pytorch-pfn-extras

Preferred Networks, Inc.

Oct 05, 2023

CONTENTS

pytorch-pfn-extras (PPE) is a collection of supplementary components to accelerate research and development in PyTorch.

CHAPTER

ONE

USER GUIDE

1.1 Quick Start

1.1.1 Quick Start

First, pytorch-pfn-extras organizes the training code implemented using PyTorch using the Trainer/Evaluator classes.

Next, it provides the following interfaces for training PyTorch models.

1. Addition of extensions for analysis and visualization
2. Runtime changes
3. Addition of custom training steps
4. Custom data handling

Step 1: Use Trainer

First, pass to the Trainer the Model and Optimizer you want to train.

Listing 1: quick_start_trainer.py

```
import pytorch_pfn_extras as ppe
import torch


class Model(torch.nn.Module):
    def __init__(self, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)
        self.linear = torch.nn.Linear(in_features=64, out_features=2)
        self.criterion = torch.nn.NLLLoss()

    def forward(self, x, target):
        y = self.linear.forward(x).log_softmax(dim=1)
        loss = self.criterion.forward(y, target)
        return {"loss": loss}


model = Model()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

(continues on next page)

(continued from previous page)

```
device = (
    "cuda:0"  # or any other PyTorch devices ('cpu', etc.) or PPE runtime names
)
epochs = 3
# Create a trainer with the defined model, optimizer, and other parameters
trainer = ppe.engine.create_trainer(
    models=model,
    optimizers=optimizer,
    max_epochs=epochs,
    evaluator=ppe.engine.create_evaluator(
        models=model,
        device=device,
    ),
    device=device,
)

# Send the model to device(GPU) for computation
ppe.to(model, device=device)

batch_size = 10
# Create 10 batches of random training data with dimension (batch_size x 64)
training_data = [
    {
        "x": torch.rand((batch_size, 64)),
        "target": torch.ones((batch_size,), dtype=torch.long),
    }
    for _ in range(10)
]
# Create 10 batches of random validation data with dimension (batch_size x 64)
validation_data = [
    {
        "x": torch.rand((batch_size, 64)),
        "target": torch.ones((batch_size,), dtype=torch.long),
    }
    for _ in range(10)
]

# Start the training and validation of the model
trainer.run(train_loader=training_data, val_loader=validation_data)

print("Finish training!")
```

Step 2: Get Log

Next, collect the logs of the training progress.

Listing 2: quick_start_log.py

```
import pytorch_pfn_extras as ppe
import torch


class Model(torch.nn.Module):
    def __init__(self, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)
        self.linear = torch.nn.Linear(in_features=64, out_features=2)
        self.criterion = torch.nn.NLLLoss()

    def forward(self, x, target):
        y = self.linear.forward(x).log_softmax(dim=1)
        loss = self.criterion.forward(y, target)
        return {"loss": loss}


model = Model()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

device = "cuda:0"
epochs = 3
trainer = ppe.engine.create_trainer(
    models=model,
    optimizers=optimizer,
    max_epochs=epochs,
    evaluator=ppe.engine.create_evaluator(
        models=model,
        device=device,
        options={
            "eval_report_keys": [
                "loss"
            ],  # Let the value of the loss be notified to the LogReport.
        },
    ),
    device=device,
    options={
        "train_report_keys": [
            "loss"
        ],  # Let the value of the loss be notified to the LogReport.
    },
)

trainer.extend(
    ppe.training.extensions.LogReport()
)  # It is an extension to collect parameters reported during training.

ppe.to(model, device=device)
```

(continues on next page)

(continued from previous page)

```
batch_size = 10
training_data = [
    {
        "x": torch.rand((batch_size, 64)),
        "target": torch.ones((batch_size,), dtype=torch.long),
    }
    for _ in range(10)
]
validation_data = [
    {
        "x": torch.rand((batch_size, 64)),
        "target": torch.ones((batch_size,), dtype=torch.long),
    }
    for _ in range(10)
]

trainer.run(train_loader=training_data, val_loader=validation_data)

print("Finish training!")
```

The logs of the collected learning progress are output to `./result/log`.

Step 3: Display of progress

Make it possible to check the progress of the learning.

Listing 3: quick_start_progress.py

```
import pytorch_pfn_extras as ppe
import torch


class Model(torch.nn.Module):
    def __init__(self, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)
        self.linear = torch.nn.Linear(in_features=64, out_features=2)
        self.criterion = torch.nn.NLLLoss()

    def forward(self, x, target):
        y = self.linear.forward(x).log_softmax(dim=1)
        loss = self.criterion.forward(y, target)
        return {"loss": loss}


model = Model()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

device = "cuda:0"
epochs = 3
trainer = ppe.engine.create_trainer(
    models=model,
```

(continues on next page)

(continued from previous page)

```
    optimizers=optimizer,
    max_epochs=epochs,
    evaluator=ppe.engine.create_evaluator(
        models=model,
        device=device,
        options={
            "eval_report_keys": ["loss"],
        },
    ),
    device=device,
    options={
        "train_report_keys": ["loss"],
    },
)

trainer.extend(ppe.training.extensions.LogReport())
trainer.extend(ppe.training.extensions.ProgressBar())
trainer.extend(
    ppe.training.extensions.PrintReport(  # Displays the collected logs interactively.
        [
            "epoch",  # epoch, iteration, elapsed_time are automatically collected by␣
↪LogReport.
            "iteration",
            "elapsed_time",
            "train/loss",  # The parameters specified by train_report_keys are collected␣
↪under keys with the 'train/' prefix.
            "val/loss",  # The parameters specified by eval_report_keys are collected␣
↪under keys with the 'val/' prefix.
        ],
    )
)

ppe.to(model, device=device)

batch_size = 10
training_data = [
    {
        "x": torch.rand((batch_size, 64)),
        "target": torch.ones((batch_size,), dtype=torch.long),
    }
    for _ in range(10)
]
validation_data = [
    {
        "x": torch.rand((batch_size, 64)),
        "target": torch.ones((batch_size,), dtype=torch.long),
    }
    for _ in range(10)
]

trainer.run(train_loader=training_data, val_loader=validation_data)
```

(continues on next page)

(continued from previous page)

```
print("Finish training!")
```

Step 4: Save Model

Finally, save the trained model.

Listing 4: quick_start_save.py

```
import pytorch_pfn_extras as ppe
import torch


class Model(torch.nn.Module):
    def __init__(self, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)
        self.linear = torch.nn.Linear(in_features=64, out_features=2)
        self.criterion = torch.nn.NLLLoss()

    def forward(self, x, target):
        y = self.linear.forward(x).log_softmax(dim=1)
        loss = self.criterion.forward(y, target)
        return {"loss": loss}


model = Model()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)


device = "cuda:0"
epochs = 3
trainer = ppe.engine.create_trainer(
    models=model,
    optimizers=optimizer,
    max_epochs=epochs,
    evaluator=ppe.engine.create_evaluator(
        models=model,
        device=device,
        options={
            "eval_report_keys": ["loss"],
        },
    ),
    device=device,
    options={
        "train_report_keys": ["loss"],
    },
)

trainer.extend(ppe.training.extensions.LogReport())
trainer.extend(ppe.training.extensions.ProgressBar())
trainer.extend(
    ppe.training.extensions.PrintReport(  # Displays the collected logs interactively.
```

(continues on next page)

(continued from previous page)

```
        [
            "epoch",  # epoch, iteration, elapsed_time are automatically collected by␣
↪LogReport.
            "iteration",
            "elapsed_time",
            "train/loss",  # The parameters specified by train_report_keys are collected␣
↪under keys with the 'train/' prefix.
            "val/loss",  # The parameters specified by eval_report_keys are collected␣
↪under keys with the 'val/' prefix.
        ],
    )
)
trainer.extend(
    ppe.training.extensions.snapshot(target=model)
)  # Save the model parameters after each epoch.

ppe.to(model, device=device)

batch_size = 10
training_data = [
    {
        "x": torch.rand((batch_size, 64)),
        "target": torch.ones((batch_size,), dtype=torch.long),
    }
    for _ in range(10)
]
validation_data = [
    {
        "x": torch.rand((batch_size, 64)),
        "target": torch.ones((batch_size,), dtype=torch.long),
    }
    for _ in range(10)
]

trainer.run(train_loader=training_data, val_loader=validation_data)

print("Finish training!")
```

The model parameters are stored with a file name that includes the time they were saved under `./result`.

Snapshots are generated using `state_dict()`. Please refer to the official PyTorch docs for how to load the model.

1.2 Trainer

1.2.1 Trainer and Evaluator

The Trainer and Evaluator provide the device-agnostic training framework for PyTorch. These APIs abstract the training process using different *runtimes*, handlers, and *logics*.

Concepts

- **Trainer** (*ppe.engine.create_trainer()*) abstracts the training loop, built on top of the *ExtensionsManager*.
- **Evaluator** (*ppe.engine.create_evaluator()*) abstracts the evaluation step and invoked from the Trainer (usually once in every epoch).
- **Runtime** (*ppe.runtime.BaseRuntime*) represents an environment used to execute models. Device-specific implementations will reside here. PPE provides the default Runtime that supports the PyTorch-native devices (*ppe.runtime.PyTorchRuntime*).
- **Handler** (*ppe.handler.Handler*) is a layer to support device-agnostic training. This is considered as a low-level API and in most cases users can just use the Handler provided by PPE.
- **Logic** (*ppe.handler.Logic*) is a set of callback functions that define the training logic (*optimizer.zero_grad()*, forward, backward, *optimizer.step()*). You can inherit the class and define your own training flow in case you need more complex training processes such as GAN.
- **Model** is a `torch.nn.Module` used for training and evaluation, whose inputs are dicts or keyword arguments and outputs of the `forward` pass is a dict.

Note that the default logic will perform `backward` in tensors returned by `model.forward` so you will need to perform the loss calculation inside the model itself.

Trainer at a glance

```
import torch
import torch.nn.functional as F

import pytorch_pfn_extras as ppe


class MyModel(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.w = torch.nn.LazyLinear(1)

    def forward(self, *, x, target):
        y = self.w(x)
        loss = F.nll_loss(y, target)
        prefix = 'train' if self.training else 'val'
        ppe.reporting.report({f'{prefix}/loss': loss.item()})
        return {'loss': loss}


model = MyModel()
optim = torch.optim.SGD(model.parameters(), lr=0.01)

extensions = [
    ppe.training.extensions.LogReport(),
    ppe.training.extensions.ProgressBar(),
    ppe.training.extensions.PrintReport(
        ['epoch', 'iteration', 'train/loss', 'val/loss']),
]
```

(continues on next page)

(continued from previous page)

```
device = 'cuda:0'  # or any other PyTorch devices ('cpu', etc.) or PPE runtime names
epochs = 10
trainer = ppe.engine.create_trainer(
    model,
    optim,
    epochs,
    evaluator=ppe.engine.create_evaluator(
        model,
        device=device,
        progress_bar=True,
    ),
    device=device,
    extensions=extensions,
)

# Move the model to the device. This is almost equivalent to
# `model.to(device)`, but supports PPE runtimes as well as the PyTorch's
# built-in devices.
ppe.to(model, device)

# Using dummy data to illustrate the minimal working example.
# Notice that dict keys match with the kwargs of the forward method.
train_loader = torch.utils.data.DataLoader(
    [{'x': torch.rand(10, 64), 'target': torch.tensor([1])} for _ in range(1)],
    num_workers=8)
val_loader = torch.utils.data.DataLoader(
    [{'x': torch.rand(10, 64), 'target': torch.tensor([1])} for _ in range(1)],
    num_workers=8)

trainer.run(train_loader, val_loader)
```

Snapshot

To obtain and save the trained model for later use you can use the *Snapshot* extension, or directly invoke *state_dict* on the trainer itself.

Handler

The `ppe.handler.Handler` object is used to help the trainer and evaluator objects in the *Logic* and *Runtime* manipulation. This class should ideally never be overriden by the user if the desired functionality can be achieved through subclassing BaseLogic or BaseRuntime.

The handler object's main responsibility is to inspect all the submodules of a module to obtain the runtimes they have associated, and then execute their callbacks accordingly. In addition, it drives the actual model execution by using the user provided Logic object and deals with asynchronous execution in runtimes that provide support for it.

Runtime

By inheriting *ppe.runtime.BaseRuntime* and implementing your own runtime, you can use your non-standard devices with the training loop.

```
class MyRuntime(BaseRuntime):
    ...

# Register MyRuntime with device name "mydev"
ppe.runtime.runtime_registry.register('mydev', MyRuntime)

ppe.to(module_or_tensor, 'mydev')
```

See *Runtimes for Custom Devices* if you are interested in implementing your own runtime.

1.2.2 Logic for Custom Training and Evaluation

In the training and evaluation engines, *ppe.handler.BaseLogic* API is in charge of abstracting the algorithmic details of the training and evaluation loops.

Logic is an object that defines multiple callbacks used through the training and evaluation processes. With logic, we can implement training of complex models such as GANs.

Users wanting to define their own Logic for training can inherit from *ppe.handler.Logic* which implements the training and evaluation steps to train a single module.

Logic functions are not exepcted to be directly called by the user. They will be invoked by the Trainer and Evaluator engines.

Default Logic (`ppe.handler.Logic`)

PPE provides a default logic that performs the forward/backward/optimizer loop for a single model. This logic allows using some torch features such as AMP autocast and GradScaler and performs the backward pass on the outputs specified by the config option backward_outputs.

CodeBlock Logic (`ppe.handler.Logic`)

With the CodeBlock API, we provide a basic logic that uses it to perform the training of a single model. Similarly to the default logic AMP features are supported but by means of the Runtime. For more information check the codeblock documentation.

1.2.3 Runtimes for Custom Devices

Note: This documentation is intended for those implementing the own device backend for PPE training framework. Most users can just skip this chapter.

The *ppe.runtime.BaseRuntime* API is in charge of abstracting the device details and performing the movement of data and modules to the corresponding device.

A runtime is an object that defines multiple callbacks used through the training, evaluation, and regular model calls. With runtimes, we can implement training in devices other than cpus or gpus with minimal changes to the user code.

Users wanting to override only a few callbacks can inherit from *ppe.runtime.PyTorchRuntime* which implements the basic functionality for cpu and gpu devices.

Runtimes must be registered by calling the `ppe.runtime.runtime_registry.register(device_name, runtime_class)` function for them to be discoverable.

Use of `ppe.to` to transfer modules and batches to custom devices

If you have defined a new runtime for a custom device the `ppe.to` function allows moving a module or a tensor to the new device by invoking the `Runtime.move_tensor` and `Runtime.move_module` when needed.

The module will be tagged by adding a attribute named `_ppe_runtime` that holds the needed runtime. It is the responsibility of the user custom runtime to perform the actual movement to the device and apply all the transformations needed to a module so it can be correctly executed.

Usually, runtime writers will need to replace the given module forward function by a new one that performs the actual device execution.

```
class MyModule(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.layer = torch.nn.Linear(10, 10)

    def forward(x):
        return self.layer(x)

class MyMagicDeviceRuntime(ppe.runtime.BaseRuntime):
    def _device_forward(self, args):
        return run_batch_in_my_device(args):

    def move_module(self, module):
        # Registers a hook to initialize the module on the first batch
        # execution
        def hook(module, *args):
            module._ppe_runtime.initialize_module(module, args)

        self.hook = module.register_forward_pre_hook(hook)
        # Change the module forward to do the computation in the device
        module.forward = self._device_forward

    def initialize_module(self, module, loader_or_batch, optimizer=None):
        create_the_module_in_my_device(module, loader_or_batch, optimizer)

# Register the runtime class
ppe.runtime.runtime_registry.register('my_device', MyMagicDeviceRuntime)

# Create a regular module
module = MyModule()
# Move the module to the device
ppe.to(module, device='my_device')

for x in my_dataloader:
    # The first iteration will create the module in the device
    # and the next ones will directly execute the module in the device instead
```

(continues on next page)

(continued from previous page)

```
    # of executing the regular pytorch `forward` call.
    y = model(x)
```

Please note that this is an oversimplified description and that developing a runtime that is 100% compatible with PyTorch requires to wrap the substitute forward function with `torch.autograd.Function` among several other concerns such as `state_dict` manipulation to ensure correcteness.

Runtime Registry

When creating a new `Runtime` class for custom needs, they need to be registered in a global `runtime_registry` object as detailed above. This object is of the `_RuntimeRegistry` type and it maintains a map of strings and `Runtime` types. The keys are the devices passed to `ppe.to` and the types will be the type of the `Runtime` object that `ppe.to` will use to treat the module or tensor. Beware that users are not supposed to interact directly with this class, only with the `runtime_registry.register` to register new runtimes.

1.2.4 CodeBlocks for Abstracting Logic Steps

The *ppe.handler.CodeBlock* API provides a mean of abstracting the actions that are possible to be done in a model in a device agnostic way.

Currently there is support for two different actions using `CodeBlock`.

- **:function:`ppe.handler.update_parameters <pytorch_pfn_extras.handler.update_parameters>`**

  takes a model, an optimizer and returns a `CodeBlock` object that performs the forward, backward and optimizer step at once.

- **:function:`ppe.handler.forward <pytorch_pfn_extras.handler.forward>`**

  takes a model and returns a `CodeBlock` object that performs only the forward pass.

Executing CodeBlocks

For executing `CodeBlock` objects we need to add an **:method:`ppe.runtime.BaseRuntime.execute <pytorch_pfn_extras.runtime.BaseRuntime.execute`** to the corresponding `Runtime` class. This method takes a `CodeBlock` and uses the information in the object to execute the `CodeBlock` in the device. Note that the **:method:`ppe.runtime.PyTorchRuntime.execute <pytorch_pfn_extras.runtime.PyTorchRuntime.execute`** method providesn support for using PyTorch AMP with autocast or gradient scaling if needed.

Moreover, you can execute `CodeBlock` objects outside the training API.

```
ppe.to(model, "cuda:0")
cblock = ppe.handler.update_parameters(model, optimizer)
outs = cblock(input_batch)
```

The only requirement is that the associated model has been assigned a device using `ppe.to`.

1.3 Extensions

1.3.1 Extensions Manager

Extensions Manager provides an interface to extend your training loop, by integrating it into your manual training loop or Ignite.

Extensions

See the *API Reference* for the list of built-in extensions.

How to use

Create an `ExtensionsManager` object and then wrap the iteration of your training loop inside the `manager.run_iteration()` context manager.

An example follows:

```
import pytorch_pfn_extras as ppe
from pytorch_pfn_extras.training import extensions

import time
import math

max_epoch = 10
iters_per_epoch = 938

# manager.extend(...) also works
my_extensions = [extensions.LogReport(),
                 extensions.ProgressBar(),
                 extensions.PrintReport(['epoch', 'iteration', 'sin', 'cos'])]

models = {}
optimizers = []
manager = ppe.training.ExtensionsManager(
    models, optimizers, max_epoch,
    extensions=my_extensions,
    iters_per_epoch=iters_per_epoch)

for epoch in range(max_epoch):
    for i in range(iters_per_epoch):
        with manager.run_iteration():
            ppe.reporting.report({
                'sin': math.sin(i * 2 * math.pi / iters_per_epoch),
                'cos': math.cos(i * 2 * math.pi / iters_per_epoch),
            })
            time.sleep(0.001)
```

In the examples folder there is a mnist using all the avaiable extensions.

Usage with Ignite

Ignite is supported by using the `IgniteExtensionsManager` with the trainer as the first argument.

The user needs to define an ignite event to report the appropiated metrics for the extensions to use them.

```
manager = ppe.training.IgniteExtensionsManager(
    trainer, models, optimizers, epochs,
    extensions=my_extensions)

@trainer.on(Events.ITERATION_COMPLETED)
def report_loss(engine):
    ppe.reporting.report({'train/loss':engine.state.output})
```

Using Evaluators

Regular PyTorch

In order to report the results of the evaluation so they can be accessed by other extensions, an `Evaluation` extension needs to be created with the argument `eval_func` set to a function that gets the current data and target batches as parameters and reports the needed metrics. Example

The test function looks has the following signature

```
def test(args, model, device, data, target):
```

and is invoked once per batch in the validation dataloader. It is important to report the current validation loss or accuracy in order to the log report to see it.

```
def test(args, model, device, data, target):
    ...
    # Final result will be average of averages of the same size
    test_loss += F.nll_loss(output, target, reduction='mean').item()
    ppe.reporting.report({'val/loss': test_loss})
    pred = output.argmax(dim=1, keepdim=True)
    correct += pred.eq(target.view_as(pred)).sum().item()
    ppe.reporting.report({'val/acc': correct/len(data)})
```

Ignite

Just use the `IgniteEvaluator` extension with the ignite created evaluator as the first parameter and you are ready to go. Example The metrics defined when creating the evaluator with `create_supervised_evaluator` will be automatically reported

```
 create_supervised_evaluator(model, metrics={'acc': Accuracy(), 'loss': Loss(F.nll_loss)}
↪, device)
```

Snapshots

It is possible to take snapshots by using the `snapshot` training extension just as in chainer.

Whenever the extension is triggered, it saves the status of the optimizer, model and extensions to the output folder in the same way as chainer. To load the snapshot and continue the training call `torch.load` and use the `ExtensionsManager.load_state_dict`example to resume the training. The snapshots can be used outside the pytorch-pfn-extras module just by accessing the models, or optimizers fields of the loaded state.

Extensions execution order

The supported extensions honours the chainer priorities for execution. However, when using Ignite. Chainer extensions are executed after any user-defined ignite events. The idea is to use ignite events to report the metrics of the model, and after this, Chainer extensions will be executed in the chainer defined order.

If you want to execute an event-handler in between chainer extensions, create a Chainer-like extension and access the ignite engine on the `.engine` attribute of the manager object passed as a parameter when your extension is called.

1.3.2 Creating Extensions

It is possible to create an extension just by passing a function which receives the manager object as an argument to the manager extend call

```
def my_extension(manager):
    print('Epoch-Iteration: {}-{}'.format(manager.epoch, manager.iteration)

manager.extend(my_extension, trigger=(1, 'iteration')
```

It is also possible to create extensions using the `ppe.training.extension.make_extension` decorator to add a specific `trigger`, `default_name`, `priority`. In addition, `initializer`, `finalizer` and `on_error` functions can be specified as well.

```
@ppe.training.extension.make_extension(finalizer=lambda: print('done'))
def my_extension(manager):
    print('Epoch-Iteration: {}-{}'.format(manager.epoch, manager.iteration)
```

Finally, it is possible to create an extension by subclassing the `ppe.training.extensions.Extension` class as shown below.

```
import pytorch_pfn_extras as ppe

class MyExtension(ppe.training.extension.Extension)
    def __init__(self, args):
        self.args = args

    def initialize(self, manager):
        """
        Automatically called before training. Optional.
        """
        pass

    def __call__(self, manager):
        """
```

(continues on next page)

(continued from previous page)

```
        Called when the associated trigger is fired.
        """
        print('Epoch-Iteration: {}-{}'.format(manager.epoch, manager.iteration)

    def state_dict(self):
        """
        Used to serialize the state. Optional.
        """
        return {'args': self.args}

    def load_state_dict(self, state):
        """
        Used to deserialize the state. Optional.
        """
        self.args = state['args']
```

1.3.3 Reporting

`reporting.Reporter` is used to collect values that users want to watch. The reporter object holds a mapping from value names to the actually observed values. We call this mapping observations.

When a value is passed to the reporter, an object called observer can be optionally attached. In this case, the name of the observer is added as the prefix of the value name. The observer name should be registered beforehand.

```
import pytorch_pfn_extras as ppe

reporter = ppe.reporting.Reporter()
observer = object()
reporter.add_observer('my_observer', observer)
observation = {}

with reporter.scope(observation):
    reporter.report({'x': 1}, observer)

print(observation)
# outputs: {'my_observer/x': 1}
```

There is also a global API to add values:

```
import pytorch_pfn_extras as ppe

reporter = ppe.reporting.Reporter()
observer = object()
reporter.add_observer('my_observer', observer)

observation = {}
with reporter:
    with ppe.reporting.report_scope(observation):
        ppe.reporting.report({'x': 1}, observer)

print(observation)
# outputs: {'my_observer/x': 1}
```

The most important application of Reporter is to report observed values from different parts of the model in the training and validation procedures. `ExtensionsManager` objects hold their own `Reporter` object with the parameters of the target module registered as observers. `report()` can be used inside the modules to report the observed values (e.g., training loss, accuracy, activation statistics, etc.).

1.3.4 Distributed Snapshot

To take snapshots when using `torch.distributed` the only needed step is to provide the `saver_rank` keyword argument to the regular snapshot extension.

```
# saver_rank is the MPI rank which will write the actual snapshot.
snapshot = extensions.snapshot(saver_rank=saver_rank)
```

To resume the training, snapshots are loaded in every worker by using the `ExtensionsManager.load_state_dict` method, or the `extensions.snapshot` `autoload` keyword argument.

1.4 Utilities

1.4.1 Lazy Modules

Lazy modules can automatically infer shapes of parameters based on the shape of the data given to the first forward invocation.

Following modules are provided:

- `ppe.nn.LazyBatchNorm1d`, `ppe.nn.LazyBatchNorm2d`, `ppe.nn.LazyBatchNorm3d`
  - Module that behaves as `torch.nn.BatchNorm[123]d` but `num_features` can be set to `None`.
  - These modles are now included as a part of PyTorch 1.9 release (torch.nn.LazyBatchNormXd, pull-request).

The following modules are now considered deprecated as now included as a part of PyTorch 1.8 release:

- `ppe.nn.LazyLinear`
  - Module that behaves as `torch.nn.Linear` but `in_features` can be set to `None`.
  - PyTorch-native implementation: (torch.nn.LazyLinear, pull-request)
- `ppe.nn.LazyConv1d`, `ppe.nn.LazyConv2d`, `ppe.nn.LazyConv3d`
  - Module that behaves as `torch.nn.Conv[123]d` but `in_channels` can be set to `None`.
  - PyTorch-native implementation: (torch.nn.LazyConvXd, pull-request)

Now that all lazy modules are merged to the upstream, we encourage you to migrate to PyTorch's lazy modules. We will keep these implementaions only for backward compatibility.

Note that you need to run a "dummy" forward to initialize lazy parameters. See the example below:

```
import torch
import torch.nn.functional as F

import pytorch_pfn_extras as ppe


class Net(torch.nn.Module):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        super().__init__()
        self.conv1 = ppe.nn.LazyConv2d(None, 20, 5, 1)
        self.conv2 = ppe.nn.LazyConv2d(None, 50, 5, 1)
        self.fc1 = ppe.nn.LazyLinear(None, 500)
        self.fc2 = ppe.nn.LazyLinear(None, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = x.flatten(start_dim=1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)


model = Net()

# Initialize lazy parameters.
dummy_input = ...
model(dummy_input)

# Pass parameters to the optimizer.
optimizer = torch.optim.SGD(
    model.parameters(), lr=args.lr, momentum=args.momentum)

# Run training loop.
# ...
```

You need to run a dummy forward before passing parameters to optimizers; otherwise optimizers cannot refer to lazily-initialized parameters. You will get a warning if you pass uninitialized lazy parameters to optimizers:

```
>>> model = ppe.nn.LazyLinear(None, 10)
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
/.../pytorch-pfn-extras/pytorch_pfn_extras/nn/modules/lazy.py:127: UserWarning:
    Use of uninitialized lazy parameter in Optimizer has been detected.
    Maybe you forgot to run forward before passing `module.parameters()` to the␣
↪optimizer?
```

* *Lazy Evaluation*

1.4.2 Config

Basic

```
from pytorch_pfn_extras.config import Config
import yaml
pre_eval_config = yaml.load('''
foo:
  bar: 'bar_value'
  ls:
    - 'first'
    - key0: 'value0'
      key1: 'value1'
baz: 'baz_value'
''')
config = Config(pre_eval_config)
```

Accessing config values:

```
print(config['/foo/ls/0'])
# 'first'
print(config['/foo/ls/1/key0'])
# 'value0'
print(config['/foo/ls'])
# ['first', {'key0': 'value0', 'key1': 'value1'}]
print(config['/baz'])
# 'baz_value'
```

Substitution

Callable Substitution

You could replace a value as the return value of a callable.

- `types` is an additional input to `Config`. `types` is a mapping from a callable's name to the actual callable.
- A sub-dictionary containing the key `type` invokes callable substitution.

```
pre_eval_config = yaml.load('''
name:
  type: concat
  x0: 'First'
  x1: 'Last'
''')

types = {
  'concat': lambda x0, x1: x0 + ' ' + x1
}

config = Config(pre_eval_config, types)
```

(continues on next page)

(continued from previous page)

```
# the value returned by
# concat(x0='First', x1='Last')
print(config['/name'])
# 'First Last'
```

Nested

```
pre_eval_config = yaml.load('''
name:
  type: concat
  x0: 'First'
  x1:
    type: concat
    x0: 'Middle'
    x1: 'Last'
''')
types = {
  'concat': lambda x0, x1: x0 + ' ' + x1
}
config = Config(pre_eval_config, types)
print(config['/name'])
# First Middle Last
```

Class

```
pre_eval_config = yaml.load('''
dataset:
  type: Dataset
  n_class: 10
''')

class Dataset(object):

    def __init__(self, n_class):
        self.n_class = n_class

types = {
  'Dataset': Dataset,
}

config = Config(pre_eval_config, types)
print(isintance(config['/dataset'], Dataset))
# True
```

Substitution by Path

Absolute

`@/absolute/path` is replaced by the value at `/absolute/path`.

```
pre_eval_config = yaml.load('''
foo: 'FOO'
boo:
  baz: '@/foo'
''')
config = Config(pre_eval_config)
print(config['/boo/baz'])
# FOO
```

Relative

Relative path is also possible using `@relative/path`.

```
pre_eval_config = yaml.load('''
foo: 'FOO'
boo:
  baz: '@../foo'
''')
config = Config(pre_eval_config)
print(config['/boo/baz'])
# FOO
```

Substitution by Attribute

`@/path/to/obj.attr_name` is replaced by:

1. Use substitution by path to get an object at `/path/to/obj`.
2. Replace the config value by `getattr(obj, attr_name)`, where `obj` is obtained at step 1.

```
pre_eval_config = yaml.load('''
dataset:
  type: Dataset
  n_class: 10
n_data: '@/dataset.n_data'
''')

class Dataset(object):

    def __init__(self, n_class):
        self.n_class = n_class
        self.n_data = 4

types = {
  'Dataset': Dataset,
```

(continues on next page)

(continued from previous page)

```
}

config = Config(pre_eval_config, types)
print(config['/n_data'])
# 4
```

Default Value by Path Substitution

`customize_type` is a decorator that sets default argument values by path substitution.

```
from pytorch_pfn_extras.config import customize_type

pre_eval_config = yaml.load('''
dataset:
  type: Dataset
n_class: 5
''')

# If n_class is not passed, the value would be config['/n_class'].
# Both absolute and relative paths are allowed.
@customize_type(n_class='/n_class')
class Dataset(object):

    def __init__(self, n_class):
        self.n_class = n_class

types = {
  'Dataset': Dataset,
}

config = Config(pre_eval_config, types)
print(config['/dataset'].n_class)
# 5
```

Ignore Substitution

Access using `config['!/path']` instead of `config['/path']`.

```
pre_eval_config = yaml.load('''
name:
  type: concat
  x0: 'First'
  x1: 'Last'
''')

types = {
  'concat': lambda x0, x1: x0 + ' ' + x1
}
```

(continues on next page)

(continued from previous page)

```
config = Config(pre_eval_config, types)
print(config['!/name'])
# {'type': 'concat', 'x0': 'First', 'x1': 'Last'}
```

Lazy Evaluation

Callable substitution is lazily executed. This means that callables that are not dependent on the accesed value do not get executed.

```
pre_eval_config = yaml.load('''
foo:
  - type: f0
  - '@/bar'
bar:
  type: f1
baz:
  type: f2
''')

def f0():
    print('f0 called')
    return 'f0_return'

def f1():
    print('f1 called')
    return 'f1_return'

def f2():
    print('f2 called')
    return 'f2_return'

types = {
  'f0': f0,
  'f1': f1,
  'f2': f2,
}

config = Config(pre_eval_config, types)
config['/foo']  # f2 does not get called
# f0 called
# f1 called
```

1.4.3 pytorch_pfn_extras.onnx

Extensions to `torch.onnx.export`.

Installation

```
pip3 install "pytorch-pfn-extras[onnx]"
```

Or

1. Install pytorch-pfn-extras normally
2. Install onnx with `pip install onnx==1.7.0`

API

pytorch_pfn_extras.onnx.export_testcase

Instead of specifying file name in `torch.onnx.export`, `pytorch_pfn_extra.onnx.export_testcase` specifies directory to output ONNX model and test case in/out.

```
import torch
import torch.nn as nn
model = nn.Sequential(nn.Linear(5, 10, bias=False))
x = torch.zeros((2, 5))

import pytorch_pfn_extras.onnx as tou
tou.export_testcase(model, x, '/path/to/output')
```

Directory structure with following will be generated to `/path/to/output`:

```
$ tree /path/to/output
/path/to/output
├── meta.json
├── model.onnx
└── test_data_set_0
    ├── input_0.pb
    └── output_0.pb
```

- This directory structure format is inspired by ONNX official test data set: (Example: node). PyTorch's ONNX tests use this format too. (Reference: export_onnx_tests_generator.py)
  - There are scripts in chainer-compiler/utils to run inference in major runtime with the directory structure. For example to inference with ONNXRuntime, run `$ python run_onnx_onnxruntime.py /path/to/output` to use `input_N.pb` as input and compare numerically with its output `output_N.pb`(N is the index of test case).
- By default `meta.json` is generated too to track git infos, date times, etc. Add `metadata=False` argument to suppress this.

`out_grad` option

If `out_grad=True` is specified gradient will be dumped too, which is useful for debugging backward. `gradient_N.pb` and `gradient_input_N.pb` would be dumped to test case directory with in/out data. `gradient_input_N.pb` is the initial value of backward, and it's default value is ones tensor with same shape of output. Use `out_grad` to specify custom initial value (`torch.Tensor` type) for it.

```
model = nn.Sequential(nn.Linear(5, 10, bias=False))
x = torch.zeros((2, 5))

import pytorch_pfn_extras.onnx as tou
tou.export_testcase(model, x, '/path/to/output', out_grad=True)
```

```
$ tree /path/to/output
/path/to/output
├── meta.json
├── model.onnx
└── test_data_set_0
    ├── gradient_0.pb
    ├── gradient_input_0.pb
    ├── input_0.pb
    └── output_0.pb
```

`model_overwrite` option

Use `model_overwrite` option to create multiple data set like following:

```
import pytorch_pfn_extras.onnx as tou
tou.export_testcase(model, x1, '/path/to/output')
tou.export_testcase(model, x2, '/path/to/output', model_overwrite=False)
```

Following is the generated test cases of the above. `test_data_set_0` is the input`x1` and is its output, `test_data_set_1` is the input `x2` and its output.

```
$ tree /path/to/output
├── meta.json
├── model.onnx
├── test_data_set_0
│   ├── input_0.pb
│   └── output_0.pb
└── test_data_set_1
    ├── input_0.pb
    └── output_0.pb
```

`strip_large_tensor_data` option

This option strips large tensor in dumped files which is useful to reduce file size in usage such as benchmarking. Not only `model.onnx`, in/out, gradient data would be affected too. `large_tensor_threshold` could be used to specify threshold of large tensor size.

```
import torchvision
model = torchvision.models.resnet50(pretrained=True)
x = torch.zeros((1, 3, 224, 224))

import pytorch_pfn_extras.onnx as tou
tou.export_testcase(model, x, '/path/to/output')
tou.export_testcase(model, x, '/path/to/output2', strip_large_tensor_data=True)
```

```
$ ls -lh /path/to/output/model.onnx
-rwxrwxrwx 1 user user 98M Jun 24 23:34 /path/to/output/model.onnx
$ ls -lh /path/to/output2/model.onnx
-rwxrwxrwx 1 user user 64K Jun 24 23:34 /path/to/output2/model.onnx
```

This feature could be called from CLI:

```
$ python -m pytorch_pfn_extras.onnx.strip_large_tensor resnet50.onnx --out_onnx_path␣
↪resnet50_slim.onnx
$ ls -lh
-rwxrwxrwx 1 user user 98M Jun 30 09:13 resnet50.onnx
-rwxrwxrwx 1 user user 64K Jun 30 09:16 resnet50_slim.onnx
```

See `$ python -m pytorch_pfn_extras.onnx.strip_large_tensor -h` for help

Notes:

If an ONNX runtime does not support no `raw_data` tensor, `unstrip_tensor.py` will resolve. See `$ python -m pytorch_pfn_extras.onnx.unstrip_tensor -h` for help

`pytorch_pfn_extras.onnx.export`

Function with same interface like `torch.onnx.export`. Unlike `torch.onnx.export`, you can use annotation feature (described below), `strip_large_tensor_data` options, or other `torch.onnx` extensions.

- `strip_large_tensor_data`: Same as `export_testcase`. Useful reducing file sizes.
- `return_output`: Returns output value of model execution. Note: Most output type would be `torch.Tensor`(not `onnx.TensorProto`)

```
model = nn.Sequential(nn.Linear(5, 10, bias=False))
x = torch.zeros((2, 5))

import io, onnx
bytesio = io.BytesIO()
pytorch_pfn_extras.onnx.export(model, x, bytesio)
onnx_proto = onnx.load(io.BytesIO(bytesio.getvalue()))
```

annotate

Feature to add custom ONNX attribute to specified `nn.Module`.

Notes:

- Annotated ONNX would be invalid ONNX format that doesn't pass check of onnx.checker.check_model.
- Only valid with `pytorch_pfn_extras.onnx.export_testcase` or `pytorch_pfn_extras.onnx.export` export.
- **Only** the first ONNX node of modules like `nn.Linear`, `nn.GroupNorm`, etc. with multiple ONNX node would be annotated
  - For example `nn.Linear` with bias is split to `MatMul` -> `Add` graph. Only `MatMul` would be annotated. This is same in `apply_annotation` (described later) too.
- Use `apply_annotation` instead when the annotation target isn't `nn.Module`.

```
import pytorch_pfn_extras.onnx as tou

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.conv = nn.Conv2d(6, 9, 3)
        self.conv2 = nn.Conv2d(9, 12, 3)
        self.linear = nn.Linear(28, 20)
        self.linear2 = nn.Linear(20, 15)

    def forward(self, x):
        h = self.conv(x)
        with tou.annotate(key='value'):
            h = self.conv2(h)
            h = self.linear(h)
        h = self.linear2(h)
        return h

model = Net()
x = torch.randn((1, 6, 32, 32))
tou.export_testcase(model, x, '/path/to/output')
onnx_proto = onnx.load(os.path.join('/path/to/output, 'model.onnx'))
print(onnx.helper.printable_graph(onnx_proto.graph))
```

```
graph torch-jit-export (
  %input.1[FLOAT, 1x6x32x32]
) initializers (
  %17[FLOAT, 28x20]
  %18[FLOAT, 20x15]
  %conv.bias[FLOAT, 9]
  %conv.weight[FLOAT, 9x6x3x3]
  %conv2.bias[FLOAT, 12]
  %conv2.weight[FLOAT, 12x9x3x3]
  %linear.bias[FLOAT, 20]
  %linear2.bias[FLOAT, 15]
) {
```

(continues on next page)

(continued from previous page)

```
  %9 = Conv[dilations = [1, 1], group = 1, kernel_shape = [3, 3], pads = [0, 0, 0, 0],␣
↪strides = [1, 1]](%input.1, %conv.weight, %conv.bias)
  %10 = Conv[dilations = [1, 1], group = 1, kernel_shape = [3, 3], key = 'value', pads =␣
↪[0, 0, 0, 0], strides = [1, 1]](%9, %conv2.weight, %conv2.bias)
  %12 = MatMul[key = 'value'](%10, %17)
  %13 = Add(%12, %linear.bias)
  %15 = MatMul(%13, %18)
  %16 = Add(%15, %linear2.bias)
  return %16
}
```

In above example `%10 = Conv` and `%12 = MatMul` has `key='value'` attribute annotated.

apply_annotation

This annotates function call instead of annotating it with `with`.

The `annotate` target is `nn.Module`, so `torch.nn.functional` couldn't be annotated

```
import torch.nn.functional as F
import pytorch_pfn_extras.onnx as tou

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.conv = nn.Conv2d(6, 9, 3)
        self.conv2 = nn.Conv2d(9, 12, 3)
        self.linear = nn.Linear(28, 20)
        self.linear2 = nn.Linear(20, 15)

    def forward(self, x):
        h = self.conv(x)
        with tou.annotate(key='value'):
            h = self.conv2(h)
            h = F.relu(h)
            h = self.linear(h)
        h = self.linear2(h)
        return h

model = Net()
x = torch.randn((1, 6, 32, 32))
tou.export_testcase(model, x, '/path/to/output')
onnx_proto = onnx.load(os.path.join('/path/to/output', 'model.onnx'))
print(onnx.helper.printable_graph(onnx_proto.graph))
```

```
graph torch-jit-export (
  %input.1[FLOAT, 1x6x32x32]
) initializers (
  %18[FLOAT, 28x20]
  %19[FLOAT, 20x15]
```

(continues on next page)

(continued from previous page)

```
  %conv.bias[FLOAT, 9]
  %conv.weight[FLOAT, 9x6x3x3]
  %conv2.bias[FLOAT, 12]
  %conv2.weight[FLOAT, 12x9x3x3]
  %linear.bias[FLOAT, 20]
  %linear2.bias[FLOAT, 15]
) {
  %9 = Conv[dilations = [1, 1], group = 1, kernel_shape = [3, 3], pads = [0, 0, 0, 0],␣
↪strides = [1, 1]](%input.1, %conv.weight, %conv.bias)
  %10 = Conv[dilations = [1, 1], group = 1, kernel_shape = [3, 3], key = 'value', pads =␣
↪[0, 0, 0, 0], strides = [1, 1]](%9, %conv2.weight, %conv2.bias)
  %11 = Relu(%10)
  %13 = MatMul[key = 'value'](%11, %18)
  %14 = Add(%13, %linear.bias)
  %16 = MatMul(%14, %19)
  %17 = Add(%16, %linear2.bias)
  return %17
}
```

`%10 = Conv` and `%13 = MatMul` has `key='value'` attribute but `%11 = Relu` hasn't. By using `apply_annotation` all node in the function is annotated.

```
import pytorch_pfn_extras.onnx as tou

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.conv = nn.Conv2d(6, 9, 3)
        self.conv2 = nn.Conv2d(9, 12, 3)
        self.linear = nn.Linear(28, 20)
        self.linear2 = nn.Linear(20, 15)

    def forward(self, x):
        h = self.conv(x)
        def _f(x):
            h = self.conv2(x)
            h = F.relu(h)
            h = self.linear(h)
            return h
        h = tou.apply_annotation(_f, h, key='value')
        h = self.linear2(h)
        return h

model = Net()
x = torch.randn((1, 6, 32, 32))
tou.export_testcase(model, x, '/path/to/outout')
onnx_proto = onnx.load(os.path.join('/path/to/output', 'model.onnx'))
print(onnx.helper.printable_graph(onnx_proto.graph))
```

```
graph torch-jit-export (
  %input.1[FLOAT, 1x6x32x32]
```

(continues on next page)

(continued from previous page)

```
) initializers (
  %18[FLOAT, 28x20]
  %19[FLOAT, 20x15]
  %conv.bias[FLOAT, 9]
  %conv.weight[FLOAT, 9x6x3x3]
  %conv2.bias[FLOAT, 12]
  %conv2.weight[FLOAT, 12x9x3x3]
  %linear.bias[FLOAT, 20]
  %linear2.bias[FLOAT, 15]
) {
  %9 = Conv[dilations = [1, 1], group = 1, kernel_shape = [3, 3], pads = [0, 0, 0, 0],␣
↪strides = [1, 1]](%input.1, %conv.weight, %conv.bias)
  %10 = Conv[dilations = [1, 1], group = 1, kernel_shape = [3, 3], key = 'value', pads =␣
↪[0, 0, 0, 0], strides = [1, 1]](%9, %conv2.weight, %conv2.bias)
  %11 = Relu[key = 'value'](%10)
  %13 = MatMul[key = 'value'](%11, %18)
  %14 = Add(%13, %linear.bias)
  %16 = MatMul(%14, %19)
  %17 = Add(%16, %linear2.bias)
  return %17
}
```

Now `%11 = Relu` is annotated with `key='value'` attribute too.

scoped_anchor

This annotates scope's beginning and end of one or modules by adding Anchor node. Node would be named `Anchor_N_start` or `Anchor_N_end` (N is a index) and with op_type Identity.

- Adding custom parameter would add ONNX attribute and this will generate invalid ONNX in checker.
- Use this with `pytorch_pfn_extras.onnx.export_testcase` or `pytorch_pfn_extras.onnx.export`.
- When scope has multiple input/output only first input/output will get Anchor node added.
- `N` of node name is the index of pair beginning/end Anchor node like `Anchor_0_start`, `Anchor_0_end`.

```
import pytorch_pfn_extras.onnx as tou

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.conv = nn.Conv2d(6, 9, 3)
        self.conv2 = nn.Conv2d(9, 12, 3)
        self.linear = nn.Linear(28, 20)
        self.linear2 = nn.Linear(20, 15)

    def forward(self, x):
        h = self.conv(x)
        with tou.scoped_anchor(key='value'):
            h = self.conv2(h)
            h = self.linear(h)
```

(continues on next page)

(continued from previous page)

```
        h = self.linear2(h)
        return h

    def forward(self, x):
        with annotate(key='value'):
            return self.add(x)

model = Net()
x = torch.randn((1, 6, 32, 32))
out_dir = tou.export_testcase(model, x, '/path/to/output')
onnx_proto = onnx.load(os.path.join('/path/to/output', 'model.onnx'))
print(onnx.helper.printable_graph(onnx_proto.graph))
```

```
graph torch-jit-export (
  %input.1[FLOAT, 1x6x32x32]
) initializers (
  %23[FLOAT, 28x20]
  %24[FLOAT, 20x15]
  %conv.bias[FLOAT, 9]
  %conv.weight[FLOAT, 9x6x3x3]
  %conv2.bias[FLOAT, 12]
  %conv2.weight[FLOAT, 12x9x3x3]
  %linear.bias[FLOAT, 20]
  %linear2.bias[FLOAT, 15]
) {
  %9 = Conv[dilations = [1, 1], group = 1, kernel_shape = [3, 3], pads = [0, 0, 0, 0],␣
↪strides = [1, 1]](%input.1, %conv.weight, %conv.bias)
  %11 = Identity[key = 'value'](%9)
  %12 = Conv[dilations = [1, 1], group = 1, kernel_shape = [3, 3], pads = [0, 0, 0, 0],␣
↪strides = [1, 1]](%11, %conv2.weight, %conv2.bias)
  %16 = MatMul(%12, %23)
  %17 = Add(%16, %linear.bias)
  %19 = Identity[key = 'value'](%17)
  %21 = MatMul(%19, %24)
  %22 = Add(%21, %linear2.bias)
  return %22
}
```

`%11 = Identity` (node name = `Anchor_0_start`) and `%19 = Identity` (node name = `Anchor_0_end`) is added. `key='value'` is added as ONNX attribute.

non-nn.Module

The target of scope is only `nn.Module`. You can add adding sub `nn.Module` instead, if scope bound doesn't match `nn.Module`.

