_step**¶ (bool) – Required for distributed training support.

- **process_group**¶ (Optional[Any]) – Specify the process group on which synchronization is called. The

- **dist_sync_fn**¶ (Optional[Callable]) – Required for distributed training support.

Example

```
>>> # Mimic R2Score in `multioutput`, `raw_values` mode:
>>> import torch
>>> from torchmetrics import MultioutputWrapper, R2Score
>>> target = torch.tensor([[0.5, 1], [-1, 1], [7, -6]])
>>> preds = torch.tensor([[0, 2], [-1, 2], [8, -5]])
>>> r2score = MultioutputWrapper(R2Score(), 2)
>>> r2score(preds, target)
[tensor(0.9654), tensor(0.9082)]
>>> # Classification metric where prediction and label tensors have different␣
↪shapes.
>>> from torchmetrics import BinnedAveragePrecision
>>> target = torch.tensor([[1, 2], [2, 0], [1, 2]])
>>> preds = torch.tensor([
...     [[.1, .8], [.8, .05], [.1, .15]],
...     [[.1, .1], [.2, .3], [.7, .6]],
...     [[.002, .4], [.95, .45], [.048, .15]]
... ])
>>> binned_avg_precision = MultioutputWrapper(BinnedAveragePrecision(3,␣
↪thresholds=5), 2)
>>> binned_avg_precision(preds, target)
[[tensor(-0.), tensor(1.0000), tensor(1.0000)], [tensor(0.3333), tensor(-0.),␣
↪tensor(0.6667)]]
```

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

compute()

> Compute metrics.
>
>> **Return type** List[Tensor]

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

> Call underlying forward methods and aggregate the results if they're non-null.
>
> We override this method to ensure that state variables get copied over on the underlying metrics.
>
>> **Return type** Any

reset()

> Reset all underlying metrics.
>
>> **Return type** None

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

> Update each underlying metric with the corresponding output.
>
>> **Return type** None

2.6 Functional metrics

2.6.1 Audio

perceptual_evaluation_speech_quality [func]

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

> PESQ (Perceptual Evaluation of Speech Quality)
>
> This is a wrapper for the pesq package [1]. Note that input will be moved to *cpu* to perform the metric calculation.
>
> ---
>
> **Note:** using this metrics requires you to have pesq install. Either install as pip install torchmetrics[audio] or pip install pesq
>
> ---
>
> **Parameters**
>
> - **preds** (Tensor) – shape [...,time]
> - **target** (Tensor) – shape [...,time]
> - **fs** (int) – sampling frequency, should be 16000 or 8000 (Hz)
> - **mode** (str) – 'wb' (wide-band) or 'nb' (narrow-band)
> - **keep_same_device** (bool) – whether to move the pesq value to the device of preds
>
> **Return type** Tensor
>
> **Returns** pesq value of shape [...]
>
> **Raises**
>
> - **ModuleNotFoundError** – If peqs package is not installed
> - **ValueError** – If fs is not either 8000 or 16000

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

Example

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

References

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

permutation_invariant_training [func]

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

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

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

> **Parameters**
>
> - **preds** (Tensor) – shape [batch, spk, . . .]
>
> - **target** (Tensor) – shape [batch, spk, . . .]
>
> - **metric_func** (Callable) – a metric function accept a batch of target and estimate, i.e. metric_func(preds[:, i, . . .], target[:, j, . . .]), and returns a batch of metric tensors [batch]
>
> - **eval_func** (str) – the function to find the best permutation, can be 'min' or 'max', i.e. the smaller the better or the larger the better.
>
> - **kwargs** (Dict[str, Any]) – additional args for metric_func
>
> **Return type** Tuple[Tensor, Tensor]
>
> **Returns** best_metric of shape [batch], best_perm of shape [batch]

Example

```
>>> from torchmetrics.functional.audio import scale_invariant_signal_distortion_
↪ratio
>>> # [batch, spk, time]
>>> preds = torch.tensor([[[-0.0579,  0.3560, -0.9604], [-0.1719,  0.3205,  0.
↪2951]]])
>>> target = torch.tensor([[[ 1.0958, -0.1648,  0.5228], [-0.4100,  1.1942, -0.
↪5103]]])
>>> best_metric, best_perm = permutation_invariant_training(
...     preds, target, scale_invariant_signal_distortion_ratio, 'max')
>>> best_metric
tensor([-5.1091])
>>> best_perm
tensor([[0, 1]])
>>> pit_permutate(preds, best_perm)
tensor([[[-0.0579,  0.3560, -0.9604],
         [-0.1719,  0.3205,  0.2951]]])
```

Reference: [1] Permutation Invariant Training of Deep Models

signal_distortion_ratio [func]

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

Signal to Distortion Ratio (SDR) [1,2,3]

> **Parameters**
>
> - **preds** (Tensor) – shape [..., time]
>
> - **target** (Tensor) – shape [..., time]
>
> - **use_cg_iter** (Optional[int]) – If provided, an iterative method is used to solve for the distortion filter coefficients instead of direct Gaussian elimination. This can speed up the computation of the metrics in case the filters are long. Using a value of 10 here has been shown to provide good accuracy in most cases and is sufficient when using this loss to train neural separation networks.
>
> - **filter_length** (int) – The length of the distortion filter allowed
>
> - **zero_mean** (bool) – When set to True, the mean of all signals is subtracted prior to computation of the metrics
>
> - **load_diag** (Optional[float]) – If provided, this small value is added to the diagonal coefficients of the system metrics when solving for the filter coefficients. This can help stabilize the metric in the case where some of the reference signals may sometimes be zero
>
> **Raises** ModuleNotFoundError – If fast-bss-eval package is not installed
>
> **Return type** Tensor
>
> **Returns** sdr value of shape [...]

Example

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

Note:

1. **when pytorch<1.8.0, numpy will be used to calculate this metric, which causes `sdr` to be** non-differentiable and slower to calculate

2. using this metrics requires you to have `fast-bss-eval` install. Either install as `pip install torchmetrics[audio]` or `pip install fast-bss-eval`

3. preds and target need to have the same dtype, otherwise target will be converted to preds' dtype

References

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

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

[3] https://github.com/fakufaku/fast_bss_eval

scale_invariant_signal_distortion_ratio [func]

`torchmetrics.functional.`**`scale_invariant_signal_distortion_ratio`**(*preds*, *target*, *zero_mean=False*)
Calculates Scale-invariant signal-to-distortion ratio (SI-SDR) metric. The SI-SDR value is in general considered an overall measure of how good a source sound.

> **Parameters**
>> • **preds** (Tensor) – shape [...,time]
>>
>> • **target** (Tensor) – shape [...,time]
>>
>> • **zero_mean** (bool) – If to zero mean target and preds or not

> **Return type** `Tensor`

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

Example

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

References

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

scale_invariant_signal_noise_ratio [func]

`torchmetrics.functional.`**`si_snr`**`(`*preds*, *target*`)`
    Scale-invariant signal-to-noise ratio (SI-SNR).

    Deprecated since version v0.7: Use `torchmetrics.functional.scale_invariant_signal_noise_ratio()`. Will be removed in v0.8.

Example

```
>>> import torch
>>> si_snr(torch.tensor([2.5, 0.0, 2.0, 8.0]), torch.tensor([3.0, -0.5, 2.0, 7.0]))
tensor(15.0918)
```

> **Return type** `Tensor`

signal_noise_ratio [func]

`torchmetrics.functional.`**`signal_noise_ratio`**`(`*preds*, *target*, *zero_mean=False*`)`
    Signal-to-noise ratio (SNR):

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

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

> **Parameters**
>
> - **preds**¶ (`Tensor`) – shape `[...,time]`
>
> - **target**¶ (`Tensor`) – shape `[...,time]`
>
> - **zero_mean**¶ (`bool`) – if to zero mean target and preds or not

> **Return type** Tensor

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

Example

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

References

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

short_time_objective_intelligibility [func]

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

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

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

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

---

> **Parameters**
> - **preds**∅ (Tensor) – shape [...,time]
> - **target**∅ (Tensor) – shape [...,time]
> - **fs**∅ (int) – sampling frequency (Hz)
> - **extended**∅ (bool) – whether to use the extended STOI described in [4]
> - **keep_same_device**∅ (bool) – whether to move the stoi value to the device of preds
>
> **Return type** Tensor
>
> **Returns** stoi value of shape [. . .]
>
> **Raises** ModuleNotFoundError – If pystoi package is not installed

Example

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

References

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

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

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

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

2.6.2 Classification

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

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

Accepts all input types listed in *Input types*.

   Parameters

   - **preds** (Tensor) – Predictions from model (probabilities, logits or labels)

   - **target** (Tensor) – Ground truth labels

   - **average** (str) – Defines the reduction that is applied. Should be one of the following:

     – 'micro' [default]: Calculate the metric globally, across all samples and classes.

- 'macro': Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).

- 'weighted': Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (tp + fn).

- 'none' or None: Calculate the metric for each class separately, and return the metric for every class.

- 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

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

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

- **mdmc_average** (Optional[str]) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the average parameter). Should be one of the following:

  - None [default]: Should be left unchanged if your data is not multi-dimensional multi-class.

  - 'samplewise': In this case, the statistics are computed separately for each sample on the N axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... (see *Input types*) as the N dimension within the sample, and computing the metric for the sample based on that.

  - 'global': In this case the N and ... dimensions of the inputs (see *Input types*) are flattened into a new N_X sample axis, i.e. the inputs are treated as if they were (N_X, C). From here on the average parameter applies as usual.

- **num_classes** (Optional[int]) – Number of classes. Necessary for 'macro', 'weighted' and None average methods.

- **threshold** (float) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.

- **top_k** (Optional[int]) – Number of highest probability or logit score predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (None) will be interpreted as 1 for these inputs.

  Should be left at default (None) for all other types of inputs.

- **multiclass** (Optional[bool]) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.

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

> **Returns** Tensor containing AUC score (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)

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.

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

> **Parameters**
>
> - **preds**⫽ (`Tensor`) – predictions from model (logits or probabilities)
>
> - **target**⫽ (`Tensor`) – Ground truth labels
>
> - **num_classes**⫽ (`Optional[int]`) – integer with number of classes for multi-label and multiclass problems. Should be set to `None` for binary problems
>
> - **pos_label**⫽ (`Optional[int]`) – integer determining the positive class. Default is `None` which for binary problem is translate to 1. For multiclass problems this argument should not be set as we iteratively change it in the range [0,num_classes-1]
>
> - **average**⫽ (`Optional[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*, *average='macro'*, *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]

- **average**⫽ (Optional[str]) – defines the reduction that is applied in the case of multiclass and multilabel input. Should be one of the following:

– `'macro'` [default]: Calculate the metric for each class separately, and average the metrics
across classes (with equal weights for each class).

– `'micro'`: Calculate the metric globally, across all samples and classes. Cannot be used
with multiclass input.

– `'weighted'`: Calculate the metric for each class separately, and average the metrics across
classes, weighting each class by its support.

– `'none'` or `None`: Calculate the metric for each class separately, and return the metric for
every class.

- **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, average=None)
[tensor(1.), tensor(1.), tensor(0.2500), tensor(0.2500), tensor(nan)]
```

calibration_error [func]

`torchmetrics.functional.`**`calibration_error`**(*preds*, *target*, *n_bins=15*, *norm='l1'*)

Computes the Top-label Calibration Error

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

L1 norm (Expected Calibration Error)

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

Infinity norm (Maximum Calibration Error)

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

L2 norm (Root Mean Square Calibration Error)

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

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

Parameters

- **preds**⏴ (Tensor) – Model output probabilities.

- **target**⏴ (Tensor) – Ground-truth target class labels.

- **n_bins**⏴ (int) – Number of bins to use when computing t. Defaults to 15.

- **norm**⏴ (str) – Norm used to compare empirical and expected probability bins. Defaults to "l1", or Expected Calibration Error.

Return type Tensor

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.

Args:

    preds: (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: target (long tensor), tensor with shape (N, ...) with ground true labels

    num_classes: Number of classes in the dataset.

    weights: Weighting type to calculate the score. Choose from

- None or 'none': no weighting

- 'linear': linear weighting

- 'quadratic': quadratic weighting

    threshold: Threshold value for binary or multi-label probabilities.

Example:

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

Return type Tensor

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 or logits from a model output or integer class values in prediction. Works with multi-dimensional preds and target, but it should be noted that additional dimensions will be flattened.

If preds and target are the same shape and preds is a float tensor, we use the `self.threshold` argument to convert into integer labels. This is the case for binary and multi-label probabilities or logits.

If preds has an extra dimension as in the case of multi-class scores we perform an argmax on `dim=1`.

If working with multilabel data, setting the *is_multilabel* argument to *True* will make sure that a confusion matrix gets calculated per label.

> **Parameters**
>
> - **preds**⫽ (`Tensor`) – (float or long tensor), Either a (`N, ...`) tensor with labels or (`N, C, ...`) where C is the number of classes, tensor with labels/logits/probabilities
>
> - **target**⫽ (`Tensor`) – `target` (long tensor), tensor with shape (`N, ...`) with ground true labels
>
> - **num_classes**⫽ (`int`) – Number of classes in the dataset.
>
> - **normalize**⫽ (`Optional`[`str`]) – Normalization mode for confusion matrix. Choose from
>
>   – `None` or `'none'`: no normalization (default)
>
>   – `'true'`: normalization over the targets (most commonly used)
>
>   – `'pred'`: normalization over the predictions
>
>   – `'all'`: normalization over the whole matrix
>
> - **threshold**⫽ (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
>
> - **multilabel**⫽ (`bool`) – determines if data is multilabel or not.

Example (binary data):