```
import pytorch_pfn_extras.onnx as tou

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
```

(continues on next page)

(continued from previous page)

```
        class _Net(nn.Module):
            def forward(self, x):
                return x + torch.ones((1,))
        self.add = _Net()

    def forward(self, x):
        with tou.scoped_anchor(key='value'):
            return self.add(x)

model = Net()
x = torch.randn((1, 6, 32, 32))
out_dir = tou.export_testcase(model, x, '/path/to/output')
onnx_proto = onnx.load(os.path.join('/path/to/output', 'model.onnx'))
print(onnx.helper.printable_graph(onnx_proto.graph))
```

```
graph torch-jit-export (
  %x.1[FLOAT, 1x6x32x32]
) {
  %2 = Identity[key = 'value'](%x.1)
  %3 = Constant[value = <Tensor>]()
  %4 = Add(%2, %3)
  %6 = Identity[key = 'value'](%4)
  return %6
}
```

Or you can use `anchor` (described below) instead.

`anchor` (Future work)

Inserts Anchor node per each arbitrarily position of `nn.Module` . Node name would be Anchor and op_type would be `Identity`.

- Note: adding extra parameter would make extended ONNX format because it would be attribute.
- Please use it with `pytorch_pfn_extras.onnx.export_testcase` or `pytorch_pfn_extras.onnx.export`.

1.4.4 CUDA (CuPy Interoperability)

- `pytorch_pfn_extras.cuda.stream(stream)`
  - Context-manager that selects a given stream. This context manager also changes the CuPy's default stream if CuPy is available. When CuPy is not available, the functionality is the same as the PyTorch's counterpart, `torch.cuda.stream()`.
- `pytorch_pfn_extras.cuda.use_torch_mempool_in_cupy()`
  - Use PyTorch's memory pool in CuPy. If you want to use PyTorch's memory pool and non-default CUDA streams, streams must be created and managed using PyTorch (using `torch.cuda.Stream()` and `pytorch_pfn_extras.cuda.stream(stream)`). This feature requires CuPy v8.0+ and PyTorch v1.5+.
- `pytorch_pfn_extras.cuda.use_default_mempool_in_cupy()`
  - Use CuPy's default memory pool in CuPy.

- `pytorch_pfn_extras.from_ndarray(ndarray)`
  - Creates a Tensor from NumPy/CuPy ndarray.
- `pytorch_pfn_extras.as_ndarray(tensor)`
  - Creates a NumPy/CuPy ndarray from Tensor.
- `pytorch_pfn_extras.get_xp(tensor_device_or_ndarray)`
  - Returns `numpy` or `cupy` module for the given object.
- `pytorch_pfn_extras.as_numpy_dtype(torch_dtype)`
  - Returns NumPy dtype for the given torch dtype.
- `pytorch_pfn_extras.from_numpy_dtype(numpy_dtype)`
  - Returns torch dtype for the given NumPy dtype.

CHAPTER

TWO

API REFERENCE

- genindex

2.1 Package

pytorch_pfn_extras

2.1.1 pytorch_pfn_extras

Functions

pytorch_pfn_extras.as_ndarray(tensor)	Creates a *numpy.ndarray* or *cupy.ndarray* from *torch.Tensor*.
pytorch_pfn_extras.as_numpy_dtype(torch_dtype)	Returns NumPy dtype for the given PyTorch dtype.
pytorch_pfn_extras.compile(module[, ...])	Compiles a module and an optimizer in a single graph using the provided backend.
pytorch_pfn_extras.from_ndarray(ndarray)	Creates a *torch.Tensor* from a *numpy.ndarray* or *cupy.ndarray*.
pytorch_pfn_extras.from_numpy_dtype(numpy_dtype)	Returns PyTorch dtype for the given NumPy dtype.
pytorch_pfn_extras.get_xp(obj)	Returns a module of ndarray implementation (*numpy* or *cupy*) for the given *obj*.
pytorch_pfn_extras.map(func, iterable[, ...])	
pytorch_pfn_extras.requires(version[, package])	
pytorch_pfn_extras.to(module_or_tensor, ...)	A function to transfer the given object to the given device.

pytorch_pfn_extras.as_ndarray

pytorch_pfn_extras.**as_ndarray**(*tensor*)

Creates a *numpy.ndarray* or *cupy.ndarray* from *torch.Tensor*.

This method returns a tensor as a NumPy or CuPy ndarray depending on where the given *tensor* resides in. The *tensor* and the returned *ndarray* share the same underlying storage. Changes to the tensor will be reflected in the *ndarray* and vice versa. Note that changes made to *ndarray* cannot be tracked in the computational graph.

Parameters
tensor (*Tensor*) –

Return type
Any

pytorch_pfn_extras.as_numpy_dtype

pytorch_pfn_extras.**as_numpy_dtype**(*torch_dtype*)

Returns NumPy dtype for the given PyTorch dtype.

Parameters
torch_dtype (*dtype*) – PyTorch's dtype object.

Returns
NumPy type object.

Return type
Any

pytorch_pfn_extras.compile

pytorch_pfn_extras.**compile**(*module*, *optimizer=None*, *backend=None*)

Compiles a module and an optimizer in a single graph using the provided backend.

Note: The backend object needs to be a callable accepting a `torch.fx.GraphModule` and a list of `torch.Tensor` and return a `Callable` as specified by https://pytorch.org/docs/2.0/dynamo/custom-backends.html#custom-backends

---

Note: Modules that are split in multiple graphs are not supported. `torch.compiled` is called with the `fullgraph=True` argument.

Parameters

- **module** (*Module*) – torch.nn.Module to be compiled
- **optimizer** (*Optional[Optimizer]*) – Optimizer object associated to the module. It will be traced and its operations included in the module graph. Some dry run operations may be performed to fully initialize the optimizer status.
- **backend** (*optional*) – Object to process the graph and compile it for custom devices, will use PyTorch dynamo by default if not specified.

Return type
Callable[[...], *Any*]

pytorch_pfn_extras.from_ndarray

pytorch_pfn_extras.**from_ndarray**(*ndarray*)

Creates a *torch.Tensor* from a *numpy.ndarray* or *cupy.ndarray*.

Unlike *torch.from_numpy*, this method may make a copy when needed, e.g. when the given *ndarray* contains the negative strides which is not supported by PyTorch.

Parameters
ndarray (*Any*) –

Return type
Tensor

pytorch_pfn_extras.from_numpy_dtype

pytorch_pfn_extras.**from_numpy_dtype**(*numpy_dtype*)

Returns PyTorch dtype for the given NumPy dtype.

Parameters
numpy_dtype (*Any*) – NumPy's dtype object.

Returns
PyTorch type object.

Return type
dtype

pytorch_pfn_extras.get_xp

pytorch_pfn_extras.**get_xp**(*obj*)

Returns a module of ndarray implementation (*numpy* or *cupy*) for the given *obj*.

The *obj* can be *torch.Tensor*, *torch.device* or NumPy/CuPy *ndarray*.

Parameters
obj (*Union[Any, Tensor]*) –

Return type
Any

pytorch_pfn_extras.map

pytorch_pfn_extras.**map**(*func*, *iterable*, *out_keys=None*, *device='cpu'*)

Parameters

- **func** (*Callable[[Any], Any]*) –
- **iterable** (*Sequence[Any]*) –
- **out_keys** (*Optional[Set[str]]*) –
- **device** (*Any*) –

Return type
Sequence[*Any*]

pytorch_pfn_extras.requires

pytorch_pfn_extras.**requires**(*version*, *package='torch'*)

Parameters

- **version** (*str*) –
- **package** (*str*) –

Return type
bool

pytorch_pfn_extras.to

pytorch_pfn_extras.**to**(*module_or_tensor*, *device*, *, *options=None*, *runtime_class=None*, *config=None*)

A function to transfer the given object to the given device.

If PyTorch's device type is given as the `device` argument, the behavior of this function is equivalent to `module_or_tensor.to(module_or_tensor, device)`.

Otherwise, this function uses the **Runtime** mechanism. This function looks for the Runtime for the device from the RuntimeRegistry and delegates the actual transfer operation to it.

See also the documentation of `ppe.runtime.BaseRuntime` for details.

Parameters

- **module_or_tensor** (*torch.nn.Module or torch.Tensor*) – An object to be transferred.
- **device** (*torch.device or str*) – The device that the input object is transferred to.
- **options** (*dict, optional*) – An options of dictionary type that is passed to `runtime_class.__init__` as an argument.
- **runtime_class** (*Optional[Type[BaseRuntime]]*) – A runtime class inherited from *BaseRuntime* class. If `None`, a runtime class is automatically selected based on the `device` argument from the runtime registry.
- **config** (*dict, optional*) – DEPRECATED. Use *options*.

Returns
A *torch.Tensor* with the specified device.

Return type
ModuleOrTensor

Modules

pytorch_pfn_extras.config

Functions

pytorch_pfn_extras.config.customize_type

pytorch_pfn_extras.config.**customize_type**(***default_kwargs*)

Parameters
default_kwargs (*Any*) –

Return type
Callable[[*Callable*[[...], *Any*]], *Callable*[[...], *Any*]]

Classes

pytorch_pfn_extras.config.Config(config[, types])

pytorch_pfn_extras.config.Config

class pytorch_pfn_extras.config.**Config**(*config*, *types=None*)

Bases: object

Methods

__init__(config[, types])
load_path(path, *[, loader, types])
update_via_args(args)

Parameters

- **config** (*Any*) –
- **types** (*Optional[Mapping[str, Callable[[...], Any]]]*) –

__init__(*config*, *types=None*)

Parameters

- **config** (*Any*) –
- **types** (*Optional[Mapping[str, Callable[[...], Any]]]*) –

Return type
None

classmethod load_path(*path*, ***, *loader=None*, *types=None*)

Parameters

- **path** (*str*) –
- **loader** (*Optional[Callable[[str], Any]]*) –
- **types** (*Optional[Mapping[str, Callable[[...], Any]]]*) –

Return type
Config

`update_via_args`(*args*)

Parameters
args (*Sequence[Tuple[str, Any]]*) –

Return type
None

pytorch_pfn_extras.config_types

Functions

pytorch_pfn_extras.config_types.load_path_with_optuna_types(...)
pytorch_pfn_extras.config_types.optuna_types(trial)

pytorch_pfn_extras.config_types.load_path_with_optuna_types

pytorch_pfn_extras.config_types.**load_path_with_optuna_types**(*path*, *trial*, *loader=None*, *types=None*)

Parameters

- **path** (*str*) –
- **trial** (*optuna.trial.Trial*) –
- **loader** (*Optional[Callable[[str], Any]]*) –
- **types** (*Optional[Dict[str, Callable[[...], Any]]]*) –

Return type
Config

pytorch_pfn_extras.config_types.optuna_types

pytorch_pfn_extras.config_types.**optuna_types**(*trial*)

Parameters
trial (*optuna.trial.Trial*) –

Return type
Dict[str, *Any*]

pytorch_pfn_extras.cuda

Functions

pytorch_pfn_extras.cuda.stream(stream)	Context-manager that selects a given stream.
pytorch_pfn_extras.cuda.use_default_mempool_in_cupy()	Use the default memory pool in CuPy.
pytorch_pfn_extras.cuda.use_torch_mempool_in_cupy()	Use the PyTorch memory pool in CuPy.

pytorch_pfn_extras.cuda.stream

pytorch_pfn_extras.cuda.**stream**(*stream*)

Context-manager that selects a given stream.

This context manager also changes the CuPy's default stream if CuPy is available. When CuPy is not available, the functionality is the same as the PyTorch's counterpart, *torch.cuda.stream()*.

Parameters
stream (*Optional[Stream]*) –

Return type
Generator[None, None, None]

pytorch_pfn_extras.cuda.use_default_mempool_in_cupy

pytorch_pfn_extras.cuda.**use_default_mempool_in_cupy**()

Use the default memory pool in CuPy.

Return type
None

pytorch_pfn_extras.cuda.use_torch_mempool_in_cupy

pytorch_pfn_extras.cuda.**use_torch_mempool_in_cupy**()

Use the PyTorch memory pool in CuPy.

If you want to use PyTorch's memory pool and non-default CUDA streams, streams must be created and managed using PyTorch (using *torch.cuda.Stream()* and *pytorch_pfn_extras.cuda.stream(stream)*).

Return type
None

pytorch_pfn_extras.dataloaders

Classes

pytorch_pfn_extras.dataloaders.DataLoader(dataset)	Data loader.

pytorch_pfn_extras.dataloaders.DataLoader

class pytorch_pfn_extras.dataloaders.**DataLoader**(*dataset*, *batch_size=1*, *shuffle=None*, *sampler=None*, *batch_sampler=None*, *num_workers=0*, *collate_fn=None*, *pin_memory=False*, *drop_last=False*, *timeout=0*, *worker_init_fn=None*, *multiprocessing_context=None*, *generator=None*, ***, *prefetch_factor=None*, *persistent_workers=False*, *pin_memory_device=''*)

Bases: `Generic[T_co]`

Data loader. Combines a dataset and a sampler, and provides an iterable over the given dataset.

The `DataLoader` supports both map-style and iterable-style datasets with single- or multi-process loading, customizing loading order and optional automatic batching (collation) and memory pinning.

See `torch.utils.data` documentation page for more details.

Parameters

- **dataset** (*Dataset*) – dataset from which to load the data.
- **batch_size** (*int, optional*) – how many samples per batch to load (default: `1`).
- **shuffle** (*bool, optional*) – set to `True` to have the data reshuffled at every epoch (default: `False`).
- **sampler** (*Sampler or Iterable, optional*) – defines the strategy to draw samples from the dataset. Can be any `Iterable` with `__len__` implemented. If specified, `shuffle` must not be specified.
- **batch_sampler** (*Sampler or Iterable, optional*) – like *sampler*, but returns a batch of indices at a time. Mutually exclusive with *batch_size*, `shuffle`, *sampler*, and *drop_last*.
- **num_workers** (*int, optional*) – how many subprocesses to use for data loading. `0` means that the data will be loaded in the main process. (default: `0`)
- **collate_fn** (*Callable, optional*) – merges a list of samples to form a mini-batch of Tensor(s). Used when using batched loading from a map-style dataset.
- **pin_memory** (*bool, optional*) – If `True`, the data loader will copy Tensors into device/CUDA pinned memory before returning them. If your data elements are a custom type, or your `collate_fn` returns a batch that is a custom type, see the example below.
- **drop_last** (*bool, optional*) – set to `True` to drop the last incomplete batch, if the dataset size is not divisible by the batch size. If `False` and the size of dataset is not divisible by the batch size, then the last batch will be smaller. (default: `False`)
- **timeout** (*numeric, optional*) – if positive, the timeout value for collecting a batch from workers. Should always be non-negative. (default: `0`)

- **worker_init_fn** (*Callable, optional*) – If not None, this will be called on each worker subprocess with the worker id (an int in [0, num_workers - 1]) as input, after seeding and before data loading. (default: None)
- **multiprocessing_context** (*str or multiprocessing.context.BaseContext, optional*) – If None, the default multiprocessing context of your operating system will be used. (default: None)
- **generator** (*torch.Generator, optional*) – If not None, this RNG will be used by RandomSampler to generate random indexes and multiprocessing to generate base_seed for workers. (default: None)
- **prefetch_factor** (*int, optional, keyword-only arg*) – Number of batches loaded in advance by each worker. 2 means there will be a total of 2 * num_workers batches prefetched across all workers. (default value depends on the set value for num_workers. If value of num_workers=0 default is None. Otherwise, if value of num_workers > 0 default is 2).
- **persistent_workers** (*bool, optional*) – If True, the data loader will not shut down the worker processes after a dataset has been consumed once. This allows to maintain the workers *Dataset* instances alive. (default: False)
- **pin_memory_device** (*str, optional*) – the device to *pin_memory* to if pin_memory is True.

Warning: If the spawn start method is used, worker_init_fn cannot be an unpicklable object, e.g., a lambda function. See multiprocessing-best-practices on more details related to multiprocessing in PyTorch.

Warning: len(dataloader) heuristic is based on the length of the sampler used. When *dataset* is an IterableDataset, it instead returns an estimate based on len(dataset) / batch_size, with proper rounding depending on *drop_last*, regardless of multi-process loading configurations. This represents the best guess PyTorch can make because PyTorch trusts user *dataset* code in correctly handling multi-process loading to avoid duplicate data.

However, if sharding results in multiple workers having incomplete last batches, this estimate can still be inaccurate, because (1) an otherwise complete batch can be broken into multiple ones and (2) more than one batch worth of samples can be dropped when *drop_last* is set. Unfortunately, PyTorch can not detect such cases in general.

See `Dataset Types`_ for more details on these two types of datasets and how IterableDataset interacts with `Multi-process data loading`_.

Warning: See reproducibility, and dataloader-workers-random-seed, and data-loading-randomness notes for random seed related questions.

Methods

__init__(dataset[, batch_size, shuffle, ...])
check_worker_number_rationality()

Attributes

multiprocessing_context
dataset
batch_size
num_workers
pin_memory
drop_last
timeout
sampler
pin_memory_device
prefetch_factor

__init__(*dataset*, *batch_size=1*, *shuffle=None*, *sampler=None*, *batch_sampler=None*, *num_workers=0*, *collate_fn=None*, *pin_memory=False*, *drop_last=False*, *timeout=0*, *worker_init_fn=None*, *multiprocessing_context=None*, *generator=None*, ***, *prefetch_factor=None*, *persistent_workers=False*, *pin_memory_device=''*)

Parameters

- **dataset** (*Dataset[T_co]*) –
- **batch_size** (*Optional[int]*) –
- **shuffle** (*Optional[bool]*) –
- **sampler** (*Optional[Union[Sampler, Iterable]]*) –
- **batch_sampler** (*Optional[Union[Sampler[List], Iterable[List]]]*) –
- **num_workers** (*int*) –
- **collate_fn** (*Optional[Callable[[List[T]], Any]]*) –
- **pin_memory** (*bool*) –
- **drop_last** (*bool*) –
- **timeout** (*float*) –

- **worker_init_fn** (*Optional[Callable[[int], None]]*) –
- **prefetch_factor** (*Optional[int]*) –
- **persistent_workers** (*bool*) –
- **pin_memory_device** (*str*) –

batch_size: Optional[int]

check_worker_number_rationality()

dataset: *Dataset*[T_co]

drop_last: bool

property multiprocessing_context

num_workers: int

pin_memory: bool

pin_memory_device: str

prefetch_factor: Optional[int]

sampler: Union[Sampler, Iterable]

timeout: float

Modules

pytorch_pfn_extras.dataloaders.dataloader
pytorch_pfn_extras.dataloaders.utils

pytorch_pfn_extras.dataloaders.dataloader

Functions

pytorch_pfn_extras.dataloaders.dataloader.default_collate(batch)	Function that takes in a batch of data and puts the elements within the batch into a tensor with an additional outer dimension - batch size.
pytorch_pfn_extras.dataloaders.dataloader.default_convert(data)	Function that converts each NumPy array element into a `torch.Tensor`.
pytorch_pfn_extras.dataloaders.dataloader.get_worker_info()	Returns the information about the current `DataLoader` iterator worker process.

pytorch_pfn_extras.dataloaders.dataloader.default_collate

pytorch_pfn_extras.dataloaders.dataloader.**default_collate**(*batch*)

Function that takes in a batch of data and puts the elements within the batch into a tensor with an additional outer dimension - batch size. The exact output type can be a `torch.Tensor`, a *Sequence* of `torch.Tensor`, a Collection of `torch.Tensor`, or left unchanged, depending on the input type. This is used as the default function for collation when *batch_size* or *batch_sampler* is defined in `DataLoader`.

Here is the general input type (based on the type of the element within the batch) to output type mapping:

- `torch.Tensor` -> `torch.Tensor` (with an added outer dimension batch size)
- NumPy Arrays -> `torch.Tensor`
- *float* -> `torch.Tensor`
- *int* -> `torch.Tensor`
- *str* -> *str* (unchanged)
- *bytes* -> *bytes* (unchanged)
- *Mapping[K, V_i]* -> *Mapping[K, default_collate([V_1, V_2, …])]*
- *NamedTuple[V1_i, V2_i, …]* -> *NamedTuple[default_collate([V1_1, V1_2, …]), default_collate([V2_1, V2_2, …]), …]*
- *Sequence[V1_i, V2_i, …]* -> *Sequence[default_collate([V1_1, V1_2, …]), default_collate([V2_1, V2_2, …]), …]*

Parameters

batch – a single batch to be collated

Examples

```
>>> # xdoctest: +SKIP
>>> # Example with a batch of `int`s:
>>> default_collate([0, 1, 2, 3])
tensor([0, 1, 2, 3])
>>> # Example with a batch of `str`s:
>>> default_collate(['a', 'b', 'c'])
['a', 'b', 'c']
>>> # Example with `Map` inside the batch:
>>> default_collate([{'A': 0, 'B': 1}, {'A': 100, 'B': 100}])
{'A': tensor([  0, 100]), 'B': tensor([  1, 100])}
>>> # Example with `NamedTuple` inside the batch:
>>> Point = namedtuple('Point', ['x', 'y'])
>>> default_collate([Point(0, 0), Point(1, 1)])
Point(x=tensor([0, 1]), y=tensor([0, 1]))
>>> # Example with `Tuple` inside the batch:
>>> default_collate([(0, 1), (2, 3)])
[tensor([0, 2]), tensor([1, 3])]
>>> # Example with `List` inside the batch:
>>> default_collate([[0, 1], [2, 3]])
[tensor([0, 2]), tensor([1, 3])]
>>> # Two options to extend `default_collate` to handle specific type
>>> # Option 1: Write custom collate function and invoke `default_collate`
```

(continues on next page)

(continued from previous page)

```
>>> def custom_collate(batch):
...     elem = batch[0]
...     if isinstance(elem, CustomType):  # Some custom condition
...         return ...
...     else:  # Fall back to `default_collate`
...         return default_collate(batch)
>>> # Option 2: In-place modify `default_collate_fn_map`
>>> def collate_customtype_fn(batch, *, collate_fn_map=None):
...     return ...
>>> default_collate_fn_map.update(CustoType, collate_customtype_fn)
>>> default_collate(batch)  # Handle `CustomType` automatically
```

pytorch_pfn_extras.dataloaders.dataloader.default_convert

pytorch_pfn_extras.dataloaders.dataloader.**default_convert**(*data*)

Function that converts each NumPy array element into a `torch.Tensor`. If the input is a *Sequence*, *Collection*, or *Mapping*, it tries to convert each element inside to a `torch.Tensor`. If the input is not an NumPy array, it is left unchanged. This is used as the default function for collation when both *batch_sampler* and *batch_size* are NOT defined in `DataLoader`.

The general input type to output type mapping is similar to that of `default_collate()`. See the description there for more details.

Parameters

data – a single data point to be converted

Examples

```
>>> # xdoctest: +SKIP
>>> # Example with `int`
>>> default_convert(0)
0
>>> # Example with NumPy array
>>> default_convert(np.array([0, 1]))
tensor([0, 1])
>>> # Example with NamedTuple
>>> Point = namedtuple('Point', ['x', 'y'])
>>> default_convert(Point(0, 0))
Point(x=0, y=0)
>>> default_convert(Point(np.array(0), np.array(0)))
Point(x=tensor(0), y=tensor(0))
>>> # Example with List
>>> default_convert([np.array([0, 1]), np.array([2, 3])])
[tensor([0, 1]), tensor([2, 3])]
```

pytorch_pfn_extras.dataloaders.dataloader.get_worker_info

pytorch_pfn_extras.dataloaders.dataloader.**get_worker_info**()

Returns the information about the current DataLoader iterator worker process.

When called in a worker, this returns an object guaranteed to have the following attributes:

- id: the current worker id.
- num_workers: the total number of workers.
- seed: the random seed set for the current worker. This value is determined by main process RNG and the worker id. See DataLoader's documentation for more details.
- dataset: the copy of the dataset object in **this** process. Note that this will be a different object in a different process than the one in the main process.

When called in the main process, this returns None.

Note: When used in a worker_init_fn passed over to DataLoader, this method can be useful to set up each worker process differently, for instance, using worker_id to configure the dataset object to only read a specific fraction of a sharded dataset, or use seed to seed other libraries used in dataset code.

Return type
Optional[*WorkerInfo*]

Classes

pytorch_pfn_extras.dataloaders.dataloader.DataLoader(dataset)	Data loader.

pytorch_pfn_extras.dataloaders.dataloader.DataLoader

class pytorch_pfn_extras.dataloaders.dataloader.**DataLoader**(*dataset*, *batch_size=1*, *shuffle=None*, *sampler=None*, *batch_sampler=None*, *num_workers=0*, *collate_fn=None*, *pin_memory=False*, *drop_last=False*, *timeout=0*, *worker_init_fn=None*, *multiprocessing_context=None*, *generator=None*, ***, *prefetch_factor=None*, *persistent_workers=False*, *pin_memory_device=''*)

Bases: Generic[T_co]

Data loader. Combines a dataset and a sampler, and provides an iterable over the given dataset.

The DataLoader supports both map-style and iterable-style datasets with single- or multi-process loading, customizing loading order and optional automatic batching (collation) and memory pinning.

See torch.utils.data documentation page for more details.

Parameters

- **dataset** (Dataset) – dataset from which to load the data.
- **batch_size** (*int, optional*) – how many samples per batch to load (default: 1).
- **shuffle** (*bool, optional*) – set to True to have the data reshuffled at every epoch (default: False).
- **sampler** (*Sampler or Iterable, optional*) – defines the strategy to draw samples from the dataset. Can be any Iterable with __len__ implemented. If specified, shuffle must not be specified.
- **batch_sampler** (*Sampler or Iterable, optional*) – like *sampler*, but returns a batch of indices at a time. Mutually exclusive with *batch_size*, shuffle, *sampler*, and *drop_last*.
- **num_workers** (*int, optional*) – how many subprocesses to use for data loading. 0 means that the data will be loaded in the main process. (default: 0)
- **collate_fn** (*Callable, optional*) – merges a list of samples to form a mini-batch of Tensor(s). Used when using batched loading from a map-style dataset.
- **pin_memory** (*bool, optional*) – If True, the data loader will copy Tensors into device/CUDA pinned memory before returning them. If your data elements are a custom type, or your collate_fn returns a batch that is a custom type, see the example below.
- **drop_last** (*bool, optional*) – set to True to drop the last incomplete batch, if the dataset size is not divisible by the batch size. If False and the size of dataset is not divisible by the batch size, then the last batch will be smaller. (default: False)
- **timeout** (*numeric, optional*) – if positive, the timeout value for collecting a batch from workers. Should always be non-negative. (default: 0)
- **worker_init_fn** (*Callable, optional*) – If not None, this will be called on each worker subprocess with the worker id (an int in [0, num_workers - 1]) as input, after seeding and before data loading. (default: None)
- **multiprocessing_context** (*str or multiprocessing.context.BaseContext, optional*) – If None, the default multiprocessing context of your operating system will be used. (default: None)
- **generator** (*torch.Generator, optional*) – If not None, this RNG will be used by RandomSampler to generate random indexes and multiprocessing to generate base_seed for workers. (default: None)
- **prefetch_factor** (*int, optional, keyword-only arg*) – Number of batches loaded in advance by each worker. 2 means there will be a total of 2 * num_workers batches prefetched across all workers. (default value depends on the set value for num_workers. If value of num_workers=0 default is None. Otherwise, if value of num_workers > 0 default is 2).
- **persistent_workers** (*bool, optional*) – If True, the data loader will not shut down the worker processes after a dataset has been consumed once. This allows to maintain the workers *Dataset* instances alive. (default: False)
- **pin_memory_device** (*str, optional*) – the device to *pin_memory* to if pin_memory is True.

Warning: If the spawn start method is used, worker_init_fn cannot be an unpicklable object, e.g., a lambda function. See multiprocessing-best-practices on more details related to multiprocessing in PyTorch.

Warning: `len(dataloader)` heuristic is based on the length of the sampler used. When *dataset* is an `IterableDataset`, it instead returns an estimate based on `len(dataset) / batch_size`, with proper rounding depending on *drop_last*, regardless of multi-process loading configurations. This represents the best guess PyTorch can make because PyTorch trusts user *dataset* code in correctly handling multi-process loading to avoid duplicate data.

However, if sharding results in multiple workers having incomplete last batches, this estimate can still be inaccurate, because (1) an otherwise complete batch can be broken into multiple ones and (2) more than one batch worth of samples can be dropped when *drop_last* is set. Unfortunately, PyTorch can not detect such cases in general.

See **`Dataset Types`_** for more details on these two types of datasets and how `IterableDataset` interacts with **`Multi-process data loading`_**.

Warning: See reproducibility, and dataloader-workers-random-seed, and data-loading-randomness notes for random seed related questions.

Methods

__init__(dataset[, batch_size, shuffle, ...])

check_worker_number_rationality()

Attributes

multiprocessing_context

dataset

batch_size

num_workers

pin_memory

drop_last

timeout

sampler

pin_memory_device

prefetch_factor

__init__(*dataset*, *batch_size=1*, *shuffle=None*, *sampler=None*, *batch_sampler=None*, *num_workers=0*, *collate_fn=None*, *pin_memory=False*, *drop_last=False*, *timeout=0*, *worker_init_fn=None*, *multiprocessing_context=None*, *generator=None*, ***, *prefetch_factor=None*, *persistent_workers=False*, *pin_memory_device=''*)

Parameters

- **dataset** (*Dataset[T_co]*) –
- **batch_size** (*Optional[int]*) –
- **shuffle** (*Optional[bool]*) –
- **sampler** (*Optional[Union[Sampler, Iterable]]*) –
- **batch_sampler** (*Optional[Union[Sampler[List], Iterable[List]]]*) –
- **num_workers** (*int*) –
- **collate_fn** (*Optional[Callable[[List[T]], Any]]*) –
- **pin_memory** (*bool*) –
- **drop_last** (*bool*) –
- **timeout** (*float*) –
- **worker_init_fn** (*Optional[Callable[[int], None]]*) –
- **prefetch_factor** (*Optional[int]*) –
- **persistent_workers** (*bool*) –
- **pin_memory_device** (*str*) –

batch_size: Optional[int]

check_worker_number_rationality()

dataset: *Dataset*[T_co]

drop_last: bool

property multiprocessing_context

num_workers: int

pin_memory: bool

pin_memory_device: str

prefetch_factor: Optional[int]

sampler: Union[Sampler, Iterable]

timeout: float

pytorch_pfn_extras.dataloaders.utils

Classes

pytorch_pfn_extras.dataloaders.utils.CollateAsDict(names)	Creates a collate function that converts inputs to a dict of tensors.

pytorch_pfn_extras.dataloaders.utils.CollateAsDict

class pytorch_pfn_extras.dataloaders.utils.**CollateAsDict**(*names*, *collate_fn=<function default_collate>*)

Bases: `object`

Creates a collate function that converts inputs to a dict of tensors.

An instantiated callable object can be feeded to `torch.utils.data.DataLoader` as a `collate_fn` option.

Parameters

- **names** (*list of str*) – Names of keys of output dict.
- **collate_fn** (*function*) – A function preprocesses inputs.

Methods

__init__(names[, collate_fn])	

__call__(**args*, ***kwargs*)

Converts inputs the dataset generated to a dictionary of tensors.

Returns (dict of Tensor):
A dictionary with keys that specified as `names` option, and values as input tensors.

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Dict[str, *Any*]

__init__(*names*, *collate_fn=<function default_collate>*)

Parameters

- **names** (*Sequence[str]*) –
- **collate_fn** (*Callable[[...], Any]*) –

Return type
None

pytorch_pfn_extras.dataset

Classes

pytorch_pfn_extras.dataset.SharedDataset(sm_size)	Dataset that caches the load samples in shared memory
pytorch_pfn_extras.dataset.TabularDataset(...)	An abstract class that represents tabular dataset.

pytorch_pfn_extras.dataset.SharedDataset

class pytorch_pfn_extras.dataset.**SharedDataset**(*sm_size*, *cache_type=<class 'pytorch_pfn_extras.dataset.shared_dataset.InfiniteCache'>*)

Bases: *Dataset*

Dataset that caches the load samples in shared memory

Args

Methods

__init__(sm_size[, cache_type])	
cache_item(idx, x)	
is_cached(idx)	

__init__(*sm_size*, *cache_type=<class 'pytorch_pfn_extras.dataset.shared_dataset.InfiniteCache'>*)

cache_item(*idx*, *x*)

is_cached(*idx*)

pytorch_pfn_extras.dataset.TabularDataset

class pytorch_pfn_extras.dataset.**TabularDataset**(**args*, ***kwds*)

Bases: *Dataset*

An abstract class that represents tabular dataset.

This class represents a tabular dataset. In a tabular dataset, all examples have the same number of elements. For example, all examples of the dataset below have three elements (`a[i]`, `b[i]`, and `c[i]`).

	a	b	c
0	a[0]	b[0]	c[0]
1	a[1]	b[1]	c[1]
2	a[2]	b[2]	c[2]
3	a[3]	b[3]	c[3]

Since an example can be represented by both tuple and dict (`(a[i], b[i], c[i])` and `{'a': a[i], 'b': b[i], 'c': c[i]}`), this class uses *mode* to indicate which representation will be used. If there is only one column, an example also can be represented by a value (`a[i]`). In this case, *mode* is `None`.

An inheritance should implement `__len__()`, *keys*, *mode* and *get_examples()*.

```
>>> import numpy as np
>>>
>>> from pytorch_pfn_extras import dataset
>>>
>>> class MyDataset(dataset.TabularDataset):
...
...     def __len__(self):
...         return 4
...
...     @property
...     def keys(self):
...          return ('a', 'b', 'c')
...
...     @property
...     def mode(self):
...          return tuple
...
...     def get_examples(self, indices, key_indices):
...          data = np.arange(12).reshape((4, 3))
...          if indices is not None:
...              data = data[indices]
...          if key_indices is not None:
...              data = data[:, list(key_indices)]
...          return tuple(data.transpose())
...
>>> dataset = MyDataset()
>>> len(dataset)
4
>>> dataset.keys
('a', 'b', 'c')
>>> dataset.astuple()[0]
(0, 1, 2)
>>> sorted(dataset.asdict()[0].items())
[('a', 0), ('b', 1), ('c', 2)]
>>>
>>> view = dataset.slice[[3, 2], ('c', 0)]
>>> len(view)
2
>>> view.keys
('c', 'a')
>>> view.astuple()[1]
(8, 6)
>>> sorted(view.asdict()[1].items())
[('a', 6), ('c', 8)]
```

Methods

`__init__`()	

asdict()	Return a view with dict mode.
astuple()	Return a view with tuple mode.
concat(*datasets)	Stack datasets along rows.
convert(data)	Convert fetched data.
fetch()	Fetch data.
get_example(i)	
get_examples(indices, key_indices)	Return a part of data.
join(*datasets)	Stack datasets along columns.
transform(keys, transform)	Apply a transform to each example.
transform_batch(keys, transform_batch)	Apply a transform to examples.
with_converter(converter)	Override the behaviour of *convert()*.

Attributes

keys	Names of columns.
mode	Mode of representation.
slice	Get a slice of dataset.

asdict()

Return a view with dict mode.

Returns
A view whose *mode* is `dict`.

astuple()

Return a view with tuple mode.

Returns
A view whose *mode* is `tuple`.

concat(**datasets*)

Stack datasets along rows.

Parameters
datasets (iterable of *TabularDataset*) – Datasets to be concatenated. All datasets must have the same *keys*.

Returns
A concatenated dataset.

convert(*data*)

Convert fetched data.

This method takes data fetched by *fetch()* and pre-process them before passing them to models. The default behaviour is converting each column into an ndarray. This behaviour can be overridden by *with_converter()*. If the dataset is constructed by *concat()* or *join()*, the converter of the first dataset is used.

Parameters
data (*tuple or dict*) – Data from *fetch()*.

Returns
A tuple or dict. Each value is an ndarray.

`fetch`()

Fetch data.

This method fetches all data of the dataset/view. Note that this method returns a column-major data (i.e. `([a[0], ..., a[3]], ..., [c[0], ... c[3]])`, `{'a': [a[0], ..., a[3]], ..., 'c': [c[0], ..., c[3]]}`, or `[a[0], ..., a[3]]`).

Returns
If *mode* is `tuple`, this method returns a tuple of lists/arrays. If *mode* is `dict`, this method returns a dict of lists/arrays.

`get_example`(*i*)

`get_examples`(*indices*, *key_indices*)

Return a part of data.

Parameters

- **indices** (*list of ints or slice*) – Indices of requested rows. If this argument is `None`, it indicates all rows.
- **key_indices** (*tuple of ints*) – Indices of requested columns. If this argument is `None`, it indicates all columns.

Returns
tuple of lists/arrays

`join`(**datasets*)

Stack datasets along columns.

Args: datasets (iterable of *TabularDataset*):
Datasets to be concatenated. All datasets must have the same length

Returns
A joined dataset.

`property keys`

Names of columns.

A tuple of strings that indicate the names of columns.

`property mode`

Mode of representation.

This indicates the type of value returned by *fetch()* and `__getitem__()`. `tuple`, `dict`, and `None` are supported.

`property slice`

Get a slice of dataset.

Parameters

- **indices** (*list/array of ints/bools or slice*) – Requested rows.
- **keys** (*tuple of ints/strs or int or str*) – Requested columns.

Returns
A view of specified range.

`transform`(*keys*, *transform*)

Apply a transform to each example.

The transformations are a list where each element is a tuple that holds the transformation signature and a callable that is the transformation itself.

The transformation signature is a tuple of 2 elements with the first one being the keys of the dataset that are taken as inputs. And the last one the outputs it produces for the transformation *keys* argument.

When multiple transformations are specified, the outputs must be disjoint or *ValueError* will be risen.

Parameters

- **keys** (*tuple of strs*) – The keys of transformed examples.
- **transform** (*list of tuples*) – A list where each element specifies a transformation with a tuple with the transformation signature and a callable that takes an example and returns transformed example. *mode* of transformed dataset is determined by the transformed examples.

Returns

A transfromed dataset.

`transform_batch`(*keys*, *transform_batch*)

Apply a transform to examples.

The transformations are a list where each element is a tuple that holds the transformation signature and a callable that is the transformation itself.

The transformation signature is a tuple of 2 elements with the first one being the keys of the dataset that are taken as inputs. And the last one the outputs it produces for the transformation *keys* argument.

When multiple transformations are specified, the outputs must be disjoint or *ValueError* will be risen.

Parameters

- **keys** (*tuple of strs*) – The keys of transformed examples.
- **transform_batch** (*list of tuples*) – A list where each element specifies a transformation with a tuple with the transformation signature and a callable that takes a batch of examples and returns a batch of transformed examples. *mode* of transformed dataset is determined by the transformed examples.

Returns

A transfromed dataset.

`with_converter`(*converter*)

Override the behaviour of *convert()*.

This method overrides *convert()*.

Parameters

converter (*callable*) – A new converter.

Returns

A dataset with the new converter.

Exceptions

pytorch_pfn_extras.dataset.ItemNotFoundException

pytorch_pfn_extras.dataset.ItemNotFoundException

exception pytorch_pfn_extras.dataset.**ItemNotFoundException**

Modules

pytorch_pfn_extras.dataset.shared_dataset
pytorch_pfn_extras.dataset.tabular

pytorch_pfn_extras.dataset.shared_dataset

Classes

pytorch_pfn_extras.dataset.shared_dataset.Cache()	
pytorch_pfn_extras.dataset.shared_dataset.InfiniteCache(sm_size)	
pytorch_pfn_extras.dataset.shared_dataset.SharedDataset(sm_size)	Dataset that caches the load samples in shared memory

pytorch_pfn_extras.dataset.shared_dataset.Cache

class pytorch_pfn_extras.dataset.shared_dataset.**Cache**

Bases: object

Methods

__init__()
add_to_cache(idx, x)
get_value(idx)
is_cached(idx)

add_to_cache(*idx*, *x*)

get_value(*idx*)

is_cached(*idx*)

pytorch_pfn_extras.dataset.shared_dataset.InfiniteCache

class pytorch_pfn_extras.dataset.shared_dataset.**InfiniteCache**(*sm_size*)

Bases: *Cache*

Methods

__init__(sm_size)
add_to_cache(idx, x)
get_value(idx)
is_cached(idx)

__init__(*sm_size*)

add_to_cache(*idx*, *x*)

get_value(*idx*)

is_cached(*idx*)

pytorch_pfn_extras.dataset.shared_dataset.SharedDataset

class pytorch_pfn_extras.dataset.shared_dataset.**SharedDataset**(*sm_size*, *cache_type=<class 'pytorch_pfn_extras.dataset.shared_dataset.InfiniteCache*

Bases: *Dataset*

Dataset that caches the load samples in shared memory

Args

Methods

__init__(sm_size[, cache_type])
cache_item(idx, x)
is_cached(idx)

__init__(*sm_size*, *cache_type=<class 'pytorch_pfn_extras.dataset.shared_dataset.InfiniteCache'>*)

cache_item(*idx*, *x*)

is_cached(*idx*)

Exceptions

pytorch_pfn_extras.dataset.shared_dataset.ItemNotFoundException	

pytorch_pfn_extras.dataset.shared_dataset.ItemNotFoundException

exception pytorch_pfn_extras.dataset.shared_dataset.**ItemNotFoundException**

pytorch_pfn_extras.dataset.tabular

Functions

pytorch_pfn_extras.dataset.tabular.from_data(data, *)	Create a TabularDataset from lists/arrays/callables.

pytorch_pfn_extras.dataset.tabular.from_data

pytorch_pfn_extras.dataset.tabular.**from_data**(*data*, *, *size=None*)

Create a TabularDataset from lists/arrays/callables.

```
>>> from pytorch_pfn_extras.dataset import tabular
>>>
>>> dataset = tabular.from_data([0, 1, 2])
>>> dataset[0]
0
>>> dataset = tabular.from_data(([0, 1, 2], [3, 4, 5]))
>>> dataset[0]
(0, 3)
>>> dataset = tabular.from_data((('a', [0, 1, 2]), ('b', [3, 4, 5])))
>>> dataset.keys
('a', 'b')
>>> dataset[0]
(0, 3)
>>> dataset = tabular.from_data({'a': [0, 1, 2], 'b': [3, 4, 5]})
>>> sorted(dataset[0].items())
[('a', 0), ('b', 3)]
>>> dataset = tabular.from_data(('a', lambda i: i * i), size=10)
>>> dataset[5]
25
```

Parameters

- **data** (*list, array, tuple, or dict*) – Data in following format.

- *list/array*
- *(str, list/array/callable)*
- *((str, ...), callable)*
- *((list/array)/(str, list/array/callable) /((key, ...), callable), ...)*
- *{str: (list/array/callable)/(str, ...): callable, ...}*

- **size** (*int*) – The length of the dataset. This argument is required when no lists/arrays exist in `data`.

Returns
A *TabularDataset*.

Classes

pytorch_pfn_extras.dataset.tabular.DelegateDataset(dataset)	A helper class to implement a TabularDataset.

pytorch_pfn_extras.dataset.tabular.DelegateDataset

class pytorch_pfn_extras.dataset.tabular.**DelegateDataset**(*dataset*)

Bases: *TabularDataset*

A helper class to implement a TabularDataset.

This class wraps an instance of *TabularDataset* and provides methods of *TabularDataset*. This class is useful to create a custom dataset class by inheriting it.

```
>>> import numpy as np
>>>
>>> from pytorch_pfn_extras.dataset import tabular
>>>
>>> class MyDataset(tabular.DelegateDataset):
...
...     def __init__(self):
...         super().__init__(tabular.from_data((
...             ('a', np.arange(10)),
...             ('b', self.get_b),
...             ('c', [3, 1, 4, 5, 9, 2, 6, 8, 7, 0]),
...             (('d', 'e'), self.get_de))))
...
...     def get_b(self, i):
...         return 'b[{}]'.format(i)
...
...     def get_de(self, i):
...         return {'d': 'd[{}]'.format(i), 'e': 'e[{}]'.format(i)}
...
>>> dataset = MyDataset()
>>> len(dataset)
10
>>> dataset.keys
```

(continues on next page)

(continued from previous page)

```
('a', 'b', 'c', 'd', 'e')
>>> dataset[0]
(0, 'b[0]', 3, 'd[0]', 'e[0]')
```

Parameters
dataset (pytorch_pfn_extras.dataset.TabularDataset) – An underlying dataset.

Methods

__init__(dataset)	

asdict()	Return a view with dict mode.
astuple()	Return a view with tuple mode.
concat(*datasets)	Stack datasets along rows.
convert(data)	Convert fetched data.
fetch()	Fetch data.
get_example(i)	
get_examples(indices, key_indices)	Return a part of data.
join(*datasets)	Stack datasets along columns.
transform(keys, transform)	Apply a transform to each example.
transform_batch(keys, transform_batch)	Apply a transform to examples.
with_converter(converter)	Override the behaviour of convert().

Attributes

keys	Names of columns.
mode	Mode of representation.
slice	Get a slice of dataset.

__init__(*dataset*)

get_examples(*indices*, *key_indices*)

Return a part of data.

Parameters

- **indices** (*list of ints or slice*) – Indices of requested rows. If this argument is None, it indicates all rows.
- **key_indices** (*tuple of ints*) – Indices of requested columns. If this argument is None, it indicates all columns.

Returns
tuple of lists/arrays

property keys

Names of columns.

A tuple of strings that indicate the names of columns.

property mode

Mode of representation.

This indicates the type of value returned by `fetch()` and `__getitem__()`. `tuple`, `dict`, and `None` are supported.

Modules

pytorch_pfn_extras.dataset.tabular.delegate_dataset	
pytorch_pfn_extras.dataset.tabular.from_data(data, *)	Create a TabularDataset from lists/arrays/callables.
pytorch_pfn_extras.dataset.tabular.tabular_dataset	

pytorch_pfn_extras.dataset.tabular.delegate_dataset

Classes

pytorch_pfn_extras.dataset.tabular.delegate_dataset.DelegateDataset(dataset)	A helper class to implement a TabularDataset.

pytorch_pfn_extras.dataset.tabular.delegate_dataset.DelegateDataset

class pytorch_pfn_extras.dataset.tabular.delegate_dataset.**DelegateDataset**(*dataset*)

Bases: *TabularDataset*

A helper class to implement a TabularDataset.

This class wraps an instance of *TabularDataset* and provides methods of *TabularDataset*. This class is useful to create a custom dataset class by inheriting it.

```
>>> import numpy as np
>>>
>>> from pytorch_pfn_extras.dataset import tabular
>>>
>>> class MyDataset(tabular.DelegateDataset):
...
...     def __init__(self):
...         super().__init__(tabular.from_data((
...             ('a', np.arange(10)),
...             ('b', self.get_b),
...             ('c', [3, 1, 4, 5, 9, 2, 6, 8, 7, 0]),
...             (('d', 'e'), self.get_de))))
...
...     def get_b(self, i):
...         return 'b[{}]'.format(i)
...
...     def get_de(self, i):
```

(continues on next page)

(continued from previous page)

```
...         return {'d': 'd[{}]'.format(i), 'e': 'e[{}]'.format(i)}
...
>>> dataset = MyDataset()
>>> len(dataset)
10
>>> dataset.keys
('a', 'b', 'c', 'd', 'e')
>>> dataset[0]
(0, 'b[0]', 3, 'd[0]', 'e[0]')
```

Parameters
dataset (pytorch_pfn_extras.dataset.TabularDataset) – An underlying dataset.

Methods

__init__(dataset)	

asdict()	Return a view with dict mode.
astuple()	Return a view with tuple mode.
concat(*datasets)	Stack datasets along rows.
convert(data)	Convert fetched data.
fetch()	Fetch data.
get_example(i)	
get_examples(indices, key_indices)	Return a part of data.
join(*datasets)	Stack datasets along columns.
transform(keys, transform)	Apply a transform to each example.
transform_batch(keys, transform_batch)	Apply a transform to examples.
with_converter(converter)	Override the behaviour of convert().

Attributes

keys	Names of columns.
mode	Mode of representation.
slice	Get a slice of dataset.

__init__(*dataset*)

get_examples(*indices*, *key_indices*)
Return a part of data.

Parameters

- **indices** (*list of ints or slice*) – Indices of requested rows. If this argument is None, it indicates all rows.
- **key_indices** (*tuple of ints*) – Indices of requested columns. If this argument is None, it indicates all columns.

Returns
tuple of lists/arrays

property keys

Names of columns.

A tuple of strings that indicate the names of columns.

property mode

Mode of representation.

This indicates the type of value returned by `fetch()` and `__getitem__()`. `tuple`, `dict`, and `None` are supported.

pytorch_pfn_extras.dataset.tabular.tabular_dataset

Classes

pytorch_pfn_extras.dataset.tabular.tabular_dataset.Dataset(...)	An abstract class representing a *Dataset*.
pytorch_pfn_extras.dataset.tabular.tabular_dataset.TabularDataset(...)	An abstract class that represents tabular dataset.

pytorch_pfn_extras.dataset.tabular.tabular_dataset.Dataset

class pytorch_pfn_extras.dataset.tabular.tabular_dataset.**Dataset**(**args*, ***kwds*)

Bases: `Generic[T_co]`

An abstract class representing a *Dataset*.

All datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite `__getitem__()`, supporting fetching a data sample for a given key. Subclasses could also optionally overwrite `__len__()`, which is expected to return the size of the dataset by many `Sampler` implementations and the default options of `DataLoader`. Subclasses could also optionally implement `__getitems__()`, for speedup batched samples loading. This method accepts list of indices of samples of batch and returns list of samples.

Note: `DataLoader` by default constructs a index sampler that yields integral indices. To make it work with a map-style dataset with non-integral indices/keys, a custom sampler must be provided.

Methods

`__init__()`

pytorch_pfn_extras.dataset.tabular.tabular_dataset.TabularDataset

class pytorch_pfn_extras.dataset.tabular.tabular_dataset.**TabularDataset**(**args*, ***kwds*)

Bases: *Dataset*

An abstract class that represents tabular dataset.

This class represents a tabular dataset. In a tabular dataset, all examples have the same number of elements. For example, all examples of the dataset below have three elements (`a[i]`, `b[i]`, and `c[i]`).

	a	b	c
0	a[0]	b[0]	c[0]
1	a[1]	b[1]	c[1]
2	a[2]	b[2]	c[2]
3	a[3]	b[3]	c[3]

Since an example can be represented by both tuple and dict (`(a[i], b[i], c[i])` and `{'a': a[i], 'b': b[i], 'c': c[i]}`), this class uses *mode* to indicate which representation will be used. If there is only one column, an example also can be represented by a value (`a[i]`). In this case, *mode* is `None`.

An inheritance should implement `__len__()`, *keys*, *mode* and *get_examples()*.

```
>>> import numpy as np
>>>
>>> from pytorch_pfn_extras import dataset
>>>
>>> class MyDataset(dataset.TabularDataset):
...
...     def __len__(self):
...         return 4
...
...     @property
...     def keys(self):
...          return ('a', 'b', 'c')
...
...     @property
...     def mode(self):
...          return tuple
...
...     def get_examples(self, indices, key_indices):
...          data = np.arange(12).reshape((4, 3))
...          if indices is not None:
...              data = data[indices]
...          if key_indices is not None:
...              data = data[:, list(key_indices)]
...          return tuple(data.transpose())
...
>>> dataset = MyDataset()
>>> len(dataset)
4
>>> dataset.keys
('a', 'b', 'c')
>>> dataset.astuple()[0]
(0, 1, 2)
```

(continues on next page)

(continued from previous page)

```
>>> sorted(dataset.asdict()[0].items())
[('a', 0), ('b', 1), ('c', 2)]
>>>
>>> view = dataset.slice[[3, 2], ('c', 0)]
>>> len(view)
2
>>> view.keys
('c', 'a')
>>> view.astuple()[1]
(8, 6)
>>> sorted(view.asdict()[1].items())
[('a', 6), ('c', 8)]
```

Methods

`__init__`()	

asdict()	Return a view with dict mode.
astuple()	Return a view with tuple mode.
concat(*datasets)	Stack datasets along rows.
convert(data)	Convert fetched data.
fetch()	Fetch data.
get_example(i)	
get_examples(indices, key_indices)	Return a part of data.
join(*datasets)	Stack datasets along columns.
transform(keys, transform)	Apply a transform to each example.
transform_batch(keys, transform_batch)	Apply a transform to examples.
with_converter(converter)	Override the behaviour of *convert()*.

Attributes

keys	Names of columns.
mode	Mode of representation.
slice	Get a slice of dataset.

asdict()

Return a view with dict mode.

Returns
A view whose *mode* is `dict`.

astuple()

Return a view with tuple mode.

Returns
A view whose *mode* is `tuple`.

concat(**datasets*)

Stack datasets along rows.

Parameters
datasets (iterable of *TabularDataset*) – Datasets to be concatenated. All datasets must have the same *keys*.

Returns
A concatenated dataset.

convert(*data*)

Convert fetched data.

This method takes data fetched by *fetch()* and pre-process them before passing them to models. The default behaviour is converting each column into an ndarray. This behaviour can be overridden by *with_converter()*. If the dataset is constructed by *concat()* or *join()*, the converter of the first dataset is used.

Parameters
data (*tuple or dict*) – Data from *fetch()*.

Returns
A tuple or dict. Each value is an ndarray.

fetch()

Fetch data.

This method fetches all data of the dataset/view. Note that this method returns a column-major data (i.e. `([a[0], ..., a[3]], ..., [c[0], ... c[3]])`, `{'a': [a[0], ..., a[3]], ..., 'c': [c[0], ..., c[3]]}`, or `[a[0], ..., a[3]]`).

Returns
If *mode* is `tuple`, this method returns a tuple of lists/arrays. If *mode* is `dict`, this method returns a dict of lists/arrays.

get_example(*i*)

get_examples(*indices*, *key_indices*)

Return a part of data.

Parameters

- **indices** (*list of ints or slice*) – Indices of requested rows. If this argument is `None`, it indicates all rows.
- **key_indices** (*tuple of ints*) – Indices of requested columns. If this argument is `None`, it indicates all columns.

Returns
tuple of lists/arrays

join(**datasets*)

Stack datasets along columns.

Args: datasets (iterable of *TabularDataset*):
Datasets to be concatenated. All datasets must have the same length

Returns
A joined dataset.

property keys

Names of columns.

A tuple of strings that indicate the names of columns.

property mode

Mode of representation.

This indicates the type of value returned by *fetch()* and `__getitem__()`. `tuple`, `dict`, and `None` are supported.

property slice

Get a slice of dataset.

Parameters

- **indices** (*list/array of ints/bools or slice*) – Requested rows.
- **keys** (*tuple of ints/strs or int or str*) – Requested columns.

Returns

A view of specified range.

transform(*keys*, *transform*)

Apply a transform to each example.

The transformations are a list where each element is a tuple that holds the transformation signature and a callable that is the transformation itself.

The transformation signature is a tuple of 2 elements with the first one being the keys of the dataset that are taken as inputs. And the last one the outputs it produces for the transformation *keys* argument.

When multiple transformations are specified, the outputs must be disjoint or *ValueError* will be risen.

Parameters

- **keys** (*tuple of strs*) – The keys of transformed examples.
- **transform** (*list of tuples*) – A list where each element specifies a transformation with a tuple with the transformation signature and a callable that takes an example and returns transformed example. *mode* of transformed dataset is determined by the transformed examples.

Returns

A transfromed dataset.

transform_batch(*keys*, *transform_batch*)

Apply a transform to examples.

The transformations are a list where each element is a tuple that holds the transformation signature and a callable that is the transformation itself.

The transformation signature is a tuple of 2 elements with the first one being the keys of the dataset that are taken as inputs. And the last one the outputs it produces for the transformation *keys* argument.

When multiple transformations are specified, the outputs must be disjoint or *ValueError* will be risen.

Parameters

- **keys** (*tuple of strs*) – The keys of transformed examples.
- **transform_batch** (*list of tuples*) – A list where each element specifies a transformation with a tuple with the transformation signature and a callable that takes a batch of

examples and returns a batch of transformed examples. *mode* of transformed dataset is determined by the transformed examples.

Returns
A transfromed dataset.

with_converter(*converter*)

Override the behaviour of *convert()*.

This method overrides *convert()*.

Parameters
converter (*callable*) – A new converter.

Returns
A dataset with the new converter.

pytorch_pfn_extras.distributed

Functions

pytorch_pfn_extras.distributed.create_distributed_subset_indices(...)	Returns a indices of a dataset to be used for the current process.
pytorch_pfn_extras.distributed.initialize_ompi_environment(*)	Initialize *torch.distributed* environments with values taken from OpenMPI.

pytorch_pfn_extras.distributed.create_distributed_subset_indices

pytorch_pfn_extras.distributed.**create_distributed_subset_indices**(*num_total_samples*, *num_replicas=None*, *rank=None*, *shuffle=True*, *seed=None*)

Returns a indices of a dataset to be used for the current process.

Parameters

- **num_total_samples** (*int*) – The size of the dataset.
- **num_replicas** (*Optional[int]*) – Number of processes participating in the training. By default, `torch.distributed.get_world_size()` is used.
- **rank** (*Optional[int]*) – Rank of the current process within *num_replicas*. By default, `torch.distributed.get_rank()` is used.
- **shuffle** (*bool*) – If `True` (default), shuffle the indices.
- **seed** (*Optional[int]*) – Random seed used to shuffle.

Return type
List[int]

pytorch_pfn_extras.distributed.initialize_ompi_environment

pytorch_pfn_extras.distributed.**initialize_ompi_environment**(*, *backend='gloo'*, *init_method='tcp'*, *world_size=1*, *rank=0*, *local_rank=0*, *addr='localhost'*, *port='1234'*)

Initialize *torch.distributed* environments with values taken from OpenMPI.

Parameters

- **backend** (*str*) – The backend to be used, only "gloo" and "nccl" are supported. Defaults to "gloo".
- **init_method** (*str*) – Initialization method used by torch, only "tcp" and "env" are supported. Defaults to "tcp".
- **world_size** (*int*) – The total world size to be used in case it is not specified in MPI env vars. Defaults to 1.
- **rank** (*int*) – The process rank to be used in case it is not specified in MPI env vars. Defaults to 0.
- **local_rank** (*int*) – The process local rank to be used in case it is not specified in MPI env vars. Defaults to 0.
- **addr** (*str*) – The address of the master process of *torch.distributed*. Defaults to "localhost"
- **port** (*str*) – The port of the master process of *torch.distributed*. Defaults to "1234"

Return type
Tuple[int, int, int]

Classes

pytorch_pfn_extras.distributed.DistributedValidationSampler(dataset)	Distributed sampler without duplication

pytorch_pfn_extras.distributed.DistributedValidationSampler

class pytorch_pfn_extras.distributed.**DistributedValidationSampler**(*dataset*, *num_replicas=None*, *rank=None*, *shuffle=True*, *seed=0*)

Bases: Sampler

Distributed sampler without duplication

This sampler splits the input dataset to each worker process in distributed setup without allowing repetition. It is for evaluation purpose such as DistributedEvaluator. This does not guarantee each worker to get the same number of samples, so for training do not use this sampler (use PyTorch DistributedSampler instead).

Methods

__init__(dataset[, num_replicas, rank, ...])	

__init__(*dataset*, *num_replicas=None*, *rank=None*, *shuffle=True*, *seed=0*)

Parameters

- **dataset** (*Sized*) –
- **num_replicas** (*Optional[int]*) –
- **rank** (*Optional[int]*) –
- **shuffle** (*bool*) –
- **seed** (*int*) –

Return type
None

pytorch_pfn_extras.engine

Functions

pytorch_pfn_extras.engine.cast(typ, val)	Cast a value to a type.
pytorch_pfn_extras.engine.create_evaluator(...)	Creates an evaluator object.
pytorch_pfn_extras.engine.create_trainer(...)	Creates a trainer object.
pytorch_pfn_extras.engine.default_transform_model(n, x)	
pytorch_pfn_extras.engine.filter_state_objects(args)	
pytorch_pfn_extras.engine.filter_state_objects_dict(args)	

pytorch_pfn_extras.engine.cast

pytorch_pfn_extras.engine.**cast**(*typ*, *val*)

Cast a value to a type.

This returns the value unchanged. To the type checker this signals that the return value has the designated type, but at runtime we intentionally don't check anything (we want this to be as fast as possible).

pytorch_pfn_extras.engine.create_evaluator

pytorch_pfn_extras.engine.**create_evaluator**(*models*, ***, *progress_bar=False*, *device='cpu'*, *metrics=None*, *logic=None*, *handler_class=None*, *options=None*, *runtime_options=None*, *profile=None*, *distributed=False*)

Creates an evaluator object. The return value of this function is expected to be fed to *ppe.engine.create_trainer* as an argument.

Parameters

- **models** (*Union[Module, Mapping[str, Module]]*) – Map of string to `torch.nn.Module` or an actual Module. In most cases, this arugment is the same as the *model* arguemnt of *ppe.engine.create_trainer*.
- **progress_bar** (*bool*) – If *True*, a progress bar is enabled in evaluation.
- **device** (*str or torch.device*) – Device name used for selecting a corresponding runtime class.
- **metrics** (*list of metrics*) – List of metrics, which computes various quantities and update output for the reporting.
- **logic** (*Optional[Logic]*) – A logic object. If *None* is given, an logic object is instantiated from the default logic class.
- **handler_class** (*Optional[Type[Handler]]*) – A handler class that instantiates a handler object. If *None* is given, *ppe.handler.Handler* is used as a default handler class.
- **options** (*Optional[Dict[str, Any]]*) – Options that are set to the handler and logic object. See the documentation of *ppe.handler.Handler* and *ppe.handler.Logic* for details.
- **runtime_options** (*Optional[Mapping[str, Any]]*) – Options that are set to the runtime object. See the documentation of *ppe.handler.Handler* for details.
- **profile** (*Optional[profile]*) – A *torch.profiler.profile* object to collect the performance metrics.
- **distributed** (*bool*) – Flag to determine whether to create a distributed-enabled evaluator. If set to True, the created evaluator will support distributed execution.

Return type
Evaluator

pytorch_pfn_extras.engine.create_trainer

pytorch_pfn_extras.engine.**create_trainer**(*models*, *optimizers*, *max_epochs*, ***, *extensions=None*, *out_dir='result'*, *stop_trigger=None*, *writer=None*, *evaluator=None*, *device='cpu'*, *logic=None*, *transform_model=<function default_transform_model>*, *handler_class=None*, *options=None*, *runtime_options=None*, *profile=None*, ***kwargs*)

Creates a trainer object.

Parameters

- **models** (*Union[Module, Mapping[str, Module]]*) – Map of string to Module or an actual Module.

- **optimizers** (*Union[Optimizer, Mapping[str, Optimizer]]*) – Map of string to Optimizer or an actual Optimizer.
- **max_epochs** (*int*) – Number of epochs in the whole training loop. Ignored if *stop_trigger* is passed as a kwarg.
- **extensions** (*Optional[Sequence[Union[extension.ExtensionLike, extension.ExtensionEntry]]]*) – List of extensions to be registered to the trainer.
- **out_dir** (*str*) – Output directory (default: `result`).
- **stop_trigger** (*trigger, optional*) – Trigger that can be consulted to determine wether training has concluded. The default is an interval trigger set to *max_epochs*
- **writer** (*Optional[writing.Writer]*) – Writer that can be used by extensions to write data to custom filesystems.
- **evaluator** (*Optional[Union[Evaluator, Tuple[Evaluator, TriggerLike], Mapping[str, Union[Evaluator, Tuple[Evaluator, TriggerLike]]]]*) – Evaluator that is used in evaluation phase. If *None* is given, the evaluation is skipped. Evaluators can be created with *pytorch_pfn_extras.engine.create_evaluator()*.
- **device** (*str or torch.device*) – Device name used for selecting a corresponding runtime class.
- **logic** (*Optional[BaseLogic]*) – A logic object. If *None* is given, an logic object is instantiated from the default logic class.
- **transform_model** (*Callable[[str, Module], Module]*) – A function to transform a model structure, often used to unwrap the a module from DDP module.
- **handler_class** (*Optional[Type[Handler]]*) – A handler class that instantiates a handler object. If *None* is given, *ppe.handler.Handler* is used as a default handler class.
- **options** (*Optional[Dict[str, Any]]*) – Options that are set to the handler and logic object. See the documentation of *ppe.handler.Handler* and *ppe.handler.Logic* for details.
- **runtime_options** (*Optional[Mapping[str, Any]]*) – Options that are set to the runtime object. See the documentation of *ppe.runtime.PyTorchRuntime* for details.
- **profile** (*Optional[profile]*) – A *torch.profiler.profile* object to collect the performance metrics.
- **kwargs** (*Any*) –

Return type
Trainer

pytorch_pfn_extras.engine.default_transform_model

pytorch_pfn_extras.engine.**default_transform_model**(*n*, *x*)

Parameters

- **n** (*str*) –
- **x** (*Module*) –

Return type
Module

pytorch_pfn_extras.engine.filter_state_objects

pytorch_pfn_extras.engine.**filter_state_objects**(*args*, *key_name=''*)

Parameters

- **args** (*Any*) –
- **key_name** (*str*) –

Return type
List[*Tuple*[str, *StateObjectProtocol*]]

pytorch_pfn_extras.engine.filter_state_objects_dict

pytorch_pfn_extras.engine.**filter_state_objects_dict**(*args*, *key_name='option'*)

Parameters

- **args** (*Dict[str, Any]*) –
- **key_name** (*str*) –

Return type
List[*Tuple*[str, *StateObjectProtocol*]]

Classes

pytorch_pfn_extras.engine.DistributedEvaluator(...)	
pytorch_pfn_extras.engine.Evaluator(handler, ...)	
pytorch_pfn_extras.engine.NamedTuple(typename)	Typed version of namedtuple.
pytorch_pfn_extras.engine.StateObjectProtocol(...)	
pytorch_pfn_extras.engine.Trainer(handler, ...)	

pytorch_pfn_extras.engine.DistributedEvaluator

class pytorch_pfn_extras.engine.**DistributedEvaluator**(*handler*, *models*, ***, *progress_bar=False*, *metrics=None*, *profile=None*)

Bases: *Evaluator*

Methods

__init__(handler, models, *[, progress_bar, ...])	
run(loader, *[, eval_len])	Executes the evaluation loop.

Parameters

- **handler** (BaseHandler) –
- **models** (*Union[Module, Mapping[str, Module]]*) –
- **progress_bar** (*bool*) –
- **metrics** (*Optional[Sequence[MetricType]]*) –
- **profile** (*Optional[profile]*) –

__init__(*handler*, *models*, *, *progress_bar=False*, *metrics=None*, *profile=None*)

Parameters

- **handler** (BaseHandler) –
- **models** (*Union[Module, Mapping[str, Module]]*) –
- **progress_bar** (*bool*) –
- **metrics** (*Optional[Sequence[MetricType]]*) –
- **profile** (*Optional[profile]*) –

pytorch_pfn_extras.engine.Evaluator

class pytorch_pfn_extras.engine.**Evaluator**(*handler*, *models*, *, *progress_bar=False*, *metrics=None*, *profile=None*)

Bases: object

Methods

__init__(handler, models, *[, progress_bar, ...])	
run(loader, *[, eval_len])	Executes the evaluation loop.

Parameters

- **handler** (BaseHandler) –
- **models** (*Union[Module, Mapping[str, Module]]*) –
- **progress_bar** (*bool*) –
- **metrics** (*Optional[Sequence[MetricType]]*) –
- **profile** (*Optional[profile]*) –

__init__(*handler*, *models*, *, *progress_bar=False*, *metrics=None*, *profile=None*)

Parameters

- **handler** (BaseHandler) –
- **models** (*Union[Module, Mapping[str, Module]]*) –
- **progress_bar** (*bool*) –
- **metrics** (*Optional[Sequence[MetricType]]*) –
- **profile** (*Optional[profile]*) –

run(*loader*, *, *eval_len=None*)

Executes the evaluation loop.

Parameters

- **loader** (*torch.utils.data.DataLoader*) – A data loader for evaluation.
- **eval_len** (*int, optional*) – The number of iterations per one evaluation epoch.

Return type

None

pytorch_pfn_extras.engine.NamedTuple

class pytorch_pfn_extras.engine.**NamedTuple**(*typename*, *fields=None*, */*, ***kwargs*)

Bases: object

Typed version of namedtuple.

Usage in Python versions >= 3.6:

```
class Employee(NamedTuple):
    name: str
    id: int
```

This is equivalent to:

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

The resulting class has an extra __annotations__ attribute, giving a dict that maps field names to types. (The field names are also in the _fields attribute, which is part of the namedtuple API.) Alternative equivalent keyword syntax is also accepted:

```
Employee = NamedTuple('Employee', name=str, id=int)
```

In Python versions <= 3.5 use:

```
Employee = NamedTuple('Employee', [('name', str), ('id', int)])
```

Methods

`__init__`()

pytorch_pfn_extras.engine.StateObjectProtocol

class pytorch_pfn_extras.engine.**StateObjectProtocol**(**args*, ***kwargs*)

Bases: `Protocol`

Methods

__init__(*args, **kwargs)
load_state_dict(state_dict)
state_dict()

__init__(**args*, ***kwargs*)

load_state_dict(*state_dict*)

Parameters
state_dict (*Dict[str, Any]*) –

Return type
None

state_dict()

Return type
Dict[str, *Any*]

pytorch_pfn_extras.engine.Trainer

class pytorch_pfn_extras.engine.**Trainer**(*handler*, ***, *evaluator*, *models*, *profile=None*, ***kwargs*)

Bases: `object`

Methods

__init__(handler, *, evaluator, models[, ...])	
extend(extension[, name, trigger, priority, ...])	
get_optimizer(name)	
is_epoch_last_iter(idx)	
load_state_dict(to_load)	
run(train_loader[, val_loader, train_len, ...])	Executes the training loop.
set_optimizer(name, optimizer)	
state_dict()	

Attributes

epoch
epoch_detail
evaluator
is_before_training
iteration
manager
models
optimizers
stop_trigger

Parameters

- **handler** (*handler_module.BaseHandler*) –
- **evaluator** (*Optional[Union[Evaluator, Tuple[Evaluator, Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]], Mapping[str, Union[Evaluator, Tuple[Evaluator, Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]]]]]*) –
- **models** (*Union[Module, Mapping[str, Module]]*) –
- **profile** (*Optional[profile]*) –

- **kwargs** (*Any*) –

__init__(*handler*, *, *evaluator*, *models*, *profile=None*, ***kwargs*)

Parameters

- **handler** (*handler_module.BaseHandler*) –
- **evaluator** (*Optional[Union[Evaluator, Tuple[Evaluator, Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]], Mapping[str, Union[Evaluator, Tuple[Evaluator, Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]]]]]*) –
- **models** (*Union[Module, Mapping[str, Module]]*) –
- **profile** (*Optional[profile]*) –
- **kwargs** (*Any*) –

property epoch: int

property epoch_detail: float

property evaluator: Optional[*Evaluator*]

extend(*extension*, *name=None*, *trigger=None*, *priority=None*, *, *call_before_training=False*, ***kwargs*)

Parameters

- **extension** (*Union[extension.ExtensionLike, ExtensionEntry]*) –
- **name** (*Optional[str]*) –
- **trigger** (*TriggerLike*) –
- **priority** (*Optional[int]*) –
- **call_before_training** (*bool*) –
- **kwargs** (*Any*) –

Return type
None

get_optimizer(*name*)

Parameters
name (*str*) –

Return type
Optimizer

property is_before_training: bool

is_epoch_last_iter(*idx*)

Parameters
idx (*int*) –

Return type
bool

property iteration: int

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

property manager: ExtensionsManager

property models: Mapping[str, Module]

property optimizers: Mapping[str, Optimizer]

run(*train_loader*, *val_loader=None*, ***, *train_len=None*, *eval_len=None*)

Executes the training loop.

Parameters

- **train_loader** (*torch.utils.data.DataLoader*) – A data loader for training.
- **val_loader** (*torch.utils.data.DataLoader, optional*) – A data loader passed to `Evaluator.run()`.
- **train_len** (*int, optional*) – The number of iterations per one training epoch. The default value is inferred from the size of training data loader.
- **eval_len** (*int, optional*) – The number of iterations per one evaluation epoch, passed to `Evaluator.run()`

Return type
None

See also:

- *pytorch_pfn_extras.training._evaluator.Evaluator()*

set_optimizer(*name*, *optimizer*)

Parameters

- **name** (*str*) –
- **optimizer** (*Optimizer*) –

Return type
None

state_dict()

Return type
Dict[str, *Any*]

property stop_trigger: *Trigger*

pytorch_pfn_extras.handler

Functions

pytorch_pfn_extras.handler.forward(block)	Returns a `CodeBlock` that performs the forward pass for the given `torch.nn.Module` or another `CodeBlock`.
pytorch_pfn_extras.handler.torch_autocast([...])	
pytorch_pfn_extras.handler.update_parameters(...)	Returns a `CodeBlock` that performs the forward, backward passes and applies the optimizer step for the given `torch.nn.Module` or another `CodeBlock`.

pytorch_pfn_extras.handler.forward

pytorch_pfn_extras.handler.**forward**(*block*)

Returns a `CodeBlock` that performs the forward pass for the given `torch.nn.Module` or another `CodeBlock`.

Parameters
block (*Callable*) – `torch.nn.Module` or `CodeBlock` to update the parameters.

Return type
CodeBlock

Returns: A `CodeBlock` object.

pytorch_pfn_extras.handler.torch_autocast

pytorch_pfn_extras.handler.**torch_autocast**(*enabled=True*)

Parameters
enabled (*bool*) –

Return type
Generator[None, None, None]

pytorch_pfn_extras.handler.update_parameters

pytorch_pfn_extras.handler.**update_parameters**(*block*, *optimizers*, *backprop_from=None*, *backprop_to=None*)

Returns a `CodeBlock` that performs the forward, backward passes and applies the optimizer step for the given `torch.nn.Module` or another `CodeBlock`.

Parameters

- **block** (*Callable*) – `torch.nn.Module` or `CodeBlock` to update the parameters.
- **optimizers** (*List[Optimizer]*) – The list of Optimizer that will be used for parameter update.
- **backprop_from** (*Optional[str]*) – Select a single output from the block execution to perform the gradient calculation.
- **backprop_to** (*Optional[Set[str]]*) – Name of the values where backpropagation will be stopped.

Return type
CodeBlock

Returns: A `CodeBlock` object.

Classes

pytorch_pfn_extras.handler.BaseHandler(...)	Base class of Handler.
pytorch_pfn_extras.handler.BaseLogic([options])	
pytorch_pfn_extras.handler.ClousureLogic([...])	A set of methods that defines the training logic.
pytorch_pfn_extras.handler.CodeBlock(func, ...)	Class that is used to specify and apply actions to a callable.
pytorch_pfn_extras.handler.CodeBlockLogic([...])	A set of methods that defines the training logic.
pytorch_pfn_extras.handler.Handler(logic, ...)	A set of callback functions to perform device-specific operations.
pytorch_pfn_extras.handler.Logic([...])	A set of methods that defines the training logic.

pytorch_pfn_extras.handler.BaseHandler

class pytorch_pfn_extras.handler.**BaseHandler**(*logic*, *options*, **args*, ***kwargs*)

Bases: `object`

Base class of Handler.

Parameters

- **logic** (*Logic*) – A logic.
- **options** (*Dict[str, Any]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Methods

__init__(logic, options, *args, **kwargs)	Base class of Handler.
consume_options(options)	A method to update options of Handler.
eval_loop_begin(evaluator)	A method called before each evaluation step.
eval_loop_end(evaluator)	A method called after running all steps of the evaluation.
eval_post_step(evaluator, batch_idx, batch, ...)	A method called after each evaluation step.
eval_setup(evaluator, loader)	A method called only once when starting a training run.
eval_step(evaluator, batch_idx, batch, ...)	Evaluation iteration.
train_cleanup(trainer)	A method called only once when compleing a training run.
train_epoch_begin(trainer, loader)	A method called when starting a new epoch.
train_epoch_end(trainer)	A method called when finishing an epoch.
train_post_step(trainer, batch_idx, batch, ...)	A method called after each training step.
train_setup(trainer, loader)	A method called only once when starting a training run.
train_step(trainer, batch_idx, batch, ...)	A training step.
train_validation_begin(trainer, evaluator)	A method called when starting a validation.
train_validation_end(trainer, evaluator)	A method called after validation.

__init__(*logic*, *options*, **args*, ***kwargs*)

Base class of Handler.

Parameters

- **logic** (*Logic*) – A logic.
- **options** (*Dict[str, Any]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

consume_options(*options*)

A method to update options of Handler.

Note that the given dict will be modified.

Parameters
options (*dict*) – Option key-values to be set.

Return type
None

eval_loop_begin(*evaluator*)

A method called before each evaluation step.

Parameters
evaluator (*Evaluator*) – The evaluator.

Return type
None

`eval_loop_end`(*evaluator*)

A method called after running all steps of the evaluation.

Parameters
evaluator (Evaluator) –

Return type
None

`eval_post_step`(*evaluator*, *batch_idx*, *batch*, *outputs*)

A method called after each evaluation step.

Parameters

- **evaluator** (Evaluator) –
- **batch_idx** (*int*) –
- **batch** (*Any*) –
- **outputs** (*Any*) –

Return type
None

`eval_setup`(*evaluator*, *loader*)

A method called only once when starting a training run. When evaluator is not given, this method is not called.

Parameters

- **evaluator** (Evaluator) –
- **loader** (*Iterable[Any]*) –

Return type
None

`eval_step`(*evaluator*, *batch_idx*, *batch*, *complete_fn*)

Evaluation iteration.

Parameters

- **evaluator** (Evaluator) –
- **batch_idx** (*int*) –
- **batch** (*Any*) –
- **complete_fn** (*Callable[[int, Any], None]*) –

Return type
None

`train_cleanup`(*trainer*)

A method called only once when compleing a training run.

Parameters
trainer (Trainer) –

Return type
None

`train_epoch_begin`(*trainer*, *loader*)

A method called when starting a new epoch.

Parameters

- **trainer** (Trainer) –
- **loader** (*Iterable[Any]*) –

Return type
None

`train_epoch_end`(*trainer*)

A method called when finishing an epoch.

Parameters
trainer (Trainer) –

Return type
None

`train_post_step`(*trainer*, *batch_idx*, *batch*, *outputs*)

A method called after each training step.

Parameters

- **trainer** (Trainer) –
- **batch_idx** (*int*) –
- **batch** (*Any*) –
- **outputs** (*Any*) –

Return type
None

`train_setup`(*trainer*, *loader*)

A method called only once when starting a training run.

Parameters

- **trainer** (Trainer) –
- **loader** (*Iterable[Any]*) –

Return type
None

`train_step`(*trainer*, *batch_idx*, *batch*, *complete_fn*)

A training step.

Parameters

- **trainer** (Trainer) –
- **batch_idx** (*int*) –
- **batch** (*Any*) –
- **complete_fn** (*Callable[[int, Any], None]*) –

Return type
None

train_validation_begin(*trainer*, *evaluator*)

A method called when starting a validation.

Parameters

- **trainer** (Trainer) –
- **evaluator** (Evaluator) –

Return type
None

train_validation_end(*trainer*, *evaluator*)

A method called after validation.

Parameters

- **trainer** (Trainer) – The trainer that calls this method.
- **evaluator** (Evaluator) – The evaluator used for validation.

Return type
None

pytorch_pfn_extras.handler.BaseLogic

class pytorch_pfn_extras.handler.**BaseLogic**(*options=None*)

Bases: object

Methods

__init__([options])	

consume_options(options)	A method to update options of Logic.
eval_step(models, batch_idx, batch)	A method for an evaluation step.
train_epoch_begin(models, epoch, loader)	A method called when starting a new epoch of training.
train_epoch_end(models, epoch)	A method called when completing an epoch of training.
train_step(models, optimizers, batch_idx, batch)	A method invokes the models forward and backward passes.
train_step_optimizers(models, optimizers, ...)	A method in charge of stepping the provided optimizers.
train_validation_begin(models)	A method called when starting a validation.
train_validation_end(models)	A method called when the validation completes.

Parameters
options (*Optional[Dict[str, Any]]*) –

__init__(*options=None*)

Parameters
options (*Optional[Dict[str, Any]]*) –

`consume_options`(*options*)

A method to update options of Logic.

Note that the given dict will be modified.

Parameters
options (*dict*) – Option key-values to be set.

Return type
None

`eval_step`(*models*, *batch_idx*, *batch*)

A method for an evaluation step.

Parameters

- **models** (*dict of torch.nn.Module*) – The models.
- **batch_idx** (*int*) – Number of steps already finished.
- **batch** (*torch.Tensor, list of torch.Tensor, dict of torch.Tensor*) – Input tensors feeded to the model of the current step.

Return type
Any

`train_epoch_begin`(*models*, *epoch*, *loader*)

A method called when starting a new epoch of training.

Parameters

- **epoch** (*int*) – Number of epochs already finished.
- **models** (*dict of torch.nn.Module*) – The models.
- **loader** (*torch.utils.data.DataLoader*) – The data loder.

Return type
None

`train_epoch_end`(*models*, *epoch*)

A method called when completing an epoch of training.

Parameters

- **epoch** (*int*) – Number of epochs already finished.
- **models** (*dict of torch.nn.Module*) – The models.

Return type
None

`train_step`(*models*, *optimizers*, *batch_idx*, *batch*)

A method invokes the models forward and backward passes.

Optimizing is left to *train_step_optimizers* since maybe the user would like to aggregate the gradients of several iterations.

Parameters

- **models** (*dict of torch.nn.Module*) – The models.
- **optimizers** (*dict of torch.optim.Optimizer*) – The optimizers.
- **batch_idx** (*int*) – Number of training steps already finished.

- **batch** (*torch.Tensor, list of torch.Tensor, dict of torch.Tensor*) – Input tensors feeded to the model of the current step.

Return type
Any

train_step_optimizers(*models*, *optimizers*, *batch_idx*)

A method in charge of stepping the provided optimizers.

Parameters

- **optimizers** (*dict of torch.optim.Optimizer*) – The optimizers.
- **batch_idx** (*int*) – Number of steps already finished.
- **models** (*Mapping[str, Module]*) –

Return type
None

train_validation_begin(*models*)

A method called when starting a validation.

Parameters
models (*dict of torch.nn.Module*) – The models.

Return type
None

train_validation_end(*models*)

A method called when the validation completes.

Parameters
models (*dict of torch.nn.Module*) – The models.

Return type
None

pytorch_pfn_extras.handler.ClousureLogic

class pytorch_pfn_extras.handler.**ClousureLogic**(*model_name='main'*, *options=None*)

Bases: *Logic*

A set of methods that defines the training logic.

Parameters

- **model_name** (*str*) – Name of the model. Default is `'main'`.
- **options** (*dict, optional*) – The configuration options.
  - **`'backward_outputs'` (list of str):**
    A list of names of outputs that require compution of the gradient.
  - **`'autocast'` (bool or dict):**
    If `True`, `torch.autocast` is enabled, using `{"enabled": True, "device_type": "cuda"}` as autocast options. The default is `False` which corresponds to the following options `{"enabled": False, "device_type": "cuda"}`. If dict, options are passed to `torch.autocast`.
  - **`'grad_scaler'` (torch.cuda.amp.GradScaler):**
    A gradient scaler that outputs are applied to.

Methods

`__init__`([model_name, options])	A set of methods that defines the training logic.
consume_options(options)	A method to update options of Logic.
`eval_step`(models, batch_idx, batch)	A method for an evaluation step.
`train_epoch_begin`(models, epoch, loader)	A method called when starting a new epoch of training.
`train_epoch_end`(models, epoch)	A method called when completing an epoch of training.
train_step(models, optimizers, batch_idx, batch)	A method invokes the model forward and backward passes and performs an optimization step.
train_step_optimizers(models, optimizers, ...)	In clousure mode, the stepping of the optimizer cannot be changed.
`train_validation_begin`(models)	A method called when starting a validation.
`train_validation_end`(models)	A method called when the validation completes.

consume_options(*options*)

A method to update options of Logic.

Note that the given dict will be modified.

Parameters
options (*dict*) – Option key-values to be set.

Return type
None

train_step(*models*, *optimizers*, *batch_idx*, *batch*)

A method invokes the model forward and backward passes and performs an optimization step.

Parameters

- **models** (*dict of torch.nn.Module*) – The models.
- **optimizers** (*dict of torch.optim.Optimizer*) – The optimizers.
- **batch_idx** (*int*) – Number of training steps already finished.
- **batch** (*torch.Tensor, list of torch.Tensor, dict of torch.Tensor*) – Input tensors feeded to the model of the current step.

Return type
Any

train_step_optimizers(*models*, *optimizers*, *batch_idx*)

In clousure mode, the stepping of the optimizer cannot be changed.

If you want to change the stepping of the optimizer, please use the normal Logic class.

Parameters

- **optimizers** (*dict of torch.optim.Optimizer*) – The optimizers.
- **batch_idx** (*int*) – Number of steps already finished.
- **models** (*Mapping[str, Module]*) –

Return type
None

pytorch_pfn_extras.handler.CodeBlock

class pytorch_pfn_extras.handler.**CodeBlock**(*func*, *optimizers*, *backprop*, *backprop_from*, *backprop_to*, *state*, *runtime*)

Bases: object

Class that is used to specify and apply actions to a callable.

CodeBlocks are used in Logic classes to write device agnostic codes, as the device runtime is in charge of doing the execution of the module with the actions requested from the codeblock

Parameters

- **func** (*Callable*) – The function to be operated according to the specified options.
- **optimizer** – The Optimizer that will be used for parameter update.
- **backprop** (*bool*) – Flag to specify if gradients are to be calculated.
- **backprop_from** (*Optional[str]*) – Select a single output from the block execution to perform the gradient calculation.
- **backprop_to** (*Optional[Set[str]]*) – Name of the values where backpropagation will be stopped.
- **state** (*Dict[str, Any]*) – Data that can be used during the CodeBlock execution.
- **optimizers** (*List[Optimizer]*) –
- **runtime** (*Any*) –

Methods

__init__(func, optimizers, backprop, ...)
load_state_dict(state)
state_dict()

Attributes

func
optimizers
backprop
backprop_from
backprop_to
state
runtime

__call__(*inputs*)

Call self as a function.

Parameters
inputs (*Any*) –

Return type
Any

__init__(*func*, *optimizers*, *backprop*, *backprop_from*, *backprop_to*, *state*, *runtime*)

Parameters

- **func** (*Callable*) –
- **optimizers** (*List[Optimizer]*) –
- **backprop** (*bool*) –
- **backprop_from** (*Optional[str]*) –
- **backprop_to** (*Optional[Set[str]]*) –
- **state** (*Dict[str, Any]*) –
- **runtime** (*Any*) –

Return type
None

backprop: bool

backprop_from: Optional[str]

backprop_to: Optional[Set[str]]

func: Callable

load_state_dict(*state*)

Parameters
state (*Dict[str, Any]*) –

Return type
None

optimizers: List[Optimizer]

runtime: Any

state: Dict[str, Any]

state_dict()

Return type
Dict[str, *Any*]

pytorch_pfn_extras.handler.CodeBlockLogic

class pytorch_pfn_extras.handler.**CodeBlockLogic**(*model_name='main'*, *options=None*)

Bases: *BaseLogic*

A set of methods that defines the training logic.

Parameters

- **model_name** (*str*) – Name of the model. Default is `'main'`.
- **options** (*dict, optional*) – The configuration options.
  - **'backward_outputs' (list of str):**
    A list of names of outputs that require compution of the gradient.

Methods

__init__([model_name, options])	A set of methods that defines the training logic.
consume_options(options)	A method to update options of Logic.
eval_step(models, batch_idx, batch)	A method for an evaluation step.
train_epoch_begin(models, epoch, loader)	A method called when starting a new epoch of training.
train_epoch_end(models, epoch)	A method called when completing an epoch of training.
train_step(models, optimizers, batch_idx, batch)	A method invokes the model forward and backward passes.
train_step_optimizers(models, optimizers, ...)	A method in charge of stepping the provided optimizers.
train_validation_begin(models)	A method called when starting a validation.
train_validation_end(models)	A method called when the validation completes.

__init__(*model_name='main'*, *options=None*)

A set of methods that defines the training logic.

Parameters

- **model_name** (*str*) – Name of the model. Default is `'main'`.
- **options** (*dict, optional*) – The configuration options.
  - **'backward_outputs' (list of str):**
    A list of names of outputs that require compution of the gradient.

Return type
None

consume_options(*options*)

A method to update options of Logic.

Note that the given dict will be modified.

Parameters
options (*dict*) – Option key-values to be set.

Return type
None

eval_step(*models*, *batch_idx*, *batch*)

A method for an evaluation step.

Parameters

- **models** (*dict of torch.nn.Module*) – The models.
- **batch_idx** (*int*) – Number of steps already finished.
- **batch** (*torch.Tensor, list of torch.Tensor, dict of torch.Tensor*) – Input tensors feeded to the model of the current step.

Return type
Any

train_epoch_begin(*models*, *epoch*, *loader*)

A method called when starting a new epoch of training.

Parameters

- **epoch** (*int*) – Number of epochs already finished.
- **models** (*dict of torch.nn.Module*) – The models.
- **loader** (*torch.utils.data.DataLoader*) – The data loder.

Return type
None

train_epoch_end(*models*, *epoch*)

A method called when completing an epoch of training.

Parameters

- **epoch** (*int*) – Number of epochs already finished.
- **models** (*dict of torch.nn.Module*) – The models.

Return type
None

train_step(*models*, *optimizers*, *batch_idx*, *batch*)

A method invokes the model forward and backward passes.

Optimizing is left to *train_step_optimizers* since maybe the user would like to aggregate the gradients of several iterations.

Parameters

- **models** (*dict of torch.nn.Module*) – The models.
- **optimizers** (*dict of torch.optim.Optimizer*) – The optimizers.
- **batch_idx** (*int*) – Number of training steps already finished.
- **batch** (*torch.Tensor, list of torch.Tensor, dict of torch.Tensor*) – Input tensors feeded to the model of the current step.

Return type
Any

train_validation_begin(*models*)

A method called when starting a validation.

Parameters
models (*dict of torch.nn.Module*) – The models.

Return type
None

`train_validation_end`(*models*)

A method called when the validation completes.

Parameters
models (*dict of torch.nn.Module*) – The models.

Return type
None

pytorch_pfn_extras.handler.Handler

`class` pytorch_pfn_extras.handler.**Handler**(*logic*, *entry_runtime*, *options*)

Bases: *BaseHandler*

A set of callback functions to perform device-specific operations.

Parameters

- **logic** (Logic) – A logic.
- **entry_runtime** (BaseRuntime) – A runtime object.
- **options** (*dict*) – The configuration options.
  - **'eval_report_keys' (list of str):**
    A list of names of outputs that are given as inputs of `reporting.report` after each evaluation step. Default is an empty list.
  - **'train_report_keys' (list of str):**
    A list of names of outputs that are given as inputs of `reporting.report` after each training step. Default is an empty list.

Methods

__init__(logic, entry_runtime, options)	A set of callback functions to perform device-specific operations.
consume_options(options)	A method to update options of Handler.
eval_loop_begin(evaluator)	A method called before each evaluation step.
eval_loop_end(evaluator)	A method called after running all steps of the evaluation.
eval_post_step(evaluator, batch_idx, batch, ...)	A method called after each evaluation step.
eval_setup(evaluator, loader)	Called only once when starting a training run.
eval_step(evaluator, batch_idx, batch, ...)	Evaluation iteration.
train_cleanup(trainer)	A method called only once when compleing a training run.
train_epoch_begin(trainer, loader)	A method called when starting a new epoch.
train_epoch_end(trainer)	A method called when finishing an epoch.
train_post_step(trainer, batch_idx, batch, ...)	A method called after each training step.
train_setup(trainer, loader)	A method called only once when starting a training run.
train_step(trainer, batch_idx, batch, ...)	A training step.
train_validation_begin(trainer, evaluator)	A method called when starting a validation.
train_validation_end(trainer, evaluator)	A method called after validation.

`__init__`(*logic*, *entry_runtime*, *options*)

A set of callback functions to perform device-specific operations.

Parameters

- **logic** (Logic) – A logic.
- **entry_runtime** (BaseRuntime) – A runtime object.
- **options** (*dict*) – The configuration options.
  - **'eval_report_keys' (list of str):** A list of names of outputs that are given as inputs of `reporting.report` after each evaluation step. Default is an empty list.
  - **'train_report_keys' (list of str):** A list of names of outputs that are given as inputs of `reporting.report` after each training step. Default is an empty list.

Return type
None

`consume_options`(*options*)

A method to update options of Handler.

Note that the given dict will be modified.

Parameters
options (*dict*) – Option key-values to be set.

Return type
None

`eval_loop_end`(*evaluator*)

A method called after running all steps of the evaluation.

Parameters
evaluator (Evaluator) – The evaluator.

Return type
None

`eval_post_step`(*evaluator*, *batch_idx*, *batch*, *outputs*)

A method called after each evaluation step.

Parameters

- **evaluator** (Evaluator) – The evaluator.
- **batch_idx** (*int*) – Number of iterations already finished.
- **batch** (*dict of torch.Tensor*) – Input tensors of this batch.
- **complete_fn** (*callable*) – A callback function called after training step.
- **outputs** (*Any*) –

Return type
None

`eval_setup`(*evaluator*, *loader*)

Called only once when starting a training run. When evaluator is not given, this method is not called.

Parameters

- **evaluator** (Evaluator) – The evaluator.
- **loader** (*torch.utils.data.DataLoader*) – The data loader.

Return type
None

eval_step(*evaluator*, *batch_idx*, *batch*, *complete_fn*)

Evaluation iteration.

Parameters

- **evaluator** (Evaluator) – The evaluator.
- **batch_idx** (*int*) – Number of iterations already finished.
- **batch** (*dict of torch.Tensor*) – Input tensors of this batch.
- **complete_fn** (*callable*) – A callback function called after training step.

Return type
None

train_cleanup(*trainer*)

A method called only once when compleing a training run.

Parameters

- **trainer** (Trainer) – The trainer that calls this method.
- **loader** (*torch.utils.data.DataLoader*) – The data loader.

Return type
None

train_epoch_begin(*trainer*, *loader*)

A method called when starting a new epoch.

Parameters

- **trainer** (Trainer) – The trainer that calls this method.
- **loader** (*torch.utils.data.DataLoader*) – The data loader.

Return type
None

train_epoch_end(*trainer*)

A method called when finishing an epoch.

Parameters
trainer (Trainer) – The trainer that calls this method.

Return type
None

train_post_step(*trainer*, *batch_idx*, *batch*, *outputs*)

A method called after each training step.

Parameters

- **trainer** (Trainer) – The trainer that calls this method.
- **batch_idx** (*int*) – Number of iterations
- **batch** (*dict of torch.Tensor*) – Input tensors of this batch.

- **outputs** (*dict of torch.Tensor*) – Output tensors of this batch.

Return type
None

train_setup(*trainer*, *loader*)

A method called only once when starting a training run.

Parameters

- **trainer** (Trainer) – The trainer that calls this method.
- **loader** (*torch.utils.data.DataLoader*) – The data loader.

Return type
None

train_step(*trainer*, *batch_idx*, *batch*, *complete_fn*)

A training step.

Parameters

- **trainer** (Trainer) – A trainer.
- **batch_idx** (*int*) – Number of iterations already finished.
- **batch** (*dict of torch.Tensor*) – Input tensors of this batch.
- **complete_fn** (*callable*) – A callback function called after training step.

Return type
None

train_validation_begin(*trainer*, *evaluator*)

A method called when starting a validation.

Parameters

- **evaluator** (Evaluator) – An evaluator.
- **trainer** (Trainer) –

Return type
None

train_validation_end(*trainer*, *evaluator*)

A method called after validation.

Parameters

- **trainer** (Trainer) – The trainer that calls this method.
- **evaluator** (Evaluator) – The evaluator used for validation.

Return type
None

pytorch_pfn_extras.handler.Logic

class pytorch_pfn_extras.handler.**Logic**(*model_name='main'*, *options=None*)

Bases: *BaseLogic*

A set of methods that defines the training logic.

Parameters

- **model_name** (*str*) – Name of the model. Default is `'main'`.
- **options** (*dict, optional*) – The configuration options.
  - **`'backward_outputs'` (list of str):**
    A list of names of outputs that require compution of the gradient.
  - **`'autocast'` (bool or dict):**
    If `True`, `torch.autocast` is enabled, using `{"enabled": True, "device_type": "cuda"}` as autocast options. The default is `False` which corresponds to the following options `{"enabled": False, "device_type": "cuda"}`. If dict, options are passed to `torch.autocast`.
  - **`'grad_scaler'` (torch.cuda.amp.GradScaler):**
    A gradient scaler that outputs are applied to.

Methods

__init__([model_name, options])	A set of methods that defines the training logic.
consume_options(options)	A method to update options of Logic.
eval_step(models, batch_idx, batch)	A method for an evaluation step.
train_epoch_begin(models, epoch, loader)	A method called when starting a new epoch of training.
train_epoch_end(models, epoch)	A method called when completing an epoch of training.
train_step(models, optimizers, batch_idx, batch)	A method invokes the model forward and backward passes.
train_step_optimizers(models, optimizers, ...)	A method in charge of stepping the provided optimizers.
train_validation_begin(models)	A method called when starting a validation.
train_validation_end(models)	A method called when the validation completes.

__init__(*model_name='main'*, *options=None*)

A set of methods that defines the training logic.

Parameters

- **model_name** (*str*) – Name of the model. Default is `'main'`.
- **options** (*dict, optional*) – The configuration options.
  - **`'backward_outputs'` (list of str):**
    A list of names of outputs that require compution of the gradient.
  - **`'autocast'` (bool or dict):**
    If `True`, `torch.autocast` is enabled, using `{"enabled": True, "device_type": "cuda"}` as autocast options. The default is `False`

which corresponds to the following options `{"enabled": False, "device_type": "cuda"}`. If dict, options are passed to `torch.autocast`.

- **`'grad_scaler'` (torch.cuda.amp.GradScaler):**
  A gradient scaler that outputs are applied to.

Return type
None

`consume_options`(*options*)

A method to update options of Logic.

Note that the given dict will be modified.

Parameters
`options` (*dict*) – Option key-values to be set.

Return type
None

`eval_step`(*models*, *batch_idx*, *batch*)

A method for an evaluation step.

Parameters

- **`models`** (*dict of torch.nn.Module*) – The models.
- **`batch_idx`** (*int*) – Number of steps already finished.
- **`batch`** (*torch.Tensor, list of torch.Tensor, dict of torch.Tensor*) – Input tensors feeded to the model of the current step.

Return type
Any

`train_epoch_begin`(*models*, *epoch*, *loader*)

A method called when starting a new epoch of training.

Parameters

- **`epoch`** (*int*) – Number of epochs already finished.
- **`models`** (*dict of torch.nn.Module*) – The models.
- **`loader`** (*torch.utils.data.DataLoader*) – The data loder.

Return type
None

`train_epoch_end`(*models*, *epoch*)

A method called when completing an epoch of training.

Parameters

- **`epoch`** (*int*) – Number of epochs already finished.
- **`models`** (*dict of torch.nn.Module*) – The models.

Return type
None

`train_step`(*models*, *optimizers*, *batch_idx*, *batch*)

A method invokes the model forward and backward passes.

Optimizing is left to *train_step_optimizers* since maybe the user would like to aggregate the gradients of several iterations.

Parameters

- **models** (*dict of torch.nn.Module*) – The models.
- **optimizers** (*dict of torch.optim.Optimizer*) – The optimizers.
- **batch_idx** (*int*) – Number of training steps already finished.
- **batch** (*torch.Tensor, list of torch.Tensor, dict of torch.Tensor*) – Input tensors feeded to the model of the current step.

Return type

Any

`train_step_optimizers`(*models*, *optimizers*, *batch_idx*)

A method in charge of stepping the provided optimizers.

Also a grad scaler will be used if defined.

Parameters

- **optimizers** (*dict of torch.optim.Optimizer*) – The optimizers.
- **batch_idx** (*int*) – Number of steps already finished.
- **models** (*Mapping[str, Module]*) –

Return type

None

`train_validation_begin`(*models*)

A method called when starting a validation.

Parameters

models (*dict of torch.nn.Module*) – The models.

Return type

None

`train_validation_end`(*models*)

A method called when the validation completes.

Parameters

models (*dict of torch.nn.Module*) – The models.

Return type

None

pytorch_pfn_extras.logging

Functions

pytorch_pfn_extras.logging.get_logger(name)	Returns a child logger to be used by applications.

pytorch_pfn_extras.logging.get_logger

pytorch_pfn_extras.logging.**get_logger**(*name*)

Returns a child logger to be used by applications.

Parameters
name (*str*) – Name used to register and retrieve the logger object.

Returns
A logging.Logger object used to log in the application code.

Return type
Logger

pytorch_pfn_extras.nn

Functions

pytorch_pfn_extras.nn.ensure(tensor[, ...])	Checks the shape and type of a tensor.

pytorch_pfn_extras.nn.ensure

pytorch_pfn_extras.nn.**ensure**(*tensor*, *shape=None*, *dtype=None*, *broadcastable=False*, *can_cast=False*)

Checks the shape and type of a tensor.

Parameters

- **shape** (*Optional[Tuple[Optional[int], ...]]*) – Tuple with the desired shape. If the input tensor shape is not compatible, *ValueError* will be raised. If *None* is set as a dimension value, that dimension will be ignored.
- **dtype** (*Optional[dtype]*) – Checks if the *dtype* of the input thensor matches the provided one.
- **broadcastable** (*bool*) – Check if the shapes are compatible using broadcasting rules.
- **can_cast** (*bool*) – Check if the input tensor can be casted to the provided type.
- **tensor** (*Tensor*) –

Return type
None

Classes

pytorch_pfn_extras.nn.Ensure(*[, shape, ...])	Module to check the shape of a tensor.
pytorch_pfn_extras.nn.ExtendedSequential()	Sequential module with extended features from chainer.
pytorch_pfn_extras.nn.LazyBatchNorm1d(...)	BatchNorm1d module with lazy weight initialization.
pytorch_pfn_extras.nn.LazyBatchNorm2d(...)	BatchNorm2d module with lazy weight initialization.
pytorch_pfn_extras.nn.LazyBatchNorm3d(...)	BatchNorm3d module with lazy weight initialization.
pytorch_pfn_extras.nn.LazyConv1d(...)	Conv1d module with lazy weight initialization.
pytorch_pfn_extras.nn.LazyConv2d(...)	Conv2d module with lazy weight initialization.
pytorch_pfn_extras.nn.LazyConv3d(...)	Conv3d module with lazy weight initialization.
pytorch_pfn_extras.nn.LazyLinear(...)	Linear module with lazy weight initialization.

pytorch_pfn_extras.nn.Ensure

class pytorch_pfn_extras.nn.**Ensure**(*, *shape=None*, *dtype=None*, *broadcastable=False*, *can_cast=False*)

Bases: Module

Module to check the shape of a tensor.

Parameters

- **shape** (*Optional[Tuple[Optional[int], ...]]*) – Tuple with the desired shape. If the input tensor shape is not compatible, *ValueError* will be raised. If *None* is set as a dimension value, that dimension will be ignored.
- **dtype** (*Optional[dtype]*) – Checks if the *dtype* of the input thensor matches the provided one.
- **broadcastable** (*bool*) – Check if the shapes are compatible using broadcasting rules.
- **can_cast** (*bool*) – Check if the input tensor can be casted to the provided type.

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

Methods

__init__(*[, shape, dtype, broadcastable, ...])	Initializes internal Module state, shared by both nn.Module and ScriptModule.
add_module(name, module)	Adds a child module to the current module.
apply(fn)	Applies fn recursively to every submodule (as returned by .children()) as well as self.
bfloat16()	Casts all floating point parameters and buffers to bfloat16 datatype.
buffers([recurse])	Returns an iterator over module buffers.
children()	Returns an iterator over immediate children modules.
compile(*args, **kwargs)	Compile this Module's forward using torch.compile().
cpu()	Moves all model parameters and buffers to the CPU.
cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to double datatype.
eval()	Sets the module in evaluation mode.

continues on next page

Table 1 – continued from previous page

extra_repr()	Set the extra representation of the module
float()	Casts all floating point parameters and buffers to float datatype.
forward(input)	Defines the computation performed at every call.
get_buffer(target)	Returns the buffer given by target if it exists, otherwise throws an error.
get_extra_state()	Returns any extra state to include in the module's state_dict.
get_parameter(target)	Returns the parameter given by target if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by target if it exists, otherwise throws an error.
half()	Casts all floating point parameters and buffers to half datatype.
ipu([device])	Moves all model parameters and buffers to the IPU.
load_state_dict(state_dict[, strict, assign])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse, ...])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse, ...])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook, *[, prepend, ...])	Registers a forward hook on the module.
register_forward_pre_hook(hook, *[, ...])	Registers a forward pre-hook on the module.
register_full_backward_hook(hook[, prepend])	Registers a backward hook on the module.
register_full_backward_pre_hook(hook[, prepend])	Registers a backward pre-hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
register_state_dict_pre_hook(hook)	These hooks will be called with arguments: self, prefix, and keep_vars before calling state_dict on self.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()

continues on next page

Table 1 – continued from previous page

state_dict(*args[, destination, prefix, ...])	Returns a dictionary containing references to the whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device[, recurse])	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Resets gradients of all model parameters.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
call_super_init	
dump_patches	

__init__(*, *shape=None*, *dtype=None*, *broadcastable=False*, *can_cast=False*)

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

Parameters

- **shape** (*Optional[Tuple[Optional[int], ...]]*) –
- **dtype** (*Optional[dtype]*) –
- **broadcastable** (*bool*) –
- **can_cast** (*bool*) –

forward(*input*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Parameters

input (*Tensor*) –

Return type

Tensor

training: **bool**

pytorch_pfn_extras.nn.ExtendedSequential

class pytorch_pfn_extras.nn.**ExtendedSequential**(**args: Module*)

class pytorch_pfn_extras.nn.**ExtendedSequential**(*arg:* OrderedDict*[str, Module]*)

Bases: Sequential

Sequential module with extended features from chainer.

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

Methods

__init__(*args)	Initializes internal Module state, shared by both nn.Module and ScriptModule.
add_module(name, module)	Adds a child module to the current module.
append(module)	Appends a given module to the end.
apply(fn)	Applies fn recursively to every submodule (as returned by .children()) as well as self.
bfloat16()	Casts all floating point parameters and buffers to bfloat16 datatype.
buffers([recurse])	Returns an iterator over module buffers.
children()	Returns an iterator over immediate children modules.
compile(*args, **kwargs)	Compile this Module's forward using torch.compile().
cpu()	Moves all model parameters and buffers to the CPU.
cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to double datatype.
eval()	Sets the module in evaluation mode.
extend(sequential)	
extra_repr()	Set the extra representation of the module
float()	Casts all floating point parameters and buffers to float datatype.
forward(input)	Defines the computation performed at every call.
get_buffer(target)	Returns the buffer given by target if it exists, otherwise throws an error.
get_extra_state()	Returns any extra state to include in the module's state_dict.
get_parameter(target)	Returns the parameter given by target if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by target if it exists, otherwise throws an error.
half()	Casts all floating point parameters and buffers to half datatype.
insert(index, module)	
ipu([device])	Moves all model parameters and buffers to the IPU.
load_state_dict(state_dict[, strict, assign])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.

continues on next page

Table 2 – continued from previous page

named_buffers([prefix, recurse, ...])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse, ...])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
pop(key)	
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook, *[, prepend, ...])	Registers a forward hook on the module.
register_forward_pre_hook(hook, *[, ...])	Registers a forward pre-hook on the module.
register_full_backward_hook(hook[, prepend])	Registers a backward hook on the module.
register_full_backward_pre_hook(hook[, prepend])	Registers a backward pre-hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
register_state_dict_pre_hook(hook)	These hooks will be called with arguments: self, prefix, and keep_vars before calling state_dict on self.
repeat(n_repeat[, mode])	Repeats this Sequential multiple times.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict(*args[, destination, prefix, ...])	Returns a dictionary containing references to the whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device[, recurse])	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Resets gradients of all model parameters.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
call_super_init	
dump_patches	

repeat(*n_repeat*, *mode='init'*)

Repeats this Sequential multiple times.

This method returns a `Sequential` object which has original *Sequential* multiple times repeatedly. The `mode` argument means how to copy this sequential to repeat.

The functions is supposed to behave the same way as *repeat* in *chainer*.

When the mode is set to `init`, the default value, modules will be copied and reinitialized by calling `reset_parameters` (or `_reset_parameters`) method.

To repeat user-defined modules, which have parameters or buffers, with mode=``init`` in this Sequential, you need to implement the `reset_parameters` or `_reset_parameters` method to the module to reinitialize parameters and (if necessary) buffers; otherwise the initialization cannot be performed and a warning message will be shown.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned `Sequential` will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting `Sequential` object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

Return type
ExtendedSequential

pytorch_pfn_extras.nn.LazyBatchNorm1d

class pytorch_pfn_extras.nn.**LazyBatchNorm1d**(*num_features*, **args*, ***kwargs*)

Bases: `_LazyBatchNorm`, `BatchNorm1d`

BatchNorm1d module with lazy weight initialization.

When `num_features` is `None`, it is determined at the first time of the forward step.

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

Methods

`__init__`(num_features, *args, **kwargs)	Initializes internal Module state, shared by both nn.Module and ScriptModule.
`add_module`(name, module)	Adds a child module to the current module.
`apply`(fn)	Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self.
`bfloat16`()	Casts all floating point parameters and buffers to `bfloat16` datatype.
`buffers`([recurse])	Returns an iterator over module buffers.
`children`()	Returns an iterator over immediate children modules.
`compile`(*args, **kwargs)	Compile this Module's forward using `torch.compile()`.
`cpu`()	Moves all model parameters and buffers to the CPU.
`cuda`([device])	Moves all model parameters and buffers to the GPU.
`double`()	Casts all floating point parameters and buffers to `double` datatype.
`eval`()	Sets the module in evaluation mode.
`extra_repr`()	Set the extra representation of the module
`float`()	Casts all floating point parameters and buffers to `float` datatype.
`forward`(input)	Defines the computation performed at every call.
`get_buffer`(target)	Returns the buffer given by `target` if it exists, otherwise throws an error.
`get_extra_state`()	Returns any extra state to include in the module's state_dict.
`get_parameter`(target)	Returns the parameter given by `target` if it exists, otherwise throws an error.
`get_submodule`(target)	Returns the submodule given by `target` if it exists, otherwise throws an error.
`half`()	Casts all floating point parameters and buffers to `half` datatype.
`ipu`([device])	Moves all model parameters and buffers to the IPU.
`load_state_dict`(state_dict[, strict, assign])	Copies parameters and buffers from `state_dict` into this module and its descendants.
`modules`()	Returns an iterator over all modules in the network.
`named_buffers`([prefix, recurse, ...])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
`named_children`()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
`named_modules`([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
`named_parameters`([prefix, recurse, ...])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
`parameters`([recurse])	Returns an iterator over module parameters.
`register_backward_hook`(hook)	Registers a backward hook on the module.
`register_buffer`(name, tensor[, persistent])	Adds a buffer to the module.
`register_forward_hook`(hook, *[, prepend, ...])	Registers a forward hook on the module.

continues on next page

Table 3 – continued from previous page

register_forward_pre_hook(hook, *[, ...])	Registers a forward pre-hook on the module.
register_full_backward_hook(hook[, prepend])	Registers a backward hook on the module.
register_full_backward_pre_hook(hook[, prepend])	Registers a backward pre-hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
register_state_dict_pre_hook(hook)	These hooks will be called with arguments: self, prefix, and keep_vars before calling state_dict on self.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
reset_parameters()	
reset_running_stats()	
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict(*args, **kwargs)	Returns a dictionary containing a whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device[, recurse])	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Resets gradients of all model parameters.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
call_super_init	
dump_patches	
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

Parameters

- **num_features** (*int*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

running_mean: *Any*

running_var: *Any*

pytorch_pfn_extras.nn.LazyBatchNorm2d

class pytorch_pfn_extras.nn.**LazyBatchNorm2d**(*num_features*, **args*, ***kwargs*)

Bases: `_LazyBatchNorm`, `BatchNorm2d`

BatchNorm2d module with lazy weight initialization.

When `num_features` is `None`, it is determined at the first time of the forward step.

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

Methods

`__init__`(num_features, *args, **kwargs)	Initializes internal Module state, shared by both nn.Module and ScriptModule.
`add_module`(name, module)	Adds a child module to the current module.
`apply`(fn)	Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self.
`bfloat16`()	Casts all floating point parameters and buffers to `bfloat16` datatype.
`buffers`([recurse])	Returns an iterator over module buffers.
`children`()	Returns an iterator over immediate children modules.
`compile`(*args, **kwargs)	Compile this Module's forward using `torch.compile()`.
`cpu`()	Moves all model parameters and buffers to the CPU.
`cuda`([device])	Moves all model parameters and buffers to the GPU.
`double`()	Casts all floating point parameters and buffers to `double` datatype.
`eval`()	Sets the module in evaluation mode.
`extra_repr`()	Set the extra representation of the module
`float`()	Casts all floating point parameters and buffers to `float` datatype.
`forward`(input)	Defines the computation performed at every call.
`get_buffer`(target)	Returns the buffer given by `target` if it exists, otherwise throws an error.
`get_extra_state`()	Returns any extra state to include in the module's state_dict.
`get_parameter`(target)	Returns the parameter given by `target` if it exists, otherwise throws an error.
`get_submodule`(target)	Returns the submodule given by `target` if it exists, otherwise throws an error.
`half`()	Casts all floating point parameters and buffers to `half` datatype.
`ipu`([device])	Moves all model parameters and buffers to the IPU.
`load_state_dict`(state_dict[, strict, assign])	Copies parameters and buffers from `state_dict` into this module and its descendants.
`modules`()	Returns an iterator over all modules in the network.

continues on next page

Table 4 – continued from previous page

named_buffers([prefix, recurse, ...])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse, ...])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook, *[, prepend, ...])	Registers a forward hook on the module.
register_forward_pre_hook(hook, *[, ...])	Registers a forward pre-hook on the module.
register_full_backward_hook(hook[, prepend])	Registers a backward hook on the module.
register_full_backward_pre_hook(hook[, prepend])	Registers a backward pre-hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
register_state_dict_pre_hook(hook)	These hooks will be called with arguments: self, prefix, and keep_vars before calling state_dict on self.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
reset_parameters()	
reset_running_stats()	
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict(*args, **kwargs)	Returns a dictionary containing a whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device[, recurse])	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Resets gradients of all model parameters.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
call_super_init	
dump_patches	
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

Parameters

- **num_features** (*int*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

running_mean: Any

running_var: Any

pytorch_pfn_extras.nn.LazyBatchNorm3d

class pytorch_pfn_extras.nn.**LazyBatchNorm3d**(*num_features*, **args*, ***kwargs*)

Bases: _LazyBatchNorm, BatchNorm3d

BatchNorm3d module with lazy weight initialization.

When num_features is None, it is determined at the first time of the forward step.

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

Methods

__init__(num_features, *args, **kwargs)	Initializes internal Module state, shared by both nn.Module and ScriptModule.
add_module(name, module)	Adds a child module to the current module.
apply(fn)	Applies fn recursively to every submodule (as returned by .children()) as well as self.
bfloat16()	Casts all floating point parameters and buffers to bfloat16 datatype.
buffers([recurse])	Returns an iterator over module buffers.
children()	Returns an iterator over immediate children modules.
compile(*args, **kwargs)	Compile this Module's forward using torch.compile().
cpu()	Moves all model parameters and buffers to the CPU.
cuda([device])	Moves all model parameters and buffers to the GPU.

continues on next page

Table 5 – continued from previous page

double()	Casts all floating point parameters and buffers to double datatype.
eval()	Sets the module in evaluation mode.
extra_repr()	Set the extra representation of the module
float()	Casts all floating point parameters and buffers to float datatype.
forward(input)	Defines the computation performed at every call.
get_buffer(target)	Returns the buffer given by target if it exists, otherwise throws an error.
get_extra_state()	Returns any extra state to include in the module's state_dict.
get_parameter(target)	Returns the parameter given by target if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by target if it exists, otherwise throws an error.
half()	Casts all floating point parameters and buffers to half datatype.
ipu([device])	Moves all model parameters and buffers to the IPU.
load_state_dict(state_dict[, strict, assign])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse, ...])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse, ...])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook, *[, prepend, ...])	Registers a forward hook on the module.
register_forward_pre_hook(hook, *[, ...])	Registers a forward pre-hook on the module.
register_full_backward_hook(hook[, prepend])	Registers a backward hook on the module.
register_full_backward_pre_hook(hook[, prepend])	Registers a backward pre-hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
register_state_dict_pre_hook(hook)	These hooks will be called with arguments: self, prefix, and keep_vars before calling state_dict on self.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.

continues on next page

Table 5 – continued from previous page

reset_parameters()	
reset_running_stats()	
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict(*args, **kwargs)	Returns a dictionary containing a whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device[, recurse])	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Resets gradients of all model parameters.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
call_super_init	
dump_patches	
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

Parameters

- **num_features** (*int*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

running_mean: Any

running_var: Any

pytorch_pfn_extras.nn.LazyConv1d

class pytorch_pfn_extras.nn.**LazyConv1d**(*in_channels*, **args*, ***kwargs*)

Bases: _LazyConvNd, Conv1d

Conv1d module with lazy weight initialization.

When `in_channels` is `None`, it is determined at the first time of the forward step.

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

Methods

`__init__`(in_channels, *args, **kwargs)	Initializes internal Module state, shared by both nn.Module and ScriptModule.
`add_module`(name, module)	Adds a child module to the current module.
`apply`(fn)	Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self.
`bfloat16`()	Casts all floating point parameters and buffers to `bfloat16` datatype.
`buffers`([recurse])	Returns an iterator over module buffers.
`children`()	Returns an iterator over immediate children modules.
`compile`(*args, **kwargs)	Compile this Module's forward using `torch.compile()`.
`cpu`()	Moves all model parameters and buffers to the CPU.
`cuda`([device])	Moves all model parameters and buffers to the GPU.
`double`()	Casts all floating point parameters and buffers to `double` datatype.
`eval`()	Sets the module in evaluation mode.
`extra_repr`()	Set the extra representation of the module
`float`()	Casts all floating point parameters and buffers to `float` datatype.
`forward`(input)	Defines the computation performed at every call.
`get_buffer`(target)	Returns the buffer given by `target` if it exists, otherwise throws an error.
`get_extra_state`()	Returns any extra state to include in the module's state_dict.
`get_parameter`(target)	Returns the parameter given by `target` if it exists, otherwise throws an error.
`get_submodule`(target)	Returns the submodule given by `target` if it exists, otherwise throws an error.
`half`()	Casts all floating point parameters and buffers to `half` datatype.
`ipu`([device])	Moves all model parameters and buffers to the IPU.
`load_state_dict`(state_dict[, strict, assign])	Copies parameters and buffers from `state_dict` into this module and its descendants.
`modules`()	Returns an iterator over all modules in the network.
`named_buffers`([prefix, recurse, ...])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
`named_children`()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

continues on next page

Table 6 – continued from previous page

named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse, ...])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook, *[, prepend, ...])	Registers a forward hook on the module.
register_forward_pre_hook(hook, *[, ...])	Registers a forward pre-hook on the module.
register_full_backward_hook(hook[, prepend])	Registers a backward hook on the module.
register_full_backward_pre_hook(hook[, prepend])	Registers a backward pre-hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
register_state_dict_pre_hook(hook)	These hooks will be called with arguments: self, prefix, and keep_vars before calling state_dict on self.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
reset_parameters()	
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict(*args, **kwargs)	Returns a dictionary containing a whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device[, recurse])	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Resets gradients of all model parameters.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
call_super_init	
dump_patches	
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

Parameters

- **in_channels** (*int*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

pytorch_pfn_extras.nn.LazyConv2d

class pytorch_pfn_extras.nn.**LazyConv2d**(*in_channels*, **args*, ***kwargs*)

Bases: _LazyConvNd, Conv2d

Conv2d module with lazy weight initialization.

When in_channels is None, it is determined at the first time of the forward step.

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

Methods

__init__(in_channels, *args, **kwargs)	Initializes internal Module state, shared by both nn.Module and ScriptModule.
add_module(name, module)	Adds a child module to the current module.
apply(fn)	Applies fn recursively to every submodule (as returned by .children()) as well as self.
bfloat16()	Casts all floating point parameters and buffers to bfloat16 datatype.
buffers([recurse])	Returns an iterator over module buffers.
children()	Returns an iterator over immediate children modules.
compile(*args, **kwargs)	Compile this Module's forward using torch.compile().
cpu()	Moves all model parameters and buffers to the CPU.
cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to double datatype.
eval()	Sets the module in evaluation mode.
extra_repr()	Set the extra representation of the module

continues on next page

Table 7 – continued from previous page

float()	Casts all floating point parameters and buffers to float datatype.
forward(input)	Defines the computation performed at every call.
get_buffer(target)	Returns the buffer given by target if it exists, otherwise throws an error.
get_extra_state()	Returns any extra state to include in the module's state_dict.
get_parameter(target)	Returns the parameter given by target if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by target if it exists, otherwise throws an error.
half()	Casts all floating point parameters and buffers to half datatype.
ipu([device])	Moves all model parameters and buffers to the IPU.
load_state_dict(state_dict[, strict, assign])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse, ...])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse, ...])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook, *[, prepend, ...])	Registers a forward hook on the module.
register_forward_pre_hook(hook, *[, ...])	Registers a forward pre-hook on the module.
register_full_backward_hook(hook[, prepend])	Registers a backward hook on the module.
register_full_backward_pre_hook(hook[, prepend])	Registers a backward pre-hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
register_state_dict_pre_hook(hook)	These hooks will be called with arguments: self, prefix, and keep_vars before calling state_dict on self.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
reset_parameters()	
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()

continues on next page

Table 7 – continued from previous page

state_dict(*args, **kwargs)	Returns a dictionary containing a whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device[, recurse])	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Resets gradients of all model parameters.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
call_super_init	
dump_patches	
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

Parameters

- **in_channels** (*int*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

pytorch_pfn_extras.nn.LazyConv3d

class pytorch_pfn_extras.nn.**LazyConv3d**(*in_channels*, **args*, ***kwargs*)

Bases: _LazyConvNd, Conv3d

Conv3d module with lazy weight initialization.

When in_channels is None, it is determined at the first time of the forward step.

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

Methods

`__init__`(in_channels, *args, **kwargs)	Initializes internal Module state, shared by both nn.Module and ScriptModule.
`add_module`(name, module)	Adds a child module to the current module.
`apply`(fn)	Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self.
`bfloat16`()	Casts all floating point parameters and buffers to `bfloat16` datatype.
`buffers`([recurse])	Returns an iterator over module buffers.
`children`()	Returns an iterator over immediate children modules.
`compile`(*args, **kwargs)	Compile this Module's forward using `torch.compile()`.
`cpu`()	Moves all model parameters and buffers to the CPU.
`cuda`([device])	Moves all model parameters and buffers to the GPU.
`double`()	Casts all floating point parameters and buffers to `double` datatype.
`eval`()	Sets the module in evaluation mode.
`extra_repr`()	Set the extra representation of the module
`float`()	Casts all floating point parameters and buffers to `float` datatype.
`forward`(input)	Defines the computation performed at every call.
`get_buffer`(target)	Returns the buffer given by `target` if it exists, otherwise throws an error.
`get_extra_state`()	Returns any extra state to include in the module's state_dict.
`get_parameter`(target)	Returns the parameter given by `target` if it exists, otherwise throws an error.
`get_submodule`(target)	Returns the submodule given by `target` if it exists, otherwise throws an error.
`half`()	Casts all floating point parameters and buffers to `half` datatype.
`ipu`([device])	Moves all model parameters and buffers to the IPU.
`load_state_dict`(state_dict[, strict, assign])	Copies parameters and buffers from `state_dict` into this module and its descendants.
`modules`()	Returns an iterator over all modules in the network.
`named_buffers`([prefix, recurse, ...])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
`named_children`()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
`named_modules`([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
`named_parameters`([prefix, recurse, ...])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
`parameters`([recurse])	Returns an iterator over module parameters.
`register_backward_hook`(hook)	Registers a backward hook on the module.
`register_buffer`(name, tensor[, persistent])	Adds a buffer to the module.
`register_forward_hook`(hook, *[, prepend, ...])	Registers a forward hook on the module.

continues on next page

Table 8 – continued from previous page

register_forward_pre_hook(hook, *[, ...])	Registers a forward pre-hook on the module.
register_full_backward_hook(hook[, prepend])	Registers a backward hook on the module.
register_full_backward_pre_hook(hook[, prepend])	Registers a backward pre-hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
register_state_dict_pre_hook(hook)	These hooks will be called with arguments: self, prefix, and keep_vars before calling state_dict on self.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
reset_parameters()	
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict(*args, **kwargs)	Returns a dictionary containing a whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device[, recurse])	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Resets gradients of all model parameters.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
call_super_init	
dump_patches	
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

Parameters

- **in_channels** (*int*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

pytorch_pfn_extras.nn.LazyLinear

class pytorch_pfn_extras.nn.**LazyLinear**(*in_features*, **args*, ***kwargs*)

Bases: *LazyInitializationMixin*, Linear

Linear module with lazy weight initialization.

When `in_features` is `None`, it is determined at the first time of the forward step.

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

Methods

__init__(in_features, *args, **kwargs)	Initializes internal Module state, shared by both nn.Module and ScriptModule.
add_module(name, module)	Adds a child module to the current module.
apply(fn)	Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self.
bfloat16()	Casts all floating point parameters and buffers to `bfloat16` datatype.
buffers([recurse])	Returns an iterator over module buffers.
children()	Returns an iterator over immediate children modules.
compile(*args, **kwargs)	Compile this Module's forward using `torch.compile()`.
cpu()	Moves all model parameters and buffers to the CPU.
cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to `double` datatype.
eval()	Sets the module in evaluation mode.
extra_repr()	Set the extra representation of the module
float()	Casts all floating point parameters and buffers to `float` datatype.
forward(input)	Defines the computation performed at every call.
get_buffer(target)	Returns the buffer given by `target` if it exists, otherwise throws an error.
get_extra_state()	Returns any extra state to include in the module's state_dict.
get_parameter(target)	Returns the parameter given by `target` if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by `target` if it exists, otherwise throws an error.
half()	Casts all floating point parameters and buffers to `half` datatype.
ipu([device])	Moves all model parameters and buffers to the IPU.
load_state_dict(state_dict[, strict, assign])	Copies parameters and buffers from `state_dict` into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse, ...])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

continues on next page

Table 9 – continued from previous page

named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse, ...])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook, *[, prepend, ...])	Registers a forward hook on the module.
register_forward_pre_hook(hook, *[, ...])	Registers a forward pre-hook on the module.
register_full_backward_hook(hook[, prepend])	Registers a backward hook on the module.
register_full_backward_pre_hook(hook[, prepend])	Registers a backward pre-hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
register_state_dict_pre_hook(hook)	These hooks will be called with arguments: self, prefix, and keep_vars before calling state_dict on self.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
reset_parameters()	
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict(*args, **kwargs)	Returns a dictionary containing a whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device[, recurse])	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Resets gradients of all model parameters.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
call_super_init	
dump_patches	
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

Parameters

- **in_features** (*int*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

__init__(*in_features*, **args*, ***kwargs*)

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

Parameters

- **in_features** (*Optional[int]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

forward(*input*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Parameters
input (*Tensor*) –

Return type
Tensor

lazy_parameter_names: Tuple[str, ...] = ('weight',)

reset_parameters()

Return type
None

Modules

pytorch_pfn_extras.nn.modules

Modules

pytorch_pfn_extras.nn.modules.ensure_shape

Functions

pytorch_pfn_extras.nn.modules.ensure_shape.ensure(tensor)	Checks the shape and type of a tensor.

pytorch_pfn_extras.nn.modules.ensure_shape.ensure

pytorch_pfn_extras.nn.modules.ensure_shape.**ensure**(*tensor*, *shape=None*, *dtype=None*, *broadcastable=False*, *can_cast=False*)

Checks the shape and type of a tensor.

Parameters

- **shape** (*Optional[Tuple[Optional[int], ...]]*) – Tuple with the desired shape. If the input tensor shape is not compatible, *ValueError* will be raised. If *None* is set as a dimension value, that dimension will be ignored.
- **dtype** (*Optional[dtype]*) – Checks if the *dtype* of the input thensor matches the provided one.
- **broadcastable** (*bool*) – Check if the shapes are compatible using broadcasting rules.
- **can_cast** (*bool*) – Check if the input tensor can be casted to the provided type.

- **tensor** (*Tensor*) –

Return type
None

Classes

pytorch_pfn_extras.nn.modules.ensure_shape.Ensure(*)	Module to check the shape of a tensor.

pytorch_pfn_extras.nn.modules.ensure_shape.Ensure

class pytorch_pfn_extras.nn.modules.ensure_shape.**Ensure**(*, *shape=None*, *dtype=None*, *broadcastable=False*, *can_cast=False*)

Bases: `Module`

Module to check the shape of a tensor.

Parameters

- **shape** (*Optional[Tuple[Optional[int], ...]]*) – Tuple with the desired shape. If the input tensor shape is not compatible, *ValueError* will be raised. If *None* is set as a dimension value, that dimension will be ignored.
- **dtype** (*Optional[dtype]*) – Checks if the *dtype* of the input thensor matches the provided one.
- **broadcastable** (*bool*) – Check if the shapes are compatible using broadcasting rules.
- **can_cast** (*bool*) – Check if the input tensor can be casted to the provided type.

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

Methods

__init__(*[, shape, dtype, broadcastable, ...])	Initializes internal Module state, shared by both nn.Module and ScriptModule.
add_module(name, module)	Adds a child module to the current module.
apply(fn)	Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self.
bfloat16()	Casts all floating point parameters and buffers to `bfloat16` datatype.
buffers([recurse])	Returns an iterator over module buffers.
children()	Returns an iterator over immediate children modules.
compile(*args, **kwargs)	Compile this Module's forward using `torch.compile()`.
cpu()	Moves all model parameters and buffers to the CPU.
cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to `double` datatype.
eval()	Sets the module in evaluation mode.
extra_repr()	Set the extra representation of the module

continues on next page

Table 10 – continued from previous page

float()	Casts all floating point parameters and buffers to float datatype.
forward(input)	Defines the computation performed at every call.
get_buffer(target)	Returns the buffer given by target if it exists, otherwise throws an error.
get_extra_state()	Returns any extra state to include in the module's state_dict.
get_parameter(target)	Returns the parameter given by target if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by target if it exists, otherwise throws an error.
half()	Casts all floating point parameters and buffers to half datatype.
ipu([device])	Moves all model parameters and buffers to the IPU.
load_state_dict(state_dict[, strict, assign])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse, ...])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse, ...])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook, *[, prepend, ...])	Registers a forward hook on the module.
register_forward_pre_hook(hook, *[, ...])	Registers a forward pre-hook on the module.
register_full_backward_hook(hook[, prepend])	Registers a backward hook on the module.
register_full_backward_pre_hook(hook[, prepend])	Registers a backward pre-hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
register_state_dict_pre_hook(hook)	These hooks will be called with arguments: self, prefix, and keep_vars before calling state_dict on self.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict(*args[, destination, prefix, ...])	Returns a dictionary containing references to the whole state of the module.

continues on next page

Table 10 – continued from previous page

to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device[, recurse])	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Resets gradients of all model parameters.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
call_super_init	
dump_patches	

__init__(*, *shape=None*, *dtype=None*, *broadcastable=False*, *can_cast=False*)

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

Parameters

- **shape** (*Optional[Tuple[Optional[int], ...]]*) –
- **dtype** (*Optional[dtype]*) –
- **broadcastable** (*bool*) –
- **can_cast** (*bool*) –

forward(*input*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Parameters

input (*Tensor*) –

Return type

Tensor

training: **bool**

pytorch_pfn_extras.nn.modules.extended_sequential

Classes

pytorch_pfn_extras.nn.modules.extended_sequential.ExtendedSequential()	Sequential module with extended features from chainer.
pytorch_pfn_extras.nn.modules.extended_sequential.TypeVar(...)	Type variable.

pytorch_pfn_extras.nn.modules.extended_sequential.ExtendedSequential

class pytorch_pfn_extras.nn.modules.extended_sequential.**ExtendedSequential**(**args: Module*)

class pytorch_pfn_extras.nn.modules.extended_sequential.**ExtendedSequential**(*arg:* OrderedDict*[str, Module]*)

Bases: `Sequential`

Sequential module with extended features from chainer.

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

Methods

`__init__`(*args)	Initializes internal Module state, shared by both nn.Module and ScriptModule.
`add_module`(name, module)	Adds a child module to the current module.
`append`(module)	Appends a given module to the end.
`apply`(fn)	Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self.
`bfloat16`()	Casts all floating point parameters and buffers to `bfloat16` datatype.
`buffers`([recurse])	Returns an iterator over module buffers.
`children`()	Returns an iterator over immediate children modules.
`compile`(*args, **kwargs)	Compile this Module's forward using `torch.compile()`.
`cpu`()	Moves all model parameters and buffers to the CPU.
`cuda`([device])	Moves all model parameters and buffers to the GPU.
`double`()	Casts all floating point parameters and buffers to `double` datatype.
`eval`()	Sets the module in evaluation mode.
`extend`(sequential)	
`extra_repr`()	Set the extra representation of the module
`float`()	Casts all floating point parameters and buffers to `float` datatype.
`forward`(input)	Defines the computation performed at every call.
`get_buffer`(target)	Returns the buffer given by `target` if it exists, otherwise throws an error.

continues on next page

Table 11 – continued from previous page

get_extra_state()	Returns any extra state to include in the module's state_dict.
get_parameter(target)	Returns the parameter given by target if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by target if it exists, otherwise throws an error.
half()	Casts all floating point parameters and buffers to half datatype.
insert(index, module)	
ipu([device])	Moves all model parameters and buffers to the IPU.
load_state_dict(state_dict[, strict, assign])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse, ...])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse, ...])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
pop(key)	
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook, *[, prepend, ...])	Registers a forward hook on the module.
register_forward_pre_hook(hook, *[, ...])	Registers a forward pre-hook on the module.
register_full_backward_hook(hook[, prepend])	Registers a backward hook on the module.
register_full_backward_pre_hook(hook[, prepend])	Registers a backward pre-hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
register_state_dict_pre_hook(hook)	These hooks will be called with arguments: self, prefix, and keep_vars before calling state_dict on self.
repeat(n_repeat[, mode])	Repeats this Sequential multiple times.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict(*args[, destination, prefix, ...])	Returns a dictionary containing references to the whole state of the module.

continues on next page

Table 11 – continued from previous page

to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device[, recurse])	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Resets gradients of all model parameters.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
call_super_init	
dump_patches	

repeat(*n_repeat*, *mode='init'*)

Repeats this Sequential multiple times.

This method returns a `Sequential` object which has original *Sequential* multiple times repeatedly. The `mode` argument means how to copy this sequential to repeat.

The functions is supposed to behave the same way as *repeat* in *chainer*.

When the mode is set to `init`, the default value, modules will be copied and reinitialized by calling `reset_parameters` (or `_reset_parameters`) method.

To repeat user-defined modules, which have parameters or buffers, with mode=``init`` in this Sequential, you need to implement the `reset_parameters` or `_reset_parameters` method to the module to reinitialize parameters and (if necessary) buffers; otherwise the initialization cannot be performed and a warning message will be shown.

Parameters

- **n_repeat** (*int*) – Number of times to repeat.
- **mode** (*str*) – It should be either `init`, `copy`, or `share`. `init` means parameters of each repeated element in the returned `Sequential` will be re-initialized, so that all elements have different initial parameters. `copy` means that the parameters will not be re-initialized but object itself will be deep-copied, so that all elements have same initial parameters but can be changed independently. `share` means all the elements which consist the resulting `Sequential` object are same object because they are shallow-copied, so that all parameters of elements are shared with each other.

Return type

ExtendedSequential

training: bool

pytorch_pfn_extras.nn.modules.extended_sequential.TypeVar

class pytorch_pfn_extras.nn.modules.extended_sequential.**TypeVar**(*name*, **constraints*, *bound=None*, *covariant=False*, *contravariant=False*)

Bases: _Final, _Immutable

Type variable.

Usage:

```
T = TypeVar('T')  # Can be anything
A = TypeVar('A', str, bytes)  # Must be str or bytes
```

Type variables exist primarily for the benefit of static type checkers. They serve as the parameters for generic types as well as for generic function definitions. See class Generic for more information on generic types. Generic functions work as follows:

def repeat(x: T, n: int) -> List[T]:
'''Return a list containing n references to x.''' return [x]*n

def longest(x: A, y: A) -> A:
'''Return the longest of two strings.''' return x if len(x) >= len(y) else y

The latter example's signature is essentially the overloading of (str, str) -> str and (bytes, bytes) -> bytes. Also note that if the arguments are instances of some subclass of str, the return type is still plain str.

At runtime, isinstance(x, T) and issubclass(C, T) will raise TypeError.

Type variables defined with covariant=True or contravariant=True can be used to declare covariant or contravariant generic types. See PEP 484 for more details. By default generic types are invariant in all type variables.

Type variables can be introspected. e.g.:

T.__name__ == 'T' T.__constraints__ == () T.__covariant__ == False T.__contravariant__ = False A.__constraints__ == (str, bytes)

Note that only type variables defined in global scope can be pickled.

Methods

__init__(name, *constraints[, bound, ...])

__init__(*name*, **constraints*, *bound=None*, *covariant=False*, *contravariant=False*)

pytorch_pfn_extras.nn.modules.lazy

Classes

pytorch_pfn_extras.nn.modules.lazy.LazyInitializationMixin(...)	A mixin for modules that lazily initialize buffers and parameters.
pytorch_pfn_extras.nn.modules.lazy.UninitializedParameter([...])	

pytorch_pfn_extras.nn.modules.lazy.LazyInitializationMixin

class pytorch_pfn_extras.nn.modules.lazy.**LazyInitializationMixin**(**args*, ***kwargs*)

Bases: `object`

A mixin for modules that lazily initialize buffers and parameters.

Unlike regular modules, subclasses of this module can initialize buffers and parameters outside of the constructor (`__init__`). This allows you to, for example, initialize parameters in `forward` method to determine the shape of the weight based on the initial input.

Be sure to run "dummy" forward once to initialize all parameters that should be trained, before passing `module.parameters()` to an optimizer; otherwise weights initialized after `module.parameters()` (e.g., in `forward` function) will never be trained.

Note that lazy modules cannot validate if the shape is correct during deserialization. Also note that the initial weights may become different from the original (non-lazy) module even if the random seed is manually configured, as the order of initialization is different from the original one; especially, `module.cuda()` may cause the initialization to run on a GPU.

The default value of lazy buffers and parameters are `torch.Tensor([])` and `UninitializedParameter()`, respectively.

Methods

__init__(*args, **kwargs)	
state_dict(*args, **kwargs)	Returns a dictionary containing a whole state of the module.

Attributes

lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

__init__(**args*, ***kwargs*)

Parameters

- **self** (*Any*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

lazy_buffer_names: Tuple[str, ...] = ()

lazy_parameter_names: Tuple[str, ...] = ()

property lazy_parmeters_determined: bool

Returns if all lazy parameters are determined.

Subclasses can perform parameters initialization after all lazy parameters are determined. Note that this may be called during `__init__`.

state_dict(**args*, ***kwargs*)

Returns a dictionary containing a whole state of the module.

This function overrides the default behavior to exclude uninitialized parameter from serialization. This is needed because we need to discriminate lazy parameters (`UninitializedParameter()`) and initialized empty parameters (``torch.nn.Parameter(torch.Tensor())`) during deserialization.

See comments of `_lazy_load_hook` for details.

Parameters

- **self** (*Any*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Dict[str, *Any*]

pytorch_pfn_extras.nn.modules.lazy.UninitializedParameter

class pytorch_pfn_extras.nn.modules.lazy.**UninitializedParameter**(*data=None*, *requires_grad=True*)

Bases: `Parameter`

Methods

`__init__`()	

`abs`()	See `torch.abs()`
`abs_`()	In-place version of `abs()`
`absolute`()	Alias for `abs()`
`absolute_`()	In-place version of `absolute()` Alias for `abs_()`
`acos`()	See `torch.acos()`
`acos_`()	In-place version of `acos()`
`acosh`()	See `torch.acosh()`
`acosh_`()	In-place version of `acosh()`
`add`(other, *[, alpha])	Add a scalar or tensor to `self` tensor.
`add_`(other, *[, alpha])	In-place version of `add()`
`addbmm`(batch1, batch2, *[, beta, alpha])	See `torch.addbmm()`
`addbmm_`(batch1, batch2, *[, beta, alpha])	In-place version of `addbmm()`
`addcdiv`(tensor1, tensor2, *[, value])	See `torch.addcdiv()`

continues on next page

Table 12 – continued from previous page

addcdiv_(tensor1, tensor2, *[, value])	In-place version of addcdiv()
addcmul(tensor1, tensor2, *[, value])	See torch.addcmul()
addcmul_(tensor1, tensor2, *[, value])	In-place version of addcmul()
addmm(mat1, mat2, *[, beta, alpha])	See torch.addmm()
addmm_(mat1, mat2, *[, beta, alpha])	In-place version of addmm()
addmv(mat, vec, *[, beta, alpha])	See torch.addmv()
addmv_(mat, vec, *[, beta, alpha])	In-place version of addmv()
addr(vec1, vec2, *[, beta, alpha])	See torch.addr()
addr_(vec1, vec2, *[, beta, alpha])	In-place version of addr()
adjoint()	Alias for adjoint()
align_as(other)	Permutes the dimensions of the self tensor to match the dimension order in the other tensor, adding size-one dims for any new names.
align_to(*names)	Permutes the dimensions of the self tensor to match the order specified in names, adding size-one dims for any new names.
all([dim, keepdim])	See torch.all()
allclose(other[, rtol, atol, equal_nan])	See torch.allclose()
amax([dim, keepdim])	See torch.amax()
amin([dim, keepdim])	See torch.amin()
aminmax(*[, dim, keepdim])	See torch.aminmax()
angle()	See torch.angle()
any([dim, keepdim])	See torch.any()
apply_(callable)	Applies the function callable to each element in the tensor, replacing each element with the value returned by callable.
arccos()	See torch.arccos()
arccos_()	In-place version of arccos()
arccosh	acosh() -> Tensor
arccosh_	acosh_() -> Tensor
arcsin()	See torch.arcsin()
arcsin_()	In-place version of arcsin()
arcsinh()	See torch.arcsinh()
arcsinh_()	In-place version of arcsinh()
arctan()	See torch.arctan()
arctan2(other)	See torch.arctan2()
arctan2_	atan2_(other) -> Tensor
arctan_()	In-place version of arctan()
arctanh()	See torch.arctanh()
arctanh_(other)	In-place version of arctanh()
argmax([dim, keepdim])	See torch.argmax()
argmin([dim, keepdim])	See torch.argmin()
argsort([dim, descending])	See torch.argsort()
argwhere()	See torch.argwhere()
as_strided(size, stride[, storage_offset])	See torch.as_strided()
as_strided_(size, stride[, storage_offset])	In-place version of as_strided()
as_strided_scatter(src, size, stride[, ...])	See torch.as_strided_scatter()
as_subclass(cls)	Makes a cls instance with the same data pointer as self.
asin()	See torch.asin()

continues on next page

Table 12 – continued from previous page

asin_()	In-place version of asin()
asinh()	See torch.asinh()
asinh_()	In-place version of asinh()
atan()	See torch.atan()
atan2(other)	See torch.atan2()
atan2_(other)	In-place version of atan2()
atan_()	In-place version of atan()
atanh()	See torch.atanh()
atanh_(other)	In-place version of atanh()
backward([gradient, retain_graph, ...])	Computes the gradient of current tensor wrt graph leaves.
baddbmm(batch1, batch2, *[, beta, alpha])	See torch.baddbmm()
baddbmm_(batch1, batch2, *[, beta, alpha])	In-place version of baddbmm()
bernoulli(*[, generator])	Returns a result tensor where each result[i] is independently sampled from Bernoulli(self[i]).
bernoulli_([p, generator])	Fills each location of self with an independent sample from Bernoulli(p).
bfloat16([memory_format])	self.bfloat16() is equivalent to self.to(torch.bfloat16).
bincount([weights, minlength])	See torch.bincount()
bitwise_and()	See torch.bitwise_and()
bitwise_and_()	In-place version of bitwise_and()
bitwise_left_shift(other)	See torch.bitwise_left_shift()
bitwise_left_shift_(other)	In-place version of bitwise_left_shift()
bitwise_not()	See torch.bitwise_not()
bitwise_not_()	In-place version of bitwise_not()
bitwise_or()	See torch.bitwise_or()
bitwise_or_()	In-place version of bitwise_or()
bitwise_right_shift(other)	See torch.bitwise_right_shift()
bitwise_right_shift_(other)	In-place version of bitwise_right_shift()
bitwise_xor()	See torch.bitwise_xor()
bitwise_xor_()	In-place version of bitwise_xor()
bmm(batch2)	See torch.bmm()
bool([memory_format])	self.bool() is equivalent to self.to(torch.bool).
broadcast_to(shape)	See torch.broadcast_to().
byte([memory_format])	self.byte() is equivalent to self.to(torch.uint8).
cauchy_([median, sigma, generator])	Fills the tensor with numbers drawn from the Cauchy distribution:
ccol_indices	
cdouble([memory_format])	self.cdouble() is equivalent to self.to(torch.complex128).
ceil()	See torch.ceil()
ceil_()	In-place version of ceil()
cfloat([memory_format])	self.cfloat() is equivalent to self.to(torch.complex64).
chalf([memory_format])	self.chalf() is equivalent to self.to(torch.complex32).

continues on next page

Table 12 – continued from previous page

char([memory_format])	self.char() is equivalent to self.to(torch.int8).
cholesky([upper])	See torch.cholesky()
cholesky_inverse([upper])	See torch.cholesky_inverse()
cholesky_solve(input2[, upper])	See torch.cholesky_solve()
chunk(chunks[, dim])	See torch.chunk()
clamp([min, max])	See torch.clamp()
clamp_([min, max])	In-place version of clamp()
clamp_max	
clamp_max_	
clamp_min	
clamp_min_	
clip([min, max])	Alias for clamp().
clip_([min, max])	Alias for clamp_().
clone(*[, memory_format])	See torch.clone()
coalesce()	Returns a coalesced copy of self if self is an uncoalesced tensor.
col_indices()	Returns the tensor containing the column indices of the self tensor when self is a sparse CSR tensor of layout sparse_csr.
conj()	See torch.conj()
conj_physical()	See torch.conj_physical()
conj_physical_()	In-place version of conj_physical()
contiguous([memory_format])	Returns a contiguous in memory tensor containing the same data as self tensor.
copy_(src[, non_blocking])	Copies the elements from src into self tensor and returns self.
copysign(other)	See torch.copysign()
copysign_(other)	In-place version of copysign()
corrcoef()	See torch.corrcoef()
cos()	See torch.cos()
cos_()	In-place version of cos()
cosh()	See torch.cosh()
cosh_()	In-place version of cosh()
count_nonzero([dim])	See torch.count_nonzero()
cov(*[, correction, fweights, aweights])	See torch.cov()
cpu([memory_format])	Returns a copy of this object in CPU memory.
cross(other[, dim])	See torch.cross()
crow_indices()	Returns the tensor containing the compressed row indices of the self tensor when self is a sparse CSR tensor of layout sparse_csr.
cuda([device, non_blocking, memory_format])	Returns a copy of this object in CUDA memory.
cummax(dim)	See torch.cummax()
cummin(dim)	See torch.cummin()
cumprod(dim[, dtype])	See torch.cumprod()
cumprod_(dim[, dtype])	In-place version of cumprod()
cumsum(dim[, dtype])	See torch.cumsum()

continues on next page

Table 12 – continued from previous page

cumsum_(dim[, dtype])	In-place version of cumsum()
data_ptr()	Returns the address of the first element of self tensor.
deg2rad()	See torch.deg2rad()
deg2rad_()	In-place version of deg2rad()
dense_dim()	Return the number of dense dimensions in a sparse tensor self.
dequantize()	Given a quantized Tensor, dequantize it and return the dequantized float Tensor.
det()	See torch.det()
detach	Returns a new Tensor, detached from the current graph.
detach_	Detaches the Tensor from the graph that created it, making it a leaf.
diag([diagonal])	See torch.diag()
diag_embed([offset, dim1, dim2])	See torch.diag_embed()
diagflat([offset])	See torch.diagflat()
diagonal([offset, dim1, dim2])	See torch.diagonal()
diagonal_scatter(src[, offset, dim1, dim2])	See torch.diagonal_scatter()
diff([n, dim, prepend, append])	See torch.diff()
digamma()	See torch.digamma()
digamma_()	In-place version of digamma()
dim()	Returns the number of dimensions of self tensor.
dim_order()	Returns a tuple of int describing the dim order or physical layout of self.
dist(other[, p])	See torch.dist()
div(value, *[, rounding_mode])	See torch.div()
div_(value, *[, rounding_mode])	In-place version of div()
divide(value, *[, rounding_mode])	See torch.divide()
divide_(value, *[, rounding_mode])	In-place version of divide()
dot(other)	See torch.dot()
double([memory_format])	self.double() is equivalent to self.to(torch.float64).
dsplit(split_size_or_sections)	See torch.dsplit()
eig([eigenvectors])	
element_size()	Returns the size in bytes of an individual element.
eq(other)	See torch.eq()
eq_(other)	In-place version of eq()
equal(other)	See torch.equal()
erf()	See torch.erf()
erf_()	In-place version of erf()
erfc()	See torch.erfc()
erfc_()	In-place version of erfc()
erfinv()	See torch.erfinv()
erfinv_()	In-place version of erfinv()
exp()	See torch.exp()
exp2()	See torch.exp2()
exp2_()	In-place version of exp2()
exp_()	In-place version of exp()

continues on next page

Table 12 – continued from previous page

expand(*sizes)	Returns a new view of the self tensor with singleton dimensions expanded to a larger size.
expand_as(other)	Expand this tensor to the same size as other.
expm1()	See torch.expm1()
expm1_()	In-place version of expm1()
exponential_([lambd, generator])	Fills self tensor with elements drawn from the exponential distribution:
fill_(value)	Fills self tensor with the specified value.
fill_diagonal_(fill_value[, wrap])	Fill the main diagonal of a tensor that has at least 2-dimensions.
fix()	See torch.fix().
fix_()	In-place version of fix()
flatten([start_dim, end_dim])	See torch.flatten()
flip(dims)	See torch.flip()
fliplr()	See torch.fliplr()
flipud()	See torch.flipud()
float([memory_format])	self.float() is equivalent to self.to(torch.float32).
float_power(exponent)	See torch.float_power()
float_power_(exponent)	In-place version of float_power()
floor()	See torch.floor()
floor_()	In-place version of floor()
floor_divide(value)	See torch.floor_divide()
floor_divide_(value)	In-place version of floor_divide()
fmax(other)	See torch.fmax()
fmin(other)	See torch.fmin()
fmod(divisor)	See torch.fmod()
fmod_(divisor)	In-place version of fmod()
frac()	See torch.frac()
frac_()	In-place version of frac()
frexp(input)	See torch.frexp()
gather(dim, index)	See torch.gather()
gcd(other)	See torch.gcd()
gcd_(other)	In-place version of gcd()
ge(other)	See torch.ge().
ge_(other)	In-place version of ge().
geometric_(p, *[, generator])	Fills self tensor with elements drawn from the geometric distribution:
geqrf()	See torch.geqrf()
ger(vec2)	See torch.ger()
get_device()	For CUDA tensors, this function returns the device ordinal of the GPU on which the tensor resides.
greater(other)	See torch.greater().
greater_(other)	In-place version of greater().
greater_equal(other)	See torch.greater_equal().
greater_equal_(other)	In-place version of greater_equal().
gt(other)	See torch.gt().
gt_(other)	In-place version of gt().
half([memory_format])	self.half() is equivalent to self.to(torch.float16).

continues on next page

Table 12 – continued from previous page

hardshrink([lambd])	See torch.nn.functional.hardshrink()
has_names	Is True if any of this tensor's dimensions are named.
heaviside(values)	See torch.heaviside()
heaviside_(values)	In-place version of heaviside()
histc([bins, min, max])	See torch.histc()
histogram(input, bins, *[, range, weight, ...])	See torch.histogram()
hsplit(split_size_or_sections)	See torch.hsplit()
hypot(other)	See torch.hypot()
hypot_(other)	In-place version of hypot()
i0()	See torch.i0()
i0_()	In-place version of i0()
igamma(other)	See torch.igamma()
igamma_(other)	In-place version of igamma()
igammac(other)	See torch.igammac()
igammac_(other)	In-place version of igammac()
index_add(dim, index, source, *[, alpha])	Out-of-place version of torch.Tensor.index_add_().
index_add_(dim, index, source, *[, alpha])	Accumulate the elements of alpha times source into the self tensor by adding to the indices in the order given in index.
index_copy(dim, index, tensor2)	Out-of-place version of torch.Tensor.index_copy_().
index_copy_(dim, index, tensor)	Copies the elements of tensor into the self tensor by selecting the indices in the order given in index.
index_fill(dim, index, value)	Out-of-place version of torch.Tensor.index_fill_().
index_fill_(dim, index, value)	Fills the elements of the self tensor with value value by selecting the indices in the order given in index.
index_put(indices, values[, accumulate])	Out-place version of index_put_().
index_put_(indices, values[, accumulate])	Puts values from the tensor values into the tensor self using the indices specified in indices (which is a tuple of Tensors).
index_reduce	
index_reduce_(dim, index, source, reduce, *)	Accumulate the elements of source into the self tensor by accumulating to the indices in the order given in index using the reduction given by the reduce argument.
index_select(dim, index)	See torch.index_select()
indices()	Return the indices tensor of a sparse COO tensor.
inner(other)	See torch.inner().
int([memory_format])	self.int() is equivalent to self.to(torch.int32).
int_repr()	Given a quantized Tensor, self.int_repr() returns a CPU Tensor with uint8_t as data type that stores the underlying uint8_t values of the given Tensor.
inverse()	See torch.inverse()
ipu([device, non_blocking, memory_format])	Returns a copy of this object in IPU memory.

continues on next page

Table 12 – continued from previous page

is_coalesced()	Returns True if self is a sparse COO tensor that is coalesced, False otherwise.
is_complex()	Returns True if the data type of self is a complex data type.
is_conj()	Returns True if the conjugate bit of self is set to true.
is_contiguous([memory_format])	Returns True if self tensor is contiguous in memory in the order specified by memory format.
is_distributed	
is_floating_point()	Returns True if the data type of self is a floating point data type.
is_inference()	See torch.is_inference()
is_neg()	Returns True if the negative bit of self is set to true.
is_nonzero	
is_pinned	Returns true if this tensor resides in pinned memory.
is_same_size	
is_set_to(tensor)	Returns True if both tensors are pointing to the exact same memory (same storage, offset, size and stride).
is_shared()	Checks if tensor is in shared memory.
is_signed()	Returns True if the data type of self is a signed data type.
isclose(other[, rtol, atol, equal_nan])	See torch.isclose()
isfinite()	See torch.isfinite()
isinf()	See torch.isinf()
isnan()	See torch.isnan()
isneginf()	See torch.isneginf()
isposinf()	See torch.isposinf()
isreal()	See torch.isreal()
istft(n_fft[, hop_length, win_length, ...])	See torch.istft()
item()	Returns the value of this tensor as a standard Python number.
kron(other)	See torch.kron()
kthvalue(k[, dim, keepdim])	See torch.kthvalue()
lcm(other)	See torch.lcm()
lcm_(other)	In-place version of lcm()
ldexp(other)	See torch.ldexp()
ldexp_(other)	In-place version of ldexp()
le(other)	See torch.le().
le_(other)	In-place version of le().
lerp(end, weight)	See torch.lerp()
lerp_(end, weight)	In-place version of lerp()
less	lt(other) -> Tensor
less_(other)	In-place version of less().
less_equal(other)	See torch.less_equal().
less_equal_(other)	In-place version of less_equal().
lgamma()	See torch.lgamma()
lgamma_()	In-place version of lgamma()
log()	See torch.log()
log10()	See torch.log10()

continues on next page

Table 12 – continued from previous page

log10_()	In-place version of log10()
log1p()	See torch.log1p()
log1p_()	In-place version of log1p()
log2()	See torch.log2()
log2_()	In-place version of log2()
log_()	In-place version of log()
log_normal_([mean, std, generator])	Fills self tensor with numbers samples from the log-normal distribution parameterized by the given mean μ and standard deviation σ.
log_softmax	
logaddexp(other)	See torch.logaddexp()
logaddexp2(other)	See torch.logaddexp2()
logcumsumexp(dim)	See torch.logcumsumexp()
logdet()	See torch.logdet()
logical_and()	See torch.logical_and()
logical_and_()	In-place version of logical_and()
logical_not()	See torch.logical_not()
logical_not_()	In-place version of logical_not()
logical_or()	See torch.logical_or()
logical_or_()	In-place version of logical_or()
logical_xor()	See torch.logical_xor()
logical_xor_()	In-place version of logical_xor()
logit()	See torch.logit()
logit_()	In-place version of logit()
logsumexp(dim[, keepdim])	See torch.logsumexp()
long([memory_format])	self.long() is equivalent to self.to(torch.int64).
lstsq(other)	
lt(other)	See torch.lt().
lt_(other)	In-place version of lt().
lu([pivot, get_infos])	See torch.lu()
lu_solve(LU_data, LU_pivots)	See torch.lu_solve()
map2_	
map_(tensor, callable)	Applies callable for each element in self tensor and the given tensor and stores the results in self tensor.
masked_fill(mask, value)	Out-of-place version of torch.Tensor.masked_fill_()
masked_fill_(mask, value)	Fills elements of self tensor with value where mask is True.
masked_scatter(mask, tensor)	Out-of-place version of torch.Tensor.masked_scatter_()
masked_scatter_(mask, source)	Copies elements from source into self tensor at positions where the mask is True.
masked_select(mask)	See torch.masked_select()
materialize(shape[, device, dtype])	Create a Parameter with the same properties of the uninitialized one.
matmul(tensor2)	See torch.matmul()

continues on next page

Table 12 – continued from previous page

matrix_exp()	See torch.matrix_exp()
matrix_power(n)	**Note:** matrix_power() is deprecated, use torch.linalg.matrix_power() instead.
max([dim, keepdim])	See torch.max()
maximum(other)	See torch.maximum()
mean([dim, keepdim, dtype])	See torch.mean()
median([dim, keepdim])	See torch.median()
min([dim, keepdim])	See torch.min()
minimum(other)	See torch.minimum()
mm(mat2)	See torch.mm()
mode([dim, keepdim])	See torch.mode()
moveaxis(source, destination)	See torch.moveaxis()
movedim(source, destination)	See torch.movedim()
msort()	See torch.msort()
mul(value)	See torch.mul().
mul_(value)	In-place version of mul().
multinomial(num_samples[, replacement, ...])	See torch.multinomial()
multiply(value)	See torch.multiply().
multiply_(value)	In-place version of multiply().
mv(vec)	See torch.mv()
mvlgamma(p)	See torch.mvlgamma()
mvlgamma_(p)	In-place version of mvlgamma()
nan_to_num([nan, posinf, neginf])	See torch.nan_to_num().
nan_to_num_([nan, posinf, neginf])	In-place version of nan_to_num().
nanmean([dim, keepdim, dtype])	See torch.nanmean()
nanmedian([dim, keepdim])	See torch.nanmedian()
nanquantile(q[, dim, keepdim, interpolation])	See torch.nanquantile()
nansum([dim, keepdim, dtype])	See torch.nansum()
narrow(dimension, start, length)	See torch.narrow().
narrow_copy(dimension, start, length)	See torch.narrow_copy().
ndimension()	Alias for dim()
ne(other)	See torch.ne().
ne_(other)	In-place version of ne().
neg()	See torch.neg()
neg_()	In-place version of neg()
negative()	See torch.negative()
negative_()	In-place version of negative()
nelement()	Alias for numel()
new	
new_empty(size, *[, dtype, device, ...])	Returns a Tensor of size size filled with uninitialized data.
new_empty_strided(size, stride[, dtype, ...])	Returns a Tensor of size size and strides stride filled with uninitialized data.
new_full(size, fill_value, *[, dtype, ...])	Returns a Tensor of size size filled with fill_value.
new_ones(size, *[, dtype, device, ...])	Returns a Tensor of size size filled with 1.

continues on next page

Table 12 – continued from previous page

new_tensor(data, *[, dtype, device, ...])	Returns a new Tensor with data as the tensor data.
new_zeros(size, *[, dtype, device, ...])	Returns a Tensor of size size filled with 0.
nextafter(other)	See torch.nextafter()
nextafter_(other)	In-place version of nextafter()
nonzero()	See torch.nonzero()
nonzero_static(input, *, size[, fill_value])	Returns a 2-D tensor where each row is the index for a non-zero value.
norm([p, dim, keepdim, dtype])	See torch.norm()
normal_([mean, std, generator])	Fills self tensor with elements samples from the normal distribution parameterized by mean and std.
not_equal(other)	See torch.not_equal().
not_equal_(other)	In-place version of not_equal().
numel()	See torch.numel()
numpy(*[, force])	Returns the tensor as a NumPy ndarray.
orgqr(input2)	See torch.orgqr()
ormqr(input2, input3[, left, transpose])	See torch.ormqr()
outer(vec2)	See torch.outer().
permute(*dims)	See torch.permute()
pin_memory()	Copies the tensor to pinned memory, if it's not already pinned.
pinverse()	See torch.pinverse()
polygamma(n)	See torch.polygamma()
polygamma_(n)	In-place version of polygamma()
positive()	See torch.positive()
pow(exponent)	See torch.pow()
pow_(exponent)	In-place version of pow()
prelu	
prod([dim, keepdim, dtype])	See torch.prod()
put(input, index, source[, accumulate])	Out-of-place version of torch.Tensor.put_().
put_(index, source[, accumulate])	Copies the elements from source into the positions specified by index.
q_per_channel_axis()	Given a Tensor quantized by linear (affine) per-channel quantization, returns the index of dimension on which per-channel quantization is applied.
q_per_channel_scales()	Given a Tensor quantized by linear (affine) per-channel quantization, returns a Tensor of scales of the underlying quantizer.
q_per_channel_zero_points()	Given a Tensor quantized by linear (affine) per-channel quantization, returns a tensor of zero_points of the underlying quantizer.
q_scale()	Given a Tensor quantized by linear(affine) quantization, returns the scale of the underlying quantizer().
q_zero_point()	Given a Tensor quantized by linear(affine) quantization, returns the zero_point of the underlying quantizer().
qr([some])	See torch.qr()
qscheme()	Returns the quantization scheme of a given QTensor.
quantile(q[, dim, keepdim, interpolation])	See torch.quantile()
rad2deg()	See torch.rad2deg()
rad2deg_()	In-place version of rad2deg()

continues on next page

Table 12 – continued from previous page

random_([from, to, generator])	Fills self tensor with numbers sampled from the discrete uniform distribution over [from, to - 1].
ravel()	see torch.ravel()
reciprocal()	See torch.reciprocal()
reciprocal_()	In-place version of reciprocal()
record_stream(stream)	Ensures that the tensor memory is not reused for another tensor until all current work queued on stream are complete.
refine_names(*names)	Refines the dimension names of self according to names.
register_hook(hook)	Registers a backward hook.
register_post_accumulate_grad_hook(hook)	Registers a backward hook that runs after grad accumulation.
reinforce(reward)	
relu	
relu_	
remainder(divisor)	See torch.remainder()
remainder_(divisor)	In-place version of remainder()
rename(*names, **rename_map)	Renames dimension names of self.
rename_(*names, **rename_map)	In-place version of rename().
renorm(p, dim, maxnorm)	See torch.renorm()
renorm_(p, dim, maxnorm)	In-place version of renorm()
repeat(*sizes)	Repeats this tensor along the specified dimensions.
repeat_interleave(repeats[, dim, output_size])	See torch.repeat_interleave().
requires_grad_([requires_grad])	Change if autograd should record operations on this tensor: sets this tensor's requires_grad attribute in-place.
reshape(*shape)	Returns a tensor with the same data and number of elements as self but with the specified shape.
reshape_as(other)	Returns this tensor as the same shape as other.
resize(*sizes)	
resize_(*sizes[, memory_format])	Resizes self tensor to the specified size.
resize_as(tensor)	
resize_as_(tensor[, memory_format])	Resizes the self tensor to be the same size as the specified tensor.
resize_as_sparse_	
resolve_conj()	See torch.resolve_conj()
resolve_neg()	See torch.resolve_neg()
retain_grad()	Enables this Tensor to have their grad populated during backward().
roll(shifts, dims)	See torch.roll()
rot90(k, dims)	See torch.rot90()
round([decimals])	See torch.round()
round_([decimals])	In-place version of round()

continues on next page

Table 12 – continued from previous page

row_indices	
rsqrt()	See torch.rsqrt()
rsqrt_()	In-place version of rsqrt()
scatter(dim, index, src)	Out-of-place version of torch.Tensor.scatter_()
scatter_(dim, index, src[, reduce])	Writes all values from the tensor src into self at the indices specified in the index tensor.
scatter_add(dim, index, src)	Out-of-place version of torch.Tensor.scatter_add_()
scatter_add_(dim, index, src)	Adds all values from the tensor src into self at the indices specified in the index tensor in a similar fashion as scatter_().
scatter_reduce(dim, index, src, reduce, *[, ...])	Out-of-place version of torch.Tensor.scatter_reduce_()
scatter_reduce_(dim, index, src, reduce, *)	Reduces all values from the src tensor to the indices specified in the index tensor in the self tensor using the applied reduction defined via the reduce argument ("sum", "prod", "mean", "amax", "amin").
select(dim, index)	See torch.select()
select_scatter(src, dim, index)	See torch.select_scatter()
set_([source, storage_offset, size, stride])	Sets the underlying storage, size, and strides.
sgn()	See torch.sgn()
sgn_()	In-place version of sgn()
*share_memory_*()	Moves the underlying storage to shared memory.
short([memory_format])	self.short() is equivalent to self.to(torch.int16).
sigmoid()	See torch.sigmoid()
sigmoid_()	In-place version of sigmoid()
sign()	See torch.sign()
sign_()	In-place version of sign()
signbit()	See torch.signbit()
sin()	See torch.sin()
sin_()	In-place version of sin()
sinc()	See torch.sinc()
sinc_()	In-place version of sinc()
sinh()	See torch.sinh()
sinh_()	In-place version of sinh()
size([dim])	Returns the size of the self tensor.
slice_scatter(src[, dim, start, end, step])	See torch.slice_scatter()
slogdet()	See torch.slogdet()
smm(mat)	See torch.smm()
softmax(dim)	Alias for torch.nn.functional.softmax().
solve(other)	
sort([dim, descending])	See torch.sort()
sparse_dim()	Return the number of sparse dimensions in a sparse tensor self.
sparse_mask(mask)	Returns a new sparse tensor with values from a strided tensor self filtered by the indices of the sparse tensor mask.

continues on next page

Table 12 – continued from previous page

sparse_resize_(size, sparse_dim, dense_dim)	Resizes self sparse tensor to the desired size and the number of sparse and dense dimensions.
sparse_resize_and_clear_(size, sparse_dim, ...)	Removes all specified elements from a sparse tensor self and resizes self to the desired size and the number of sparse and dense dimensions.
split(split_size[, dim])	See torch.split()
split_with_sizes	
sqrt()	See torch.sqrt()
sqrt_()	In-place version of sqrt()
square()	See torch.square()
square_()	In-place version of square()
squeeze([dim])	See torch.squeeze()
squeeze_([dim])	In-place version of squeeze()
sspaddmm(mat1, mat2, *[, beta, alpha])	See torch.sspaddmm()
std([dim, correction, keepdim])	See torch.std()
stft(n_fft[, hop_length, win_length, ...])	See torch.stft()
storage()	Returns the underlying TypedStorage.
storage_offset()	Returns self tensor's offset in the underlying storage in terms of number of storage elements (not bytes).
storage_type()	Returns the type of the underlying storage.
stride(dim)	Returns the stride of self tensor.
sub(other, *[, alpha])	See torch.sub().
sub_(other, *[, alpha])	In-place version of sub()
subtract(other, *[, alpha])	See torch.subtract().
subtract_(other, *[, alpha])	In-place version of subtract().
sum([dim, keepdim, dtype])	See torch.sum()
sum_to_size(*size)	Sum this tensor to size.
svd([some, compute_uv])	See torch.svd()
swapaxes(axis0, axis1)	See torch.swapaxes()
swapaxes_(axis0, axis1)	In-place version of swapaxes()
swapdims(dim0, dim1)	See torch.swapdims()
swapdims_(dim0, dim1)	In-place version of swapdims()
symeig([eigenvectors])	
t()	See torch.t()
t_()	In-place version of t()
take(indices)	See torch.take()
take_along_dim(indices, dim)	See torch.take_along_dim()
tan()	See torch.tan()
tan_()	In-place version of tan()
tanh()	See torch.tanh()
tanh_()	In-place version of tanh()
tensor_split(indices_or_sections[, dim])	See torch.tensor_split()
tile(dims)	See torch.tile()
to(*args, **kwargs)	Performs Tensor dtype and/or device conversion.
to_dense([dtype, masked_grad])	Creates a strided copy of self if self is not a strided tensor, otherwise returns self.
to_mkldnn()	Returns a copy of the tensor in torch.mkldnn layout.
to_padded_tensor(padding[, output_size])	See to_padded_tensor()

continues on next page

Table 12 – continued from previous page

to_sparse(sparseDims)	Returns a sparse copy of the tensor.
to_sparse_bsc(blocksize, dense_dim)	Convert a tensor to a block sparse column (BSC) storage format of given blocksize.
to_sparse_bsr(blocksize, dense_dim)	Convert a tensor to a block sparse row (BSR) storage format of given blocksize.
to_sparse_coo()	Convert a tensor to coordinate format.
to_sparse_csc()	Convert a tensor to compressed column storage (CSC) format.
to_sparse_csr([dense_dim])	Convert a tensor to compressed row storage format (CSR).
tolist()	Returns the tensor as a (nested) list.
topk(k[, dim, largest, sorted])	See torch.topk()
trace()	See torch.trace()
transpose(dim0, dim1)	See torch.transpose()
transpose_(dim0, dim1)	In-place version of transpose()
triangular_solve(A[, upper, transpose, ...])	See torch.triangular_solve()
tril([diagonal])	See torch.tril()
tril_([diagonal])	In-place version of tril()
triu([diagonal])	See torch.triu()
triu_([diagonal])	In-place version of triu()
true_divide(value)	See torch.true_divide()
true_divide_(value)	In-place version of true_divide_()
trunc()	See torch.trunc()
trunc_()	In-place version of trunc()
type([dtype, non_blocking])	Returns the type if *dtype* is not provided, else casts this object to the specified type.
type_as(tensor)	Returns this tensor cast to the type of the given tensor.
unbind([dim])	See torch.unbind()
unflatten(dim, sizes)	See torch.unflatten().
unfold(dimension, size, step)	Returns a view of the original tensor which contains all slices of size size from self tensor in the dimension dimension.
uniform_([from, to, generator])	Fills self tensor with numbers sampled from the continuous uniform distribution:
unique([sorted, return_inverse, ...])	Returns the unique elements of the input tensor.
unique_consecutive([return_inverse, ...])	Eliminates all but the first element from every consecutive group of equivalent elements.
unsafe_chunk(chunks[, dim])	See torch.unsafe_chunk()
unsafe_split(split_size[, dim])	See torch.unsafe_split()
unsafe_split_with_sizes	
unsqueeze(dim)	See torch.unsqueeze()
unsqueeze_(dim)	In-place version of unsqueeze()
untyped_storage()	Returns the underlying UntypedStorage.
values()	Return the values tensor of a sparse COO tensor.
var([dim, correction, keepdim])	See torch.var()
vdot(other)	See torch.vdot()
view(*shape)	Returns a new tensor with the same data as the self tensor but of a different shape.
view_as(other)	View this tensor as the same size as other.
vsplit(split_size_or_sections)	See torch.vsplit()

continues on next page

Table 12 – continued from previous page

where(condition, y)	self.where(condition, y) is equivalent to torch.where(condition, self, y).
xlogy(other)	See torch.xlogy()
xlogy_(other)	In-place version of xlogy()
xpu([device, non_blocking, memory_format])	Returns a copy of this object in XPU memory.
zero_()	Fills self tensor with zeros.

Attributes

H	Returns a view of a matrix (2-D tensor) conjugated and transposed.
T	Returns a view of this tensor with its dimensions reversed.
data	
device	Is the torch.device where this Tensor is.
dtype	
grad	This attribute is None by default and becomes a Tensor the first time a call to backward() computes gradients for self.
grad_fn	
imag	Returns a new tensor containing imaginary values of the self tensor.
is_cpu	Is True if the Tensor is stored on the CPU, False otherwise.
is_cuda	Is True if the Tensor is stored on the GPU, False otherwise.
is_ipu	Is True if the Tensor is stored on the IPU, False otherwise.
is_leaf	All Tensors that have requires_grad which is False will be leaf Tensors by convention.
is_meta	Is True if the Tensor is a meta tensor, False otherwise.
is_mkldnn	
is_mps	Is True if the Tensor is stored on the MPS device, False otherwise.
is_nested	
is_ort	
is_quantized	Is True if the Tensor is quantized, False otherwise.
is_sparse	Is True if the Tensor uses sparse COO storage layout, False otherwise.
is_sparse_csr	Is True if the Tensor uses sparse CSR storage layout, False otherwise.

continues on next page

Table 13 – continued from previous page

is_vulkan	
is_xla	Is True if the Tensor is stored on an XLA device, False otherwise.
is_xpu	Is True if the Tensor is stored on the XPU, False otherwise.
itemsize	Alias for element_size()
layout	
mH	Accessing this property is equivalent to calling adjoint().
mT	Returns a view of this tensor with the last two dimensions transposed.
name	
names	Stores names for each of this tensor's dimensions.
nbytes	Returns the number of bytes consumed by the "view" of elements of the Tensor if the Tensor does not use sparse storage layout.
ndim	Alias for dim()
output_nr	
real	Returns a new tensor containing real values of the self tensor for a complex-valued input tensor.
requires_grad	Is True if gradients need to be computed for this Tensor, False otherwise.
retains_grad	Is True if this Tensor is non-leaf and its grad is enabled to be populated during backward(), False otherwise.
shape	Returns the size of the self tensor.
volatile	

property is_leaf: bool

All Tensors that have requires_grad which is False will be leaf Tensors by convention.

For Tensors that have requires_grad which is True, they will be leaf Tensors if they were created by the user. This means that they are not the result of an operation and so grad_fn is None.

Only leaf Tensors will have their grad populated during a call to backward(). To get grad populated for non-leaf Tensors, you can use retain_grad().

Example:

```
>>> a = torch.rand(10, requires_grad=True)
>>> a.is_leaf
True
>>> b = torch.rand(10, requires_grad=True).cuda()
>>> b.is_leaf
False
# b was created by the operation that cast a cpu Tensor into a cuda Tensor
>>> c = torch.rand(10, requires_grad=True) + 2
>>> c.is_leaf
```

(continues on next page)

(continued from previous page)