```
>>> from torchmetrics import ConfusionMatrix
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0, 1, 0, 0])
>>> confmat = ConfusionMatrix(num_classes=2)
>>> confmat(preds, target)
tensor([[2, 0],
        [1, 1]])
```

Example (multiclass data):

```
>>> target = torch.tensor([2, 1, 0, 0])
>>> preds = torch.tensor([2, 1, 0, 1])
>>> confmat = ConfusionMatrix(num_classes=3)
>>> confmat(preds, target)
tensor([[1, 1, 0],
        [0, 1, 0],
        [0, 0, 1]])
```

Example (multilabel data):

```
>>> target = torch.tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = torch.tensor([[0, 0, 1], [1, 0, 1]])
>>> confmat = ConfusionMatrix(num_classes=3, multilabel=True)
>>> confmat(preds, target)
tensor([[[1, 0], [0, 1]],
        [[1, 0], [1, 0]],
        [[0, 1], [0, 1]]])
```

> **Return type** Tensor

dice_score [func]

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

Compute dice score from prediction scores.

> **Parameters**
>
> - **preds**¶ (Tensor) – estimated probabilities
>
> - **target**¶ (Tensor) – ground-truth labels
>
> - **bg**¶ (bool) – whether to also compute dice for the background
>
> - **nan_score**¶ (float) – score to return, if a NaN occurs during computation
>
> - **no_fg_score**¶ (float) – 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_score [func]

torchmetrics.functional.**f1_score**(*preds*, *target*, *beta=1.0*, *average='micro'*, *mdmc_average=None*,
                                    ignore_index=None, *num_classes=None*, *threshold=0.5*, *top_k=None*,
                                    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 or logits from a model output or integer class values in prediction. Works with multi-dimensional preds and target.

If preds and target are the same shape and preds is a float tensor, we use the `self.threshold` argument to convert into integer labels. This is the case for binary and multi-label probabilities or logits.

If preds has an extra dimension as in the case of multi-class scores we perform an argmax on `dim=1`.

The reduction method (how the precision scores are aggregated) is controlled by the `average` parameter, and additionally by the `mdmc_average` parameter in the multi-dimensional multi-class case. Accepts all inputs listed in *Input types*.

> **Parameters**
>
> - **preds**⫽ (`Tensor`) – Predictions from model (probabilities, logits or labels)
>
> - **target**⫽ (`Tensor`) – Ground truth values
>
> - **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`.
>
>   ---
>
>   ---
>
>   **Note:** If `'none'` and a given class doesn't occur in the *preds* or *target*, the value for the class will be `nan`.
>
>   ---
>
> - **mdmc_average**⫽ (`Optional[str]`) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
>
>   - `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
>
>   - `'samplewise'`: In this case, the statistics are computed separately for each sample on the `N` axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes `...` (see *Input types*) as the `N` dimension within the sample, and computing the metric for the sample based on that.

– 'global': In this case the `N` and `...` dimensions of the inputs (see *Input types*) are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were (`N_X`, `C`). From here on the `average` parameter applies as usual.

- **ignore_index** (`Optional[int]`) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or `'none'`, the score for the ignored class will be returned as `nan`.

- **num_classes** (`Optional[int]`) – Number of classes. Necessary for `'macro'`, `'weighted'` and `None` average methods.

- **threshold** (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.

- **top_k** (`Optional[int]`) – Number of highest probability or logit score predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (`None`) will be interpreted as 1 for these inputs.

  Should be left at default (`None`) for all other types of inputs.

- **multiclass** (`Optional[bool]`) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.

Return type `Tensor`

Returns

The shape of the returned tensor depends on the `average` parameter

- If `average in ['micro', 'macro', 'weighted', 'samples']`, a one-element tensor will be returned

- If `average in ['none', None]`, the shape will be (`C,`), where `C` stands for the number of classes

Example

```
>>> from torchmetrics.functional import f1_score
>>> target = torch.tensor([0, 1, 2, 0, 1, 2])
>>> preds = torch.tensor([0, 2, 1, 0, 0, 1])
>>> f1_score(preds, target, num_classes=3)
tensor(0.3333)
```

fbeta_score [func]

torchmetrics.functional.**fbeta_score**(*preds*, *target*, *beta=1.0*, *average='micro'*, *mdmc_average=None*, *ignore_index=None*, *num_classes=None*, *threshold=0.5*, *top_k=None*, *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 or logits from a model output or integer class values in prediction. Works with multi-dimensional preds and target.

If preds and target are the same shape and preds is a float tensor, we use the `self.threshold` argument to convert into integer labels. This is the case for binary and multi-label logits or probabilities.

If preds has an extra dimension as in the case of multi-class scores we perform an argmax on `dim=1`.

The reduction method (how the precision scores are aggregated) is controlled by the `average` parameter, and additionally by the `mdmc_average` parameter in the multi-dimensional multi-class case. Accepts all inputs listed in *Input types*.

> **Parameters**
>
> - **preds**⫝ (Tensor) – Predictions from model (probabilities, logits or labels)
>
> - **target**⫝ (Tensor) – Ground truth values
>
> - **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`.
>
>   ---
>
>   ---
>
>   **Note:**   If `'none'` and a given class doesn't occur in the *preds* or *target*, the value for the class will be `nan`.
>
>   ---
>
> - **mdmc_average**⫝ (Optional[str]) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
>
>   - None [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
>
>   - `'samplewise'`: In this case, the statistics are computed separately for each sample on the N axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes `...` (see *Input types*) as the N dimension within the sample, and computing the metric for the sample based on that.
>
>   - `'global'`: In this case the N and `...` dimensions of the inputs (see *Input types*) are flattened into a new N_X sample axis, i.e. the inputs are treated as if they were (N_X, C). From here on the `average` parameter applies as usual.
>
> - **ignore_index**⫝ (Optional[int]) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or `'none'`, the score for the ignored class will be returned as `nan`.
>
> - **num_classes**⫝ (Optional[int]) – Number of classes.   Necessary for `'macro'`, `'weighted'` and None average methods.

- **threshold**⫚ (float) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.

- **top_k**⫚ (Optional[int]) – Number of highest probability or logit score predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (None) will be interpreted as 1 for these inputs.

  Should be left at default (None) for all other types of inputs.

- **multiclass**⫚ (Optional[bool]) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.

Return type Tensor

Returns

The shape of the returned tensor depends on the average parameter

- If average in ['micro', 'macro', 'weighted', 'samples'], a one-element tensor will be returned

- If average in ['none', None], the shape will be (C,), where C stands for the number of classes

Example

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

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 (probabilities, logits or labels)

- **target**⫚ (Tensor) – Ground truth

- **threshold**⫚ (float) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.

Example

```
>>> from torchmetrics.functional import hamming_distance
>>> target = torch.tensor([[0, 1], [1, 1]])
>>> preds = torch.tensor([[0, 1], [0, 1]])
>>> hamming_distance(preds, target)
tensor(0.2500)
```

> **Return type** Tensor

hinge_loss [func]

torchmetrics.functional.**hinge_loss**(*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 $y \in 0, ..., \mathrm{C}$ 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**
   >
   > - **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_loss
>>> target = torch.tensor([0, 1, 1])
>>> preds = torch.tensor([-2.2, 2.4, 0.1])
>>> hinge_loss(preds, target)
tensor(0.3000)
```

Example (default / multiclass case):

```
>>> target = torch.tensor([0, 1, 2])
>>> preds = torch.tensor([[-1.0, 0.9, 0.2], [0.5, -1.1, 0.8], [2.2, -0.5, 0.3]])
>>> hinge_loss(preds, target)
tensor(2.9000)
```

Example (multiclass example, one vs all mode):

```
>>> target = torch.tensor([0, 1, 2])
>>> preds = torch.tensor([[-1.0, 0.9, 0.2], [0.5, -1.1, 0.8], [2.2, -0.5, 0.3]])
>>> hinge_loss(preds, target, multiclass_mode="one-vs-all")
tensor([2.2333, 1.5000, 1.2333])
```

> **Return type** Tensor

jaccard_index [func]

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

Computes Jaccard index

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

Where: A and B are both tensors of the same size, containing integer class values. They may be subject to conversion from input data (see description below).

Note that it is different from box IoU.

If preds and target are the same shape and preds is a float tensor, we use the `self.threshold` argument to convert into integer labels. This is the case for binary and multi-label probabilities.

If pred has an extra dimension as in the case of multi-class scores we perform an argmax on `dim=1`.

> **Parameters**
>
> - **preds**⫽ (Tensor) – tensor containing predictions from model (probabilities, or labels) with shape [N, d1, d2, ...]
> - **target**⫽ (Tensor) – tensor containing ground truth labels with shape [N, d1, d2, ...]
> - **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.

- **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 jaccard_index
>>> target = torch.randint(0, 2, (10, 25, 25))
>>> pred = torch.tensor(target)
>>> pred[2:5, 7:13, 9:15] = 1 - pred[2:5, 7:13, 9:15]
>>> jaccard_index(pred, target)
tensor(0.9660)
```

kl_divergence [func]

torchmetrics.functional.**kl_divergence**(*p*, *q*, *log_prob=False*, *reduction='mean'*)
    Computes KL divergence

$$D_{KL}(P||Q) = \sum_{x \in \mathcal{X}} P(x) \log \frac{P(x)}{Qx}$$

Where P and Q are probability distributions where P usually represents a distribution over data and Q is often a prior or approximation of P. It should be noted that the KL divergence is a non-symetrical metric i.e. $D_{KL}(P||Q) \neq D_{KL}(Q||P)$.

Parameters

- **p**⫟ (Tensor) – data distribution with shape [N, d]

- **q**⫟ (Tensor) – prior or approximate distribution with shape [N, d]

- **log_prob**⫟ (bool) – bool indicating if input is log-probabilities or probabilities. If given as probabilities, will normalize to make sure the distributes sum to 1

- **reduction**⫟ (Optional[str]) – Determines how to reduce over the N/batch dimension:

  - 'mean' [default]: Averages score across samples

- **'sum'**: Sum score across samples

- **'none'** or None: Returns score per sample

Example

```
>>> import torch
>>> p = torch.tensor([[0.36, 0.48, 0.16]])
>>> q = torch.tensor([[1/3, 1/3, 1/3]])
>>> kl_divergence(p, q)
tensor(0.0853)
```

> **Return type** Tensor

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.

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.

Note: If either the positive class or negative class is completly missing in the target tensor, the roc values are not well defined in this case and a tensor of zeros will be returned (either fpr or tpr depending on what class is missing) together with an warning.

> **Parameters**
> - **preds**⫮ (Tensor) – predictions from model (logits or probabilities)
> - **target**⫮ (Tensor) – ground truth values
> - **num_classes**⫮ (Optional[int]) – integer with number of classes for multi-label and multiclass problems. Should be set to None for binary problems
> - **pos_label**⫮ (Optional[int]) – integer determining the positive class. Default is None which for binary problem is translate to 1. For multiclass problems this argument should not be set as we iteratively change it in the range [0,num_classes-1]
> - **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)

Computes Precision

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

Where TP and FP represent the number of true positives and false positives respectively. With the use of `top_k` parameter, this metric can generalize to Precision@K.

The reduction method (how the precision scores are aggregated) is controlled by the `average` parameter, and additionally by the `mdmc_average` parameter in the multi-dimensional multi-class case. Accepts all inputs listed in *Input types*.

Parameters

- **preds**⫽ (Tensor) – Predictions from model (probabilities, logits or labels)

- **target**⫽ (Tensor) – Ground truth values

- **average**⫽ (str) – Defines the reduction that is applied. Should be one of the following:

  - 'micro' [default]: Calculate the metric globally, across all samples and classes.

  - 'macro': Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).

  - 'weighted': Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (tp + fn).

  - 'none' or None: Calculate the metric for each class separately, and return the metric for every class.

  - 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

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

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

- **mdmc_average**⫽ (Optional[str]) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the average parameter). Should be one of the following:

  - None [default]: Should be left unchanged if your data is not multi-dimensional multi-class.

  - 'samplewise': In this case, the statistics are computed separately for each sample on the N axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... (see *Input types*) as the N dimension within the sample, and computing the metric for the sample based on that.

  - 'global': In this case the N and ... dimensions of the inputs (see *Input types*) are flattened into a new N_X sample axis, i.e. the inputs are treated as if they were (N_X, C). From here on the average parameter applies as usual.

- **ignore_index**⫽ (Optional[int]) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and average=None or 'none', the score for the ignored class will be returned as nan.

- **num_classes**⫽ (Optional[int]) – Number of classes. Necessary for 'macro', 'weighted' and None average methods.

- **threshold**⫽ (float) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.

- **top_k**⫽ (Optional[int]) – Number of highest probability or logit score predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (None) will be interpreted as 1 for these inputs.

  Should be left at default (None) for all other types of inputs.

- **multiclass**¶ (Optional[bool]) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.

Return type `Tensor`

Returns

The shape of the returned tensor depends on the `average` parameter

- If `average in ['micro', 'macro', 'weighted', 'samples']`, a one-element tensor will be returned

- If `average in ['none', None]`, the shape will be `(C,)`, where C stands for the number of classes

Raises

- `ValueError` – If `average` is not one of `"micro"`, `"macro"`, `"weighted"`, `"samples"`, `"none"` or `None`.

- `ValueError` – If `mdmc_average` is not one of `None`, `"samplewise"`, `"global"`.

- `ValueError` – If `average` is set but `num_classes` is not provided.

- `ValueError` – If `num_classes` is set and `ignore_index` is not in the range `[0, num_classes)`.

Example

```
>>> from torchmetrics.functional import precision
>>> preds  = torch.tensor([2, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> precision(preds, target, average='macro', num_classes=3)
tensor(0.1667)
>>> precision(preds, target, average='micro')
tensor(0.2500)
```

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)`

Computes Precision

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

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

Where TPtext{FN}`` and FP represent the number of true positives, false negatives and false positives respicitively. With the use of `top_k` parameter, this metric can generalize to Recall@K and Precision@K.

The reduction method (how the recall scores are aggregated) is controlled by the `average` parameter, and additionally by the `mdmc_average` parameter in the multi-dimensional multi-class case. Accepts all inputs listed in *Input types*.

Parameters

- **preds**⫽ (Tensor) – Predictions from model (probabilities, logits or labels)

- **target**⫽ (Tensor) – Ground truth values

- **average**⫽ (str) – Defines the reduction that is applied. Should be one of the following:

  - 'micro' [default]: Calculate the metric globally, across all samples and classes.

  - 'macro': Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).

  - 'weighted': Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (tp + fn).

  - 'none' or None: Calculate the metric for each class separately, and return the metric for every class.

  - 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

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

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

- **mdmc_average**⫽ (Optional[str]) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the average parameter). Should be one of the following:

  - None [default]: Should be left unchanged if your data is not multi-dimensional multi-class.

  - 'samplewise': In this case, the statistics are computed separately for each sample on the N axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... (see *Input types*) as the N dimension within the sample, and computing the metric for the sample based on that.

  - 'global': In this case the N and ... dimensions of the inputs (see *Input types*) are flattened into a new N_X sample axis, i.e. the inputs are treated as if they were (N_X, C). From here on the average parameter applies as usual.

- **ignore_index**⫽ (Optional[int]) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and average=None or 'none', the score for the ignored class will be returned as nan.

- **num_classes**⫽ (Optional[int]) – Number of classes. Necessary for 'macro', 'weighted' and None average methods.

- **threshold**⫽ (float) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.

- **top_k**⫽ (Optional[int]) – Number of highest probability or logit score predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (None) will be interpreted as 1 for these inputs.

  Should be left at default (None) for all other types of inputs.

- **multiclass**⫽ (Optional[bool]) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.

Returns

precision and recall. Their shape depends on the `average` parameter

- If `average in ['micro', 'macro', 'weighted', 'samples']`, they are a single element tensor

- If `average in ['none', None]`, they are a tensor of shape `(C, )`, where `C` stands for the number of classes

Return type The function returns a tuple with two elements

Raises

- **ValueError** – If `average` is not one of `"micro"`, `"macro"`, `"weighted"`, `"samples"`, `"none"` or `None`.

- **ValueError** – If `mdmc_average` is not one of `None`, `"samplewise"`, `"global"`.

- **ValueError** – If `average` is set but `num_classes` is not provided.

- **ValueError** – If `num_classes` is set and `ignore_index` is not in the range `[0, num_classes)`.

Example

```
>>> from torchmetrics.functional import precision_recall
>>> preds  = torch.tensor([2, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> precision_recall(preds, target, average='macro', num_classes=3)
(tensor(0.1667), tensor(0.3333))
>>> precision_recall(preds, target, average='micro')
(tensor(0.2500), tensor(0.2500))
```

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 for multi-label and multiclass problems. Should be set to `None` for binary problems

- **pos_label**⫽ (Optional[int]) – integer determining the positive class. Default is `None` which for binary problem is translate to 1. For multiclass problems this argument should not be set as we iteratively change it in the range [0,num_classes-1]

- **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*)

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, logits 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`.
>
> ---
>
> ---
>
> **Note:** If `'none'` and a given class doesn't occur in the *preds* or *target*, the value for the class will be `nan`.
>
> ---
>
> - **mdmc_average**‖ (Optional[str]) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
>
>   - None [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
>
>   - `'samplewise'`: In this case, the statistics are computed separately for each sample on the N axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes `...` (see *Input types*) as the N dimension within the sample, and computing the metric for the sample based on that.
>
>   - `'global'`: In this case the N and `...` dimensions of the inputs (see *Input types*) are flattened into a new N_X sample axis, i.e. the inputs are treated as if they were (N_X, C). From here on the `average` parameter applies as usual.

- **ignore_index** (Optional[int]) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and average=None or 'none', the score for the ignored class will be returned as nan.

- **num_classes** (Optional[int]) – Number of classes. Necessary for 'macro', 'weighted' and None average methods.

- **threshold** (float) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.

- **top_k** (Optional[int]) – Number of highest probability or logit score predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (None) will be interpreted as 1 for these inputs.

  Should be left at default (None) for all other types of inputs.

- **multiclass** (Optional[bool]) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.

Return type Tensor

Returns

The shape of the returned tensor depends on the average parameter

- If average in ['micro', 'macro', 'weighted', 'samples'], a one-element tensor will be returned

- If average in ['none', None], the shape will be (C,), where C stands for the number of classes

Raises

- **ValueError** – If average is not one of "micro", "macro", "weighted", "samples", "none" or None.

- **ValueError** – If mdmc_average is not one of None, "samplewise", "global".

- **ValueError** – If average is set but num_classes is not provided.

- **ValueError** – If num_classes is set and ignore_index is not in the range [0, num_classes).

Example

```
>>> from torchmetrics.functional import 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)
```

specificity [func]

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

> Computes Specificity
>
> $$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$
>
> Where TN and FP represent the number of true negatives and false positives respectively. With the use of top_k parameter, this metric can generalize to Specificity@K.
>
> The reduction method (how the specificity scores are aggregated) is controlled by the average parameter, and additionally by the mdmc_average parameter in the multi-dimensional multi-class case. Accepts all inputs listed in *Input types*.
>
> > **Parameters**
> >
> > - **preds**⫽ (Tensor) – Predictions from model (probabilities, or labels)
> >
> > - **target**⫽ (Tensor) – Ground truth values
> >
> > - **average**⫽ (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 (tn + fp).
> >
> >   - 'none' or None: Calculate the metric for each class separately, and return the metric for every class.

– `'samples'`: Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

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

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

- **mdmc_average** *(Optional[str])* – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:

  – `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class.

  – `'samplewise'`: In this case, the statistics are computed separately for each sample on the `N` axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes `...` (see *Input types*) as the `N` dimension within the sample, and computing the metric for the sample based on that.

  – `'global'`: In this case the `N` and `...` dimensions of the inputs (see *Input types*) are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were `(N_X, C)`. From here on the `average` parameter applies as usual.

- **ignore_index** *(Optional[int])* – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or `'none'`, the score for the ignored class will be returned as `nan`.

- **num_classes** *(Optional[int])* – Number of classes. Necessary for `'macro'`, `'weighted'` and `None` average methods.

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

- **top_k** *(Optional[int])* – Number of highest probability entries for each sample to convert to 1s - relevant only for inputs with probability predictions. If this parameter is set for multi-label inputs, it will take precedence over `threshold`. For (multi-dim) multi-class inputs, this parameter defaults to 1.

  Should be left unset (`None`) for inputs with label predictions.

- **multiclass** *(Optional[bool])* – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.

Return type `Tensor`

Returns

The shape of the returned tensor depends on the `average` parameter

- If `average in ['micro', 'macro', 'weighted', 'samples']`, a one-element tensor will be returned

- If `average in ['none', None]`, the shape will be `(C,)`, where C stands for the number of classes

Raises

- **ValueError** – If average is not one of "micro", "macro", "weighted", "samples", "none" or None.

- **ValueError** – If mdmc_average is not one of None, "samplewise", "global".

- **ValueError** – If average is set but num_classes is not provided.

- **ValueError** – If num_classes is set and ignore_index is not in the range [0, num_classes).

Example

```
>>> from torchmetrics.functional import specificity
>>> preds  = torch.tensor([2, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> specificity(preds, target, average='macro', num_classes=3)
tensor(0.6111)
>>> specificity(preds, target, average='micro')
tensor(0.6250)
```

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*)

Computes the number of true positives, false positives, true negatives, false negatives. Related to Type I and Type II errors and the confusion matrix.

The reduction method (how the statistics are aggregated) is controlled by the reduce parameter, and additionally by the mdmc_reduce parameter in the multi-dimensional multi-class case. Accepts all inputs listed in *Input types*.

> **Parameters**
>
> - **preds**⫻ (Tensor) – Predictions from model (probabilities, logits or labels)
>
> - **target**⫻ (Tensor) – Ground truth values
>
> - **threshold**⫻ (float) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
>
> - **top_k**⫻ (Optional[int]) – Number of highest probability or logit score predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (None) will be interpreted as 1 for these inputs.
>
>   Should be left at default (None) for all other types of inputs.
>
> - **reduce**⫻ (str) – Defines the reduction that is applied. Should be one of the following:
>
>   - 'micro' [default]: Counts the statistics by summing over all [sample, class] combinations (globally). Each statistic is represented by a single integer.
>
>   - 'macro': Counts the statistics for each class separately (over all samples). Each statistic is represented by a (C,) tensor. Requires num_classes to be set.
>
>   - 'samples': Counts the statistics for each sample separately (over all classes). Each statistic is represented by a (N,) 1d tensor.

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

- **num_classes**⫽ (`Optional[int]`) – Number of classes. Necessary for (multi-dimensional) multi-class or multi-label data.

- **ignore_index**⫽ (`Optional[int]`) – Specify a class (label) to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `reduce='macro'`, the class statistics for the ignored class will all be returned as `-1`.

- **mdmc_reduce**⫽ (`Optional[str]`) – Defines how the multi-dimensional multi-class inputs are 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**(*x*, *argmax_dim=1*)

Converts a tensor of probabilities to a dense label tensor.

Parameters

- **x**⫿ (Tensor) – probabilities to get the categorical label [N, d1, d2, …]

- **argmax_dim**⫿ (int) – dimension to apply

Return type Tensor

Returns A tensor with categorical labels [N, d2, …]

Example

```
>>> x = torch.tensor([[0.2, 0.5], [0.9, 0.1]])
>>> to_categorical(x)
tensor([1, 0])
```

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]])
```

2.6.3 Image

image_gradients [func]

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

Computes Gradient Computation of Image of a given image using finite difference.

> **Parameters img** (Tensor) – An (N, C, H, W) input tensor where C is the number of image
> channels
>
> **Return type** Tuple[Tensor, Tensor]
>
> **Returns** Tuple of (dy, dx) with each gradient of shape [N, C, H, W]
>
> **Raises**
>
> > - **TypeError** – If img is not of the type <torch.Tensor>.
> >
> > - **RuntimeError** – If img is not a 4D tensor.

> **Example**

```
>>> from torchmetrics.functional import image_gradients
>>> image = torch.arange(0, 1*1*5*5, dtype=torch.float32)
>>> image = torch.reshape(image, (1, 1, 5, 5))
>>> dy, dx = image_gradients(image)
>>> dy[0, 0, :, :]
tensor([[5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [0., 0., 0., 0., 0.]])
```

Note: The implementation follows the 1-step finite difference method as followed by the TF implementation. The values are organized such that the gradient of [I(x+1, y)-[I(x, y)]] are at the (x, y) location

---

structural_similarity_index_measure [func]

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

Computes Structural 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.
>>
>> • **k2**⫽ (float) – Parameter of SSIM.
>
> **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 structural_similarity_index_measure
>>> preds = torch.rand([16, 1, 16, 16])
>>> target = preds * 0.75
>>> structural_similarity_index_measure(preds, target)
tensor(0.9219)
```

multiscale_structural_similarity_index_measure [func]

torchmetrics.functional.**multiscale_structural_similarity_index_measure**(*preds*, *target*, *kernel_size=(11, 11)*, *sigma=(1.5, 1.5)*, *reduction='elementwise_mean'*, *data_range=None*, *k1=0.01*, *k2=0.03*, *betas=(0.0448, 0.2856, 0.3001, 0.2363, 0.1333)*, *normalize=None*)

Computes MultiScaleSSIM, Multi-scale Structual Similarity Index Measure, which is a generalization of Structual Similarity Index Measure by incorporating image details at different resolution scores.

> **Parameters**
>
> - **preds**⫲ (Tensor) – Predictions from model of shape *[N, C, H, W]*
>
> - **target**⫲ (Tensor) – Ground truth values of shape *[N, C, H, W]*
>
> - **kernel_size**⫲ (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 structural similarity index measure.
>
> - **k2**⫲ (float) – Parameter of structural similarity index measure.
>
> - **betas**⫲ (Tuple[float, ...]) – Exponent parameters for individual similarities and contrastive sensitivies returned by different image resolutions.
>
> - **normalize**⫲ (Optional[Literal['relu', 'simple']]) – When MultiScaleSSIM loss is used for training, it is desirable to use normalizes to improve the training stability. This *normalize* argument is out of scope of the original implementation [1], and it is adapted from https://github.com/jorge-pessoa/pytorch-msssim instead.
>
> **Return type** Tensor
>
> **Returns** Tensor with Multi-Scale SSIM score
>
> **Raises**

- **TypeError** – If `preds` and `target` don't have the same data type.

- **ValueError** – If `preds` and `target` don't have `BxCxHxW` shape.

- **ValueError** – If the length of `kernel_size` or `sigma` is not 2.

- **ValueError** – If one of the elements of `kernel_size` is not an `odd positive number`.

- **ValueError** – If one of the elements of `sigma` is not a `positive number`.

Example

```
>>> from torchmetrics.functional import multiscale_structural_similarity_index_
↪measure
>>> preds = torch.rand([1, 1, 256, 256], generator=torch.manual_seed(42))
>>> target = preds * 0.75
>>> multiscale_structural_similarity_index_measure(preds, target)
tensor(0.9558)
```

References: [1] Multi-Scale Structural Similarity For Image Quality Assessment by Zhou Wang, Eero P. Simoncelli and Alan C. Bovik MultiScaleSSIM

peak_signal_noise_ratio [func]

torchmetrics.functional.**peak_signal_noise_ratio**(*preds*, *target*, *data_range=None*, *base=10.0*, *reduction='elementwise_mean'*, *dim=None*)

Computes the peak signal-to-noise ratio.

Parameters

- **preds**⫝ (Tensor) – estimated signal

- **target**⫝ (Tensor) – 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 peak_signal_noise_ratio
>>> pred = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
>>> target = torch.tensor([[3.0, 2.0], [1.0, 0.0]])
>>> peak_signal_noise_ratio(pred, target)
tensor(2.5527)
```

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

2.6.4 Regression

cosine_similarity [func]

torchmetrics.functional.**cosine_similarity**(*preds*, *target*, *reduction='sum'*)

Computes the Cosine Similarity between targets and predictions:

$$cos_{sim}(x, y) = \frac{x \cdot y}{||x|| \cdot ||y||} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2} \sqrt{\sum_{i=1}^{n} y_i^2}}$$

where y is a tensor of target values, and x is a tensor of predictions.