```
False
# c was created by the addition operation
>>> d = torch.rand(10).cuda()
>>> d.is_leaf
True
# d does not require gradients and so has no operation creating it (that is␣
↪tracked by the autograd engine)
>>> e = torch.rand(10).cuda().requires_grad_()
>>> e.is_leaf
True
# e requires gradients and has no operations creating it
>>> f = torch.rand(10, requires_grad=True, device="cuda")
>>> f.is_leaf
True
# f requires grad, has no operation creating it
```

materialize(*shape*, *device=None*, *dtype=None*)

Create a Parameter with the same properties of the uninitialized one. Given a shape, it materializes a parameter in the same device and with the same *dtype* as the current one or the specified ones in the arguments.

Parameters

- **shape** (*Tuple[int, ...]*) – (tuple): the shape for the materialized tensor.
- **device** (`torch.device`) – the desired device of the parameters and buffers in this module. Optional.
- **dtype** (`torch.dtype`) – the desired floating point type of the floating point parameters and buffers in this module. Optional.

Return type
None

share_memory_()

Moves the underlying storage to shared memory.

This is a no-op if the underlying storage is already in shared memory and for CUDA tensors. Tensors in shared memory cannot be resized.

Return type
UninitializedParameter

pytorch_pfn_extras.nn.modules.lazy_batchnorm

Classes

pytorch_pfn_extras.nn.modules.lazy_batchnorm.LazyBatchNorm1d(...)	BatchNorm1d module with lazy weight initialization.
pytorch_pfn_extras.nn.modules.lazy_batchnorm.LazyBatchNorm2d(...)	BatchNorm2d module with lazy weight initialization.
pytorch_pfn_extras.nn.modules.lazy_batchnorm.LazyBatchNorm3d(...)	BatchNorm3d module with lazy weight initialization.
pytorch_pfn_extras.nn.modules.lazy_batchnorm.LazyInitializationMixin(...)	A mixin for modules that lazily initialize buffers and parameters.
pytorch_pfn_extras.nn.modules.lazy_batchnorm.UninitializedParameter([...])	

pytorch_pfn_extras.nn.modules.lazy_batchnorm.LazyBatchNorm1d

class pytorch_pfn_extras.nn.modules.lazy_batchnorm.**LazyBatchNorm1d**(*num_features*, **args*, ***kwargs*)

Bases: `_LazyBatchNorm`, `BatchNorm1d`

BatchNorm1d module with lazy weight initialization.

When `num_features` is `None`, it is determined at the first time of the forward step.

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

Methods

`__init__`(num_features, *args, **kwargs)	Initializes internal Module state, shared by both nn.Module and ScriptModule.
`add_module`(name, module)	Adds a child module to the current module.
`apply`(fn)	Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self.
`bfloat16`()	Casts all floating point parameters and buffers to `bfloat16` datatype.
`buffers`([recurse])	Returns an iterator over module buffers.
`children`()	Returns an iterator over immediate children modules.
`compile`(*args, **kwargs)	Compile this Module's forward using `torch.compile()`.
`cpu`()	Moves all model parameters and buffers to the CPU.
`cuda`([device])	Moves all model parameters and buffers to the GPU.
`double`()	Casts all floating point parameters and buffers to `double` datatype.
`eval`()	Sets the module in evaluation mode.
`extra_repr`()	Set the extra representation of the module
`float`()	Casts all floating point parameters and buffers to `float` datatype.
`forward`(input)	Defines the computation performed at every call.

continues on next page

Table 14 – continued from previous page

get_buffer(target)	Returns the buffer given by target if it exists, otherwise throws an error.
get_extra_state()	Returns any extra state to include in the module's state_dict.
get_parameter(target)	Returns the parameter given by target if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by target if it exists, otherwise throws an error.
half()	Casts all floating point parameters and buffers to half datatype.
ipu([device])	Moves all model parameters and buffers to the IPU.
load_state_dict(state_dict[, strict, assign])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse, ...])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse, ...])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook, *[, prepend, ...])	Registers a forward hook on the module.
register_forward_pre_hook(hook, *[, ...])	Registers a forward pre-hook on the module.
register_full_backward_hook(hook[, prepend])	Registers a backward hook on the module.
register_full_backward_pre_hook(hook[, prepend])	Registers a backward pre-hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
register_state_dict_pre_hook(hook)	These hooks will be called with arguments: self, prefix, and keep_vars before calling state_dict on self.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
reset_parameters()	
reset_running_stats()	
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()

continues on next page

Table 14 – continued from previous page

state_dict(*args, **kwargs)	Returns a dictionary containing a whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device[, recurse])	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Resets gradients of all model parameters.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
call_super_init	
dump_patches	
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

Parameters

- **num_features** (*Optional[int]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

affine: bool

eps: float

momentum: float

num_batches_tracked: Optional[Tensor]

num_features: Optional[int]

running_mean: Any

running_var: Any

track_running_stats: bool

training: bool

pytorch_pfn_extras.nn.modules.lazy_batchnorm.LazyBatchNorm2d

class pytorch_pfn_extras.nn.modules.lazy_batchnorm.**LazyBatchNorm2d**(*num_features*, **args*, ***kwargs*)

Bases: _LazyBatchNorm, BatchNorm2d

BatchNorm2d module with lazy weight initialization.

When num_features is None, it is determined at the first time of the forward step.

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

Methods

__init__(num_features, *args, **kwargs)	Initializes internal Module state, shared by both nn.Module and ScriptModule.
add_module(name, module)	Adds a child module to the current module.
apply(fn)	Applies fn recursively to every submodule (as returned by .children()) as well as self.
bfloat16()	Casts all floating point parameters and buffers to bfloat16 datatype.
buffers([recurse])	Returns an iterator over module buffers.
children()	Returns an iterator over immediate children modules.
compile(*args, **kwargs)	Compile this Module's forward using torch.compile().
cpu()	Moves all model parameters and buffers to the CPU.
cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to double datatype.
eval()	Sets the module in evaluation mode.
extra_repr()	Set the extra representation of the module
float()	Casts all floating point parameters and buffers to float datatype.
forward(input)	Defines the computation performed at every call.
get_buffer(target)	Returns the buffer given by target if it exists, otherwise throws an error.
get_extra_state()	Returns any extra state to include in the module's state_dict.
get_parameter(target)	Returns the parameter given by target if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by target if it exists, otherwise throws an error.
half()	Casts all floating point parameters and buffers to half datatype.
ipu([device])	Moves all model parameters and buffers to the IPU.
load_state_dict(state_dict[, strict, assign])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse, ...])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

continues on next page

Table 15 – continued from previous page

named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse, ...])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook, *[, prepend, ...])	Registers a forward hook on the module.
register_forward_pre_hook(hook, *[, ...])	Registers a forward pre-hook on the module.
register_full_backward_hook(hook[, prepend])	Registers a backward hook on the module.
register_full_backward_pre_hook(hook[, prepend])	Registers a backward pre-hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
register_state_dict_pre_hook(hook)	These hooks will be called with arguments: self, prefix, and keep_vars before calling state_dict on self.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
reset_parameters()	
reset_running_stats()	
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict(*args, **kwargs)	Returns a dictionary containing a whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device[, recurse])	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Resets gradients of all model parameters.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
call_super_init	
dump_patches	
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

Parameters

- **num_features** (*Optional[int]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

affine: bool

eps: float

momentum: float

num_batches_tracked: Optional[Tensor]

num_features: Optional[int]

running_mean: Any

running_var: Any

track_running_stats: bool

training: bool

pytorch_pfn_extras.nn.modules.lazy_batchnorm.LazyBatchNorm3d

class pytorch_pfn_extras.nn.modules.lazy_batchnorm.**LazyBatchNorm3d**(*num_features*, **args*, ***kwargs*)

Bases: _LazyBatchNorm, BatchNorm3d

BatchNorm3d module with lazy weight initialization.

When num_features is None, it is determined at the first time of the forward step.

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

Methods

`__init__`(num_features, *args, **kwargs)	Initializes internal Module state, shared by both nn.Module and ScriptModule.
`add_module`(name, module)	Adds a child module to the current module.
`apply`(fn)	Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self.
`bfloat16`()	Casts all floating point parameters and buffers to `bfloat16` datatype.
`buffers`([recurse])	Returns an iterator over module buffers.
`children`()	Returns an iterator over immediate children modules.
`compile`(*args, **kwargs)	Compile this Module's forward using `torch.compile()`.
`cpu`()	Moves all model parameters and buffers to the CPU.
`cuda`([device])	Moves all model parameters and buffers to the GPU.
`double`()	Casts all floating point parameters and buffers to `double` datatype.
`eval`()	Sets the module in evaluation mode.
`extra_repr`()	Set the extra representation of the module
`float`()	Casts all floating point parameters and buffers to `float` datatype.
`forward`(input)	Defines the computation performed at every call.
`get_buffer`(target)	Returns the buffer given by `target` if it exists, otherwise throws an error.
`get_extra_state`()	Returns any extra state to include in the module's state_dict.
`get_parameter`(target)	Returns the parameter given by `target` if it exists, otherwise throws an error.
`get_submodule`(target)	Returns the submodule given by `target` if it exists, otherwise throws an error.
`half`()	Casts all floating point parameters and buffers to `half` datatype.
`ipu`([device])	Moves all model parameters and buffers to the IPU.
`load_state_dict`(state_dict[, strict, assign])	Copies parameters and buffers from `state_dict` into this module and its descendants.
`modules`()	Returns an iterator over all modules in the network.
`named_buffers`([prefix, recurse, ...])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
`named_children`()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
`named_modules`([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
`named_parameters`([prefix, recurse, ...])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
`parameters`([recurse])	Returns an iterator over module parameters.
`register_backward_hook`(hook)	Registers a backward hook on the module.
`register_buffer`(name, tensor[, persistent])	Adds a buffer to the module.
`register_forward_hook`(hook, *[, prepend, ...])	Registers a forward hook on the module.

continues on next page

Table 16 – continued from previous page

register_forward_pre_hook(hook, *[, ...])	Registers a forward pre-hook on the module.
register_full_backward_hook(hook[, prepend])	Registers a backward hook on the module.
register_full_backward_pre_hook(hook[, prepend])	Registers a backward pre-hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
register_state_dict_pre_hook(hook)	These hooks will be called with arguments: self, prefix, and keep_vars before calling state_dict on self.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
reset_parameters()	
reset_running_stats()	
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict(*args, **kwargs)	Returns a dictionary containing a whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device[, recurse])	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Resets gradients of all model parameters.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
call_super_init	
dump_patches	
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

Parameters

- **num_features** (*Optional[int]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

affine: bool

eps: float

momentum: float

num_batches_tracked: Optional[Tensor]

num_features: Optional[int]

running_mean: Any

running_var: Any

track_running_stats: bool

training: bool

pytorch_pfn_extras.nn.modules.lazy_batchnorm.LazyInitializationMixin

class pytorch_pfn_extras.nn.modules.lazy_batchnorm.**LazyInitializationMixin**(**args*, ***kwargs*)

Bases: object

A mixin for modules that lazily initialize buffers and parameters.

Unlike regular modules, subclasses of this module can initialize buffers and parameters outside of the constructor (`__init__`). This allows you to, for example, initialize parameters in `forward` method to determine the shape of the weight based on the initial input.

Be sure to run "dummy" forward once to initialize all parameters that should be trained, before passing `module.parameters()` to an optimizer; otherwise weights initialized after `module.parameters()` (e.g., in `forward` function) will never be trained.

Note that lazy modules cannot validate if the shape is correct during deserialization. Also note that the initial weights may become different from the original (non-lazy) module even if the random seed is manually configured, as the order of initialization is different from the original one; especially, `module.cuda()` may cause the initialization to run on a GPU.

The default value of lazy buffers and parameters are `torch.Tensor([])` and `UninitializedParameter()`, respectively.

Methods

__init__(*args, **kwargs)	
state_dict(*args, **kwargs)	Returns a dictionary containing a whole state of the module.

Attributes

lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

__init__(**args*, ***kwargs*)

Parameters

- **self** (*Any*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

lazy_buffer_names: Tuple[str, ...] = ()

lazy_parameter_names: Tuple[str, ...] = ()

property lazy_parmeters_determined: bool

Returns if all lazy parameters are determined.

Subclasses can perform parameters initialization after all lazy parameters are determined. Note that this may be called during `__init__`.

state_dict(**args*, ***kwargs*)

Returns a dictionary containing a whole state of the module.

This function overrides the default behavior to exclude uninitialized parameter from serialization. This is needed because we need to discriminate lazy parameters (`UninitializedParameter()`) and initialized empty parameters (``torch.nn.Parameter(torch.Tensor())`) during deserialization.

See comments of `_lazy_load_hook` for details.

Parameters

- **self** (*Any*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Dict[str, *Any*]

pytorch_pfn_extras.nn.modules.lazy_batchnorm.UninitializedParameter

class pytorch_pfn_extras.nn.modules.lazy_batchnorm.**UninitializedParameter**(*data=None*, *requires_grad=True*)

Bases: Parameter

Methods

__init__()	
abs()	See torch.abs()
abs_()	In-place version of abs()
absolute()	Alias for abs()
absolute_()	In-place version of absolute() Alias for abs_()
acos()	See torch.acos()
acos_()	In-place version of acos()
acosh()	See torch.acosh()
acosh_()	In-place version of acosh()
add(other, *[, alpha])	Add a scalar or tensor to self tensor.
add_(other, *[, alpha])	In-place version of add()
addbmm(batch1, batch2, *[, beta, alpha])	See torch.addbmm()
addbmm_(batch1, batch2, *[, beta, alpha])	In-place version of addbmm()
addcdiv(tensor1, tensor2, *[, value])	See torch.addcdiv()
addcdiv_(tensor1, tensor2, *[, value])	In-place version of addcdiv()
addcmul(tensor1, tensor2, *[, value])	See torch.addcmul()
addcmul_(tensor1, tensor2, *[, value])	In-place version of addcmul()
addmm(mat1, mat2, *[, beta, alpha])	See torch.addmm()
addmm_(mat1, mat2, *[, beta, alpha])	In-place version of addmm()
addmv(mat, vec, *[, beta, alpha])	See torch.addmv()
addmv_(mat, vec, *[, beta, alpha])	In-place version of addmv()
addr(vec1, vec2, *[, beta, alpha])	See torch.addr()
addr_(vec1, vec2, *[, beta, alpha])	In-place version of addr()
adjoint()	Alias for adjoint()
align_as(other)	Permutes the dimensions of the self tensor to match the dimension order in the other tensor, adding size-one dims for any new names.
align_to(*names)	Permutes the dimensions of the self tensor to match the order specified in names, adding size-one dims for any new names.
all([dim, keepdim])	See torch.all()
allclose(other[, rtol, atol, equal_nan])	See torch.allclose()
amax([dim, keepdim])	See torch.amax()
amin([dim, keepdim])	See torch.amin()
aminmax(*[, dim, keepdim])	See torch.aminmax()
angle()	See torch.angle()
any([dim, keepdim])	See torch.any()
apply_(callable)	Applies the function callable to each element in the tensor, replacing each element with the value returned by callable.
arccos()	See torch.arccos()

continues on next page

Table 17 – continued from previous page

arccos_()	In-place version of arccos()
arccosh	acosh() -> Tensor
arccosh_	acosh_() -> Tensor
arcsin()	See torch.arcsin()
arcsin_()	In-place version of arcsin()
arcsinh()	See torch.arcsinh()
arcsinh_()	In-place version of arcsinh()
arctan()	See torch.arctan()
arctan2(other)	See torch.arctan2()
arctan2_	atan2_(other) -> Tensor
arctan_()	In-place version of arctan()
arctanh()	See torch.arctanh()
arctanh_(other)	In-place version of arctanh()
argmax([dim, keepdim])	See torch.argmax()
argmin([dim, keepdim])	See torch.argmin()
argsort([dim, descending])	See torch.argsort()
argwhere()	See torch.argwhere()
as_strided(size, stride[, storage_offset])	See torch.as_strided()
as_strided_(size, stride[, storage_offset])	In-place version of as_strided()
as_strided_scatter(src, size, stride[, ...])	See torch.as_strided_scatter()
as_subclass(cls)	Makes a cls instance with the same data pointer as self.
asin()	See torch.asin()
asin_()	In-place version of asin()
asinh()	See torch.asinh()
asinh_()	In-place version of asinh()
atan()	See torch.atan()
atan2(other)	See torch.atan2()
atan2_(other)	In-place version of atan2()
atan_()	In-place version of atan()
atanh()	See torch.atanh()
atanh_(other)	In-place version of atanh()
backward([gradient, retain_graph, ...])	Computes the gradient of current tensor wrt graph leaves.
baddbmm(batch1, batch2, *[, beta, alpha])	See torch.baddbmm()
baddbmm_(batch1, batch2, *[, beta, alpha])	In-place version of baddbmm()
bernoulli(*[, generator])	Returns a result tensor where each result[i] is independently sampled from Bernoulli(self[i]).
bernoulli_([p, generator])	Fills each location of self with an independent sample from Bernoulli(p).
bfloat16([memory_format])	self.bfloat16() is equivalent to self.to(torch.bfloat16).
bincount([weights, minlength])	See torch.bincount()
bitwise_and()	See torch.bitwise_and()
bitwise_and_()	In-place version of bitwise_and()
bitwise_left_shift(other)	See torch.bitwise_left_shift()
bitwise_left_shift_(other)	In-place version of bitwise_left_shift()
bitwise_not()	See torch.bitwise_not()
bitwise_not_()	In-place version of bitwise_not()
bitwise_or()	See torch.bitwise_or()

continues on next page

Table 17 – continued from previous page

bitwise_or_()	In-place version of bitwise_or()
bitwise_right_shift(other)	See torch.bitwise_right_shift()
bitwise_right_shift_(other)	In-place version of bitwise_right_shift()
bitwise_xor()	See torch.bitwise_xor()
bitwise_xor_()	In-place version of bitwise_xor()
bmm(batch2)	See torch.bmm()
bool([memory_format])	self.bool() is equivalent to self.to(torch.bool).
broadcast_to(shape)	See torch.broadcast_to().
byte([memory_format])	self.byte() is equivalent to self.to(torch.uint8).
cauchy_([median, sigma, generator])	Fills the tensor with numbers drawn from the Cauchy distribution:
ccol_indices	
cdouble([memory_format])	self.cdouble() is equivalent to self.to(torch.complex128).
ceil()	See torch.ceil()
ceil_()	In-place version of ceil()
cfloat([memory_format])	self.cfloat() is equivalent to self.to(torch.complex64).
chalf([memory_format])	self.chalf() is equivalent to self.to(torch.complex32).
char([memory_format])	self.char() is equivalent to self.to(torch.int8).
cholesky([upper])	See torch.cholesky()
cholesky_inverse([upper])	See torch.cholesky_inverse()
cholesky_solve(input2[, upper])	See torch.cholesky_solve()
chunk(chunks[, dim])	See torch.chunk()
clamp([min, max])	See torch.clamp()
clamp_([min, max])	In-place version of clamp()
clamp_max	
clamp_max_	
clamp_min	
clamp_min_	
clip([min, max])	Alias for clamp().
clip_([min, max])	Alias for clamp_().
clone(*[, memory_format])	See torch.clone()
coalesce()	Returns a coalesced copy of self if self is an uncoalesced tensor.
col_indices()	Returns the tensor containing the column indices of the self tensor when self is a sparse CSR tensor of layout sparse_csr.
conj()	See torch.conj()
conj_physical()	See torch.conj_physical()
conj_physical_()	In-place version of conj_physical()

continues on next page

Table 17 – continued from previous page

contiguous([memory_format])	Returns a contiguous in memory tensor containing the same data as self tensor.
copy_(src[, non_blocking])	Copies the elements from src into self tensor and returns self.
copysign(other)	See torch.copysign()
copysign_(other)	In-place version of copysign()
corrcoef()	See torch.corrcoef()
cos()	See torch.cos()
cos_()	In-place version of cos()
cosh()	See torch.cosh()
cosh_()	In-place version of cosh()
count_nonzero([dim])	See torch.count_nonzero()
cov(*[, correction, fweights, aweights])	See torch.cov()
cpu([memory_format])	Returns a copy of this object in CPU memory.
cross(other[, dim])	See torch.cross()
crow_indices()	Returns the tensor containing the compressed row indices of the self tensor when self is a sparse CSR tensor of layout sparse_csr.
cuda([device, non_blocking, memory_format])	Returns a copy of this object in CUDA memory.
cummax(dim)	See torch.cummax()
cummin(dim)	See torch.cummin()
cumprod(dim[, dtype])	See torch.cumprod()
cumprod_(dim[, dtype])	In-place version of cumprod()
cumsum(dim[, dtype])	See torch.cumsum()
cumsum_(dim[, dtype])	In-place version of cumsum()
data_ptr()	Returns the address of the first element of self tensor.
deg2rad()	See torch.deg2rad()
deg2rad_()	In-place version of deg2rad()
dense_dim()	Return the number of dense dimensions in a sparse tensor self.
dequantize()	Given a quantized Tensor, dequantize it and return the dequantized float Tensor.
det()	See torch.det()
detach	Returns a new Tensor, detached from the current graph.
detach_	Detaches the Tensor from the graph that created it, making it a leaf.
diag([diagonal])	See torch.diag()
diag_embed([offset, dim1, dim2])	See torch.diag_embed()
diagflat([offset])	See torch.diagflat()
diagonal([offset, dim1, dim2])	See torch.diagonal()
diagonal_scatter(src[, offset, dim1, dim2])	See torch.diagonal_scatter()
diff([n, dim, prepend, append])	See torch.diff()
digamma()	See torch.digamma()
digamma_()	In-place version of digamma()
dim()	Returns the number of dimensions of self tensor.
dim_order()	Returns a tuple of int describing the dim order or physical layout of self.
dist(other[, p])	See torch.dist()
div(value, *[, rounding_mode])	See torch.div()

continues on next page

Table 17 – continued from previous page

div_(value, *[, rounding_mode])	In-place version of div()
divide(value, *[, rounding_mode])	See torch.divide()
divide_(value, *[, rounding_mode])	In-place version of divide()
dot(other)	See torch.dot()
double([memory_format])	self.double() is equivalent to self.to(torch.float64).
dsplit(split_size_or_sections)	See torch.dsplit()
eig([eigenvectors])	
element_size()	Returns the size in bytes of an individual element.
eq(other)	See torch.eq()
eq_(other)	In-place version of eq()
equal(other)	See torch.equal()
erf()	See torch.erf()
erf_()	In-place version of erf()
erfc()	See torch.erfc()
erfc_()	In-place version of erfc()
erfinv()	See torch.erfinv()
erfinv_()	In-place version of erfinv()
exp()	See torch.exp()
exp2()	See torch.exp2()
exp2_()	In-place version of exp2()
exp_()	In-place version of exp()
expand(*sizes)	Returns a new view of the self tensor with singleton dimensions expanded to a larger size.
expand_as(other)	Expand this tensor to the same size as other.
expm1()	See torch.expm1()
expm1_()	In-place version of expm1()
exponential_([lambd, generator])	Fills self tensor with elements drawn from the exponential distribution:
fill_(value)	Fills self tensor with the specified value.
fill_diagonal_(fill_value[, wrap])	Fill the main diagonal of a tensor that has at least 2-dimensions.
fix()	See torch.fix().
fix_()	In-place version of fix()
flatten([start_dim, end_dim])	See torch.flatten()
flip(dims)	See torch.flip()
fliplr()	See torch.fliplr()
flipud()	See torch.flipud()
float([memory_format])	self.float() is equivalent to self.to(torch.float32).
float_power(exponent)	See torch.float_power()
float_power_(exponent)	In-place version of float_power()
floor()	See torch.floor()
floor_()	In-place version of floor()
floor_divide(value)	See torch.floor_divide()
floor_divide_(value)	In-place version of floor_divide()
fmax(other)	See torch.fmax()
fmin(other)	See torch.fmin()
fmod(divisor)	See torch.fmod()

continues on next page

Table 17 – continued from previous page

fmod_(divisor)	In-place version of fmod()
frac()	See torch.frac()
frac_()	In-place version of frac()
frexp(input)	See torch.frexp()
gather(dim, index)	See torch.gather()
gcd(other)	See torch.gcd()
gcd_(other)	In-place version of gcd()
ge(other)	See torch.ge().
ge_(other)	In-place version of ge().
geometric_(p, *[, generator])	Fills self tensor with elements drawn from the geometric distribution:
geqrf()	See torch.geqrf()
ger(vec2)	See torch.ger()
get_device()	For CUDA tensors, this function returns the device ordinal of the GPU on which the tensor resides.
greater(other)	See torch.greater().
greater_(other)	In-place version of greater().
greater_equal(other)	See torch.greater_equal().
greater_equal_(other)	In-place version of greater_equal().
gt(other)	See torch.gt().
gt_(other)	In-place version of gt().
half([memory_format])	self.half() is equivalent to self.to(torch.float16).
hardshrink([lambd])	See torch.nn.functional.hardshrink()
has_names	Is True if any of this tensor's dimensions are named.
heaviside(values)	See torch.heaviside()
heaviside_(values)	In-place version of heaviside()
histc([bins, min, max])	See torch.histc()
histogram(input, bins, *[, range, weight, ...])	See torch.histogram()
hsplit(split_size_or_sections)	See torch.hsplit()
hypot(other)	See torch.hypot()
hypot_(other)	In-place version of hypot()
i0()	See torch.i0()
i0_()	In-place version of i0()
igamma(other)	See torch.igamma()
igamma_(other)	In-place version of igamma()
igammac(other)	See torch.igammac()
igammac_(other)	In-place version of igammac()
index_add(dim, index, source, *[, alpha])	Out-of-place version of torch.Tensor.index_add_().
index_add_(dim, index, source, *[, alpha])	Accumulate the elements of alpha times source into the self tensor by adding to the indices in the order given in index.
index_copy(dim, index, tensor2)	Out-of-place version of torch.Tensor.index_copy_().
index_copy_(dim, index, tensor)	Copies the elements of tensor into the self tensor by selecting the indices in the order given in index.
index_fill(dim, index, value)	Out-of-place version of torch.Tensor.index_fill_().

continues on next page

Table 17 – continued from previous page

index_fill_(dim, index, value)	Fills the elements of the self tensor with value value by selecting the indices in the order given in index.
index_put(indices, values[, accumulate])	Out-place version of index_put_().
index_put_(indices, values[, accumulate])	Puts values from the tensor values into the tensor self using the indices specified in indices (which is a tuple of Tensors).
index_reduce	
index_reduce_(dim, index, source, reduce, *)	Accumulate the elements of source into the self tensor by accumulating to the indices in the order given in index using the reduction given by the reduce argument.
index_select(dim, index)	See torch.index_select()
indices()	Return the indices tensor of a sparse COO tensor.
inner(other)	See torch.inner().
int([memory_format])	self.int() is equivalent to self.to(torch.int32).
int_repr()	Given a quantized Tensor, self.int_repr() returns a CPU Tensor with uint8_t as data type that stores the underlying uint8_t values of the given Tensor.
inverse()	See torch.inverse()
ipu([device, non_blocking, memory_format])	Returns a copy of this object in IPU memory.
is_coalesced()	Returns True if self is a sparse COO tensor that is coalesced, False otherwise.
is_complex()	Returns True if the data type of self is a complex data type.
is_conj()	Returns True if the conjugate bit of self is set to true.
is_contiguous([memory_format])	Returns True if self tensor is contiguous in memory in the order specified by memory format.
is_distributed	
is_floating_point()	Returns True if the data type of self is a floating point data type.
is_inference()	See torch.is_inference()
is_neg()	Returns True if the negative bit of self is set to true.
is_nonzero	
is_pinned	Returns true if this tensor resides in pinned memory.
is_same_size	
is_set_to(tensor)	Returns True if both tensors are pointing to the exact same memory (same storage, offset, size and stride).
is_shared()	Checks if tensor is in shared memory.
is_signed()	Returns True if the data type of self is a signed data type.
isclose(other[, rtol, atol, equal_nan])	See torch.isclose()
isfinite()	See torch.isfinite()
isinf()	See torch.isinf()
isnan()	See torch.isnan()

continues on next page

Table 17 – continued from previous page

isneginf()	See torch.isneginf()
isposinf()	See torch.isposinf()
isreal()	See torch.isreal()
istft(n_fft[, hop_length, win_length, ...])	See torch.istft()
item()	Returns the value of this tensor as a standard Python number.
kron(other)	See torch.kron()
kthvalue(k[, dim, keepdim])	See torch.kthvalue()
lcm(other)	See torch.lcm()
lcm_(other)	In-place version of lcm()
ldexp(other)	See torch.ldexp()
ldexp_(other)	In-place version of ldexp()
le(other)	See torch.le().
le_(other)	In-place version of le().
lerp(end, weight)	See torch.lerp()
lerp_(end, weight)	In-place version of lerp()
less	lt(other) -> Tensor
less_(other)	In-place version of less().
less_equal(other)	See torch.less_equal().
less_equal_(other)	In-place version of less_equal().
lgamma()	See torch.lgamma()
lgamma_()	In-place version of lgamma()
log()	See torch.log()
log10()	See torch.log10()
log10_()	In-place version of log10()
log1p()	See torch.log1p()
log1p_()	In-place version of log1p()
log2()	See torch.log2()
log2_()	In-place version of log2()
log_()	In-place version of log()
log_normal_([mean, std, generator])	Fills self tensor with numbers samples from the log-normal distribution parameterized by the given mean μ and standard deviation σ.
log_softmax	
logaddexp(other)	See torch.logaddexp()
logaddexp2(other)	See torch.logaddexp2()
logcumsumexp(dim)	See torch.logcumsumexp()
logdet()	See torch.logdet()
logical_and()	See torch.logical_and()
logical_and_()	In-place version of logical_and()
logical_not()	See torch.logical_not()
logical_not_()	In-place version of logical_not()
logical_or()	See torch.logical_or()
logical_or_()	In-place version of logical_or()
logical_xor()	See torch.logical_xor()
logical_xor_()	In-place version of logical_xor()
logit()	See torch.logit()
logit_()	In-place version of logit()
logsumexp(dim[, keepdim])	See torch.logsumexp()

continues on next page

Table 17 – continued from previous page

long([memory_format])	self.long() is equivalent to self.to(torch.int64).
lstsq(other)	
lt(other)	See torch.lt().
lt_(other)	In-place version of lt().
lu([pivot, get_infos])	See torch.lu()
lu_solve(LU_data, LU_pivots)	See torch.lu_solve()
map2_	
map_(tensor, callable)	Applies callable for each element in self tensor and the given tensor and stores the results in self tensor.
masked_fill(mask, value)	Out-of-place version of torch.Tensor.masked_fill_()
masked_fill_(mask, value)	Fills elements of self tensor with value where mask is True.
masked_scatter(mask, tensor)	Out-of-place version of torch.Tensor.masked_scatter_()
masked_scatter_(mask, source)	Copies elements from source into self tensor at positions where the mask is True.
masked_select(mask)	See torch.masked_select()
materialize(shape[, device, dtype])	Create a Parameter with the same properties of the uninitialized one.
matmul(tensor2)	See torch.matmul()
matrix_exp()	See torch.matrix_exp()
matrix_power(n)	**Note:** matrix_power() is deprecated, use torch.linalg.matrix_power() instead.
max([dim, keepdim])	See torch.max()
maximum(other)	See torch.maximum()
mean([dim, keepdim, dtype])	See torch.mean()
median([dim, keepdim])	See torch.median()
min([dim, keepdim])	See torch.min()
minimum(other)	See torch.minimum()
mm(mat2)	See torch.mm()
mode([dim, keepdim])	See torch.mode()
moveaxis(source, destination)	See torch.moveaxis()
movedim(source, destination)	See torch.movedim()
msort()	See torch.msort()
mul(value)	See torch.mul().
mul_(value)	In-place version of mul().
multinomial(num_samples[, replacement, ...])	See torch.multinomial()
multiply(value)	See torch.multiply().
multiply_(value)	In-place version of multiply().
mv(vec)	See torch.mv()
mvlgamma(p)	See torch.mvlgamma()
mvlgamma_(p)	In-place version of mvlgamma()

continues on next page

Table 17 – continued from previous page

nan_to_num([nan, posinf, neginf])	See torch.nan_to_num().
nan_to_num_([nan, posinf, neginf])	In-place version of nan_to_num().
nanmean([dim, keepdim, dtype])	See torch.nanmean()
nanmedian([dim, keepdim])	See torch.nanmedian()
nanquantile(q[, dim, keepdim, interpolation])	See torch.nanquantile()
nansum([dim, keepdim, dtype])	See torch.nansum()
narrow(dimension, start, length)	See torch.narrow().
narrow_copy(dimension, start, length)	See torch.narrow_copy().
ndimension()	Alias for dim()
ne(other)	See torch.ne().
ne_(other)	In-place version of ne().
neg()	See torch.neg()
neg_()	In-place version of neg()
negative()	See torch.negative()
negative_()	In-place version of negative()
nelement()	Alias for numel()
new	
new_empty(size, *[, dtype, device, ...])	Returns a Tensor of size size filled with uninitialized data.
new_empty_strided(size, stride[, dtype, ...])	Returns a Tensor of size size and strides stride filled with uninitialized data.
new_full(size, fill_value, *[, dtype, ...])	Returns a Tensor of size size filled with fill_value.
new_ones(size, *[, dtype, device, ...])	Returns a Tensor of size size filled with 1.
new_tensor(data, *[, dtype, device, ...])	Returns a new Tensor with data as the tensor data.
new_zeros(size, *[, dtype, device, ...])	Returns a Tensor of size size filled with 0.
nextafter(other)	See torch.nextafter()
nextafter_(other)	In-place version of nextafter()
nonzero()	See torch.nonzero()
nonzero_static(input, *, size[, fill_value])	Returns a 2-D tensor where each row is the index for a non-zero value.
norm([p, dim, keepdim, dtype])	See torch.norm()
normal_([mean, std, generator])	Fills self tensor with elements samples from the normal distribution parameterized by mean and std.
not_equal(other)	See torch.not_equal().
not_equal_(other)	In-place version of not_equal().
numel()	See torch.numel()
numpy(*[, force])	Returns the tensor as a NumPy ndarray.
orgqr(input2)	See torch.orgqr()
ormqr(input2, input3[, left, transpose])	See torch.ormqr()
outer(vec2)	See torch.outer().
permute(*dims)	See torch.permute()
pin_memory()	Copies the tensor to pinned memory, if it's not already pinned.
pinverse()	See torch.pinverse()
polygamma(n)	See torch.polygamma()
polygamma_(n)	In-place version of polygamma()
positive()	See torch.positive()
pow(exponent)	See torch.pow()

continues on next page

Table 17 – continued from previous page

pow_(exponent)	In-place version of pow()
prelu	
prod([dim, keepdim, dtype])	See torch.prod()
put(input, index, source[, accumulate])	Out-of-place version of torch.Tensor.put_().
put_(index, source[, accumulate])	Copies the elements from source into the positions specified by index.
q_per_channel_axis()	Given a Tensor quantized by linear (affine) per-channel quantization, returns the index of dimension on which per-channel quantization is applied.
q_per_channel_scales()	Given a Tensor quantized by linear (affine) per-channel quantization, returns a Tensor of scales of the underlying quantizer.
q_per_channel_zero_points()	Given a Tensor quantized by linear (affine) per-channel quantization, returns a tensor of zero_points of the underlying quantizer.
q_scale()	Given a Tensor quantized by linear(affine) quantization, returns the scale of the underlying quantizer().
q_zero_point()	Given a Tensor quantized by linear(affine) quantization, returns the zero_point of the underlying quantizer().
qr([some])	See torch.qr()
qscheme()	Returns the quantization scheme of a given QTensor.
quantile(q[, dim, keepdim, interpolation])	See torch.quantile()
rad2deg()	See torch.rad2deg()
rad2deg_()	In-place version of rad2deg()
random_([from, to, generator])	Fills self tensor with numbers sampled from the discrete uniform distribution over [from, to - 1].
ravel()	see torch.ravel()
reciprocal()	See torch.reciprocal()
reciprocal_()	In-place version of reciprocal()
record_stream(stream)	Ensures that the tensor memory is not reused for another tensor until all current work queued on stream are complete.
refine_names(*names)	Refines the dimension names of self according to names.
register_hook(hook)	Registers a backward hook.
register_post_accumulate_grad_hook(hook)	Registers a backward hook that runs after grad accumulation.
reinforce(reward)	
relu	
relu_	
remainder(divisor)	See torch.remainder()
remainder_(divisor)	In-place version of remainder()
rename(*names, **rename_map)	Renames dimension names of self.
rename_(*names, **rename_map)	In-place version of rename().
renorm(p, dim, maxnorm)	See torch.renorm()
renorm_(p, dim, maxnorm)	In-place version of renorm()

continues on next page

Table 17 – continued from previous page

repeat(*sizes)	Repeats this tensor along the specified dimensions.
repeat_interleave(repeats[, dim, output_size])	See torch.repeat_interleave().
requires_grad_([requires_grad])	Change if autograd should record operations on this tensor: sets this tensor's requires_grad attribute in-place.
reshape(*shape)	Returns a tensor with the same data and number of elements as self but with the specified shape.
reshape_as(other)	Returns this tensor as the same shape as other.
resize(*sizes)	
resize_(*sizes[, memory_format])	Resizes self tensor to the specified size.
resize_as(tensor)	
resize_as_(tensor[, memory_format])	Resizes the self tensor to be the same size as the specified tensor.
resize_as_sparse_	
resolve_conj()	See torch.resolve_conj()
resolve_neg()	See torch.resolve_neg()
retain_grad()	Enables this Tensor to have their grad populated during backward().
roll(shifts, dims)	See torch.roll()
rot90(k, dims)	See torch.rot90()
round([decimals])	See torch.round()
round_([decimals])	In-place version of round()
row_indices	
rsqrt()	See torch.rsqrt()
rsqrt_()	In-place version of rsqrt()
scatter(dim, index, src)	Out-of-place version of torch.Tensor.scatter_()
scatter_(dim, index, src[, reduce])	Writes all values from the tensor src into self at the indices specified in the index tensor.
scatter_add(dim, index, src)	Out-of-place version of torch.Tensor.scatter_add_()
scatter_add_(dim, index, src)	Adds all values from the tensor src into self at the indices specified in the index tensor in a similar fashion as scatter_().
scatter_reduce(dim, index, src, reduce, *[, ...])	Out-of-place version of torch.Tensor.scatter_reduce_()
scatter_reduce_(dim, index, src, reduce, *)	Reduces all values from the src tensor to the indices specified in the index tensor in the self tensor using the applied reduction defined via the reduce argument ("sum", "prod", "mean", "amax", "amin").
select(dim, index)	See torch.select()
select_scatter(src, dim, index)	See torch.select_scatter()
set_([source, storage_offset, size, stride])	Sets the underlying storage, size, and strides.
sgn()	See torch.sgn()
sgn_()	In-place version of sgn()
*share_memory_*()	Moves the underlying storage to shared memory.

continues on next page

Table 17 – continued from previous page

short([memory_format])	self.short() is equivalent to self.to(torch.int16).
sigmoid()	See torch.sigmoid()
sigmoid_()	In-place version of sigmoid()
sign()	See torch.sign()
sign_()	In-place version of sign()
signbit()	See torch.signbit()
sin()	See torch.sin()
sin_()	In-place version of sin()
sinc()	See torch.sinc()
sinc_()	In-place version of sinc()
sinh()	See torch.sinh()
sinh_()	In-place version of sinh()
size([dim])	Returns the size of the self tensor.
slice_scatter(src[, dim, start, end, step])	See torch.slice_scatter()
slogdet()	See torch.slogdet()
smm(mat)	See torch.smm()
softmax(dim)	Alias for torch.nn.functional.softmax().
solve(other)	
sort([dim, descending])	See torch.sort()
sparse_dim()	Return the number of sparse dimensions in a sparse tensor self.
sparse_mask(mask)	Returns a new sparse tensor with values from a strided tensor self filtered by the indices of the sparse tensor mask.
sparse_resize_(size, sparse_dim, dense_dim)	Resizes self sparse tensor to the desired size and the number of sparse and dense dimensions.
sparse_resize_and_clear_(size, sparse_dim, ...)	Removes all specified elements from a sparse tensor self and resizes self to the desired size and the number of sparse and dense dimensions.
split(split_size[, dim])	See torch.split()
split_with_sizes	
sqrt()	See torch.sqrt()
sqrt_()	In-place version of sqrt()
square()	See torch.square()
square_()	In-place version of square()
squeeze([dim])	See torch.squeeze()
squeeze_([dim])	In-place version of squeeze()
sspaddmm(mat1, mat2, *[, beta, alpha])	See torch.sspaddmm()
std([dim, correction, keepdim])	See torch.std()
stft(n_fft[, hop_length, win_length, ...])	See torch.stft()
storage()	Returns the underlying TypedStorage.
storage_offset()	Returns self tensor's offset in the underlying storage in terms of number of storage elements (not bytes).
storage_type()	Returns the type of the underlying storage.
stride(dim)	Returns the stride of self tensor.
sub(other, *[, alpha])	See torch.sub().
sub_(other, *[, alpha])	In-place version of sub()
subtract(other, *[, alpha])	See torch.subtract().

continues on next page

Table 17 – continued from previous page

subtract_(other, *[, alpha])	In-place version of subtract().
sum([dim, keepdim, dtype])	See torch.sum()
sum_to_size(*size)	Sum this tensor to size.
svd([some, compute_uv])	See torch.svd()
swapaxes(axis0, axis1)	See torch.swapaxes()
swapaxes_(axis0, axis1)	In-place version of swapaxes()
swapdims(dim0, dim1)	See torch.swapdims()
swapdims_(dim0, dim1)	In-place version of swapdims()
symeig([eigenvectors])	
t()	See torch.t()
t_()	In-place version of t()
take(indices)	See torch.take()
take_along_dim(indices, dim)	See torch.take_along_dim()
tan()	See torch.tan()
tan_()	In-place version of tan()
tanh()	See torch.tanh()
tanh_()	In-place version of tanh()
tensor_split(indices_or_sections[, dim])	See torch.tensor_split()
tile(dims)	See torch.tile()
to(*args, **kwargs)	Performs Tensor dtype and/or device conversion.
to_dense([dtype, masked_grad])	Creates a strided copy of self if self is not a strided tensor, otherwise returns self.
to_mkldnn()	Returns a copy of the tensor in torch.mkldnn layout.
to_padded_tensor(padding[, output_size])	See to_padded_tensor()
to_sparse(sparseDims)	Returns a sparse copy of the tensor.
to_sparse_bsc(blocksize, dense_dim)	Convert a tensor to a block sparse column (BSC) storage format of given blocksize.
to_sparse_bsr(blocksize, dense_dim)	Convert a tensor to a block sparse row (BSR) storage format of given blocksize.
to_sparse_coo()	Convert a tensor to coordinate format.
to_sparse_csc()	Convert a tensor to compressed column storage (CSC) format.
to_sparse_csr([dense_dim])	Convert a tensor to compressed row storage format (CSR).
tolist()	Returns the tensor as a (nested) list.
topk(k[, dim, largest, sorted])	See torch.topk()
trace()	See torch.trace()
transpose(dim0, dim1)	See torch.transpose()
transpose_(dim0, dim1)	In-place version of transpose()
triangular_solve(A[, upper, transpose, ...])	See torch.triangular_solve()
tril([diagonal])	See torch.tril()
tril_([diagonal])	In-place version of tril()
triu([diagonal])	See torch.triu()
triu_([diagonal])	In-place version of triu()
true_divide(value)	See torch.true_divide()
true_divide_(value)	In-place version of true_divide_()
trunc()	See torch.trunc()
trunc_()	In-place version of trunc()

continues on next page

Table 17 – continued from previous page

type([dtype, non_blocking])	Returns the type if *dtype* is not provided, else casts this object to the specified type.
type_as(tensor)	Returns this tensor cast to the type of the given tensor.
unbind([dim])	See torch.unbind()
unflatten(dim, sizes)	See torch.unflatten().
unfold(dimension, size, step)	Returns a view of the original tensor which contains all slices of size size from self tensor in the dimension dimension.
uniform_([from, to, generator])	Fills self tensor with numbers sampled from the continuous uniform distribution:
unique([sorted, return_inverse, ...])	Returns the unique elements of the input tensor.
unique_consecutive([return_inverse, ...])	Eliminates all but the first element from every consecutive group of equivalent elements.
unsafe_chunk(chunks[, dim])	See torch.unsafe_chunk()
unsafe_split(split_size[, dim])	See torch.unsafe_split()
unsafe_split_with_sizes	
unsqueeze(dim)	See torch.unsqueeze()
unsqueeze_(dim)	In-place version of unsqueeze()
untyped_storage()	Returns the underlying UntypedStorage.
values()	Return the values tensor of a sparse COO tensor.
var([dim, correction, keepdim])	See torch.var()
vdot(other)	See torch.vdot()
view(*shape)	Returns a new tensor with the same data as the self tensor but of a different shape.
view_as(other)	View this tensor as the same size as other.
vsplit(split_size_or_sections)	See torch.vsplit()
where(condition, y)	self.where(condition, y) is equivalent to torch.where(condition, self, y).
xlogy(other)	See torch.xlogy()
xlogy_(other)	In-place version of xlogy()
xpu([device, non_blocking, memory_format])	Returns a copy of this object in XPU memory.
zero_()	Fills self tensor with zeros.

Attributes

H	Returns a view of a matrix (2-D tensor) conjugated and transposed.
T	Returns a view of this tensor with its dimensions reversed.
data	
device	Is the torch.device where this Tensor is.
dtype	
grad	This attribute is None by default and becomes a Tensor the first time a call to backward() computes gradients for self.

continues on next page

Table 18 – continued from previous page

grad_fn	
imag	Returns a new tensor containing imaginary values of the self tensor.
is_cpu	Is True if the Tensor is stored on the CPU, False otherwise.
is_cuda	Is True if the Tensor is stored on the GPU, False otherwise.
is_ipu	Is True if the Tensor is stored on the IPU, False otherwise.
is_leaf	All Tensors that have requires_grad which is False will be leaf Tensors by convention.
is_meta	Is True if the Tensor is a meta tensor, False otherwise.
is_mkldnn	
is_mps	Is True if the Tensor is stored on the MPS device, False otherwise.
is_nested	
is_ort	
is_quantized	Is True if the Tensor is quantized, False otherwise.
is_sparse	Is True if the Tensor uses sparse COO storage layout, False otherwise.
is_sparse_csr	Is True if the Tensor uses sparse CSR storage layout, False otherwise.
is_vulkan	
is_xla	Is True if the Tensor is stored on an XLA device, False otherwise.
is_xpu	Is True if the Tensor is stored on the XPU, False otherwise.
itemsize	Alias for element_size()
layout	
mH	Accessing this property is equivalent to calling adjoint().
mT	Returns a view of this tensor with the last two dimensions transposed.
name	
names	Stores names for each of this tensor's dimensions.
nbytes	Returns the number of bytes consumed by the "view" of elements of the Tensor if the Tensor does not use sparse storage layout.
ndim	Alias for dim()
output_nr	
real	Returns a new tensor containing real values of the self tensor for a complex-valued input tensor.

continues on next page

Table 18 – continued from previous page

requires_grad	Is True if gradients need to be computed for this Tensor, False otherwise.
retains_grad	Is True if this Tensor is non-leaf and its grad is enabled to be populated during backward(), False otherwise.
shape	Returns the size of the self tensor.
volatile	

property is_leaf: bool

All Tensors that have requires_grad which is False will be leaf Tensors by convention.

For Tensors that have requires_grad which is True, they will be leaf Tensors if they were created by the user. This means that they are not the result of an operation and so grad_fn is None.

Only leaf Tensors will have their grad populated during a call to backward(). To get grad populated for non-leaf Tensors, you can use retain_grad().

Example:

```
>>> a = torch.rand(10, requires_grad=True)
>>> a.is_leaf
True
>>> b = torch.rand(10, requires_grad=True).cuda()
>>> b.is_leaf
False
# b was created by the operation that cast a cpu Tensor into a cuda Tensor
>>> c = torch.rand(10, requires_grad=True) + 2
>>> c.is_leaf
False
# c was created by the addition operation
>>> d = torch.rand(10).cuda()
>>> d.is_leaf
True
# d does not require gradients and so has no operation creating it (that is␣
↪tracked by the autograd engine)
>>> e = torch.rand(10).cuda().requires_grad_()
>>> e.is_leaf
True
# e requires gradients and has no operations creating it
>>> f = torch.rand(10, requires_grad=True, device="cuda")
>>> f.is_leaf
True
# f requires grad, has no operation creating it
```

materialize(*shape*, *device=None*, *dtype=None*)

Create a Parameter with the same properties of the uninitialized one. Given a shape, it materializes a parameter in the same device and with the same *dtype* as the current one or the specified ones in the arguments.

Parameters

- **shape** (*Tuple[int, ...]*) – (tuple): the shape for the materialized tensor.
- **device** (torch.device) – the desired device of the parameters and buffers in this module. Optional.

- **dtype** (`torch.dtype`) – the desired floating point type of the floating point parameters and buffers in this module. Optional.

Return type
None

`share_memory_()`

Moves the underlying storage to shared memory.

This is a no-op if the underlying storage is already in shared memory and for CUDA tensors. Tensors in shared memory cannot be resized.

Return type
UninitializedParameter

pytorch_pfn_extras.nn.modules.lazy_conv

Classes

pytorch_pfn_extras.nn.modules.lazy_conv.LazyConv1d(...)	Conv1d module with lazy weight initialization.
pytorch_pfn_extras.nn.modules.lazy_conv.LazyConv2d(...)	Conv2d module with lazy weight initialization.
pytorch_pfn_extras.nn.modules.lazy_conv.LazyConv3d(...)	Conv3d module with lazy weight initialization.
pytorch_pfn_extras.nn.modules.lazy_conv.LazyInitializationMixin(...)	A mixin for modules that lazily initialize buffers and parameters.
pytorch_pfn_extras.nn.modules.lazy_conv.UninitializedParameter([...])	

pytorch_pfn_extras.nn.modules.lazy_conv.LazyConv1d

class pytorch_pfn_extras.nn.modules.lazy_conv.**LazyConv1d**(*in_channels*, **args*, ***kwargs*)

Bases: `_LazyConvNd`, `Conv1d`

Conv1d module with lazy weight initialization.

When `in_channels` is `None`, it is determined at the first time of the forward step.

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

Methods

`__init__`(in_channels, *args, **kwargs)	Initializes internal Module state, shared by both nn.Module and ScriptModule.
`add_module`(name, module)	Adds a child module to the current module.
`apply`(fn)	Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self.
`bfloat16`()	Casts all floating point parameters and buffers to `bfloat16` datatype.
`buffers`([recurse])	Returns an iterator over module buffers.

continues on next page

Table 19 – continued from previous page

children()	Returns an iterator over immediate children modules.
compile(*args, **kwargs)	Compile this Module's forward using torch.compile().
cpu()	Moves all model parameters and buffers to the CPU.
cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to double datatype.
eval()	Sets the module in evaluation mode.
extra_repr()	Set the extra representation of the module
float()	Casts all floating point parameters and buffers to float datatype.
forward(input)	Defines the computation performed at every call.
get_buffer(target)	Returns the buffer given by target if it exists, otherwise throws an error.
get_extra_state()	Returns any extra state to include in the module's state_dict.
get_parameter(target)	Returns the parameter given by target if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by target if it exists, otherwise throws an error.
half()	Casts all floating point parameters and buffers to half datatype.
ipu([device])	Moves all model parameters and buffers to the IPU.
load_state_dict(state_dict[, strict, assign])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse, ...])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse, ...])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook, *[, prepend, ...])	Registers a forward hook on the module.
register_forward_pre_hook(hook, *[, ...])	Registers a forward pre-hook on the module.
register_full_backward_hook(hook[, prepend])	Registers a backward hook on the module.
register_full_backward_pre_hook(hook[, prepend])	Registers a backward pre-hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.

continues on next page

Table 19 – continued from previous page

register_state_dict_pre_hook(hook)	These hooks will be called with arguments: self, prefix, and keep_vars before calling state_dict on self.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
reset_parameters()	
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict(*args, **kwargs)	Returns a dictionary containing a whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device[, recurse])	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Resets gradients of all model parameters.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
call_super_init	
dump_patches	
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

Parameters

- **in_channels** (*Optional[int]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

bias: Optional[Tensor]

dilation: Tuple[int, ...]

groups: int

in_channels: Optional[int]

kernel_size: Tuple[int, ...]

out_channels: int

output_padding: Tuple[int, ...]

padding: Union[str, Tuple[int, ...]]

padding_mode: str

stride: Tuple[int, ...]

training: bool

transposed: bool

weight: Tensor

pytorch_pfn_extras.nn.modules.lazy_conv.LazyConv2d

class pytorch_pfn_extras.nn.modules.lazy_conv.**LazyConv2d**(*in_channels*, **args*, ***kwargs*)

Bases: _LazyConvNd, Conv2d

Conv2d module with lazy weight initialization.

When `in_channels` is `None`, it is determined at the first time of the forward step.

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

Methods

__init__(in_channels, *args, **kwargs)	Initializes internal Module state, shared by both nn.Module and ScriptModule.
add_module(name, module)	Adds a child module to the current module.
apply(fn)	Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self.
bfloat16()	Casts all floating point parameters and buffers to `bfloat16` datatype.
buffers([recurse])	Returns an iterator over module buffers.
children()	Returns an iterator over immediate children modules.
compile(*args, **kwargs)	Compile this Module's forward using `torch.compile()`.
cpu()	Moves all model parameters and buffers to the CPU.
cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to `double` datatype.
eval()	Sets the module in evaluation mode.
extra_repr()	Set the extra representation of the module
float()	Casts all floating point parameters and buffers to `float` datatype.
forward(input)	Defines the computation performed at every call.
get_buffer(target)	Returns the buffer given by `target` if it exists, otherwise throws an error.
get_extra_state()	Returns any extra state to include in the module's state_dict.

continues on next page

Table 20 – continued from previous page

get_parameter(target)	Returns the parameter given by target if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by target if it exists, otherwise throws an error.
half()	Casts all floating point parameters and buffers to half datatype.
ipu([device])	Moves all model parameters and buffers to the IPU.
load_state_dict(state_dict[, strict, assign])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse, ...])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse, ...])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook, *[, prepend, ...])	Registers a forward hook on the module.
register_forward_pre_hook(hook, *[, ...])	Registers a forward pre-hook on the module.
register_full_backward_hook(hook[, prepend])	Registers a backward hook on the module.
register_full_backward_pre_hook(hook[, prepend])	Registers a backward pre-hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
register_state_dict_pre_hook(hook)	These hooks will be called with arguments: self, prefix, and keep_vars before calling state_dict on self.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
reset_parameters()	
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict(*args, **kwargs)	Returns a dictionary containing a whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device[, recurse])	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.

continues on next page

Table 20 – continued from previous page

xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Resets gradients of all model parameters.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
call_super_init	
dump_patches	
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

Parameters

- **in_channels** (*Optional[int]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

bias: Optional[Tensor]

dilation: Tuple[int, ...]

groups: int

in_channels: Optional[int]

kernel_size: Tuple[int, ...]

out_channels: int

output_padding: Tuple[int, ...]

padding: Union[str, Tuple[int, ...]]

padding_mode: str

stride: Tuple[int, ...]

training: bool

transposed: bool

weight: Tensor

pytorch_pfn_extras.nn.modules.lazy_conv.LazyConv3d

class pytorch_pfn_extras.nn.modules.lazy_conv.**LazyConv3d**(*in_channels*, **args*, ***kwargs*)

Bases: _LazyConvNd, Conv3d

Conv3d module with lazy weight initialization.

When in_channels is None, it is determined at the first time of the forward step.

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

Methods

__init__(in_channels, *args, **kwargs)	Initializes internal Module state, shared by both nn.Module and ScriptModule.
add_module(name, module)	Adds a child module to the current module.
apply(fn)	Applies fn recursively to every submodule (as returned by .children()) as well as self.
bfloat16()	Casts all floating point parameters and buffers to bfloat16 datatype.
buffers([recurse])	Returns an iterator over module buffers.
children()	Returns an iterator over immediate children modules.
compile(*args, **kwargs)	Compile this Module's forward using torch.compile().
cpu()	Moves all model parameters and buffers to the CPU.
cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to double datatype.
eval()	Sets the module in evaluation mode.
extra_repr()	Set the extra representation of the module
float()	Casts all floating point parameters and buffers to float datatype.
forward(input)	Defines the computation performed at every call.
get_buffer(target)	Returns the buffer given by target if it exists, otherwise throws an error.
get_extra_state()	Returns any extra state to include in the module's state_dict.
get_parameter(target)	Returns the parameter given by target if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by target if it exists, otherwise throws an error.
half()	Casts all floating point parameters and buffers to half datatype.
ipu([device])	Moves all model parameters and buffers to the IPU.
load_state_dict(state_dict[, strict, assign])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse, ...])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

continues on next page

Table 21 – continued from previous page

named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse, ...])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook, *[, prepend, ...])	Registers a forward hook on the module.
register_forward_pre_hook(hook, *[, ...])	Registers a forward pre-hook on the module.
register_full_backward_hook(hook[, prepend])	Registers a backward hook on the module.
register_full_backward_pre_hook(hook[, prepend])	Registers a backward pre-hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
register_state_dict_pre_hook(hook)	These hooks will be called with arguments: self, prefix, and keep_vars before calling state_dict on self.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
reset_parameters()	
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict(*args, **kwargs)	Returns a dictionary containing a whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device[, recurse])	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Resets gradients of all model parameters.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
call_super_init	
dump_patches	
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

Parameters

- **in_channels** (*Optional[int]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

bias: Optional[Tensor]

dilation: Tuple[int, ...]

groups: int

in_channels: Optional[int]

kernel_size: Tuple[int, ...]

out_channels: int

output_padding: Tuple[int, ...]

padding: Union[str, Tuple[int, ...]]

padding_mode: str

stride: Tuple[int, ...]

training: bool

transposed: bool

weight: Tensor

pytorch_pfn_extras.nn.modules.lazy_conv.LazyInitializationMixin

class pytorch_pfn_extras.nn.modules.lazy_conv.**LazyInitializationMixin**(**args*, ***kwargs*)

Bases: `object`

A mixin for modules that lazily initialize buffers and parameters.

Unlike regular modules, subclasses of this module can initialize buffers and parameters outside of the constructor (`__init__`). This allows you to, for example, initialize parameters in `forward` method to determine the shape of the weight based on the initial input.

Be sure to run "dummy" forward once to initialize all parameters that should be trained, before passing `module.parameters()` to an optimizer; otherwise weights initialized after `module.parameters()` (e.g., in `forward` function) will never be trained.

Note that lazy modules cannot validate if the shape is correct during deserialization. Also note that the initial weights may become different from the original (non-lazy) module even if the random seed is manually configured, as the order of initialization is different from the original one; especially, `module.cuda()` may cause the initialization to run on a GPU.

The default value of lazy buffers and parameters are `torch.Tensor([])` and `UninitializedParameter()`, respectively.

Methods

__init__(*args, **kwargs)	
state_dict(*args, **kwargs)	Returns a dictionary containing a whole state of the module.

Attributes

lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

__init__(**args*, ***kwargs*)

Parameters

- **self** (*Any*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

lazy_buffer_names: Tuple[str, ...] = ()

lazy_parameter_names: Tuple[str, ...] = ()

property lazy_parmeters_determined: bool

Returns if all lazy parameters are determined.

Subclasses can perform parameters initialization after all lazy parameters are determined. Note that this may be called during `__init__`.

state_dict(**args*, ***kwargs*)

Returns a dictionary containing a whole state of the module.

This function overrides the default behavior to exclude uninitialized parameter from serialization. This is needed because we need to discriminate lazy parameters (`UninitializedParameter()`) and initialized empty parameters (``torch.nn.Parameter(torch.Tensor())`) during deserialization.

See comments of `_lazy_load_hook` for details.

Parameters

- **self** (*Any*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Dict[str, *Any*]

pytorch_pfn_extras.nn.modules.lazy_conv.UninitializedParameter

class pytorch_pfn_extras.nn.modules.lazy_conv.**UninitializedParameter**(*data=None*, *requires_grad=True*)

Bases: `Parameter`

Methods

`__init__`()	
`abs`()	See `torch.abs()`
`abs_`()	In-place version of `abs()`
`absolute`()	Alias for `abs()`
`absolute_`()	In-place version of `absolute()` Alias for `abs_()`
`acos`()	See `torch.acos()`
`acos_`()	In-place version of `acos()`
`acosh`()	See `torch.acosh()`
`acosh_`()	In-place version of `acosh()`
`add`(other, *[, alpha])	Add a scalar or tensor to `self` tensor.
`add_`(other, *[, alpha])	In-place version of `add()`
`addbmm`(batch1, batch2, *[, beta, alpha])	See `torch.addbmm()`
`addbmm_`(batch1, batch2, *[, beta, alpha])	In-place version of `addbmm()`
`addcdiv`(tensor1, tensor2, *[, value])	See `torch.addcdiv()`

continues on next page

Table 22 – continued from previous page

addcdiv_(tensor1, tensor2, *[, value])	In-place version of addcdiv()
addcmul(tensor1, tensor2, *[, value])	See torch.addcmul()
addcmul_(tensor1, tensor2, *[, value])	In-place version of addcmul()
addmm(mat1, mat2, *[, beta, alpha])	See torch.addmm()
addmm_(mat1, mat2, *[, beta, alpha])	In-place version of addmm()
addmv(mat, vec, *[, beta, alpha])	See torch.addmv()
addmv_(mat, vec, *[, beta, alpha])	In-place version of addmv()
addr(vec1, vec2, *[, beta, alpha])	See torch.addr()
addr_(vec1, vec2, *[, beta, alpha])	In-place version of addr()
adjoint()	Alias for adjoint()
align_as(other)	Permutes the dimensions of the self tensor to match the dimension order in the other tensor, adding size-one dims for any new names.
align_to(*names)	Permutes the dimensions of the self tensor to match the order specified in names, adding size-one dims for any new names.
all([dim, keepdim])	See torch.all()
allclose(other[, rtol, atol, equal_nan])	See torch.allclose()
amax([dim, keepdim])	See torch.amax()
amin([dim, keepdim])	See torch.amin()
aminmax(*[, dim, keepdim])	See torch.aminmax()
angle()	See torch.angle()
any([dim, keepdim])	See torch.any()
apply_(callable)	Applies the function callable to each element in the tensor, replacing each element with the value returned by callable.
arccos()	See torch.arccos()
arccos_()	In-place version of arccos()
arccosh	acosh() -> Tensor
arccosh_	acosh_() -> Tensor
arcsin()	See torch.arcsin()
arcsin_()	In-place version of arcsin()
arcsinh()	See torch.arcsinh()
arcsinh_()	In-place version of arcsinh()
arctan()	See torch.arctan()
arctan2(other)	See torch.arctan2()
arctan2_	atan2_(other) -> Tensor
arctan_()	In-place version of arctan()
arctanh()	See torch.arctanh()
arctanh_(other)	In-place version of arctanh()
argmax([dim, keepdim])	See torch.argmax()
argmin([dim, keepdim])	See torch.argmin()
argsort([dim, descending])	See torch.argsort()
argwhere()	See torch.argwhere()
as_strided(size, stride[, storage_offset])	See torch.as_strided()
as_strided_(size, stride[, storage_offset])	In-place version of as_strided()
as_strided_scatter(src, size, stride[, ...])	See torch.as_strided_scatter()
as_subclass(cls)	Makes a cls instance with the same data pointer as self.
asin()	See torch.asin()

continues on next page

Table 22 – continued from previous page

asin_()	In-place version of asin()
asinh()	See torch.asinh()
asinh_()	In-place version of asinh()
atan()	See torch.atan()
atan2(other)	See torch.atan2()
atan2_(other)	In-place version of atan2()
atan_()	In-place version of atan()
atanh()	See torch.atanh()
atanh_(other)	In-place version of atanh()
backward([gradient, retain_graph, ...])	Computes the gradient of current tensor wrt graph leaves.
baddbmm(batch1, batch2, *[, beta, alpha])	See torch.baddbmm()
baddbmm_(batch1, batch2, *[, beta, alpha])	In-place version of baddbmm()
bernoulli(*[, generator])	Returns a result tensor where each result[i] is independently sampled from Bernoulli(self[i]).
bernoulli_([p, generator])	Fills each location of self with an independent sample from Bernoulli(p).
bfloat16([memory_format])	self.bfloat16() is equivalent to self.to(torch.bfloat16).
bincount([weights, minlength])	See torch.bincount()
bitwise_and()	See torch.bitwise_and()
bitwise_and_()	In-place version of bitwise_and()
bitwise_left_shift(other)	See torch.bitwise_left_shift()
bitwise_left_shift_(other)	In-place version of bitwise_left_shift()
bitwise_not()	See torch.bitwise_not()
bitwise_not_()	In-place version of bitwise_not()
bitwise_or()	See torch.bitwise_or()
bitwise_or_()	In-place version of bitwise_or()
bitwise_right_shift(other)	See torch.bitwise_right_shift()
bitwise_right_shift_(other)	In-place version of bitwise_right_shift()
bitwise_xor()	See torch.bitwise_xor()
bitwise_xor_()	In-place version of bitwise_xor()
bmm(batch2)	See torch.bmm()
bool([memory_format])	self.bool() is equivalent to self.to(torch.bool).
broadcast_to(shape)	See torch.broadcast_to().
byte([memory_format])	self.byte() is equivalent to self.to(torch.uint8).
cauchy_([median, sigma, generator])	Fills the tensor with numbers drawn from the Cauchy distribution:
ccol_indices	
cdouble([memory_format])	self.cdouble() is equivalent to self.to(torch.complex128).
ceil()	See torch.ceil()
ceil_()	In-place version of ceil()
cfloat([memory_format])	self.cfloat() is equivalent to self.to(torch.complex64).
chalf([memory_format])	self.chalf() is equivalent to self.to(torch.complex32).

continues on next page

Table 22 – continued from previous page

char([memory_format])	self.char() is equivalent to self.to(torch.int8).
cholesky([upper])	See torch.cholesky()
cholesky_inverse([upper])	See torch.cholesky_inverse()
cholesky_solve(input2[, upper])	See torch.cholesky_solve()
chunk(chunks[, dim])	See torch.chunk()
clamp([min, max])	See torch.clamp()
clamp_([min, max])	In-place version of clamp()
clamp_max	
clamp_max_	
clamp_min	
clamp_min_	
clip([min, max])	Alias for clamp().
clip_([min, max])	Alias for clamp_().
clone(*[, memory_format])	See torch.clone()
coalesce()	Returns a coalesced copy of self if self is an uncoalesced tensor.
col_indices()	Returns the tensor containing the column indices of the self tensor when self is a sparse CSR tensor of layout sparse_csr.
conj()	See torch.conj()
conj_physical()	See torch.conj_physical()
conj_physical_()	In-place version of conj_physical()
contiguous([memory_format])	Returns a contiguous in memory tensor containing the same data as self tensor.
copy_(src[, non_blocking])	Copies the elements from src into self tensor and returns self.
copysign(other)	See torch.copysign()
copysign_(other)	In-place version of copysign()
corrcoef()	See torch.corrcoef()
cos()	See torch.cos()
cos_()	In-place version of cos()
cosh()	See torch.cosh()
cosh_()	In-place version of cosh()
count_nonzero([dim])	See torch.count_nonzero()
cov(*[, correction, fweights, aweights])	See torch.cov()
cpu([memory_format])	Returns a copy of this object in CPU memory.
cross(other[, dim])	See torch.cross()
crow_indices()	Returns the tensor containing the compressed row indices of the self tensor when self is a sparse CSR tensor of layout sparse_csr.
cuda([device, non_blocking, memory_format])	Returns a copy of this object in CUDA memory.
cummax(dim)	See torch.cummax()
cummin(dim)	See torch.cummin()
cumprod(dim[, dtype])	See torch.cumprod()
cumprod_(dim[, dtype])	In-place version of cumprod()
cumsum(dim[, dtype])	See torch.cumsum()

continues on next page

Table 22 – continued from previous page

cumsum_(dim[, dtype])	In-place version of cumsum()
data_ptr()	Returns the address of the first element of self tensor.
deg2rad()	See torch.deg2rad()
deg2rad_()	In-place version of deg2rad()
dense_dim()	Return the number of dense dimensions in a sparse tensor self.
dequantize()	Given a quantized Tensor, dequantize it and return the dequantized float Tensor.
det()	See torch.det()
detach	Returns a new Tensor, detached from the current graph.
detach_	Detaches the Tensor from the graph that created it, making it a leaf.
diag([diagonal])	See torch.diag()
diag_embed([offset, dim1, dim2])	See torch.diag_embed()
diagflat([offset])	See torch.diagflat()
diagonal([offset, dim1, dim2])	See torch.diagonal()
diagonal_scatter(src[, offset, dim1, dim2])	See torch.diagonal_scatter()
diff([n, dim, prepend, append])	See torch.diff()
digamma()	See torch.digamma()
digamma_()	In-place version of digamma()
dim()	Returns the number of dimensions of self tensor.
dim_order()	Returns a tuple of int describing the dim order or physical layout of self.
dist(other[, p])	See torch.dist()
div(value, *[, rounding_mode])	See torch.div()
div_(value, *[, rounding_mode])	In-place version of div()
divide(value, *[, rounding_mode])	See torch.divide()
divide_(value, *[, rounding_mode])	In-place version of divide()
dot(other)	See torch.dot()
double([memory_format])	self.double() is equivalent to self.to(torch.float64).
dsplit(split_size_or_sections)	See torch.dsplit()
eig([eigenvectors])	
element_size()	Returns the size in bytes of an individual element.
eq(other)	See torch.eq()
eq_(other)	In-place version of eq()
equal(other)	See torch.equal()
erf()	See torch.erf()
erf_()	In-place version of erf()
erfc()	See torch.erfc()
erfc_()	In-place version of erfc()
erfinv()	See torch.erfinv()
erfinv_()	In-place version of erfinv()
exp()	See torch.exp()
exp2()	See torch.exp2()
exp2_()	In-place version of exp2()
exp_()	In-place version of exp()

continues on next page

Table 22 – continued from previous page

expand(*sizes)	Returns a new view of the self tensor with singleton dimensions expanded to a larger size.
expand_as(other)	Expand this tensor to the same size as other.
expm1()	See torch.expm1()
expm1_()	In-place version of expm1()
exponential_([lambd, generator])	Fills self tensor with elements drawn from the exponential distribution:
fill_(value)	Fills self tensor with the specified value.
fill_diagonal_(fill_value[, wrap])	Fill the main diagonal of a tensor that has at least 2-dimensions.
fix()	See torch.fix().
fix_()	In-place version of fix()
flatten([start_dim, end_dim])	See torch.flatten()
flip(dims)	See torch.flip()
fliplr()	See torch.fliplr()
flipud()	See torch.flipud()
float([memory_format])	self.float() is equivalent to self.to(torch.float32).
float_power(exponent)	See torch.float_power()
float_power_(exponent)	In-place version of float_power()
floor()	See torch.floor()
floor_()	In-place version of floor()
floor_divide(value)	See torch.floor_divide()
floor_divide_(value)	In-place version of floor_divide()
fmax(other)	See torch.fmax()
fmin(other)	See torch.fmin()
fmod(divisor)	See torch.fmod()
fmod_(divisor)	In-place version of fmod()
frac()	See torch.frac()
frac_()	In-place version of frac()
frexp(input)	See torch.frexp()
gather(dim, index)	See torch.gather()
gcd(other)	See torch.gcd()
gcd_(other)	In-place version of gcd()
ge(other)	See torch.ge().
ge_(other)	In-place version of ge().
geometric_(p, *[, generator])	Fills self tensor with elements drawn from the geometric distribution:
geqrf()	See torch.geqrf()
ger(vec2)	See torch.ger()
get_device()	For CUDA tensors, this function returns the device ordinal of the GPU on which the tensor resides.
greater(other)	See torch.greater().
greater_(other)	In-place version of greater().
greater_equal(other)	See torch.greater_equal().
greater_equal_(other)	In-place version of greater_equal().
gt(other)	See torch.gt().
gt_(other)	In-place version of gt().
half([memory_format])	self.half() is equivalent to self.to(torch.float16).

continues on next page

Table 22 – continued from previous page

hardshrink([lambd])	See torch.nn.functional.hardshrink()
has_names	Is True if any of this tensor's dimensions are named.
heaviside(values)	See torch.heaviside()
heaviside_(values)	In-place version of heaviside()
histc([bins, min, max])	See torch.histc()
histogram(input, bins, *[, range, weight, ...])	See torch.histogram()
hsplit(split_size_or_sections)	See torch.hsplit()
hypot(other)	See torch.hypot()
hypot_(other)	In-place version of hypot()
i0()	See torch.i0()
i0_()	In-place version of i0()
igamma(other)	See torch.igamma()
igamma_(other)	In-place version of igamma()
igammac(other)	See torch.igammac()
igammac_(other)	In-place version of igammac()
index_add(dim, index, source, *[, alpha])	Out-of-place version of torch.Tensor.index_add_().
index_add_(dim, index, source, *[, alpha])	Accumulate the elements of alpha times source into the self tensor by adding to the indices in the order given in index.
index_copy(dim, index, tensor2)	Out-of-place version of torch.Tensor.index_copy_().
index_copy_(dim, index, tensor)	Copies the elements of tensor into the self tensor by selecting the indices in the order given in index.
index_fill(dim, index, value)	Out-of-place version of torch.Tensor.index_fill_().
index_fill_(dim, index, value)	Fills the elements of the self tensor with value value by selecting the indices in the order given in index.
index_put(indices, values[, accumulate])	Out-place version of index_put_().
index_put_(indices, values[, accumulate])	Puts values from the tensor values into the tensor self using the indices specified in indices (which is a tuple of Tensors).
index_reduce	
index_reduce_(dim, index, source, reduce, *)	Accumulate the elements of source into the self tensor by accumulating to the indices in the order given in index using the reduction given by the reduce argument.
index_select(dim, index)	See torch.index_select()
indices()	Return the indices tensor of a sparse COO tensor.
inner(other)	See torch.inner().
int([memory_format])	self.int() is equivalent to self.to(torch.int32).
int_repr()	Given a quantized Tensor, self.int_repr() returns a CPU Tensor with uint8_t as data type that stores the underlying uint8_t values of the given Tensor.
inverse()	See torch.inverse()
ipu([device, non_blocking, memory_format])	Returns a copy of this object in IPU memory.

continues on next page

Table 22 – continued from previous page

is_coalesced()	Returns True if self is a sparse COO tensor that is coalesced, False otherwise.
is_complex()	Returns True if the data type of self is a complex data type.
is_conj()	Returns True if the conjugate bit of self is set to true.
is_contiguous([memory_format])	Returns True if self tensor is contiguous in memory in the order specified by memory format.
is_distributed	
is_floating_point()	Returns True if the data type of self is a floating point data type.
is_inference()	See torch.is_inference()
is_neg()	Returns True if the negative bit of self is set to true.
is_nonzero	
is_pinned	Returns true if this tensor resides in pinned memory.
is_same_size	
is_set_to(tensor)	Returns True if both tensors are pointing to the exact same memory (same storage, offset, size and stride).
is_shared()	Checks if tensor is in shared memory.
is_signed()	Returns True if the data type of self is a signed data type.
isclose(other[, rtol, atol, equal_nan])	See torch.isclose()
isfinite()	See torch.isfinite()
isinf()	See torch.isinf()
isnan()	See torch.isnan()
isneginf()	See torch.isneginf()
isposinf()	See torch.isposinf()
isreal()	See torch.isreal()
istft(n_fft[, hop_length, win_length, ...])	See torch.istft()
item()	Returns the value of this tensor as a standard Python number.
kron(other)	See torch.kron()
kthvalue(k[, dim, keepdim])	See torch.kthvalue()
lcm(other)	See torch.lcm()
lcm_(other)	In-place version of lcm()
ldexp(other)	See torch.ldexp()
ldexp_(other)	In-place version of ldexp()
le(other)	See torch.le().
le_(other)	In-place version of le().
lerp(end, weight)	See torch.lerp()
lerp_(end, weight)	In-place version of lerp()
less	lt(other) -> Tensor
less_(other)	In-place version of less().
less_equal(other)	See torch.less_equal().
less_equal_(other)	In-place version of less_equal().
lgamma()	See torch.lgamma()
lgamma_()	In-place version of lgamma()
log()	See torch.log()
log10()	See torch.log10()

continues on next page

Table 22 – continued from previous page

log10_()	In-place version of log10()
log1p()	See torch.log1p()
log1p_()	In-place version of log1p()
log2()	See torch.log2()
log2_()	In-place version of log2()
log_()	In-place version of log()
log_normal_([mean, std, generator])	Fills self tensor with numbers samples from the log-normal distribution parameterized by the given mean μ and standard deviation σ.
log_softmax	
logaddexp(other)	See torch.logaddexp()
logaddexp2(other)	See torch.logaddexp2()
logcumsumexp(dim)	See torch.logcumsumexp()
logdet()	See torch.logdet()
logical_and()	See torch.logical_and()
logical_and_()	In-place version of logical_and()
logical_not()	See torch.logical_not()
logical_not_()	In-place version of logical_not()
logical_or()	See torch.logical_or()
logical_or_()	In-place version of logical_or()
logical_xor()	See torch.logical_xor()
logical_xor_()	In-place version of logical_xor()
logit()	See torch.logit()
logit_()	In-place version of logit()
logsumexp(dim[, keepdim])	See torch.logsumexp()
long([memory_format])	self.long() is equivalent to self.to(torch.int64).
lstsq(other)	
lt(other)	See torch.lt().
lt_(other)	In-place version of lt().
lu([pivot, get_infos])	See torch.lu()
lu_solve(LU_data, LU_pivots)	See torch.lu_solve()
map2_	
map_(tensor, callable)	Applies callable for each element in self tensor and the given tensor and stores the results in self tensor.
masked_fill(mask, value)	Out-of-place version of torch.Tensor.masked_fill_()
masked_fill_(mask, value)	Fills elements of self tensor with value where mask is True.
masked_scatter(mask, tensor)	Out-of-place version of torch.Tensor.masked_scatter_()
masked_scatter_(mask, source)	Copies elements from source into self tensor at positions where the mask is True.
masked_select(mask)	See torch.masked_select()
materialize(shape[, device, dtype])	Create a Parameter with the same properties of the uninitialized one.
matmul(tensor2)	See torch.matmul()

continues on next page

Table 22 – continued from previous page

matrix_exp()	See torch.matrix_exp()
matrix_power(n)	**Note:** matrix_power() is deprecated, use torch.linalg.matrix_power() instead.
max([dim, keepdim])	See torch.max()
maximum(other)	See torch.maximum()
mean([dim, keepdim, dtype])	See torch.mean()
median([dim, keepdim])	See torch.median()
min([dim, keepdim])	See torch.min()
minimum(other)	See torch.minimum()
mm(mat2)	See torch.mm()
mode([dim, keepdim])	See torch.mode()
moveaxis(source, destination)	See torch.moveaxis()
movedim(source, destination)	See torch.movedim()
msort()	See torch.msort()
mul(value)	See torch.mul().
mul_(value)	In-place version of mul().
multinomial(num_samples[, replacement, ...])	See torch.multinomial()
multiply(value)	See torch.multiply().
multiply_(value)	In-place version of multiply().
mv(vec)	See torch.mv()
mvlgamma(p)	See torch.mvlgamma()
mvlgamma_(p)	In-place version of mvlgamma()
nan_to_num([nan, posinf, neginf])	See torch.nan_to_num().
nan_to_num_([nan, posinf, neginf])	In-place version of nan_to_num().
nanmean([dim, keepdim, dtype])	See torch.nanmean()
nanmedian([dim, keepdim])	See torch.nanmedian()
nanquantile(q[, dim, keepdim, interpolation])	See torch.nanquantile()
nansum([dim, keepdim, dtype])	See torch.nansum()
narrow(dimension, start, length)	See torch.narrow().
narrow_copy(dimension, start, length)	See torch.narrow_copy().
ndimension()	Alias for dim()
ne(other)	See torch.ne().
ne_(other)	In-place version of ne().
neg()	See torch.neg()
neg_()	In-place version of neg()
negative()	See torch.negative()
negative_()	In-place version of negative()
nelement()	Alias for numel()
new	
new_empty(size, *[, dtype, device, ...])	Returns a Tensor of size size filled with uninitialized data.
new_empty_strided(size, stride[, dtype, ...])	Returns a Tensor of size size and strides stride filled with uninitialized data.
new_full(size, fill_value, *[, dtype, ...])	Returns a Tensor of size size filled with fill_value.
new_ones(size, *[, dtype, device, ...])	Returns a Tensor of size size filled with 1.

continues on next page

Table 22 – continued from previous page

new_tensor(data, *[, dtype, device, ...])	Returns a new Tensor with data as the tensor data.
new_zeros(size, *[, dtype, device, ...])	Returns a Tensor of size size filled with 0.
nextafter(other)	See torch.nextafter()
nextafter_(other)	In-place version of nextafter()
nonzero()	See torch.nonzero()
nonzero_static(input, *, size[, fill_value])	Returns a 2-D tensor where each row is the index for a non-zero value.
norm([p, dim, keepdim, dtype])	See torch.norm()
normal_([mean, std, generator])	Fills self tensor with elements samples from the normal distribution parameterized by mean and std.
not_equal(other)	See torch.not_equal().
not_equal_(other)	In-place version of not_equal().
numel()	See torch.numel()
numpy(*[, force])	Returns the tensor as a NumPy ndarray.
orgqr(input2)	See torch.orgqr()
ormqr(input2, input3[, left, transpose])	See torch.ormqr()
outer(vec2)	See torch.outer().
permute(*dims)	See torch.permute()
pin_memory()	Copies the tensor to pinned memory, if it's not already pinned.
pinverse()	See torch.pinverse()
polygamma(n)	See torch.polygamma()
polygamma_(n)	In-place version of polygamma()
positive()	See torch.positive()
pow(exponent)	See torch.pow()
pow_(exponent)	In-place version of pow()
prelu	
prod([dim, keepdim, dtype])	See torch.prod()
put(input, index, source[, accumulate])	Out-of-place version of torch.Tensor.put_().
put_(index, source[, accumulate])	Copies the elements from source into the positions specified by index.
q_per_channel_axis()	Given a Tensor quantized by linear (affine) per-channel quantization, returns the index of dimension on which per-channel quantization is applied.
q_per_channel_scales()	Given a Tensor quantized by linear (affine) per-channel quantization, returns a Tensor of scales of the underlying quantizer.
q_per_channel_zero_points()	Given a Tensor quantized by linear (affine) per-channel quantization, returns a tensor of zero_points of the underlying quantizer.
q_scale()	Given a Tensor quantized by linear(affine) quantization, returns the scale of the underlying quantizer().
q_zero_point()	Given a Tensor quantized by linear(affine) quantization, returns the zero_point of the underlying quantizer().
qr([some])	See torch.qr()
qscheme()	Returns the quantization scheme of a given QTensor.
quantile(q[, dim, keepdim, interpolation])	See torch.quantile()
rad2deg()	See torch.rad2deg()
rad2deg_()	In-place version of rad2deg()

continues on next page

Table 22 – continued from previous page

random_([from, to, generator])	Fills self tensor with numbers sampled from the discrete uniform distribution over [from, to - 1].
ravel()	see torch.ravel()
reciprocal()	See torch.reciprocal()
reciprocal_()	In-place version of reciprocal()
record_stream(stream)	Ensures that the tensor memory is not reused for another tensor until all current work queued on stream are complete.
refine_names(*names)	Refines the dimension names of self according to names.
register_hook(hook)	Registers a backward hook.
register_post_accumulate_grad_hook(hook)	Registers a backward hook that runs after grad accumulation.
reinforce(reward)	
relu	
relu_	
remainder(divisor)	See torch.remainder()
remainder_(divisor)	In-place version of remainder()
rename(*names, **rename_map)	Renames dimension names of self.
rename_(*names, **rename_map)	In-place version of rename().
renorm(p, dim, maxnorm)	See torch.renorm()
renorm_(p, dim, maxnorm)	In-place version of renorm()
repeat(*sizes)	Repeats this tensor along the specified dimensions.
repeat_interleave(repeats[, dim, output_size])	See torch.repeat_interleave().
requires_grad_([requires_grad])	Change if autograd should record operations on this tensor: sets this tensor's requires_grad attribute in-place.
reshape(*shape)	Returns a tensor with the same data and number of elements as self but with the specified shape.
reshape_as(other)	Returns this tensor as the same shape as other.
resize(*sizes)	
resize_(*sizes[, memory_format])	Resizes self tensor to the specified size.
resize_as(tensor)	
resize_as_(tensor[, memory_format])	Resizes the self tensor to be the same size as the specified tensor.
resize_as_sparse_	
resolve_conj()	See torch.resolve_conj()
resolve_neg()	See torch.resolve_neg()
retain_grad()	Enables this Tensor to have their grad populated during backward().
roll(shifts, dims)	See torch.roll()
rot90(k, dims)	See torch.rot90()
round([decimals])	See torch.round()
round_([decimals])	In-place version of round()

continues on next page

Table 22 – continued from previous page

row_indices	
rsqrt()	See torch.rsqrt()
rsqrt_()	In-place version of rsqrt()
scatter(dim, index, src)	Out-of-place version of torch.Tensor.scatter_()
scatter_(dim, index, src[, reduce])	Writes all values from the tensor src into self at the indices specified in the index tensor.
scatter_add(dim, index, src)	Out-of-place version of torch.Tensor.scatter_add_()
scatter_add_(dim, index, src)	Adds all values from the tensor src into self at the indices specified in the index tensor in a similar fashion as scatter_().
scatter_reduce(dim, index, src, reduce, *[, ...])	Out-of-place version of torch.Tensor.scatter_reduce_()
scatter_reduce_(dim, index, src, reduce, *)	Reduces all values from the src tensor to the indices specified in the index tensor in the self tensor using the applied reduction defined via the reduce argument ("sum", "prod", "mean", "amax", "amin").
select(dim, index)	See torch.select()
select_scatter(src, dim, index)	See torch.select_scatter()
set_([source, storage_offset, size, stride])	Sets the underlying storage, size, and strides.
sgn()	See torch.sgn()
sgn_()	In-place version of sgn()
*share_memory_*()	Moves the underlying storage to shared memory.
short([memory_format])	self.short() is equivalent to self.to(torch.int16).
sigmoid()	See torch.sigmoid()
sigmoid_()	In-place version of sigmoid()
sign()	See torch.sign()
sign_()	In-place version of sign()
signbit()	See torch.signbit()
sin()	See torch.sin()
sin_()	In-place version of sin()
sinc()	See torch.sinc()
sinc_()	In-place version of sinc()
sinh()	See torch.sinh()
sinh_()	In-place version of sinh()
size([dim])	Returns the size of the self tensor.
slice_scatter(src[, dim, start, end, step])	See torch.slice_scatter()
slogdet()	See torch.slogdet()
smm(mat)	See torch.smm()
softmax(dim)	Alias for torch.nn.functional.softmax().
solve(other)	
sort([dim, descending])	See torch.sort()
sparse_dim()	Return the number of sparse dimensions in a sparse tensor self.
sparse_mask(mask)	Returns a new sparse tensor with values from a strided tensor self filtered by the indices of the sparse tensor mask.

continues on next page

Table 22 – continued from previous page

sparse_resize_(size, sparse_dim, dense_dim)	Resizes self sparse tensor to the desired size and the number of sparse and dense dimensions.
sparse_resize_and_clear_(size, sparse_dim, ...)	Removes all specified elements from a sparse tensor self and resizes self to the desired size and the number of sparse and dense dimensions.
split(split_size[, dim])	See torch.split()
split_with_sizes	
sqrt()	See torch.sqrt()
sqrt_()	In-place version of sqrt()
square()	See torch.square()
square_()	In-place version of square()
squeeze([dim])	See torch.squeeze()
squeeze_([dim])	In-place version of squeeze()
sspaddmm(mat1, mat2, *[, beta, alpha])	See torch.sspaddmm()
std([dim, correction, keepdim])	See torch.std()
stft(n_fft[, hop_length, win_length, ...])	See torch.stft()
storage()	Returns the underlying TypedStorage.
storage_offset()	Returns self tensor's offset in the underlying storage in terms of number of storage elements (not bytes).
storage_type()	Returns the type of the underlying storage.
stride(dim)	Returns the stride of self tensor.
sub(other, *[, alpha])	See torch.sub().
sub_(other, *[, alpha])	In-place version of sub()
subtract(other, *[, alpha])	See torch.subtract().
subtract_(other, *[, alpha])	In-place version of subtract().
sum([dim, keepdim, dtype])	See torch.sum()
sum_to_size(*size)	Sum this tensor to size.
svd([some, compute_uv])	See torch.svd()
swapaxes(axis0, axis1)	See torch.swapaxes()
swapaxes_(axis0, axis1)	In-place version of swapaxes()
swapdims(dim0, dim1)	See torch.swapdims()
swapdims_(dim0, dim1)	In-place version of swapdims()
symeig([eigenvectors])	
t()	See torch.t()
t_()	In-place version of t()
take(indices)	See torch.take()
take_along_dim(indices, dim)	See torch.take_along_dim()
tan()	See torch.tan()
tan_()	In-place version of tan()
tanh()	See torch.tanh()
tanh_()	In-place version of tanh()
tensor_split(indices_or_sections[, dim])	See torch.tensor_split()
tile(dims)	See torch.tile()
to(*args, **kwargs)	Performs Tensor dtype and/or device conversion.
to_dense([dtype, masked_grad])	Creates a strided copy of self if self is not a strided tensor, otherwise returns self.
to_mkldnn()	Returns a copy of the tensor in torch.mkldnn layout.
to_padded_tensor(padding[, output_size])	See to_padded_tensor()

continues on next page

Table 22 – continued from previous page

to_sparse(sparseDims)	Returns a sparse copy of the tensor.
to_sparse_bsc(blocksize, dense_dim)	Convert a tensor to a block sparse column (BSC) storage format of given blocksize.
to_sparse_bsr(blocksize, dense_dim)	Convert a tensor to a block sparse row (BSR) storage format of given blocksize.
to_sparse_coo()	Convert a tensor to coordinate format.
to_sparse_csc()	Convert a tensor to compressed column storage (CSC) format.
to_sparse_csr([dense_dim])	Convert a tensor to compressed row storage format (CSR).
tolist()	Returns the tensor as a (nested) list.
topk(k[, dim, largest, sorted])	See torch.topk()
trace()	See torch.trace()
transpose(dim0, dim1)	See torch.transpose()
transpose_(dim0, dim1)	In-place version of transpose()
triangular_solve(A[, upper, transpose, ...])	See torch.triangular_solve()
tril([diagonal])	See torch.tril()
tril_([diagonal])	In-place version of tril()
triu([diagonal])	See torch.triu()
triu_([diagonal])	In-place version of triu()
true_divide(value)	See torch.true_divide()
true_divide_(value)	In-place version of true_divide_()
trunc()	See torch.trunc()
trunc_()	In-place version of trunc()
type([dtype, non_blocking])	Returns the type if *dtype* is not provided, else casts this object to the specified type.
type_as(tensor)	Returns this tensor cast to the type of the given tensor.
unbind([dim])	See torch.unbind()
unflatten(dim, sizes)	See torch.unflatten().
unfold(dimension, size, step)	Returns a view of the original tensor which contains all slices of size size from self tensor in the dimension dimension.
uniform_([from, to, generator])	Fills self tensor with numbers sampled from the continuous uniform distribution:
unique([sorted, return_inverse, ...])	Returns the unique elements of the input tensor.
unique_consecutive([return_inverse, ...])	Eliminates all but the first element from every consecutive group of equivalent elements.
unsafe_chunk(chunks[, dim])	See torch.unsafe_chunk()
unsafe_split(split_size[, dim])	See torch.unsafe_split()
unsafe_split_with_sizes	
unsqueeze(dim)	See torch.unsqueeze()
unsqueeze_(dim)	In-place version of unsqueeze()
untyped_storage()	Returns the underlying UntypedStorage.
values()	Return the values tensor of a sparse COO tensor.
var([dim, correction, keepdim])	See torch.var()
vdot(other)	See torch.vdot()
view(*shape)	Returns a new tensor with the same data as the self tensor but of a different shape.
view_as(other)	View this tensor as the same size as other.
vsplit(split_size_or_sections)	See torch.vsplit()

continues on next page

Table 22 – continued from previous page

where(condition, y)	self.where(condition, y) is equivalent to torch.where(condition, self, y).
xlogy(other)	See torch.xlogy()
xlogy_(other)	In-place version of xlogy()
xpu([device, non_blocking, memory_format])	Returns a copy of this object in XPU memory.
zero_()	Fills self tensor with zeros.

Attributes

H	Returns a view of a matrix (2-D tensor) conjugated and transposed.
T	Returns a view of this tensor with its dimensions reversed.
data	
device	Is the torch.device where this Tensor is.
dtype	
grad	This attribute is None by default and becomes a Tensor the first time a call to backward() computes gradients for self.
grad_fn	
imag	Returns a new tensor containing imaginary values of the self tensor.
is_cpu	Is True if the Tensor is stored on the CPU, False otherwise.
is_cuda	Is True if the Tensor is stored on the GPU, False otherwise.
is_ipu	Is True if the Tensor is stored on the IPU, False otherwise.
is_leaf	All Tensors that have requires_grad which is False will be leaf Tensors by convention.
is_meta	Is True if the Tensor is a meta tensor, False otherwise.
is_mkldnn	
is_mps	Is True if the Tensor is stored on the MPS device, False otherwise.
is_nested	
is_ort	
is_quantized	Is True if the Tensor is quantized, False otherwise.
is_sparse	Is True if the Tensor uses sparse COO storage layout, False otherwise.
is_sparse_csr	Is True if the Tensor uses sparse CSR storage layout, False otherwise.

continues on next page

Table 23 – continued from previous page

is_vulkan	
is_xla	Is True if the Tensor is stored on an XLA device, False otherwise.
is_xpu	Is True if the Tensor is stored on the XPU, False otherwise.
itemsize	Alias for element_size()
layout	
mH	Accessing this property is equivalent to calling adjoint().
mT	Returns a view of this tensor with the last two dimensions transposed.
name	
names	Stores names for each of this tensor's dimensions.
nbytes	Returns the number of bytes consumed by the "view" of elements of the Tensor if the Tensor does not use sparse storage layout.
ndim	Alias for dim()
output_nr	
real	Returns a new tensor containing real values of the self tensor for a complex-valued input tensor.
requires_grad	Is True if gradients need to be computed for this Tensor, False otherwise.
retains_grad	Is True if this Tensor is non-leaf and its grad is enabled to be populated during backward(), False otherwise.
shape	Returns the size of the self tensor.
volatile	

property is_leaf: bool

All Tensors that have requires_grad which is False will be leaf Tensors by convention.

For Tensors that have requires_grad which is True, they will be leaf Tensors if they were created by the user. This means that they are not the result of an operation and so grad_fn is None.

Only leaf Tensors will have their grad populated during a call to backward(). To get grad populated for non-leaf Tensors, you can use retain_grad().

Example:

```
>>> a = torch.rand(10, requires_grad=True)
>>> a.is_leaf
True
>>> b = torch.rand(10, requires_grad=True).cuda()
>>> b.is_leaf
False
# b was created by the operation that cast a cpu Tensor into a cuda Tensor
>>> c = torch.rand(10, requires_grad=True) + 2
>>> c.is_leaf
```

(continues on next page)

(continued from previous page)