Parameters

- **preds**¶ (Tensor) – Predicted tensor with shape (N,d)

- **target**¶ (Tensor) – Ground truth tensor with shape (N,d)

- **reduction**¶ (str) – The method of reducing along the batch dimension using sum, mean or taking the individual scores

Example

```
>>> from torchmetrics.functional.regression import cosine_similarity
>>> target = torch.tensor([[1, 2, 3, 4],
...                        [1, 2, 3, 4]])
>>> preds = torch.tensor([[1, 2, 3, 4],
...                       [-1, -2, -3, -4]])
>>> cosine_similarity(preds, target, 'none')
tensor([ 1.0000, -1.0000])
```

Return type Tensor

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]]

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_absolute_percentage_error [func]

torchmetrics.functional.**mean_absolute_percentage_error**(*preds*, *target*)
    Computes mean absolute percentage error.

> **Parameters**
>
> - **preds**⫫ (Tensor) – estimated labels
>
> - **target**⫫ (Tensor) – ground truth labels
>
> **Return type** Tensor
>
> **Returns** Tensor with MAPE

Note: The epsilon value is taken from scikit-learn's implementation of MAPE.

Example

```
>>> from torchmetrics.functional import mean_absolute_percentage_error
>>> target = torch.tensor([1, 10, 1e6])
>>> preds = torch.tensor([0.9, 15, 1.2e6])
>>> mean_absolute_percentage_error(preds, target)
tensor(0.2667)
```

mean_squared_error [func]

torchmetrics.functional.**mean_squared_error**(*preds*, *target*, *squared=True*)
    Computes mean squared error.

> **Parameters**
>
> - **preds**⫫ (Tensor) – estimated labels
>
> - **target**⫫ (Tensor) – ground truth labels
>
> - **squared**⫫ (bool) – returns RMSE value if set to False
>
> **Return type** Tensor
>
> **Returns** Tensor with MSE

Example

```
>>> from torchmetrics.functional import mean_squared_error
>>> x = torch.tensor([0., 1, 2, 3])
>>> y = torch.tensor([0., 1, 2, 2])
>>> mean_squared_error(x, y)
tensor(0.2500)
```

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

r2_score [func]

torchmetrics.functional.**r2_score**(*preds*, *target*, *adjusted=0*, *multioutput='uniform_average'*)
    Computes r2 score also known as R2 Score_Coefficient Determination:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

where $SS_{res} = \sum_i (y_i - f(x_i))^2$ is the sum of residual squares, and $SS_{tot} = \sum_i (y_i - \bar{y})^2$ is total sum of squares. Can also calculate adjusted r2 score given by

$$R_{adj}^2 = 1 - \frac{(1 - R^2)(n - 1)}{n - k - 1}$$

where the parameter k (the number of independent regressors) should be provided as the `adjusted` argument.

> **Parameters**
>
> - **preds**¶ (Tensor) – estimated labels
>
> - **target**¶ (Tensor) – ground truth labels
>
> - **adjusted**¶ (int) – number of independent regressors for calculating adjusted r2 score. 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 r2_score
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> r2_score(preds, target)
tensor(0.9486)
```

```
>>> target = torch.tensor([[0.5, 1], [-1, 1], [7, -6]])
>>> preds = torch.tensor([[0, 2], [-1, 2], [8, -5]])
>>> r2_score(preds, target, multioutput='raw_values')
tensor([0.9654, 0.9082])
```

> **Return type** Tensor

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

symmetric_mean_absolute_percentage_error [func]

torchmetrics.functional.**symmetric_mean_absolute_percentage_error**(*preds*, *target*)
Computes symmetric mean absolute percentage error (SMAPE):

$$\text{SMAPE} = \frac{2}{n} \sum_{1}^{n} \frac{max(|y_i - \hat{y}_i|}{|y_i| + |\hat{y}_i|, \epsilon)}$$

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

> **Parameters**
>
> - **preds**¶ (Tensor) – estimated labels

- **target** *⫻* (Tensor) – ground truth labels

Return type Tensor

Returns Tensor with SMAPE.

Example

```
>>> from torchmetrics.functional import symmetric_mean_absolute_percentage_error
>>> target = torch.tensor([1, 10, 1e6])
>>> preds = torch.tensor([0.9, 15, 1.2e6])
>>> symmetric_mean_absolute_percentage_error(preds, target)
tensor(0.2290)
```

tweedie_deviance_score [func]

torchmetrics.functional.**tweedie_deviance_score**(*preds*, *targets*, *power=0.0*)

Computes the Tweedie Deviance Score between targets and predictions:

$$deviance\_score(\hat{y}, y) = \begin{cases} (\hat{y} - y)^2, & \text{for } power = 0 \\ 2 * (y * log(\frac{y}{\hat{y}}) + \hat{y} - y), & \text{for } power = 1 \\ 2 * (log(\frac{\hat{y}}{y}) + \frac{y}{\hat{y}} - 1), & \text{for } power = 2 \\ 2 * (\frac{(max(y,0))^2}{(1-power)(2-power)} - \frac{y(\hat{y})^{1-power}}{1-power} + \frac{(\hat{y})^{2-power}}{2-power}), & \text{otherwise} \end{cases}$$

where y is a tensor of targets values, and $\hat{y}$ is a tensor of predictions.

Parameters

- **preds** *⫻* (Tensor) – Predicted tensor with shape (N,...)

- **targets** *⫻* (Tensor) – Ground truth tensor with shape (N,...)

- **power** *⫻* (float) –

  - power < 0 : Extreme stable distribution. (Requires: preds > 0.)

  - power = 0 : Normal distribution. (Requires: targets and preds can be any real numbers.)

  - power = 1 : Poisson distribution. (Requires: targets >= 0 and y_pred > 0.)

  - 1 < p < 2 : Compound Poisson distribution. (Requires: targets >= 0 and preds > 0.)

  - power = 2 : Gamma distribution. (Requires: targets > 0 and preds > 0.)

  - power = 3 : Inverse Gaussian distribution. (Requires: targets > 0 and preds > 0.)

  - otherwise : Positive stable distribution. (Requires: targets > 0 and preds > 0.)

Example

```
>>> from torchmetrics.functional import tweedie_deviance_score
>>> targets = torch.tensor([1.0, 2.0, 3.0, 4.0])
>>> preds = torch.tensor([4.0, 3.0, 2.0, 1.0])
>>> tweedie_deviance_score(preds, targets, power=2)
tensor(1.2083)
```

> **Return type** Tensor

2.6.5 Pairwise Metrics

pairwise_cosine_similarity [func]

torchmetrics.functional.**pairwise_cosine_similarity**(*x*, *y=None*, *reduction=None*, *zero_diagonal=None*)

Calculates pairwise cosine similarity:

$$s_{cos}(x, y) = \frac{<x, y>}{||x|| \cdot ||y||} = \frac{\sum_{d=1}^{D} x_d \cdot y_d}{\sqrt{\sum_{d=1}^{D} x_i^2} \cdot \sqrt{\sum_{d=1}^{D} x_i^2}}$$

If both *x* and *y* are passed in, the calculation will be performed pairwise between the rows of *x* and *y*. If only *x* is passed in, the calculation will be performed between the rows of *x*.

> **Parameters**
>
> - **x** ¶ (Tensor) – Tensor with shape [N, d]
> - **y** ¶ (Optional[Tensor]) – Tensor with shape [M, d], optional
> - **reduction** ¶ (Optional[str]) – reduction to apply along the last dimension. Choose between *'mean'*, *'sum'* (applied along column dimension) or *'none'*, *None* for no reduction
> - **zero_diagonal** ¶ (Optional[bool]) – if the diagonal of the distance matrix should be set to 0. If only *x* is given this defaults to *True* else if *y* is also given it defaults to *False*
>
> **Return type** Tensor
>
> **Returns** A [N,N] matrix of distances if only x is given, else a [N,M] matrix

Example

```
>>> import torch
>>> from torchmetrics.functional import pairwise_cosine_similarity
>>> x = torch.tensor([[2, 3], [3, 5], [5, 8]], dtype=torch.float32)
>>> y = torch.tensor([[1, 0], [2, 1]], dtype=torch.float32)
>>> pairwise_cosine_similarity(x, y)
tensor([[0.5547, 0.8682],
        [0.5145, 0.8437],
        [0.5300, 0.8533]])
>>> pairwise_cosine_similarity(x)
tensor([[0.0000, 0.9989, 0.9996],
        [0.9989, 0.0000, 0.9998],
        [0.9996, 0.9998, 0.0000]])
```

pairwise_euclidean_distance [func]

torchmetrics.functional.**pairwise_euclidean_distance**(*x*, *y=None*, *reduction=None*,
zero_diagonal=None)

Calculates pairwise euclidean distances:

$$d_{euc}(x, y) = ||x - y||_2 = \sqrt{\sum_{d=1}^{D}(x_d - y_d)^2}$$

If both *x* and *y* are passed in, the calculation will be performed pairwise between the rows of *x* and *y*. If only *x* is passed in, the calculation will be performed between the rows of *x*.

> **Parameters**
>
> - **x**¶ (Tensor) – Tensor with shape [N, d]
>
> - **y**¶ (Optional[Tensor]) – Tensor with shape [M, d], optional
>
> - **reduction**¶ (Optional[str]) – reduction to apply along the last dimension. Choose between *'mean'*, *'sum'* (applied along column dimension) or *'none'*, *None* for no reduction
>
> - **zero_diagonal**¶ (Optional[bool]) – if the diagonal of the distance matrix should be set to 0. If only *x* is given this defaults to *True* else if *y* is also given it defaults to *False*
>
> **Return type** Tensor
>
> **Returns** A [N,N] matrix of distances if only x is given, else a [N,M] matrix

Example

```
>>> import torch
>>> from torchmetrics.functional import pairwise_euclidean_distance
>>> x = torch.tensor([[2, 3], [3, 5], [5, 8]], dtype=torch.float32)
>>> y = torch.tensor([[1, 0], [2, 1]], dtype=torch.float32)
>>> pairwise_euclidean_distance(x, y)
tensor([[3.1623, 2.0000],
        [5.3852, 4.1231],
        [8.9443, 7.6158]])
>>> pairwise_euclidean_distance(x)
tensor([[0.0000, 2.2361, 5.8310],
        [2.2361, 0.0000, 3.6056],
        [5.8310, 3.6056, 0.0000]])
```

pairwise_linear_similarity [func]

torchmetrics.functional.**pairwise_linear_similarity**(*x*, *y=None*, *reduction=None*,
zero_diagonal=None)

Calculates pairwise linear similarity:

$$s_{lin}(x, y) = <x, y> = \sum_{d=1}^{D} x_d \cdot y_d$$

If both *x* and *y* are passed in, the calculation will be performed pairwise between the rows of *x* and *y*. If only *x* is passed in, the calculation will be performed between the rows of *x*.

Parameters

- **x**⏎ (Tensor) – Tensor with shape [N, d]

- **y**⏎ (Optional[Tensor]) – Tensor with shape [M, d], optional

- **reduction**⏎ (Optional[str]) – reduction to apply along the last dimension. Choose between *'mean'*, *'sum'* (applied along column dimension) or *'none'*, *None* for no reduction

- **zero_diagonal**⏎ (Optional[bool]) – if the diagonal of the distance matrix should be set to 0. If only *x* is given this defaults to *True* else if *y* is also given it defaults to *False*

Return type Tensor

Returns A [N,N] matrix of distances if only x is given, else a [N,M] matrix

Example

```
>>> import torch
>>> from torchmetrics.functional import pairwise_linear_similarity
>>> x = torch.tensor([[2, 3], [3, 5], [5, 8]], dtype=torch.float32)
>>> y = torch.tensor([[1, 0], [2, 1]], dtype=torch.float32)
>>> pairwise_linear_similarity(x, y)
tensor([[ 2.,  7.],
        [ 3., 11.],
        [ 5., 18.]])
>>> pairwise_linear_similarity(x)
tensor([[ 0., 21., 34.],
        [21.,  0., 55.],
        [34., 55.,  0.]])
```

pairwise_manhattan_distance [func]

torchmetrics.functional.**pairwise_manhattan_distance**(*x*, *y=None*, *reduction=None*, *zero_diagonal=None*)

Calculates pairwise manhattan distance:

$$d_{man}(x, y) = ||x - y||_1 = \sum_{d=1}^{D} |x_d - y_d|$$

If both *x* and *y* are passed in, the calculation will be performed pairwise between the rows of *x* and *y*. If only *x* is passed in, the calculation will be performed between the rows of *x*.

Parameters

- **x**⏎ (Tensor) – Tensor with shape [N, d]

- **y**⏎ (Optional[Tensor]) – Tensor with shape [M, d], optional

- **reduction**⏎ (Optional[str]) – reduction to apply along the last dimension. Choose between *'mean'*, *'sum'* (applied along column dimension) or *'none'*, *None* for no reduction

- **zero_diagonal**⏎ (Optional[bool]) – if the diagonal of the distance matrix should be set to 0. If only *x* is given this defaults to *True* else if *y* is also given it defaults to *False*

Return type Tensor

Returns A [N,N] matrix of distances if only x is given, else a [N,M] matrix

Example

```
>>> import torch
>>> from torchmetrics.functional import pairwise_manhattan_distance
>>> x = torch.tensor([[2, 3], [3, 5], [5, 8]], dtype=torch.float32)
>>> y = torch.tensor([[1, 0], [2, 1]], dtype=torch.float32)
>>> pairwise_manhattan_distance(x, y)
tensor([[ 4.,  2.],
        [ 7.,  5.],
        [12., 10.]])
>>> pairwise_manhattan_distance(x)
tensor([[0., 3., 8.],
        [3., 0., 5.],
        [8., 5., 0.]])
```

2.6.6 Retrieval

retrieval_average_precision [func]

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

   Computes average precision (for information retrieval), as explained in IR Average precision.

   preds and target should be of the same shape and live on the same device. If no target is True, 0 is returned. target must be either *bool* or *integers* and preds must be *float*, otherwise an error is raised.

   Parameters

   - **preds**⫲ (Tensor) – estimated probabilities of each document to be relevant.

   - **target**⫲ (Tensor) – ground truth about each document being relevant or not.

   Return type Tensor

   Returns a single-value tensor with the average precision (AP) of the predictions preds w.r.t. the labels target.

Example

```
>>> from torchmetrics.functional import retrieval_average_precision
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([True, False, True])
>>> retrieval_average_precision(preds, target)
tensor(0.8333)
```

retrieval_reciprocal_rank [func]

torchmetrics.functional.**retrieval_reciprocal_rank**(*preds*, *target*)
>   Computes reciprocal rank (for information retrieval). See Mean Reciprocal Rank
>
>   `preds` and `target` should be of the same shape and live on the same device. If no `target` is `True`, 0 is returned. `target` must be either *bool* or *integers* and `preds` must be *float*, otherwise an error is raised.
>
> > **Parameters**
> >
> > - **preds**/ (Tensor) – estimated probabilities of each document to be relevant.
> >
> > - **target**/ (Tensor) – ground truth about each document being relevant or not.
> >
> > **Return type** Tensor
> >
> > **Returns** a single-value tensor with the reciprocal rank (RR) of the predictions `preds` wrt the labels `target`.

Example

```
>>> from torchmetrics.functional import retrieval_reciprocal_rank
>>> preds = torch.tensor([0.2, 0.3, 0.5])
>>> target = torch.tensor([False, True, False])
>>> retrieval_reciprocal_rank(preds, target)
tensor(0.5000)
```

retrieval_precision [func]

torchmetrics.functional.**retrieval_precision**(*preds*, *target*, *k=None*)
>   Computes the precision metric (for information retrieval). Precision is the fraction of relevant documents among all the retrieved documents.
>
>   `preds` and `target` should be of the same shape and live on the same device. If no `target` is `True`, `0` is returned. `target` must be either *bool* or *integers* and `preds` must be *float*, otherwise an error is raised. If you want to measure Precision@K, `k` must be a positive integer.
>
> > **Parameters**
> >
> > - **preds**/ (Tensor) – estimated probabilities of each document to be relevant.
> >
> > - **target**/ (Tensor) – ground truth about each document being relevant or not.
> >
> > - **k**/ (Optional[int]) – consider only the top k elements (default: *None*, which considers them all)
> >
> > **Return type** Tensor
> >
> > **Returns** a single-value tensor with the precision (at k) of the predictions `preds` w.r.t. the labels `target`.
> >
> > **Raises** **ValueError** – If k parameter is not *None* or an integer larger than 0

Example

```
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([True, False, True])
>>> retrieval_precision(preds, target, k=2)
tensor(0.5000)
```

retrieval_r_precision [func]

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

Computes the r-precision metric (for information retrieval). R-Precision is the fraction of relevant documents among all the top k retrieved documents where k is equal to the total number of relevant documents.

preds and target should be of the same shape and live on the same device. If no target is True, 0 is returned. target must be either *bool* or *integers* and preds must be *float*, otherwise an error is raised. If you want to measure Precision@K, k must be a positive integer.

> **Parameters**
>
> - **preds**⫽ (Tensor) – estimated probabilities of each document to be relevant.
>
> - **target**⫽ (Tensor) – ground truth about each document being relevant or not.
>
> **Return type** Tensor
>
> **Returns** a single-value tensor with the r-precision of the predictions preds w.r.t. the labels target.

Example

```
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([True, False, True])
>>> retrieval_r_precision(preds, target)
tensor(0.5000)
```

retrieval_recall [func]

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

Computes the recall metric (for information retrieval). Recall is the fraction of relevant documents retrieved among all the relevant documents.

preds and target should be of the same shape and live on the same device. If no target is True, 0 is returned. target must be either *bool* or *integers* and preds must be *float*, otherwise an error is raised. If you want to measure Recall@K, k must be a positive integer.

> **Parameters**
>
> - **preds**⫽ (Tensor) – estimated probabilities of each document to be relevant.
>
> - **target**⫽ (Tensor) – ground truth about each document being relevant or not.
>
> - **k**⫽ (Optional[int]) – consider only the top k elements (default: *None*, which considers them all)
>
> **Return type** Tensor
>
> **Returns** a single-value tensor with the recall (at k) of the predictions preds w.r.t. the labels target.

Raises `ValueError` – If k parameter is not *None* or an integer larger than 0

Example

```
>>> from  torchmetrics.functional import retrieval_recall
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([True, False, True])
>>> retrieval_recall(preds, target, k=2)
tensor(0.5000)
```

retrieval_fall_out [func]

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

Computes the Fall-out (for information retrieval), as explained in IR Fall-out Fall-out is the fraction of non-relevant documents retrieved among all the non-relevant documents.

`preds` and `target` should be of the same shape and live on the same device. If no `target` is `True`, `0` is returned. `target` must be either *bool* or *integers* and `preds` must be *float*, otherwise an error is raised. If you want to measure Fall-out@K, k must be a positive integer.

Parameters

- **preds**⫫ (Tensor) – estimated probabilities of each document to be relevant.

- **target**⫫ (Tensor) – ground truth about each document being relevant or not.

- **k**⫫ (Optional[int]) – consider only the top k elements (default: *None*, which considers them all)

Return type Tensor

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

Raises `ValueError` – If k parameter is not *None* or an integer larger than 0

Example

```
>>> from  torchmetrics.functional import retrieval_fall_out
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([True, False, True])
>>> retrieval_fall_out(preds, target, k=2)
tensor(1.)
```

retrieval_normalized_dcg [func]

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

Computes Normalized Discounted Cumulative Gain (for information retrieval).

`preds` and `target` should be of the same shape and live on the same device. `target` must be either *bool* or *integers* and `preds` must be *float*, otherwise an error is raised.

Parameters

- **preds**⫫ (Tensor) – estimated probabilities of each document to be relevant.

- **target** (Tensor) – ground truth about each document relevance.
- **k** (Optional[int]) – consider only the top k elements (default: *None*, which considers them all)

Return type Tensor

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

Raises **ValueError** – If k parameter is not *None* or an integer larger than 0

Example

```
>>> from torchmetrics.functional import retrieval_normalized_dcg
>>> preds = torch.tensor([.1, .2, .3, 4, 70])
>>> target = torch.tensor([10, 0, 0, 1, 5])
>>> retrieval_normalized_dcg(preds, target)
tensor(0.6957)
```

retrieval_hit_rate [func]

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

Computes the hit rate (for information retrieval). The hit rate is 1.0 if there is at least one relevant document among all the top *k* retrieved documents.

`preds` and `target` should be of the same shape and live on the same device. If no `target` is True, `0` is returned. `target` must be either *bool* or *integers* and `preds` must be *float*, otherwise an error is raised. If you want to measure HitRate@K, **k** must be a positive integer.

Parameters

- **preds** (Tensor) – estimated probabilities of each document to be relevant.
- **target** (Tensor) – ground truth about each document being relevant or not.
- **k** (Optional[int]) – consider only the top k elements (default: *None*, which considers them all)

Return type Tensor

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

Raises **ValueError** – If k parameter is not *None* or an integer larger than 0

Example

```
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([True, False, True])
>>> retrieval_hit_rate(preds, target, k=2)
tensor(1.)
```

2.6.7 Text

bert_score [func]

torchmetrics.functional.text.bert.**bert_score**(*preds*, *target*, *model_name_or_path=None*,
num_layers=None, *all_layers=False*, *model=None*,
user_tokenizer=None, *user_forward_fn=None*,
verbose=False, *idf=False*, *device=None*,
max_length=512, *batch_size=64*, *num_threads=4*,
return_hash=False, *lang='en'*,
rescale_with_baseline=False, *baseline_path=None*,
baseline_url=None)

Bert_score Evaluating Text Generation leverages the pre-trained contextual embeddings from BERT and matches words in candidate and reference sentences by cosine similarity. It has been shown to correlate with human judgment on sentence-level and system-level evaluation. Moreover, BERTScore computes precision, recall, and F1 measure, which can be useful for evaluating different language generation tasks.

This implemenation follows the original implementation from BERT_score

> **Parameters**
>
> - **preds**¶ (Union[List[str], Dict[str, Tensor]]) – Either an iterable of predicted sentences or a *Dict[str, torch.Tensor]* containing *input_ids* and *attention_mask torch.Tensor*.
>
> - **target**¶ (Union[List[str], Dict[str, Tensor]]) – Either an iterable of target sentences or a *Dict[str, torch.Tensor]* containing *input_ids* and *attention_mask torch.Tensor*.
>
> - **model_name_or_path**¶ (Optional[str]) – A name or a model path used to load *transformers* pretrained model.
>
> - **num_layers**¶ (Optional[int]) – A layer of representation to use.
>
> - **all_layers**¶ (bool) – An indication of whether the representation from all model's layers should be used. If *all_layers = True*, the argument *num_layers* is ignored.
>
> - **model**¶ (Optional[Module]) – A user's own model. Must be of *torch.nn.Module* instance.
>
> - **user_tokenizer**¶ (Optional[Any]) – A user's own tokenizer used with the own model. This must be an instance with the *__call__* method. This method must take an iterable of sentences (*List[str]*) and must return a python dictionary containing *"input_ids"* and *"attention_mask"* represented by *torch.Tensor*. It is up to the user's model of whether *"input_ids"* is a *torch.Tensor* of input ids or embedding vectors. This tokenizer must prepend an equivalent of *[CLS]* token and append an equivalent of *[SEP]* token as *transformers* tokenizer does.
>
> - **user_forward_fn**¶ (Optional[Callable[[Module, Dict[str, Tensor]], Tensor]]) – A user's own forward function used in a combination with *user_model*. This function must take *user_model* and a python dictionary of containing *"input_ids"* and *"attention_mask"* represented by *torch.Tensor* as an input and return the model's output represented by the single *torch.Tensor*.
>
> - **verbose**¶ (bool) – An indication of whether a progress bar to be displayed during the embeddings calculation.
>
> - **idf**¶ (bool) – An indication of whether normalization using inverse document frequencies should be used.
>
> - **device**¶ (Union[str, device, None]) – A device to be used for calculation.
>
> - **max_length**¶ (int) – A maximum length of input sequences. Sequences longer than *max_length* are to be trimmed.

- **batch_size** (int) – A batch size used for model processing.

- **num_threads** (int) – A number of threads to use for a dataloader.

- **return_hash** (bool) – An indication of whether the correspodning *hash_code* should be returned.

- **lang** (str) – A language of input sentences. It is used when the scores are rescaled with a baseline.

- **rescale_with_baseline** (bool) – An indication of whether bertscore should be rescaled with a pre-computed baseline. When a pretrained model from *transformers* model is used, the corresponding baseline is downloaded from the original *bert-score* package from BERT_score if available. In other cases, please specify a path to the baseline csv/tsv file, which must follow the formatting of the files from BERT_score

- **baseline_path** (Optional[str]) – A path to the user's own local csv/tsv file with the baseline scale.

- **baseline_url** (Optional[str]) – A url path to the user's own csv/tsv file with the baseline scale.

Return type Dict[str, Union[List[float], str]]

Returns Python dictionary containing the keys *precision*, *recall* and *f1* with corresponding values.

Deprecated since version v0.7: Args: predictions: This argument is deprecated in favor of *preds* and will be removed in v0.8. references: This argument is deprecated in favor of *target* and will be removed in v0.8.

Raises

- **ValueError** – If *len(preds) != len(target)*.

- **ModuleNotFoundError** – If *tqdm* package is required and not installed.

- **ModuleNotFoundError** – If transformers package is required and not installed.

- **ValueError** – If num_layer is larger than the number of the model layers.

- **ValueError** – If invalid input is provided.

Example

```
>>> from torchmetrics.functional.text.bert import bert_score
>>> preds = ["hello there", "general kenobi"]
>>> target = ["hello there", "master kenobi"]
>>> from pprint import pprint
>>> pprint(bert_score(preds, target))
{'f1': [0.999..., 0.996...],
 'precision': [0.999..., 0.996...],
 'recall': [0.999..., 0.996...]}
```

bleu_score [func]

torchmetrics.functional.**bleu_score**(*preds*, *target*, *n_gram=4*, *smooth=False*)

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

> **Parameters**
>
> - **preds**∥ (Union[str, Sequence[str]]) – An iterable of machine translated corpus
> - **target**∥ (Sequence[Union[str, Sequence[str]]]) – An iterable of iterables of reference corpus
> - **n_gram**∥ (int) – Gram value ranged from 1 to 4 (Default 4)
> - **smooth**∥ (bool) – Whether or not to apply smoothing – see [2]
>
> **Return type** Tensor
>
> **Returns** Tensor with BLEU Score

Deprecated since version v0.7: Args: translate_corpus: This argument is deprecated in favor of *preds* and will be removed in v0.8. reference_corpus: This argument is deprecated in favor of *target* and will be removed in v0.8.

Example

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

References

[1] BLEU: a Method for Automatic Evaluation of Machine Translation by Papineni, Kishore, Salim Roukos, Todd Ward, and Wei-Jing Zhu BLEU

[2] Automatic Evaluation of Machine Translation Quality Using Longest Common Subsequence and Skip-Bigram Statistics by Chin-Yew Lin and Franz Josef Och Machine Translation Evolution

char_error_rate [func]

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

character error rate is a common metric of the performance of an automatic speech recognition system. This value indicates the percentage of characters that were incorrectly predicted. The lower the value, the better the performance of the ASR system with a CER of 0 being a perfect score. :type _sphinx_paramlinks_torchmetrics.functional.char_error_rate.preds: Union[str, List[str]] :param _sphinx_paramlinks_torchmetrics.functional.char_error_rate.preds: Transcription(s) to score as a string or list of strings :type _sphinx_paramlinks_torchmetrics.functional.char_error_rate.target: Union[str, List[str]] :param _sphinx_paramlinks_torchmetrics.functional.char_error_rate.target: Reference(s) for each speech input as a string or list of strings

> **Return type** Tensor
>
> **Returns** Character error rate score

Deprecated since version v0.7: Args: predictions: This argument is deprecated in favor of *preds* and will be removed in v0.8. references: This argument is deprecated in favor of *target* and will be removed in v0.8.

Examples

```
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> char_error_rate(preds=preds, target=target)
tensor(0.3415)
```

chrf_score [func]

torchmetrics.functional.**chrf_score**(*preds*, *target*, *n_char_order=6*, *n_word_order=2*, *beta=2.0*, *lowercase=False*, *whitespace=False*, *return_sentence_level_score=False*)

Calculate chrF score of machine translated text with one or more references. This implementation supports both chrF score computation introduced in [1] and chrF++ score introduced in chrF++ score. This implementation follows the implmenetaions from https://github.com/m-popovic/chrF and https://github.com/mjpost/sacrebleu/blob/master/sacrebleu/metrics/chrf.py.

Parameters

- **preds** (Union[str, Sequence[str]]) – An iterable of hypothesis corpus.

- **target** (Sequence[Union[str, Sequence[str]]]) – An iterable of iterables of reference corpus.

- **n_char_order** (int) – A character n-gram order. If *n_char_order=6*, the metrics refers to the official chrF/chrF++.

- **n_word_order** (int) – A word n-gram order. If *n_word_order=2*, the metric refers to the official chrF++. If *n_word_order=0*, the metric is equivalent to the original chrF.

- **beta** (float) – A parameter determining an importance of recall w.r.t. precision. If *beta=1*, their importance is equal.

- **lowercase** (bool) – An indication whether to enable case-insesitivity.

- **whitespace** (bool) – An indication whether to keep whitespaces during character n-gram extraction.

- **return_sentence_level_score** (bool) – An indication whether a sentence-level chrF/chrF++ score to be returned.

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

Returns A corpus-level chrF/chrF++ score. (Optionally) A list of sentence-level chrF/chrF++ scores if *return_sentence_level_score=True*.

Raises

- **ValueError** – If n_char_order is not an integer greater than or equal to 1.

- **ValueError** – If n_word_order is not an integer greater than or equal to 0.

- **ValueError** – If beta is smaller than 0.

Example

```
>>> from torchmetrics.functional import chrf_score
>>> preds = ['the cat is on the mat']
>>> target = [['there is a cat on the mat', 'a cat is on the mat']]
>>> chrf_score(preds, target)
tensor(0.8640)
```

References

[1] chrF: character n-gram F-score for automatic MT evaluation by Maja Popović chrF score [2] chrF++: words helping character n-grams by Maja Popović chrF++ score

extended_edit_distance [func]

torchmetrics.functional.**extended_edit_distance**(*preds*, *target*, *language='en'*, *return_sentence_level_score=False*, *alpha=2.0*, *rho=0.3*, *deletion=0.2*, *insertion=1.0*)

Computes extended edit distance score (ExtendedEditDistance) [1] for strings or list of strings. The metric utilises the Levenshtein distance and extends it by adding an additional jump operation.

> **Parameters**
>
> - **preds** (Union[str, Sequence[str]]) – An iterable of hypothesis corpus.
>
> - **target** (Sequence[Union[str, Sequence[str]]]) – An iterable of iterables of reference corpus.
>
> - **language** (Literal['en', 'ja']) – Language used in sentences. Only supports English (en) and Japanese (ja) for now. Defaults to en
>
> - **return_sentence_level_score** (bool) – An indication of whether sentence-level EED score is to be returned.
>
> - **alpha** (float) – optimal jump penalty, penalty for jumps between characters
>
> - **rho** (float) – coverage cost, penalty for repetition of characters
>
> - **deletion** (float) – penalty for deletion of character
>
> - **insertion** (float) – penalty for insertion or substitution of character
>
> **Return type** Union[Tensor, Tuple[Tensor, Tensor]]
>
> **Returns** Extended edit distance score as a tensor

Example

```
>>> from torchmetrics.functional import extended_edit_distance
>>> preds = ["this is the prediction", "here is an other sample"]
>>> target = ["this is the reference", "here is another one"]
>>> extended_edit_distance(preds=preds, target=target)
tensor(0.3078)
```

References

[1] P. Stanchev, W. Wang, and H. Ney, "EED: Extended Edit Distance Measure for Machine Translation", submitted to WMT 2019. ExtendedEditDistance

match_error_rate [func]

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

Match error rate is a metric of the performance of an automatic speech recognition system. This value indicates the percentage of words that were incorrectly predicted and inserted. The lower the value, the better the performance of the ASR system with a MatchErrorRate of 0 being a perfect score.

> **Parameters**
>
> - **preds**⫽ (Union[str, List[str]]) – Transcription(s) to score as a string or list of strings
>
> - **target**⫽ (Union[str, List[str]]) – Reference(s) for each speech input as a string or list of strings
>
> **Return type** Tensor
>
> **Returns** Match error rate score

Examples

```
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> match_error_rate(preds=preds, target=target)
tensor(0.4444)
```

rouge_score [func]

torchmetrics.functional.text.rouge.**rouge_score**(*preds*, *target*, *accumulate='best'*, *use_stemmer=False*,
rouge_keys=('rouge1', 'rouge2', 'rougeL', 'rougeLsum'))

Calculate Calculate Rouge Score , used for automatic summarization.

> **Parameters**
>
> - **preds**⫽ (Union[str, Sequence[str]]) – An iterable of predicted sentences or a single predicted sentence.
>
> - **target**⫽ (Union[str, Sequence[str], Sequence[Sequence[str]]]) – An iterable of iterables of target sentences or an iterable of target sentences or a single target sentence.
>
> - **accumulate**⫽ (Literal['avg', 'best']) – Useful incase of multi-reference rouge score. - avg takes the avg of all references with respect to predictions - best takes the best fmeasure score obtained between prediction and multiple corresponding references.
>
> - **use_stemmer**⫽ (bool) – Use Porter stemmer to strip word suffixes to improve matching.
>
> - **rouge_keys**⫽ (Union[str, Tuple[str, ...]]) – A list of rouge types to calculate. Keys that are allowed are rougeL, rougeLsum, and rouge1 through rouge9.
>
> **Return type** Dict[str, Tensor]
>
> **Returns** Python dictionary of rouge scores for each input rouge key.

Example

```
>>> from torchmetrics.functional.text.rouge import rouge_score
>>> preds = "My name is John"
>>> target = "Is your name John"
>>> from pprint import pprint
>>> pprint(rouge_score(preds, target))
{'rouge1_fmeasure': tensor(0.7500),
 'rouge1_precision': tensor(0.7500),
 'rouge1_recall': tensor(0.7500),
 'rouge2_fmeasure': tensor(0.),
 'rouge2_precision': tensor(0.),
 'rouge2_recall': tensor(0.),
 'rougeL_fmeasure': tensor(0.5000),
 'rougeL_precision': tensor(0.5000),
 'rougeL_recall': tensor(0.5000),
 'rougeLsum_fmeasure': tensor(0.5000),
 'rougeLsum_precision': tensor(0.5000),
 'rougeLsum_recall': tensor(0.5000)}
```

Raises

- **ModuleNotFoundError** – If the python package `nltk` is not installed.

- **ValueError** – If any of the `rouge_keys` does not belong to the allowed set of keys.

References

[1] ROUGE: A Package for Automatic Evaluation of Summaries by Chin-Yew Lin. https://aclanthology.org/W04-1013/

sacre_bleu_score [func]

torchmetrics.functional.**sacre_bleu_score**(*preds*, *target*, *n_gram=4*, *smooth=False*, *tokenize='13a'*, *lowercase=False*)

Calculate BLEU score [1] of machine translated text with one or more references. This implementation follows the behaviour of SacreBLEU [2] implementation from https://github.com/mjpost/sacrebleu.

Parameters

- **preds** (Sequence[str]) – An iterable of machine translated corpus

- **target** (Sequence[Sequence[str]]) – An iterable of iterables of reference corpus

- **n_gram** (int) – Gram value ranged from 1 to 4 (Default 4)

- **smooth** (bool) – Whether or not to apply smoothing – see [2]

- **tokenize** (Literal['none', '13a', 'zh', 'intl', 'char']) – Tokenization technique to be used. (Default '13a') Supported tokenization: ['none', '13a', 'zh', 'intl', 'char']

- **lowercase** (bool) – If `True`, BLEU score over lowercased text is calculated.

Return type Tensor

Returns Tensor with BLEU Score

Deprecated since version v0.7: Args: translate_corpus: This argument is deprecated in favor of *preds* and will be removed in v0.8. reference_corpus: This argument is deprecated in favor of *target* and will be removed in v0.8.

Example

```
>>> from torchmetrics.functional import sacre_bleu_score
>>> preds = ['the cat is on the mat']
>>> target = [['there is a cat on the mat', 'a cat is on the mat']]
>>> sacre_bleu_score(preds, target)
tensor(0.7598)
```

References

[1] BLEU: a Method for Automatic Evaluation of Machine Translation by Papineni, Kishore, Salim Roukos, Todd Ward, and Wei-Jing Zhu BLEU

[2] A Call for Clarity in Reporting BLEU Scores by Matt Post.

[3] Automatic Evaluation of Machine Translation Quality Using Longest Common Subsequence and Skip-Bigram Statistics by Chin-Yew Lin and Franz Josef Och Machine Translation Evolution

squad [func]

`torchmetrics.functional.`**`squad`**(*preds*, *target*)

Calculate SQuAD Metric .

> **Parameters**
>
> - **preds**¶ (Union[Dict[str, str], List[Dict[str, str]]]) – A Dictionary or List of Dictionary-s that map *id* and *prediction_text* to the respective values.
>
>   Example prediction:
>
>   ```
>   {"prediction_text": "TorchMetrics is awesome", "id": "123"}
>   ```
>
> - **target**¶ (Union[Dict[str, Union[str, Dict[str, Union[List[str], List[int]]]]], List[Dict[str, Union[str, Dict[str, Union[List[str], List[int]]]]]]]) – A Dictionary or List of Dictionary-s that contain the *answers* and *id* in the SQuAD Format.
>
>   Example target:
>
>   ```
>   {
>       'answers': [{'answer_start': [1], 'text': ['This is a test answer
>   ↪']}],
>       'id': '1',
>   }
>   ```
>
>   Reference SQuAD Format:
>
>   ```
>   {
>       'answers': {'answer_start': [1], 'text': ['This is a test text']}
>   ↪,
>       'context': 'This is a test context.',
>   ```
>
> (continues on next page)

```
        'id': '1',
        'question': 'Is this a test?',
        'title': 'train test'
}
```

Return type Dict[str, Tensor]

Returns Dictionary containing the F1 score, Exact match score for the batch.

Example

```
>>> from torchmetrics.functional.text.squad import squad
>>> preds = [{"prediction_text": "1976", "id": "56e10a3be3433e1400422b22"}]
>>> target = [{"answers": {"answer_start": [97], "text": ["1976"]},"id":
→"56e10a3be3433e1400422b22"}]
>>> squad(preds, target)
{'exact_match': tensor(100.), 'f1': tensor(100.)}
```

Raises KeyError – If the required keys are missing in either predictions or targets.

References

[1] SQuAD: 100,000+ Questions for Machine Comprehension of Text by Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, Percy Liang SQuAD Metric .

translation_edit_rate [func]

torchmetrics.functional.**translation_edit_rate**(*preds*, *target*, *normalize=False*, *no_punctuation=False*, *lowercase=True*, *asian_support=False*, *return_sentence_level_score=False*)

Calculate Translation edit rate (TER) of machine translated text with one or more references. This implementation follows the implmenetaions from https://github.com/mjpost/sacrebleu/blob/master/sacrebleu/metrics/ter.py. The *sacrebleu* implmenetation is a near-exact reimplementation of the Tercom algorithm, produces identical results on all "sane" outputs.

Parameters

- **preds**⫫ (Union[str, Sequence[str]]) – An iterable of hypothesis corpus.

- **target**⫫ (Sequence[Union[str, Sequence[str]]]) – An iterable of iterables of reference corpus.

- **normalize**⫫ (bool) – An indication whether a general tokenization to be applied.

- **no_punctuation**⫫ (bool) – An indication whteher a punctuation to be removed from the sentences.

- **lowercase**⫫ (bool) – An indication whether to enable case-insesitivity.

- **asian_support**⫫ (bool) – An indication whether asian characters to be processed.

- **return_sentence_level_score**⫫ (bool) – An indication whether a sentence-level TER to be returned.

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

Returns A corpus-level translation edit rate (TER). (Optionally) A list of sentence-level translation_edit_rate (TER) if *return_sentence_level_score=True*.

Example

```
>>> preds = ['the cat is on the mat']
>>> target = [['there is a cat on the mat', 'a cat is on the mat']]
>>> translation_edit_rate(preds, target)
tensor(0.1538)
```

References: [1] A Study of Translation Edit Rate with Targeted Human Annotation by Mathew Snover, Bonnie Dorr, Richard Schwartz, Linnea Micciulla and John Makhoul TER

wer [func]

`torchmetrics.functional.`**`wer`**`(`*predictions*`, `*references*`)`

Word error rate (WER) is a common metric of the performance of an automatic speech recognition system.

Deprecated since version v0.7: Use `torchmetrics.fuctional.word_error_rate()`. Will be removed in v0.8.

Examples

```
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> wer(preds=preds, target=target)
tensor(0.5000)
```

> **Return type** Tensor

word_information_lost [func]

`torchmetrics.functional.`**`word_information_lost`**`(`*preds*`, `*target*`)`

Word Information Lost rate is a metric of the performance of an automatic speech recognition system. This value indicates the percentage of characters that were incorrectly predicted. The lower the value, the better the performance of the ASR system with a Word Information Lost rate of 0 being a perfect score.

> **Parameters**
>
> - **preds**� (Union[str, List[str]]) – Transcription(s) to score as a string or list of strings
>
> - **target**⍟ (Union[str, List[str]]) – Reference(s) for each speech input as a string or list of strings
>
> **Return type** Tensor
>
> **Returns** Word Information Lost rate

Examples

```
>>> from torchmetrics.functional import word_information_lost
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> word_information_lost(preds, target)
tensor(0.6528)
```

word_information_preserved [func]

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

Word Information Preserved rate is a metric of the performance of an automatic speech recognition system. This value indicates the percentage of characters that were incorrectly predicted. The lower the value, the better the performance of the ASR system with a Word Information preserved rate of 0 being a perfect score.

> **Parameters**
>
> - **preds**⁋ (Union[str, List[str]]) – Transcription(s) to score as a string or list of strings
>
> - **total**⁋ – Reference(s) for each speech input as a string or list of strings
>
> **Return type** Tensor
>
> **Returns** Word Information preserved rate

Examples

```
>>> from torchmetrics.functional import word_information_preserved
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> word_information_preserved(preds, target)
tensor(0.3472)
```

2.7 TorchMetrics Governance

This document describes governance processes we follow in developing TorchMetrics.

2.7.1 Persons of Interest

Leads

- Nicki Skafte (skaftenicki)
- Jirka Borovec (Borda)
- Justus Schock (justusschock)

Core Maintainers

- Luca Di Liello (lucadiliello)
- Daniel Stancl (stancld)
- Maxim Grechkin (maximsch2)
- Changsheng Quan (quancs)

Alumni

- Ananya Harsh Jha (ananyahjha93)
- Teddy Koker (teddykoker)

2.7.2 Releases

We release a new minor version (e.g., 0.5.0) every few months and bugfix releases if needed. The minor versions contain new features, API changes, deprecations, removals, potential backward-incompatible changes and also all previous bugfixes included in any bugfix release. With every release, we publish a changelog where we list additions, removals, changed functionality and fixes.

2.7.3 Project Management and Decision Making

The decision what goes into a release is governed by the *staff contributors and leaders* of Lightning development. Whenever possible, discussion happens publicly on GitHub and includes the whole community. When a consensus is reached, staff and core contributors assign milestones and labels to the issue and/or pull request and start tracking the development. It is possible that priorities change over time.

Commits to the project are exclusively to be added by pull requests on GitHub and anyone in the community is welcome to review them. However, reviews submitted by code owners have higher weight and it is necessary to get the approval of code owners before a pull request can be merged. Additional requirements may apply case by case.

2.7.4 API Evolution

Torchmetrics development is driven by research and best practices in a rapidly developing field of AI and machine learning. Change is inevitable and when it happens, the Torchmetric team is committed to minimizing user friction and maximizing ease of transition from one version to the next. We take backward compatibility and reproducibility very seriously.

For API removal, renaming or other forms of backward-incompatible changes, the procedure is:

1. A deprecation process is initiated at version X, producing warning messages at runtime and in the documentation.
2. Calls to the deprecated API remain unchanged in their function during the deprecation phase.
3. One minor versions in the future at version X+1 the breaking change takes effect.

The "X+1" rule is a recommendation and not a strict requirement. Longer deprecation cycles may apply for some cases.

2.8 Contributor Covenant Code of Conduct

2.8.1 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.

2.8.2 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

2.8.3 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.

2.8.4 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.

2.8.5 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.

2.8.6 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

2.9 Contributing

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

2.9.1 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.

2.9.2 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

```python
from typing import Optional


def my_func(param_a: int, param_b: Optional[float] = None) -> str:
    """Sample function.

    Args:
        param_a: first parameter
        param_b: second parameter

    Return:
        sum of both numbers

    Example:
        Sample doctest example...
        >>> my_func(1, 2)
        3

    .. note:: If you want to add something.
    """
    p = param_b if param_b else 0
    return str(param_a + p)
```

When updating the docs make sure to build them first locally and visually inspect the html files (in the browser) for formatting errors. In certain cases, a missing blank line or a wrong indent can lead to a broken layout. Run these commands

```
make docs
```

and open `docs/build/html/index.html` in your browser.

Notes:

- You need to have LaTeX installed for rendering math equations. You can for example install TeXLive by doing one of the following:

    - on Ubuntu (Linux) run `apt-get install texlive` or otherwise follow the instructions on the TeXLive website

    - use the RTD docker image

- with PL used class meta you need to use python 3.7 or higher

When you send a PR the continuous integration will run tests and build the docs.

Testing

Local: Testing your work locally will help you speed up the process since it allows you to focus on particular (failing) test-cases. To setup a local development environment, install both local and test dependencies:

```
python -m pip install -r requirements/test.txt
python -m pip install pre-commit
```

You can run the full test-case in your terminal via this make script:

```
make test
# or natively
python -m pytest torchmetrics tests
```

Note: if your computer does not have multi-GPU nor TPU these tests are skipped.

GitHub Actions: For convenience, you can also use your own GHActions building which will be triggered with each commit. This is useful if you do not test against all required dependency versions.

2.10 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

Note: we move fast, but still we preserve 0.1 version (one feature release) back compatibility.

2.10.1 [0.7.1] - 2022-02-03

[0.7.1] - Changed

- Used `torch.bucketize` in calibration error when `torch>1.8` for faster computations (#769)
- Improve mAP performance (#742)

[0.7.1] - Fixed

- Fixed check for available modules (#772)
- Fixed Matthews correlation coefficient when the denominator is 0 (#781)

2.10.2 [0.7.0] - 2022-01-17

[0.7.0] - Added

- Added NLP metrics:
  - `MatchErrorRate` (#619)
  - `WordInfoLost` and `WordInfoPreserved` (#630)
  - `SQuAD` (#623)
  - `CHRFScore` (#641)

- `TranslationEditRate` (#646)

- `ExtendedEditDistance` (#668)

- Added `MultiScaleSSIM` into image metrics (#679)

- Added Signal to Distortion Ratio (SDR) to audio package (#565)

- Added `MinMaxMetric` to wrappers (#556)

- Added `ignore_index` to retrieval metrics (#676)

- Added support for multi references in `ROUGEScore` (#680)

- Added a default VSCode devcontainer configuration (#621)

[0.7.0] - Changed

- Scalar metrics will now consistently have additional dimensions squeezed (#622)

- Metrics having third party dependencies removed from global import (#463)

- Untokenized for `BLEUScore` input stay consistent with all the other text metrics (#640)

- Arguments reordered for `TER`, `BLEUScore`, `SacreBLEUScore`, `CHRFScore` now expect input order as predictions first and target second (#696)

- Changed dtype of metric state from `torch.float` to `torch.long` in `ConfusionMatrix` to accommodate larger values (#715)

- Unify `preds`, `target` input argument's naming across all text metrics (#723, #727)

  - `bert`, `bleu`, `chrf`, `sacre_bleu`, `wip`, `wil`, `cer`, `ter`, `wer`, `mer`, `rouge`, `squad`

[0.7.0] - Deprecated

- Renamed IoU -> Jaccard Index (#662)

- Renamed text WER metric (#714)

- `functional.wer` -> `functional.word_error_rate`

- `WER` -> `WordErrorRate`

- Renamed correlation coefficient classes: (#710)

  - `MatthewsCorrcoef` -> `MatthewsCorrCoef`

  - `PearsonCorrcoef` -> `PearsonCorrCoef`

  - `SpearmanCorrcoef` -> `SpearmanCorrCoef`

- Renamed audio STOI metric: (#753, #758)

  - `audio.STOI` to `audio.ShortTimeObjectiveIntelligibility`

  - `functional.audio.stoi` to `functional.audio.short_time_objective_intelligibility`

- Renamed audio PESQ metrics: (#751)

  - `functional.audio.pesq` -> `functional.audio.perceptual_evaluation_speech_quality`

  - `audio.PESQ` -> `audio.PerceptualEvaluationSpeechQuality`

- Renamed audio SDR metrics: (#711)

- – `functional.sdr` -> `functional.signal_distortion_ratio`
  - – `functional.si_sdr` -> `functional.scale_invariant_signal_distortion_ratio`
  - – `SDR` -> `SignalDistortionRatio`
  - – `SI_SDR` -> `ScaleInvariantSignalDistortionRatio`
- Renamed audio SNR metrics: (#712)
  - – `functional.snr` -> `functional.signal_distortion_ratio`
  - – `functional.si_snr` -> `functional.scale_invariant_signal_noise_ratio`
  - – `SNR` -> `SignalNoiseRatio`
  - – `SI_SNR` -> `ScaleInvariantSignalNoiseRatio`
- Renamed F-score metrics: (#731, #740)
  - – `functional.f1` -> `functional.f1_score`
  - – `F1` -> `F1Score`
  - – `functional.fbeta` -> `functional.fbeta_score`
  - – `FBeta` -> `FBetaScore`
- Renamed Hinge metric: (#734)
  - – `functional.hinge` -> `functional.hinge_loss`
  - – `Hinge` -> `HingeLoss`
- Renamed image PSNR metrics (#732)
  - – `functional.psnr` -> `functional.peak_signal_noise_ratio`
  - – `PSNR` -> `PeakSignalNoiseRatio`
- Renamed image PIT metric: (#737)
  - – `functional.pit` -> `functional.permutation_invariant_training`
  - – `PIT` -> `PermutationInvariantTraining`
- Renamed image SSIM metric: (#747)
  - – `functional.ssim` -> `functional.scale_invariant_signal_noise_ratio`
  - – `SSIM` -> `StructuralSimilarityIndexMeasure`
- Renamed detection `MAP` to `MeanAveragePrecision` metric (#754)
- Renamed Fidelity & LPIPS image metric: (#752)
  - – `image.FID` -> `image.FrechetInceptionDistance`
  - – `image.KID` -> `image.KernelInceptionDistance`
  - – `image.LPIPS` -> `image.LearnedPerceptualImagePatchSimilarity`

[0.7.0] - Removed

- Removed `embedding_similarity` metric (#638)
- Removed argument `concatenate_texts` from `wer` metric (#638)
- Removed arguments `newline_sep` and `decimal_places` from `rouge` metric (#638)

[0.7.0] - Fixed

- Fixed MetricCollection kwargs filtering when no `kwargs` are present in update signature (#707)

2.10.3 [0.6.2] - 2021-12-15

[0.6.2] - Fixed

- Fixed `torch.sort` currently does not support bool `dtype` on CUDA (#665)
- Fixed mAP properly checks if ground truths are empty (#684)
- Fixed initialization of tensors to be on correct device for `MAP` metric (#673)

2.10.4 [0.6.1] - 2021-12-06

[0.6.1] - Changed

- Migrate MAP metrics from pycocotools to PyTorch (#632)
- Use `torch.topk` instead of `torch.argsort` in retrieval precision for speedup (#627)

[0.6.1] - Fixed

- Fix empty predictions in MAP metric (#594, #610, #624)
- Fix edge case of AUROC with `average=weighted` on GPU (#606)
- Fixed `forward` in compositional metrics (#645)

2.10.5 [0.6.0] - 2021-10-28

[0.6.0] - Added

- Added audio metrics:
  - Perceptual Evaluation of Speech Quality (PESQ) (#353)
  - Short-Time Objective Intelligibility (STOI) (#353)
- Added Information retrieval metrics:
  - `RetrievalRPrecision` (#577)
  - `RetrievalHitRate` (#576)
- Added NLP metrics:

- – SacreBLEUScore ([#546](#))
- – CharErrorRate ([#575](#))
- Added other metrics:
  - – Tweedie Deviance Score ([#499](#))
  - – Learned Perceptual Image Patch Similarity (LPIPS) ([#431](#))
- Added `MAP` (mean average precision) metric to new detection package ([#467](#))
- Added support for float targets in `nDCG` metric ([#437](#))
- Added `average` argument to `AveragePrecision` metric for reducing multi-label and multi-class problems ([#477](#))
- Added `MultioutputWrapper` ([#510](#))
- Added metric sweeping:
  - – `higher_is_better` as constant attribute ([#544](#))
  - – `higher_is_better` to rest of codebase ([#584](#))
- Added simple aggregation metrics: `SumMetric`, `MeanMetric`, `CatMetric`, `MinMetric`, `MaxMetric` ([#506](#))
- Added pairwise submodule with metrics ([#553](#))
  - – `pairwise_cosine_similarity`
  - – `pairwise_euclidean_distance`
  - – `pairwise_linear_similarity`
  - – `pairwise_manhatten_distance`

[0.6.0] - Changed

- `AveragePrecision` will now as default output the `macro` average for multilabel and multiclass problems ([#477](#))
- `half`, `double`, `float` will no longer change the dtype of the metric states. Use `metric.set_dtype` instead ([#493](#))
- Renamed `AverageMeter` to `MeanMetric` ([#506](#))
- Changed `is_differentiable` from property to a constant attribute ([#551](#))
- ROC and AUROC will no longer throw an error when either the positive or negative class is missing. Instead return 0 score and give a warning

[0.6.0] - Deprecated

- Deprecated `functional.self_supervised.embedding_similarity` in favour of new pairwise submodule

[0.6.0] - Removed

- Removed `dtype` property ([#493](#))

[0.6.0] - Fixed

- Fixed bug in F1 with `average='macro'` and `ignore_index!=None` ([#495](#))
- Fixed bug in `pit` by using the returned first result to initialize device and type ([#533](#))
- Fixed `SSIM` metric using too much memory ([#539](#))
- Fixed bug where `device` property was not properly update when metric was a child of a module (#542)

2.10.6 [0.5.1] - 2021-08-30

[0.5.1] - Added

- Added `device` and `dtype` properties ([#462](#))
- Added `TextTester` class for robustly testing text metrics ([#450](#))

[0.5.1] - Changed

- Added support for float targets in `nDCG` metric ([#437](#))

[0.5.1] - Removed

- Removed `rouge-score` as dependency for text package ([#443](#))
- Removed `jiwer` as dependency for text package ([#446](#))
- Removed `bert-score` as dependency for text package ([#473](#))

[0.5.1] - Fixed

- Fixed ranking of samples in `SpearmanCorrCoef` metric ([#448](#))
- Fixed bug where compositional metrics where unable to sync because of type mismatch ([#454](#))
- Fixed metric hashing ([#478](#))
- Fixed `BootStrapper` metrics not working on GPU ([#462](#))
- Fixed the semantic ordering of kernel height and width in `SSIM` metric ([#474](#))

2.10.7 [0.5.0] - 2021-08-09

[0.5.0] - Added

- Added **Text-related (NLP) metrics**:
    - Word Error Rate (WER) ([#383](#))
    - ROUGE ([#399](#))
    - BERT score ([#424](#))
    - BLUE score ([#360](#))
- Added `MetricTracker` wrapper metric for keeping track of the same metric over multiple epochs ([#238](#))
- Added other metrics:
    - Symmetric Mean Absolute Percentage error (SMAPE) ([#375](#))
    - Calibration error ([#394](#))
    - Permutation Invariant Training (PIT) ([#384](#))
- Added support in `nDCG` metric for target with values larger than 1 ([#349](#))
- Added support for negative targets in `nDCG` metric ([#378](#))
- Added `None` as reduction option in `CosineSimilarity` metric ([#400](#))
- Allowed passing labels in (n_samples, n_classes) to `AveragePrecision` ([#386](#))

[0.5.0] - Changed

- Moved `psnr` and `ssim` from `functional.regression.*` to `functional.image.*` ([#382](#))
- Moved `image_gradient` from `functional.image_gradients` to `functional.image.gradients` ([#381](#))
- Moved `R2Score` from `regression.r2score` to `regression.r2` ([#371](#))
- Pearson metric now only store 6 statistics instead of all predictions and targets ([#380](#))
- Use `torch.argmax` instead of `torch.topk` when k=1 for better performance ([#419](#))
- Moved check for number of samples in R2 score to support single sample updating ([#426](#))

[0.5.0] - Deprecated

- Rename `r2score` >> `r2_score` and `kldivergence` >> `kl_divergence` in `functional` ([#371](#))
- Moved `bleu_score` from `functional.nlp` to `functional.text.bleu` ([#360](#))

[0.5.0] - Removed

- Removed restriction that `threshold` has to be in (0,1) range to support logit input ([#351](#) [#401](#))

- Removed restriction that `preds` could not be bigger than `num_classes` to support logit input ([#357](#))

- Removed module `regression.psnr` and `regression.ssim` ([#382](#)):

- Removed ([#379](#)):

    - function `functional.mean_relative_error`

    - `num_thresholds` argument in `BinnedPrecisionRecallCurve`

[0.5.0] - Fixed

- Fixed bug where classification metrics with `average='macro'` would lead to wrong result if a class was missing ([#303](#))

- Fixed `weighted`, `multi-class` AUROC computation to allow for 0 observations of some class, as contribution to final AUROC is 0 ([#376](#))

- Fixed that `_forward_cache` and `_computed` attributes are also moved to the correct device if metric is moved ([#413](#))

- Fixed calculation in `IoU` metric when using `ignore_index` argument ([#328](#))

2.10.8 [0.4.1] - 2021-07-05

[0.4.1] - Changed

- Extend typing ([#330](#), [#332](#), [#333](#), [#335](#), [#314](#))

[0.4.1] - Fixed

- Fixed DDP by `is_sync` logic to `Metric` ([#339](#))

2.10.9 [0.4.0] - 2021-06-29

[0.4.0] - Added

- Added **Image-related metrics**:

    - Fréchet inception distance (FID) ([#213](#))

    - Kernel Inception Distance (KID) ([#301](#))

    - Inception Score ([#299](#))

    - KL divergence ([#247](#))

- Added **Audio metrics**: SNR, SI_SDR, SI_SNR ([#292](#))

- Added other metrics:

    - Cosine Similarity ([#305](#))

    - Specificity ([#210](#))

– Mean Absolute Percentage error (MAPE) ([#248](#))

- Added `add_metrics` method to `MetricCollection` for adding additional metrics after initialization ([#221](#))

- Added pre-gather reduction in the case of `dist_reduce_fx="cat"` to reduce communication cost ([#217](#))

- Added better error message for AUROC when `num_classes` is not provided for multiclass input ([#244](#))

- Added support for unnormalized scores (e.g. logits) in `Accuracy`, `Precision`, `Recall`, `FBeta`, `F1`, `StatScore`, `Hamming`, `ConfusionMatrix` metrics ([#200](#))

- Added `squared` argument to `MeanSquaredError` for computing RMSE ([#249](#))

- Added `is_differentiable` property to `ConfusionMatrix`, `F1`, `FBeta`, `Hamming`, `Hinge`, `IOU`, `MatthewsCorrcoef`, `Precision`, `Recall`, `PrecisionRecallCurve`, `ROC`, `StatScores` ([#253](#))

- Added `sync` and `sync_context` methods for manually controlling when metric states are synced ([#302](#))

[0.4.0] - Changed

- Forward cache is reset when `reset` method is called ([#260](#))

- Improved per-class metric handling for imbalanced datasets for `precision`, `recall`, `precision_recall`, `fbeta`, `f1`, `accuracy`, and `specificity` ([#204](#))

- Decorated `torch.jit.unused` to `MetricCollection` forward ([#307](#))

- Renamed `thresholds` argument to binned metrics for manually controlling the thresholds ([#322](#))

- Extend typing ([#324](#), [#326](#), [#327](#))

[0.4.0] - Deprecated

- Deprecated `functional.mean_relative_error`, use `functional.mean_absolute_percentage_error` ([#248](#))

- Deprecated `num_thresholds` argument in `BinnedPrecisionRecallCurve` ([#322](#))

[0.4.0] - Removed

- Removed argument `is_multiclass` ([#319](#))

[0.4.0] - Fixed

- AUC can also support more dimensional inputs when all but one dimension are of size 1 ([#242](#))

- Fixed `dtype` of modular metrics after reset has been called ([#243](#))

- Fixed calculation in `matthews_corrcoef` to correctly match formula ([#321](#))

2.10.10 [0.3.2] - 2021-05-10

[0.3.2] - Added

- Added `is_differentiable` property:
    - To `AUC`, `AUROC`, `CohenKappa` and `AveragePrecision` (#178)
    - To `PearsonCorrCoef`, `SpearmanCorrcoef`, `R2Score` and `ExplainedVariance` (#225)

[0.3.2] - Changed

- `MetricCollection` should return metrics with prefix on `items()`, `keys()` (#209)
- Calling `compute` before `update` will now give warning (#164)

[0.3.2] - Removed

- Removed `numpy` as direct dependency (#212)

[0.3.2] - Fixed

- Fixed auc calculation and add tests (#197)
- Fixed loading persisted metric states using `load_state_dict()` (#202)
- Fixed PSNR not working with DDP (#214)
- Fixed metric calculation with unequal batch sizes (#220)
- Fixed metric concatenation for list states for zero-dim input (#229)
- Fixed numerical instability in `AUROC` metric for large input (#230)

2.10.11 [0.3.1] - 2021-04-21

- Cleaning remaining inconsistency and fix PL develop integration (#191, #192, #193, #194)

2.10.12 [0.3.0] - 2021-04-20

[0.3.0] - Added

- Added `BootStrapper` to easily calculate confidence intervals for metrics (#101)
- Added Binned metrics (#128)
- Added metrics for Information Retrieval ((PL^5032)):
    - `RetrievalMAP` (PL^5032)
    - `RetrievalMRR` (#119)
    - `RetrievalPrecision` (#139)
    - `RetrievalRecall` (#146)
    - `RetrievalNormalizedDCG` (#160)

- – `RetrievalFallOut` (#161)
- Added other metrics:
    - – `CohenKappa` (#69)
    - – `MatthewsCorrcoef` (#98)
    - – `PearsonCorrcoef` (#157)
    - – `SpearmanCorrcoef` (#158)
    - – `Hinge` (#120)
- Added `average='micro'` as an option in AUROC for multilabel problems (#110)
- Added multilabel support to ROC metric (#114)
- Added testing for `half` precision (#77, #135)
- Added `AverageMeter` for ad-hoc averages of values (#138)
- Added `prefix` argument to `MetricCollection` (#70)
- Added `__getitem__` as metric arithmetic operation (#142)
- Added property `is_differentiable` to metrics and test for differentiability (#154)
- Added support for `average`, `ignore_index` and `mdmc_average` in `Accuracy` metric (#166)
- Added `postfix` arg to `MetricCollection` (#188)

[0.3.0] - Changed

- Changed `ExplainedVariance` from storing all preds/targets to tracking 5 statistics (#68)
- Changed behaviour of `confusionmatrix` for multilabel data to better match `multilabel_confusion_matrix` from sklearn (#134)
- Updated FBeta arguments (#111)
- Changed `reset` method to use `detach.clone()` instead of `deepcopy` when resetting to default (#163)
- Metrics passed as dict to `MetricCollection` will now always be in deterministic order (#173)
- Allowed `MetricCollection` pass metrics as arguments (#176)

[0.3.0] - Deprecated

- Rename argument `is_multiclass` -> `multiclass` (#162)

[0.3.0] - Removed

- Prune remaining deprecated (#92)

[0.3.0] - Fixed

- Fixed when `_stable_1d_sort` to work when n>=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)

2.10.13 [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)

2.10.14 [0.1.0] - 2021-02-22

- Added `Accuracy` metric now generalizes to Top-k accuracy for (multi-dimensional) multi-class inputs using the `top_k` parameter (PL^4838)

- Added `Accuracy` metric now enables the computation of subset accuracy for multi-label or multi-dimensional multi-class inputs with the `subset_accuracy` parameter (PL^4838)

- Added `HammingDistance` metric to compute the hamming distance (loss) (PL^4838)

- Added `StatScores` metric to compute the number of true positives, false positives, true negatives and false negatives (PL^4839)

- Added `R2Score` metric (PL^5241)

- Added `MetricCollection` (PL^4318)

- Added `.clone()` method to metrics (PL^4318)

- Added `IoU` class interface (PL^4704)

- The `Recall` and `Precision` metrics (and their functional counterparts `recall` and `precision`) can now be generalized to Recall@K and Precision@K with the use of `top_k` parameter (PL^4842)

- Added compositional metrics (PL^5464)

- Added AUC/AUROC class interface (PL^5479)

- Added `QuantizationAwareTraining` callback (PL^5706)

- Added `ConfusionMatrix` class interface (PL^4348)

- Added multiclass AUROC metric (PL^4236)

- Added `PrecisionRecallCurve, ROC, AveragePrecision` class metric (PL^4549)

- Classification metrics overhaul (PL^4837)

- Added `F1` class metric (PL^4656)

- Added metrics aggregation in Horovod and fixed early stopping (PL^3775)

- Added `persistent(mode)` method to metrics, to enable and disable metric states being added to `state_dict` (PL^4482)

- Added unification of regression metrics (PL^4166)

- Added persistent flag to `Metric.add_state` (PL^4195)

- Added classification metrics (PL^4043)

- Added new Metrics API. (PL^3868, PL^3921)

- Added EMB similarity (PL^3349)

- Added SSIM metrics (PL^2671)

- Added BLEU metrics (PL^2535)

INDICES AND TABLES

- genindex
- modindex
- search

INDEX