```
False
# c was created by the addition operation
>>> d = torch.rand(10).cuda()
>>> d.is_leaf
True
# d does not require gradients and so has no operation creating it (that is␣
↪tracked by the autograd engine)
>>> e = torch.rand(10).cuda().requires_grad_()
>>> e.is_leaf
True
# e requires gradients and has no operations creating it
>>> f = torch.rand(10, requires_grad=True, device="cuda")
>>> f.is_leaf
True
# f requires grad, has no operation creating it
```

materialize(*shape*, *device=None*, *dtype=None*)

Create a Parameter with the same properties of the uninitialized one. Given a shape, it materializes a parameter in the same device and with the same *dtype* as the current one or the specified ones in the arguments.

Parameters

- **shape** (*Tuple[int, ...]*) – (tuple): the shape for the materialized tensor.
- **device** (`torch.device`) – the desired device of the parameters and buffers in this module. Optional.
- **dtype** (`torch.dtype`) – the desired floating point type of the floating point parameters and buffers in this module. Optional.

Return type
None

share_memory_()

Moves the underlying storage to shared memory.

This is a no-op if the underlying storage is already in shared memory and for CUDA tensors. Tensors in shared memory cannot be resized.

Return type
UninitializedParameter

pytorch_pfn_extras.nn.modules.lazy_linear

Classes

pytorch_pfn_extras.nn.modules.lazy_linear.LazyInitializationMixin(...)	A mixin for modules that lazily initialize buffers and parameters.
pytorch_pfn_extras.nn.modules.lazy_linear.LazyLinear(...)	Linear module with lazy weight initialization.
pytorch_pfn_extras.nn.modules.lazy_linear.UninitializedParameter([...])	

pytorch_pfn_extras.nn.modules.lazy_linear.LazyInitializationMixin

class pytorch_pfn_extras.nn.modules.lazy_linear.**LazyInitializationMixin**(**args*, ***kwargs*)

Bases: `object`

A mixin for modules that lazily initialize buffers and parameters.

Unlike regular modules, subclasses of this module can initialize buffers and parameters outside of the constructor (`__init__`). This allows you to, for example, initialize parameters in `forward` method to determine the shape of the weight based on the initial input.

Be sure to run "dummy" forward once to initialize all parameters that should be trained, before passing `module.parameters()` to an optimizer; otherwise weights initialized after `module.parameters()` (e.g., in `forward` function) will never be trained.

Note that lazy modules cannot validate if the shape is correct during deserialization. Also note that the initial weights may become different from the original (non-lazy) module even if the random seed is manually configured, as the order of initialization is different from the original one; especially, `module.cuda()` may cause the initialization to run on a GPU.

The default value of lazy buffers and parameters are `torch.Tensor([])` and `UninitializedParameter()`, respectively.

Methods

__init__(*args, **kwargs)	
state_dict(*args, **kwargs)	Returns a dictionary containing a whole state of the module.

Attributes

lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

__init__(**args*, ***kwargs*)

Parameters

- **self** (*Any*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

lazy_buffer_names: Tuple[str, ...] = ()

lazy_parameter_names: Tuple[str, ...] = ()

property lazy_parmeters_determined: bool

Returns if all lazy parameters are determined.

Subclasses can perform parameters initialization after all lazy parameters are determined. Note that this may be called during `__init__`.

state_dict(**args*, ***kwargs*)

Returns a dictionary containing a whole state of the module.

This function overrides the default behavior to exclude uninitialized parameter from serialization. This is needed because we need to discriminate lazy parameters (`UninitializedParameter()`) and initialized empty parameters (``torch.nn.Parameter(torch.Tensor())`) during deserialization.

See comments of `_lazy_load_hook` for details.

Parameters

- **self** (*Any*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Dict[str, *Any*]

pytorch_pfn_extras.nn.modules.lazy_linear.LazyLinear

class pytorch_pfn_extras.nn.modules.lazy_linear.**LazyLinear**(*in_features*, **args*, ***kwargs*)

Bases: *LazyInitializationMixin*, `Linear`

Linear module with lazy weight initialization.

When `in_features` is `None`, it is determined at the first time of the forward step.

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

Methods

__init__(in_features, *args, **kwargs)	Initializes internal Module state, shared by both nn.Module and ScriptModule.
add_module(name, module)	Adds a child module to the current module.
apply(fn)	Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self.
bfloat16()	Casts all floating point parameters and buffers to `bfloat16` datatype.
buffers([recurse])	Returns an iterator over module buffers.
children()	Returns an iterator over immediate children modules.
compile(*args, **kwargs)	Compile this Module's forward using `torch.compile()`.
cpu()	Moves all model parameters and buffers to the CPU.

continues on next page

Table 24 – continued from previous page

cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to double datatype.
eval()	Sets the module in evaluation mode.
extra_repr()	Set the extra representation of the module
float()	Casts all floating point parameters and buffers to float datatype.
forward(input)	Defines the computation performed at every call.
get_buffer(target)	Returns the buffer given by target if it exists, otherwise throws an error.
get_extra_state()	Returns any extra state to include in the module's state_dict.
get_parameter(target)	Returns the parameter given by target if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by target if it exists, otherwise throws an error.
half()	Casts all floating point parameters and buffers to half datatype.
ipu([device])	Moves all model parameters and buffers to the IPU.
load_state_dict(state_dict[, strict, assign])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse, ...])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse, ...])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook, *[, prepend, ...])	Registers a forward hook on the module.
register_forward_pre_hook(hook, *[, ...])	Registers a forward pre-hook on the module.
register_full_backward_hook(hook[, prepend])	Registers a backward hook on the module.
register_full_backward_pre_hook(hook[, prepend])	Registers a backward pre-hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
register_state_dict_pre_hook(hook)	These hooks will be called with arguments: self, prefix, and keep_vars before calling state_dict on self.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.

continues on next page

Table 24 – continued from previous page

reset_parameters()	
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict(*args, **kwargs)	Returns a dictionary containing a whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device[, recurse])	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Resets gradients of all model parameters.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
call_super_init	
dump_patches	
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

Parameters

- **in_features** (*Optional[int]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

__init__(*in_features*, **args*, ***kwargs*)

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

Parameters

- **in_features** (*Optional[int]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type

None

forward(*input*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Parameters
input (*Tensor*) –

Return type
Tensor

in_features: int

lazy_parameter_names: Tuple[str, ...] = ('weight',)

out_features: int

reset_parameters()

Return type
None

training: bool

weight: Tensor

pytorch_pfn_extras.nn.modules.lazy_linear.UninitializedParameter

class pytorch_pfn_extras.nn.modules.lazy_linear.**UninitializedParameter**(*data=None*, *requires_grad=True*)

Bases: `Parameter`

Methods

`__init__`()	

`abs`()	See `torch.abs()`
`abs_`()	In-place version of `abs()`
`absolute`()	Alias for `abs()`
`absolute_`()	In-place version of `absolute()` Alias for `abs_()`
`acos`()	See `torch.acos()`
`acos_`()	In-place version of `acos()`
`acosh`()	See `torch.acosh()`
`acosh_`()	In-place version of `acosh()`
`add`(other, *[, alpha])	Add a scalar or tensor to `self` tensor.
`add_`(other, *[, alpha])	In-place version of `add()`
`addbmm`(batch1, batch2, *[, beta, alpha])	See `torch.addbmm()`
`addbmm_`(batch1, batch2, *[, beta, alpha])	In-place version of `addbmm()`
`addcdiv`(tensor1, tensor2, *[, value])	See `torch.addcdiv()`
`addcdiv_`(tensor1, tensor2, *[, value])	In-place version of `addcdiv()`
`addcmul`(tensor1, tensor2, *[, value])	See `torch.addcmul()`

continues on next page

Table 25 – continued from previous page

addcmul_(tensor1, tensor2, *[, value])	In-place version of addcmul()
addmm(mat1, mat2, *[, beta, alpha])	See torch.addmm()
addmm_(mat1, mat2, *[, beta, alpha])	In-place version of addmm()
addmv(mat, vec, *[, beta, alpha])	See torch.addmv()
addmv_(mat, vec, *[, beta, alpha])	In-place version of addmv()
addr(vec1, vec2, *[, beta, alpha])	See torch.addr()
addr_(vec1, vec2, *[, beta, alpha])	In-place version of addr()
adjoint()	Alias for adjoint()
align_as(other)	Permutes the dimensions of the self tensor to match the dimension order in the other tensor, adding size-one dims for any new names.
align_to(*names)	Permutes the dimensions of the self tensor to match the order specified in names, adding size-one dims for any new names.
all([dim, keepdim])	See torch.all()
allclose(other[, rtol, atol, equal_nan])	See torch.allclose()
amax([dim, keepdim])	See torch.amax()
amin([dim, keepdim])	See torch.amin()
aminmax(*[, dim, keepdim])	See torch.aminmax()
angle()	See torch.angle()
any([dim, keepdim])	See torch.any()
apply_(callable)	Applies the function callable to each element in the tensor, replacing each element with the value returned by callable.
arccos()	See torch.arccos()
arccos_()	In-place version of arccos()
arccosh	acosh() -> Tensor
arccosh_	acosh_() -> Tensor
arcsin()	See torch.arcsin()
arcsin_()	In-place version of arcsin()
arcsinh()	See torch.arcsinh()
arcsinh_()	In-place version of arcsinh()
arctan()	See torch.arctan()
arctan2(other)	See torch.arctan2()
arctan2_	atan2_(other) -> Tensor
arctan_()	In-place version of arctan()
arctanh()	See torch.arctanh()
arctanh_(other)	In-place version of arctanh()
argmax([dim, keepdim])	See torch.argmax()
argmin([dim, keepdim])	See torch.argmin()
argsort([dim, descending])	See torch.argsort()
argwhere()	See torch.argwhere()
as_strided(size, stride[, storage_offset])	See torch.as_strided()
as_strided_(size, stride[, storage_offset])	In-place version of as_strided()
as_strided_scatter(src, size, stride[, ...])	See torch.as_strided_scatter()
as_subclass(cls)	Makes a cls instance with the same data pointer as self.
asin()	See torch.asin()
asin_()	In-place version of asin()
asinh()	See torch.asinh()

continues on next page

Table 25 – continued from previous page

asinh_()	In-place version of asinh()
atan()	See torch.atan()
atan2(other)	See torch.atan2()
atan2_(other)	In-place version of atan2()
atan_()	In-place version of atan()
atanh()	See torch.atanh()
atanh_(other)	In-place version of atanh()
backward([gradient, retain_graph, ...])	Computes the gradient of current tensor wrt graph leaves.
baddbmm(batch1, batch2, *[, beta, alpha])	See torch.baddbmm()
baddbmm_(batch1, batch2, *[, beta, alpha])	In-place version of baddbmm()
bernoulli(*[, generator])	Returns a result tensor where each result[i] is independently sampled from Bernoulli(self[i]).
bernoulli_([p, generator])	Fills each location of self with an independent sample from Bernoulli(p).
bfloat16([memory_format])	self.bfloat16() is equivalent to self.to(torch.bfloat16).
bincount([weights, minlength])	See torch.bincount()
bitwise_and()	See torch.bitwise_and()
bitwise_and_()	In-place version of bitwise_and()
bitwise_left_shift(other)	See torch.bitwise_left_shift()
bitwise_left_shift_(other)	In-place version of bitwise_left_shift()
bitwise_not()	See torch.bitwise_not()
bitwise_not_()	In-place version of bitwise_not()
bitwise_or()	See torch.bitwise_or()
bitwise_or_()	In-place version of bitwise_or()
bitwise_right_shift(other)	See torch.bitwise_right_shift()
bitwise_right_shift_(other)	In-place version of bitwise_right_shift()
bitwise_xor()	See torch.bitwise_xor()
bitwise_xor_()	In-place version of bitwise_xor()
bmm(batch2)	See torch.bmm()
bool([memory_format])	self.bool() is equivalent to self.to(torch.bool).
broadcast_to(shape)	See torch.broadcast_to().
byte([memory_format])	self.byte() is equivalent to self.to(torch.uint8).
cauchy_([median, sigma, generator])	Fills the tensor with numbers drawn from the Cauchy distribution:
ccol_indices	
cdouble([memory_format])	self.cdouble() is equivalent to self.to(torch.complex128).
ceil()	See torch.ceil()
ceil_()	In-place version of ceil()
cfloat([memory_format])	self.cfloat() is equivalent to self.to(torch.complex64).
chalf([memory_format])	self.chalf() is equivalent to self.to(torch.complex32).
char([memory_format])	self.char() is equivalent to self.to(torch.int8).
cholesky([upper])	See torch.cholesky()

continues on next page

Table 25 – continued from previous page

cholesky_inverse([upper])	See torch.cholesky_inverse()
cholesky_solve(input2[, upper])	See torch.cholesky_solve()
chunk(chunks[, dim])	See torch.chunk()
clamp([min, max])	See torch.clamp()
clamp_([min, max])	In-place version of clamp()
clamp_max	
clamp_max_	
clamp_min	
clamp_min_	
clip([min, max])	Alias for clamp().
clip_([min, max])	Alias for clamp_().
clone(*[, memory_format])	See torch.clone()
coalesce()	Returns a coalesced copy of self if self is an uncoalesced tensor.
col_indices()	Returns the tensor containing the column indices of the self tensor when self is a sparse CSR tensor of layout sparse_csr.
conj()	See torch.conj()
conj_physical()	See torch.conj_physical()
conj_physical_()	In-place version of conj_physical()
contiguous([memory_format])	Returns a contiguous in memory tensor containing the same data as self tensor.
copy_(src[, non_blocking])	Copies the elements from src into self tensor and returns self.
copysign(other)	See torch.copysign()
copysign_(other)	In-place version of copysign()
corrcoef()	See torch.corrcoef()
cos()	See torch.cos()
cos_()	In-place version of cos()
cosh()	See torch.cosh()
cosh_()	In-place version of cosh()
count_nonzero([dim])	See torch.count_nonzero()
cov(*[, correction, fweights, aweights])	See torch.cov()
cpu([memory_format])	Returns a copy of this object in CPU memory.
cross(other[, dim])	See torch.cross()
crow_indices()	Returns the tensor containing the compressed row indices of the self tensor when self is a sparse CSR tensor of layout sparse_csr.
cuda([device, non_blocking, memory_format])	Returns a copy of this object in CUDA memory.
cummax(dim)	See torch.cummax()
cummin(dim)	See torch.cummin()
cumprod(dim[, dtype])	See torch.cumprod()
cumprod_(dim[, dtype])	In-place version of cumprod()
cumsum(dim[, dtype])	See torch.cumsum()
cumsum_(dim[, dtype])	In-place version of cumsum()
data_ptr()	Returns the address of the first element of self tensor.

continues on next page

Table 25 – continued from previous page

deg2rad()	See torch.deg2rad()
deg2rad_()	In-place version of deg2rad()
dense_dim()	Return the number of dense dimensions in a sparse tensor self.
dequantize()	Given a quantized Tensor, dequantize it and return the dequantized float Tensor.
det()	See torch.det()
detach	Returns a new Tensor, detached from the current graph.
detach_	Detaches the Tensor from the graph that created it, making it a leaf.
diag([diagonal])	See torch.diag()
diag_embed([offset, dim1, dim2])	See torch.diag_embed()
diagflat([offset])	See torch.diagflat()
diagonal([offset, dim1, dim2])	See torch.diagonal()
diagonal_scatter(src[, offset, dim1, dim2])	See torch.diagonal_scatter()
diff([n, dim, prepend, append])	See torch.diff()
digamma()	See torch.digamma()
digamma_()	In-place version of digamma()
dim()	Returns the number of dimensions of self tensor.
dim_order()	Returns a tuple of int describing the dim order or physical layout of self.
dist(other[, p])	See torch.dist()
div(value, *[, rounding_mode])	See torch.div()
div_(value, *[, rounding_mode])	In-place version of div()
divide(value, *[, rounding_mode])	See torch.divide()
divide_(value, *[, rounding_mode])	In-place version of divide()
dot(other)	See torch.dot()
double([memory_format])	self.double() is equivalent to self.to(torch.float64).
dsplit(split_size_or_sections)	See torch.dsplit()
eig([eigenvectors])	
element_size()	Returns the size in bytes of an individual element.
eq(other)	See torch.eq()
eq_(other)	In-place version of eq()
equal(other)	See torch.equal()
erf()	See torch.erf()
erf_()	In-place version of erf()
erfc()	See torch.erfc()
erfc_()	In-place version of erfc()
erfinv()	See torch.erfinv()
erfinv_()	In-place version of erfinv()
exp()	See torch.exp()
exp2()	See torch.exp2()
exp2_()	In-place version of exp2()
exp_()	In-place version of exp()
expand(*sizes)	Returns a new view of the self tensor with singleton dimensions expanded to a larger size.
expand_as(other)	Expand this tensor to the same size as other.
expm1()	See torch.expm1()

continues on next page

Table 25 – continued from previous page

expm1_()	In-place version of expm1()
exponential_([lambd, generator])	Fills self tensor with elements drawn from the exponential distribution:
fill_(value)	Fills self tensor with the specified value.
fill_diagonal_(fill_value[, wrap])	Fill the main diagonal of a tensor that has at least 2-dimensions.
fix()	See torch.fix().
fix_()	In-place version of fix()
flatten([start_dim, end_dim])	See torch.flatten()
flip(dims)	See torch.flip()
fliplr()	See torch.fliplr()
flipud()	See torch.flipud()
float([memory_format])	self.float() is equivalent to self.to(torch.float32).
float_power(exponent)	See torch.float_power()
float_power_(exponent)	In-place version of float_power()
floor()	See torch.floor()
floor_()	In-place version of floor()
floor_divide(value)	See torch.floor_divide()
floor_divide_(value)	In-place version of floor_divide()
fmax(other)	See torch.fmax()
fmin(other)	See torch.fmin()
fmod(divisor)	See torch.fmod()
fmod_(divisor)	In-place version of fmod()
frac()	See torch.frac()
frac_()	In-place version of frac()
frexp(input)	See torch.frexp()
gather(dim, index)	See torch.gather()
gcd(other)	See torch.gcd()
gcd_(other)	In-place version of gcd()
ge(other)	See torch.ge().
ge_(other)	In-place version of ge().
geometric_(p, *[, generator])	Fills self tensor with elements drawn from the geometric distribution:
geqrf()	See torch.geqrf()
ger(vec2)	See torch.ger()
get_device()	For CUDA tensors, this function returns the device ordinal of the GPU on which the tensor resides.
greater(other)	See torch.greater().
greater_(other)	In-place version of greater().
greater_equal(other)	See torch.greater_equal().
greater_equal_(other)	In-place version of greater_equal().
gt(other)	See torch.gt().
gt_(other)	In-place version of gt().
half([memory_format])	self.half() is equivalent to self.to(torch.float16).
hardshrink([lambd])	See torch.nn.functional.hardshrink()
has_names	Is True if any of this tensor's dimensions are named.
heaviside(values)	See torch.heaviside()
heaviside_(values)	In-place version of heaviside()

continues on next page

Table 25 – continued from previous page

histc([bins, min, max])	See torch.histc()
histogram(input, bins, *[, range, weight, ...])	See torch.histogram()
hsplit(split_size_or_sections)	See torch.hsplit()
hypot(other)	See torch.hypot()
hypot_(other)	In-place version of hypot()
i0()	See torch.i0()
i0_()	In-place version of i0()
igamma(other)	See torch.igamma()
igamma_(other)	In-place version of igamma()
igammac(other)	See torch.igammac()
igammac_(other)	In-place version of igammac()
index_add(dim, index, source, *[, alpha])	Out-of-place version of torch.Tensor.index_add_().
index_add_(dim, index, source, *[, alpha])	Accumulate the elements of alpha times source into the self tensor by adding to the indices in the order given in index.
index_copy(dim, index, tensor2)	Out-of-place version of torch.Tensor.index_copy_().
index_copy_(dim, index, tensor)	Copies the elements of tensor into the self tensor by selecting the indices in the order given in index.
index_fill(dim, index, value)	Out-of-place version of torch.Tensor.index_fill_().
index_fill_(dim, index, value)	Fills the elements of the self tensor with value value by selecting the indices in the order given in index.
index_put(indices, values[, accumulate])	Out-place version of index_put_().
index_put_(indices, values[, accumulate])	Puts values from the tensor values into the tensor self using the indices specified in indices (which is a tuple of Tensors).
index_reduce	
index_reduce_(dim, index, source, reduce, *)	Accumulate the elements of source into the self tensor by accumulating to the indices in the order given in index using the reduction given by the reduce argument.
index_select(dim, index)	See torch.index_select()
indices()	Return the indices tensor of a sparse COO tensor.
inner(other)	See torch.inner().
int([memory_format])	self.int() is equivalent to self.to(torch.int32).
int_repr()	Given a quantized Tensor, self.int_repr() returns a CPU Tensor with uint8_t as data type that stores the underlying uint8_t values of the given Tensor.
inverse()	See torch.inverse()
ipu([device, non_blocking, memory_format])	Returns a copy of this object in IPU memory.
is_coalesced()	Returns True if self is a sparse COO tensor that is coalesced, False otherwise.
is_complex()	Returns True if the data type of self is a complex data type.
is_conj()	Returns True if the conjugate bit of self is set to true.

continues on next page

Table 25 – continued from previous page

is_contiguous([memory_format])	Returns True if self tensor is contiguous in memory in the order specified by memory format.
is_distributed	
is_floating_point()	Returns True if the data type of self is a floating point data type.
is_inference()	See torch.is_inference()
is_neg()	Returns True if the negative bit of self is set to true.
is_nonzero	
is_pinned	Returns true if this tensor resides in pinned memory.
is_same_size	
is_set_to(tensor)	Returns True if both tensors are pointing to the exact same memory (same storage, offset, size and stride).
is_shared()	Checks if tensor is in shared memory.
is_signed()	Returns True if the data type of self is a signed data type.
isclose(other[, rtol, atol, equal_nan])	See torch.isclose()
isfinite()	See torch.isfinite()
isinf()	See torch.isinf()
isnan()	See torch.isnan()
isneginf()	See torch.isneginf()
isposinf()	See torch.isposinf()
isreal()	See torch.isreal()
istft(n_fft[, hop_length, win_length, ...])	See torch.istft()
item()	Returns the value of this tensor as a standard Python number.
kron(other)	See torch.kron()
kthvalue(k[, dim, keepdim])	See torch.kthvalue()
lcm(other)	See torch.lcm()
lcm_(other)	In-place version of lcm()
ldexp(other)	See torch.ldexp()
ldexp_(other)	In-place version of ldexp()
le(other)	See torch.le().
le_(other)	In-place version of le().
lerp(end, weight)	See torch.lerp()
lerp_(end, weight)	In-place version of lerp()
less	lt(other) -> Tensor
less_(other)	In-place version of less().
less_equal(other)	See torch.less_equal().
less_equal_(other)	In-place version of less_equal().
lgamma()	See torch.lgamma()
lgamma_()	In-place version of lgamma()
log()	See torch.log()
log10()	See torch.log10()
log10_()	In-place version of log10()
log1p()	See torch.log1p()
log1p_()	In-place version of log1p()
log2()	See torch.log2()
log2_()	In-place version of log2()

continues on next page

Table 25 – continued from previous page

log_()	In-place version of log()
log_normal_([mean, std, generator])	Fills self tensor with numbers samples from the log-normal distribution parameterized by the given mean μ and standard deviation σ.
log_softmax	
logaddexp(other)	See torch.logaddexp()
logaddexp2(other)	See torch.logaddexp2()
logcumsumexp(dim)	See torch.logcumsumexp()
logdet()	See torch.logdet()
logical_and()	See torch.logical_and()
logical_and_()	In-place version of logical_and()
logical_not()	See torch.logical_not()
logical_not_()	In-place version of logical_not()
logical_or()	See torch.logical_or()
logical_or_()	In-place version of logical_or()
logical_xor()	See torch.logical_xor()
logical_xor_()	In-place version of logical_xor()
logit()	See torch.logit()
logit_()	In-place version of logit()
logsumexp(dim[, keepdim])	See torch.logsumexp()
long([memory_format])	self.long() is equivalent to self.to(torch.int64).
lstsq(other)	
lt(other)	See torch.lt().
lt_(other)	In-place version of lt().
lu([pivot, get_infos])	See torch.lu()
lu_solve(LU_data, LU_pivots)	See torch.lu_solve()
map2_	
map_(tensor, callable)	Applies callable for each element in self tensor and the given tensor and stores the results in self tensor.
masked_fill(mask, value)	Out-of-place version of torch.Tensor.masked_fill_()
masked_fill_(mask, value)	Fills elements of self tensor with value where mask is True.
masked_scatter(mask, tensor)	Out-of-place version of torch.Tensor.masked_scatter_()
masked_scatter_(mask, source)	Copies elements from source into self tensor at positions where the mask is True.
masked_select(mask)	See torch.masked_select()
materialize(shape[, device, dtype])	Create a Parameter with the same properties of the uninitialized one.
matmul(tensor2)	See torch.matmul()
matrix_exp()	See torch.matrix_exp()

continues on next page

Table 25 – continued from previous page

matrix_power(n)	**Note:** matrix_power() is deprecated, use torch.linalg.matrix_power() instead.
max([dim, keepdim])	See torch.max()
maximum(other)	See torch.maximum()
mean([dim, keepdim, dtype])	See torch.mean()
median([dim, keepdim])	See torch.median()
min([dim, keepdim])	See torch.min()
minimum(other)	See torch.minimum()
mm(mat2)	See torch.mm()
mode([dim, keepdim])	See torch.mode()
moveaxis(source, destination)	See torch.moveaxis()
movedim(source, destination)	See torch.movedim()
msort()	See torch.msort()
mul(value)	See torch.mul().
mul_(value)	In-place version of mul().
multinomial(num_samples[, replacement, ...])	See torch.multinomial()
multiply(value)	See torch.multiply().
multiply_(value)	In-place version of multiply().
mv(vec)	See torch.mv()
mvlgamma(p)	See torch.mvlgamma()
mvlgamma_(p)	In-place version of mvlgamma()
nan_to_num([nan, posinf, neginf])	See torch.nan_to_num().
nan_to_num_([nan, posinf, neginf])	In-place version of nan_to_num().
nanmean([dim, keepdim, dtype])	See torch.nanmean()
nanmedian([dim, keepdim])	See torch.nanmedian()
nanquantile(q[, dim, keepdim, interpolation])	See torch.nanquantile()
nansum([dim, keepdim, dtype])	See torch.nansum()
narrow(dimension, start, length)	See torch.narrow().
narrow_copy(dimension, start, length)	See torch.narrow_copy().
ndimension()	Alias for dim()
ne(other)	See torch.ne().
ne_(other)	In-place version of ne().
neg()	See torch.neg()
neg_()	In-place version of neg()
negative()	See torch.negative()
negative_()	In-place version of negative()
nelement()	Alias for numel()
new	
new_empty(size, *[, dtype, device, ...])	Returns a Tensor of size size filled with uninitialized data.
new_empty_strided(size, stride[, dtype, ...])	Returns a Tensor of size size and strides stride filled with uninitialized data.
new_full(size, fill_value, *[, dtype, ...])	Returns a Tensor of size size filled with fill_value.
new_ones(size, *[, dtype, device, ...])	Returns a Tensor of size size filled with 1.
new_tensor(data, *[, dtype, device, ...])	Returns a new Tensor with data as the tensor data.

continues on next page

Table 25 – continued from previous page

new_zeros(size, *[, dtype, device, ...])	Returns a Tensor of size size filled with 0.
nextafter(other)	See torch.nextafter()
nextafter_(other)	In-place version of nextafter()
nonzero()	See torch.nonzero()
nonzero_static(input, *, size[, fill_value])	Returns a 2-D tensor where each row is the index for a non-zero value.
norm([p, dim, keepdim, dtype])	See torch.norm()
normal_([mean, std, generator])	Fills self tensor with elements samples from the normal distribution parameterized by mean and std.
not_equal(other)	See torch.not_equal().
not_equal_(other)	In-place version of not_equal().
numel()	See torch.numel()
numpy(*[, force])	Returns the tensor as a NumPy ndarray.
orgqr(input2)	See torch.orgqr()
ormqr(input2, input3[, left, transpose])	See torch.ormqr()
outer(vec2)	See torch.outer().
permute(*dims)	See torch.permute()
pin_memory()	Copies the tensor to pinned memory, if it's not already pinned.
pinverse()	See torch.pinverse()
polygamma(n)	See torch.polygamma()
polygamma_(n)	In-place version of polygamma()
positive()	See torch.positive()
pow(exponent)	See torch.pow()
pow_(exponent)	In-place version of pow()
prelu	
prod([dim, keepdim, dtype])	See torch.prod()
put(input, index, source[, accumulate])	Out-of-place version of torch.Tensor.put_().
put_(index, source[, accumulate])	Copies the elements from source into the positions specified by index.
q_per_channel_axis()	Given a Tensor quantized by linear (affine) per-channel quantization, returns the index of dimension on which per-channel quantization is applied.
q_per_channel_scales()	Given a Tensor quantized by linear (affine) per-channel quantization, returns a Tensor of scales of the underlying quantizer.
q_per_channel_zero_points()	Given a Tensor quantized by linear (affine) per-channel quantization, returns a tensor of zero_points of the underlying quantizer.
q_scale()	Given a Tensor quantized by linear(affine) quantization, returns the scale of the underlying quantizer().
q_zero_point()	Given a Tensor quantized by linear(affine) quantization, returns the zero_point of the underlying quantizer().
qr([some])	See torch.qr()
qscheme()	Returns the quantization scheme of a given QTensor.
quantile(q[, dim, keepdim, interpolation])	See torch.quantile()
rad2deg()	See torch.rad2deg()
rad2deg_()	In-place version of rad2deg()

continues on next page

Table 25 – continued from previous page

random_([from, to, generator])	Fills self tensor with numbers sampled from the discrete uniform distribution over [from, to - 1].
ravel()	see torch.ravel()
reciprocal()	See torch.reciprocal()
reciprocal_()	In-place version of reciprocal()
record_stream(stream)	Ensures that the tensor memory is not reused for another tensor until all current work queued on stream are complete.
refine_names(*names)	Refines the dimension names of self according to names.
register_hook(hook)	Registers a backward hook.
register_post_accumulate_grad_hook(hook)	Registers a backward hook that runs after grad accumulation.
reinforce(reward)	
relu	
relu_	
remainder(divisor)	See torch.remainder()
remainder_(divisor)	In-place version of remainder()
rename(*names, **rename_map)	Renames dimension names of self.
rename_(*names, **rename_map)	In-place version of rename().
renorm(p, dim, maxnorm)	See torch.renorm()
renorm_(p, dim, maxnorm)	In-place version of renorm()
repeat(*sizes)	Repeats this tensor along the specified dimensions.
repeat_interleave(repeats[, dim, output_size])	See torch.repeat_interleave().
requires_grad_([requires_grad])	Change if autograd should record operations on this tensor: sets this tensor's requires_grad attribute in-place.
reshape(*shape)	Returns a tensor with the same data and number of elements as self but with the specified shape.
reshape_as(other)	Returns this tensor as the same shape as other.
resize(*sizes)	
resize_(*sizes[, memory_format])	Resizes self tensor to the specified size.
resize_as(tensor)	
resize_as_(tensor[, memory_format])	Resizes the self tensor to be the same size as the specified tensor.
resize_as_sparse_	
resolve_conj()	See torch.resolve_conj()
resolve_neg()	See torch.resolve_neg()
retain_grad()	Enables this Tensor to have their grad populated during backward().
roll(shifts, dims)	See torch.roll()
rot90(k, dims)	See torch.rot90()
round([decimals])	See torch.round()
round_([decimals])	In-place version of round()

continues on next page

Table 25 – continued from previous page

row_indices	
rsqrt()	See torch.rsqrt()
rsqrt_()	In-place version of rsqrt()
scatter(dim, index, src)	Out-of-place version of torch.Tensor.scatter_()
scatter_(dim, index, src[, reduce])	Writes all values from the tensor src into self at the indices specified in the index tensor.
scatter_add(dim, index, src)	Out-of-place version of torch.Tensor.scatter_add_()
scatter_add_(dim, index, src)	Adds all values from the tensor src into self at the indices specified in the index tensor in a similar fashion as scatter_().
scatter_reduce(dim, index, src, reduce, *[, ...])	Out-of-place version of torch.Tensor.scatter_reduce_()
scatter_reduce_(dim, index, src, reduce, *)	Reduces all values from the src tensor to the indices specified in the index tensor in the self tensor using the applied reduction defined via the reduce argument ("sum", "prod", "mean", "amax", "amin").
select(dim, index)	See torch.select()
select_scatter(src, dim, index)	See torch.select_scatter()
set_([source, storage_offset, size, stride])	Sets the underlying storage, size, and strides.
sgn()	See torch.sgn()
sgn_()	In-place version of sgn()
*share_memory_*()	Moves the underlying storage to shared memory.
short([memory_format])	self.short() is equivalent to self.to(torch.int16).
sigmoid()	See torch.sigmoid()
sigmoid_()	In-place version of sigmoid()
sign()	See torch.sign()
sign_()	In-place version of sign()
signbit()	See torch.signbit()
sin()	See torch.sin()
sin_()	In-place version of sin()
sinc()	See torch.sinc()
sinc_()	In-place version of sinc()
sinh()	See torch.sinh()
sinh_()	In-place version of sinh()
size([dim])	Returns the size of the self tensor.
slice_scatter(src[, dim, start, end, step])	See torch.slice_scatter()
slogdet()	See torch.slogdet()
smm(mat)	See torch.smm()
softmax(dim)	Alias for torch.nn.functional.softmax().
solve(other)	
sort([dim, descending])	See torch.sort()
sparse_dim()	Return the number of sparse dimensions in a sparse tensor self.
sparse_mask(mask)	Returns a new sparse tensor with values from a strided tensor self filtered by the indices of the sparse tensor mask.

continues on next page

Table 25 – continued from previous page

sparse_resize_(size, sparse_dim, dense_dim)	Resizes self sparse tensor to the desired size and the number of sparse and dense dimensions.
sparse_resize_and_clear_(size, sparse_dim, ...)	Removes all specified elements from a sparse tensor self and resizes self to the desired size and the number of sparse and dense dimensions.
split(split_size[, dim])	See torch.split()
split_with_sizes	
sqrt()	See torch.sqrt()
sqrt_()	In-place version of sqrt()
square()	See torch.square()
square_()	In-place version of square()
squeeze([dim])	See torch.squeeze()
squeeze_([dim])	In-place version of squeeze()
sspaddmm(mat1, mat2, *[, beta, alpha])	See torch.sspaddmm()
std([dim, correction, keepdim])	See torch.std()
stft(n_fft[, hop_length, win_length, ...])	See torch.stft()
storage()	Returns the underlying TypedStorage.
storage_offset()	Returns self tensor's offset in the underlying storage in terms of number of storage elements (not bytes).
storage_type()	Returns the type of the underlying storage.
stride(dim)	Returns the stride of self tensor.
sub(other, *[, alpha])	See torch.sub().
sub_(other, *[, alpha])	In-place version of sub()
subtract(other, *[, alpha])	See torch.subtract().
subtract_(other, *[, alpha])	In-place version of subtract().
sum([dim, keepdim, dtype])	See torch.sum()
sum_to_size(*size)	Sum this tensor to size.
svd([some, compute_uv])	See torch.svd()
swapaxes(axis0, axis1)	See torch.swapaxes()
swapaxes_(axis0, axis1)	In-place version of swapaxes()
swapdims(dim0, dim1)	See torch.swapdims()
swapdims_(dim0, dim1)	In-place version of swapdims()
symeig([eigenvectors])	
t()	See torch.t()
t_()	In-place version of t()
take(indices)	See torch.take()
take_along_dim(indices, dim)	See torch.take_along_dim()
tan()	See torch.tan()
tan_()	In-place version of tan()
tanh()	See torch.tanh()
tanh_()	In-place version of tanh()
tensor_split(indices_or_sections[, dim])	See torch.tensor_split()
tile(dims)	See torch.tile()
to(*args, **kwargs)	Performs Tensor dtype and/or device conversion.
to_dense([dtype, masked_grad])	Creates a strided copy of self if self is not a strided tensor, otherwise returns self.
to_mkldnn()	Returns a copy of the tensor in torch.mkldnn layout.
to_padded_tensor(padding[, output_size])	See to_padded_tensor()

continues on next page

Table 25 – continued from previous page

to_sparse(sparseDims)	Returns a sparse copy of the tensor.
to_sparse_bsc(blocksize, dense_dim)	Convert a tensor to a block sparse column (BSC) storage format of given blocksize.
to_sparse_bsr(blocksize, dense_dim)	Convert a tensor to a block sparse row (BSR) storage format of given blocksize.
to_sparse_coo()	Convert a tensor to coordinate format.
to_sparse_csc()	Convert a tensor to compressed column storage (CSC) format.
to_sparse_csr([dense_dim])	Convert a tensor to compressed row storage format (CSR).
tolist()	Returns the tensor as a (nested) list.
topk(k[, dim, largest, sorted])	See torch.topk()
trace()	See torch.trace()
transpose(dim0, dim1)	See torch.transpose()
transpose_(dim0, dim1)	In-place version of transpose()
triangular_solve(A[, upper, transpose, ...])	See torch.triangular_solve()
tril([diagonal])	See torch.tril()
tril_([diagonal])	In-place version of tril()
triu([diagonal])	See torch.triu()
triu_([diagonal])	In-place version of triu()
true_divide(value)	See torch.true_divide()
true_divide_(value)	In-place version of true_divide_()
trunc()	See torch.trunc()
trunc_()	In-place version of trunc()
type([dtype, non_blocking])	Returns the type if *dtype* is not provided, else casts this object to the specified type.
type_as(tensor)	Returns this tensor cast to the type of the given tensor.
unbind([dim])	See torch.unbind()
unflatten(dim, sizes)	See torch.unflatten().
unfold(dimension, size, step)	Returns a view of the original tensor which contains all slices of size size from self tensor in the dimension dimension.
uniform_([from, to, generator])	Fills self tensor with numbers sampled from the continuous uniform distribution:
unique([sorted, return_inverse, ...])	Returns the unique elements of the input tensor.
unique_consecutive([return_inverse, ...])	Eliminates all but the first element from every consecutive group of equivalent elements.
unsafe_chunk(chunks[, dim])	See torch.unsafe_chunk()
unsafe_split(split_size[, dim])	See torch.unsafe_split()
unsafe_split_with_sizes	
unsqueeze(dim)	See torch.unsqueeze()
unsqueeze_(dim)	In-place version of unsqueeze()
untyped_storage()	Returns the underlying UntypedStorage.
values()	Return the values tensor of a sparse COO tensor.
var([dim, correction, keepdim])	See torch.var()
vdot(other)	See torch.vdot()
view(*shape)	Returns a new tensor with the same data as the self tensor but of a different shape.
view_as(other)	View this tensor as the same size as other.
vsplit(split_size_or_sections)	See torch.vsplit()

continues on next page

Table 25 – continued from previous page

where(condition, y)	self.where(condition, y) is equivalent to torch.where(condition, self, y).
xlogy(other)	See torch.xlogy()
xlogy_(other)	In-place version of xlogy()
xpu([device, non_blocking, memory_format])	Returns a copy of this object in XPU memory.
zero_()	Fills self tensor with zeros.

Attributes

H	Returns a view of a matrix (2-D tensor) conjugated and transposed.
T	Returns a view of this tensor with its dimensions reversed.
data	
device	Is the torch.device where this Tensor is.
dtype	
grad	This attribute is None by default and becomes a Tensor the first time a call to backward() computes gradients for self.
grad_fn	
imag	Returns a new tensor containing imaginary values of the self tensor.
is_cpu	Is True if the Tensor is stored on the CPU, False otherwise.
is_cuda	Is True if the Tensor is stored on the GPU, False otherwise.
is_ipu	Is True if the Tensor is stored on the IPU, False otherwise.
is_leaf	All Tensors that have requires_grad which is False will be leaf Tensors by convention.
is_meta	Is True if the Tensor is a meta tensor, False otherwise.
is_mkldnn	
is_mps	Is True if the Tensor is stored on the MPS device, False otherwise.
is_nested	
is_ort	
is_quantized	Is True if the Tensor is quantized, False otherwise.
is_sparse	Is True if the Tensor uses sparse COO storage layout, False otherwise.
is_sparse_csr	Is True if the Tensor uses sparse CSR storage layout, False otherwise.

continues on next page

Table 26 – continued from previous page

is_vulkan	
is_xla	Is True if the Tensor is stored on an XLA device, False otherwise.
is_xpu	Is True if the Tensor is stored on the XPU, False otherwise.
itemsize	Alias for element_size()
layout	
mH	Accessing this property is equivalent to calling adjoint().
mT	Returns a view of this tensor with the last two dimensions transposed.
name	
names	Stores names for each of this tensor's dimensions.
nbytes	Returns the number of bytes consumed by the "view" of elements of the Tensor if the Tensor does not use sparse storage layout.
ndim	Alias for dim()
output_nr	
real	Returns a new tensor containing real values of the self tensor for a complex-valued input tensor.
requires_grad	Is True if gradients need to be computed for this Tensor, False otherwise.
retains_grad	Is True if this Tensor is non-leaf and its grad is enabled to be populated during backward(), False otherwise.
shape	Returns the size of the self tensor.
volatile	

property is_leaf: bool

All Tensors that have requires_grad which is False will be leaf Tensors by convention.

For Tensors that have requires_grad which is True, they will be leaf Tensors if they were created by the user. This means that they are not the result of an operation and so grad_fn is None.

Only leaf Tensors will have their grad populated during a call to backward(). To get grad populated for non-leaf Tensors, you can use retain_grad().

Example:

```
>>> a = torch.rand(10, requires_grad=True)
>>> a.is_leaf
True
>>> b = torch.rand(10, requires_grad=True).cuda()
>>> b.is_leaf
False
# b was created by the operation that cast a cpu Tensor into a cuda Tensor
>>> c = torch.rand(10, requires_grad=True) + 2
>>> c.is_leaf
```

(continues on next page)

(continued from previous page)

```
False
# c was created by the addition operation
>>> d = torch.rand(10).cuda()
>>> d.is_leaf
True
# d does not require gradients and so has no operation creating it (that is␣
↪tracked by the autograd engine)
>>> e = torch.rand(10).cuda().requires_grad_()
>>> e.is_leaf
True
# e requires gradients and has no operations creating it
>>> f = torch.rand(10, requires_grad=True, device="cuda")
>>> f.is_leaf
True
# f requires grad, has no operation creating it
```

materialize(*shape*, *device=None*, *dtype=None*)

Create a Parameter with the same properties of the uninitialized one. Given a shape, it materializes a parameter in the same device and with the same *dtype* as the current one or the specified ones in the arguments.

Parameters

- **shape** (*Tuple[int, ...]*) – (tuple): the shape for the materialized tensor.
- **device** (`torch.device`) – the desired device of the parameters and buffers in this module. Optional.
- **dtype** (`torch.dtype`) – the desired floating point type of the floating point parameters and buffers in this module. Optional.

Return type
None

share_memory_()

Moves the underlying storage to shared memory.

This is a no-op if the underlying storage is already in shared memory and for CUDA tensors. Tensors in shared memory cannot be resized.

Return type
UninitializedParameter

pytorch_pfn_extras.nn.parallel

Classes

pytorch_pfn_extras.nn.parallel.DistributedDataParallel(module)	Module for distributed data parallelism

pytorch_pfn_extras.nn.parallel.DistributedDataParallel

class pytorch_pfn_extras.nn.parallel.**DistributedDataParallel**(*module*, *broadcast_buffers=True*, *negotiate_grads=True*, *process_group=None*, *reduce_function=None*, *broadcast_function=None*, ***kwargs*)

Bases: `Module`

Module for distributed data parallelism

This class synchronizes the gradients and the buffers after backward computations.

Parameters

- **module** (*Module*) – torch.nn.Module object to be trained
- **broadcast_buffers** (*bool*) – Boolean flag to broadcast buffers after backward computations. Broadcasting buffers may be helpful when the module includes BatchNormalization. However, it will degrade training throughput. (default: *True*)
- **negotiate_grads** (*bool*) – Boolean flag to choose gradients to be sent before all-reduce. This flag is necessary when the computation graph of the module is dynamic. (default: *True*)
- **process_group** (*Optional[ProcessGroup]*) – Process group used for broadcasting and reducing. (default: *torch.distributed.group.WORLD*)
- **reduce_function** (*Optional[Callable[[Sequence[Tensor], Optional[ProcessGroup]], None]]*) – All-reduce function
- **broadcast_function** (*Optional[Callable[[Sequence[Tensor], Optional[ProcessGroup]], None]]*) – Broadcast function
- **kwargs** (*Any*) –

This module receives keyword arguments for the compatibility with *torch.nn.parallel.DistributedDataParallel*. It shows a warning when setting the ignored arguments.

Methods

`__init__`(module[, broadcast_buffers, ...])	This module receives keyword arguments for the compatibility with *torch.nn.parallel.DistributedDataParallel*.
`add_module`(name, module)	Adds a child module to the current module.
`apply`(fn)	Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self.
`bfloat16`()	Casts all floating point parameters and buffers to `bfloat16` datatype.
`buffers`([recurse])	Returns an iterator over module buffers.
`children`()	Returns an iterator over immediate children modules.
`compile`(*args, **kwargs)	Compile this Module's forward using `torch.compile()`.
`cpu`()	Moves all model parameters and buffers to the CPU.
`cuda`([device])	Moves all model parameters and buffers to the GPU.

continues on next page

Table 27 – continued from previous page

double()	Casts all floating point parameters and buffers to double datatype.
eval()	Sets the module in evaluation mode.
extra_repr()	Set the extra representation of the module
float()	Casts all floating point parameters and buffers to float datatype.
forward(*args, **kwargs)	Defines the computation performed at every call.
get_buffer(target)	Returns the buffer given by target if it exists, otherwise throws an error.
get_extra_state()	Returns any extra state to include in the module's state_dict.
get_parameter(target)	Returns the parameter given by target if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by target if it exists, otherwise throws an error.
half()	Casts all floating point parameters and buffers to half datatype.
ipu([device])	Moves all model parameters and buffers to the IPU.
load_state_dict(state_dict[, strict])	Copies parameters and buffers from *state_dict* into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse, ...])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse, ...])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
no_sync()	A context manager to disable synchronization after backward
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_comm_hook(hook)	Registers a hook function.
register_forward_hook(hook, *[, prepend, ...])	Registers a forward hook on the module.
register_forward_pre_hook(hook, *[, ...])	Registers a forward pre-hook on the module.
register_full_backward_hook(hook[, prepend])	Registers a backward hook on the module.
register_full_backward_pre_hook(hook[, prepend])	Registers a backward pre-hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
register_state_dict_pre_hook(hook)	These hooks will be called with arguments: self, prefix, and keep_vars before calling state_dict on self.

continues on next page

Table 27 – continued from previous page

requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
set_extra_state(state)	This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict()	Returns a dictionary containing references to the whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device[, recurse])	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Resets gradients of all model parameters.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Mapping[str, Tensor])
call_super_init	
dump_patches	

T_destination

alias of TypeVar('T_destination', bound=Mapping[str, Tensor])

__init__(*module*, *broadcast_buffers=True*, *negotiate_grads=True*, *process_group=None*, *reduce_function=None*, *broadcast_function=None*, ***kwargs*)

This module receives keyword arguments for the compatibility with *torch.nn.parallel.DistributedDataParallel*. It shows a warning when setting the ignored arguments.

Parameters

- **module** (*Module*) –
- **broadcast_buffers** (*bool*) –
- **negotiate_grads** (*bool*) –
- **process_group** (*Optional[ProcessGroup]*) –
- **reduce_function** (*Optional[Callable[[Sequence[Tensor], Optional[ProcessGroup]], None]]*) –
- **broadcast_function** (*Optional[Callable[[Sequence[Tensor], Optional[ProcessGroup]], None]]*) –
- **kwargs** (*Any*) –

Return type

None

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

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

Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Warning: If `assign` is `True` the optimizer must be created after the call to *load_state_dict*.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`
- **assign** (*bool, optional*) – whether to assign items in the state dictionary to their corresponding keys in the module instead of copying them inplace into the module's current parameters and buffers. When `False`, the properties of the tensors in the current module are preserved while when `True`, the properties of the Tensors in the state dict are preserved. Default: `False`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type
`NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

`no_sync`()

A context manager to disable synchronization after backward

Return type
Generator[None, None, None]

register_comm_hook(*hook*)

Registers a hook function. This module will invoke the hook before starting the synchronization.

Args: hook: Callable object that will be invoked before synchronization

Parameters
hook (*Callable[[DistributedDataParallel], None]*) –

Return type
RemovableHandle

state_dict()

Returns a dictionary containing references to the 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.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

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

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

Parameters

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

Returns
a dictionary containing a whole state of the module

Return type
dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

training: bool

Modules

pytorch_pfn_extras.nn.parallel.distributed

pytorch_pfn_extras.nn.parallel.distributed

Functions

pytorch_pfn_extras.nn.parallel.distributed.contextmanager(func)	@contextmanager decorator.
pytorch_pfn_extras.nn.parallel.distributed.get_foreach_wrapper()	
pytorch_pfn_extras.nn.parallel.distributed.record(tag)	

pytorch_pfn_extras.nn.parallel.distributed.contextmanager

pytorch_pfn_extras.nn.parallel.distributed.**contextmanager**(*func*)

@contextmanager decorator.

Typical usage:

@contextmanager def some_generator(<arguments>):

<setup> try:

yield <value>

finally:
<cleanup>

This makes this:

with some_generator(<arguments>) as <variable>:
<body>

equivalent to this:

<setup> try:

<variable> = <value> <body>

finally:
<cleanup>

pytorch_pfn_extras.nn.parallel.distributed.get_foreach_wrapper

pytorch_pfn_extras.nn.parallel.distributed.**get_foreach_wrapper**()

Return type
_ForEachWrapper

pytorch_pfn_extras.nn.parallel.distributed.record

pytorch_pfn_extras.nn.parallel.distributed.**record**(*tag*, *metric=None*, *use_cuda=False*, *enable=True*, *device='cpu'*)

Parameters

- **tag** (*Optional[str]*) –
- **metric** (*Optional[str]*) –
- **use_cuda** (*bool*) –
- **enable** (*bool*) –
- **device** (*DeviceLike*) –

Return type
Generator[*_ReportNotification*, None, None]

Classes

pytorch_pfn_extras.nn.parallel.distributed.DistributedDataParallel(module)	Module for distributed data parallelism
pytorch_pfn_extras.nn.parallel.distributed.OrderedDict	Dictionary that remembers insertion order
pytorch_pfn_extras.nn.parallel.distributed.TypeVar(...)	Type variable.
pytorch_pfn_extras.nn.parallel.distributed.Variable	
pytorch_pfn_extras.nn.parallel.distributed.record_function(name)	Context manager/function decorator that adds a label to a block of Python code (or function) when running autograd profiler.

pytorch_pfn_extras.nn.parallel.distributed.DistributedDataParallel

class pytorch_pfn_extras.nn.parallel.distributed.**DistributedDataParallel**(*module*, *broadcast_buffers=True*, *negotiate_grads=True*, *process_group=None*, *reduce_function=None*, *broadcast_function=None*, ***kwargs*)

Bases: Module

Module for distributed data parallelism

This class synchronizes the gradients and the buffers after backward computations.

Parameters

- **module** (*Module*) – torch.nn.Module object to be trained
- **broadcast_buffers** (*bool*) – Boolean flag to broadcast buffers after backward computations. Broadcasting buffers may be helpful when the module includes BatchNormalization. However, it will degrade training throughput. (default: *True*)
- **negotiate_grads** (*bool*) – Boolean flag to choose gradients to be sent before all-reduce. This flag is necessary when the computation graph of the module is dynamic. (default: *True*)
- **process_group** (*Optional[ProcessGroup]*) – Process group used for broadcasting and reducing. (default: *torch.distributed.group.WORLD*)
- **reduce_function** (*Optional[Callable[[Sequence[Tensor], Optional[ProcessGroup]], None]]*) – All-reduce function
- **broadcast_function** (*Optional[Callable[[Sequence[Tensor], Optional[ProcessGroup]], None]]*) – Broadcast function
- **kwargs** (*Any*) –

This module receives keyword arguments for the compatibility with *torch.nn.parallel.DistributedDataParallel*. It shows a warning when setting the ignored arguments.

Methods

__init__(module[, broadcast_buffers, ...])	This module receives keyword arguments for the compatibility with *torch.nn.parallel.DistributedDataParallel*.
add_module(name, module)	Adds a child module to the current module.
apply(fn)	Applies fn recursively to every submodule (as returned by .children()) as well as self.
bfloat16()	Casts all floating point parameters and buffers to bfloat16 datatype.
buffers([recurse])	Returns an iterator over module buffers.
children()	Returns an iterator over immediate children modules.
compile(*args, **kwargs)	Compile this Module's forward using torch.compile().
cpu()	Moves all model parameters and buffers to the CPU.
cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to double datatype.
eval()	Sets the module in evaluation mode.
extra_repr()	Set the extra representation of the module
float()	Casts all floating point parameters and buffers to float datatype.
forward(*args, **kwargs)	Defines the computation performed at every call.

continues on next page

Table 28 – continued from previous page

get_buffer(target)	Returns the buffer given by target if it exists, otherwise throws an error.
get_extra_state()	Returns any extra state to include in the module's state_dict.
get_parameter(target)	Returns the parameter given by target if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by target if it exists, otherwise throws an error.
half()	Casts all floating point parameters and buffers to half datatype.
ipu([device])	Moves all model parameters and buffers to the IPU.
load_state_dict(state_dict[, strict])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse, ...])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse, ...])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
no_sync()	A context manager to disable synchronization after backward
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_comm_hook(hook)	Registers a hook function.
register_forward_hook(hook, *[, prepend, ...])	Registers a forward hook on the module.
register_forward_pre_hook(hook, *[, ...])	Registers a forward pre-hook on the module.
register_full_backward_hook(hook[, prepend])	Registers a backward hook on the module.
register_full_backward_pre_hook(hook[, prepend])	Registers a backward pre-hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
register_state_dict_pre_hook(hook)	These hooks will be called with arguments: self, prefix, and keep_vars before calling state_dict on self.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict()	Returns a dictionary containing references to the whole state of the module.

continues on next page

Table 28 – continued from previous page

to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device[, recurse])	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Resets gradients of all model parameters.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Mapping[str, Tensor])
call_super_init	
dump_patches	

T_destination

alias of TypeVar('T_destination', bound=Mapping[str, Tensor])

__init__(*module*, *broadcast_buffers=True*, *negotiate_grads=True*, *process_group=None*, *reduce_function=None*, *broadcast_function=None*, ***kwargs*)

This module receives keyword arguments for the compatibility with *torch.nn.parallel.DistributedDataParallel*. It shows a warning when setting the ignored arguments.

Parameters

- **module** (*Module*) –
- **broadcast_buffers** (*bool*) –
- **negotiate_grads** (*bool*) –
- **process_group** (*Optional[ProcessGroup]*) –
- **reduce_function** (*Optional[Callable[[Sequence[Tensor], Optional[ProcessGroup]], None]]*) –
- **broadcast_function** (*Optional[Callable[[Sequence[Tensor], Optional[ProcessGroup]], None]]*) –
- **kwargs** (*Any*) –

Return type
None

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

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

Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

> **Warning:** If `assign` is `True` the optimizer must be created after the call to *load_state_dict*.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`
- **assign** (*bool, optional*) – whether to assign items in the state dictionary to their corresponding keys in the module instead of copying them inplace into the module's current parameters and buffers. When `False`, the properties of the tensors in the current module are preserved while when `True`, the properties of the Tensors in the state dict are preserved. Default: `False`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type
`NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

`no_sync`()

A context manager to disable synchronization after backward

Return type
Generator[None, None, None]

`register_comm_hook`(*hook*)

Registers a hook function. This module will invoke the hook before starting the synchronization.

Args: hook: Callable object that will be invoked before synchronization

Parameters
hook (*Callable[[DistributedDataParallel], None]*) –

Return type
RemovableHandle

`state_dict()`

Returns a dictionary containing references to the 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.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

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

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

Parameters

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

Returns
a dictionary containing a whole state of the module

Return type
dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

`training: bool`

pytorch_pfn_extras.nn.parallel.distributed.OrderedDict

`class pytorch_pfn_extras.nn.parallel.distributed.OrderedDict`

Bases: `dict`

Dictionary that remembers insertion order

Methods

__init__(*args, **kwargs)	
clear()	
copy()	
fromkeys([value])	Create a new ordered dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
move_to_end(key[, last])	Move an existing element to the end (or beginning if last is false).
pop(k[,d])	value.
popitem([last])	Remove and return a (key, value) pair from the dictionary.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

__init__(**args*, ***kwargs*)

clear() → None. Remove all items from od.

copy() → a shallow copy of od

fromkeys(*value=None*)

Create a new ordered dictionary with keys from iterable and values set to value.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

move_to_end(*key*, *last=True*)

Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding

value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem(*last=True*)

Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault(*key*, *default=None*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

pytorch_pfn_extras.nn.parallel.distributed.TypeVar

class pytorch_pfn_extras.nn.parallel.distributed.**TypeVar**(*name*, **constraints*, *bound=None*, *covariant=False*, *contravariant=False*)

Bases: _Final, _Immutable

Type variable.

Usage:

```
T = TypeVar('T')  # Can be anything
A = TypeVar('A', str, bytes)  # Must be str or bytes
```

Type variables exist primarily for the benefit of static type checkers. They serve as the parameters for generic types as well as for generic function definitions. See class Generic for more information on generic types. Generic functions work as follows:

def repeat(x: T, n: int) -> List[T]:
'''Return a list containing n references to x.''' return [x]*n

def longest(x: A, y: A) -> A:
'''Return the longest of two strings.''' return x if len(x) >= len(y) else y

The latter example's signature is essentially the overloading of (str, str) -> str and (bytes, bytes) -> bytes. Also note that if the arguments are instances of some subclass of str, the return type is still plain str.

At runtime, isinstance(x, T) and issubclass(C, T) will raise TypeError.

Type variables defined with covariant=True or contravariant=True can be used to declare covariant or contravariant generic types. See PEP 484 for more details. By default generic types are invariant in all type variables.

Type variables can be introspected. e.g.:

T.__name__ == 'T' T.__constraints__ == () T.__covariant__ == False T.__contravariant__ = False A.__constraints__ == (str, bytes)

Note that only type variables defined in global scope can be pickled.

Methods

__init__(name, *constraints[, bound, ...])

__init__(*name*, **constraints*, *bound=None*, *covariant=False*, *contravariant=False*)

pytorch_pfn_extras.nn.parallel.distributed.Variable

class pytorch_pfn_extras.nn.parallel.distributed.**Variable**

Bases: _LegacyVariableBase

Methods

__init__()

pytorch_pfn_extras.nn.parallel.distributed.record_function

class pytorch_pfn_extras.nn.parallel.distributed.**record_function**(*name*, *args=None*)

Bases: ContextDecorator

Context manager/function decorator that adds a label to a block of Python code (or function) when running autograd profiler. It is useful when tracing the code profile.

Parameters

- **name** (*str*) – Label assigned to the block of code.
- **node_id** (*int*) – ID of node, for distributed profiling. Unset in
- **cases.** (*non-distributed*) –
- **args** (*Optional[str]*) –

Example

```
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_AUTOGRAD_PROFILER)
>>> x = torch.randn((1, 1), requires_grad=True)
>>> with torch.autograd.profiler.profile() as prof:
...     y = x ** 2
...     with torch.autograd.profiler.record_function("label-z"): # label the block
...         z = y ** 3
...     y.backward()
...
>>> # xdoctest: +IGNORE_WANT
>>> # NOTE: some columns were removed for brevity
>>> print(prof.key_averages().table(sort_by="self_cpu_time_total"))
-----------------------------------  ---------------  ---------------  -------------
↪--
```

(continues on next page)

(continued from previous page)

```
Name                                  Self CPU total %  CPU time avg     Number of␣
↪Calls
-----------------------------------  ---------------  ---------------  -------------
↪--
pow                                   60.77%           47.470us         3
mul                                   21.73%           25.465us         2
PowBackward0                          12.03%           121.891us        1
torch::autograd::AccumulateGrad       2.70%            6.324us          1
label-z                               2.13%            12.421us         1
torch::autograd::GraphRoot            0.64%            1.503us          1
-----------------------------------  ---------------  ---------------  -------------
↪--
Self CPU time total: 234.344us
CUDA time total: 0.000us
```

Methods

__init__(name[, args])	

__init__(*name*, *args=None*)

Parameters

- **name** (*str*) –
- **args** (*Optional[str]*) –

pytorch_pfn_extras.onnx

Functions

pytorch_pfn_extras.onnx.annotate(**attrs)	Annotation parameters to the target function.
pytorch_pfn_extras.onnx.apply_annotation(fn, ...)	Annotation applier to the target function
pytorch_pfn_extras.onnx.as_output(name, value)	
pytorch_pfn_extras.onnx.export(model, args, f)	Export model into ONNX Graph.
pytorch_pfn_extras.onnx.export_testcase(...)	Export model and I/O tensors of the model in protobuf format.
pytorch_pfn_extras.onnx.grad(output, inputs)	
pytorch_pfn_extras.onnx.is_large_tensor(...)	
pytorch_pfn_extras.onnx.load_model(f[, ...])	Load model from ONNX file.
pytorch_pfn_extras.onnx.no_grad(fn, *args, ...)	
pytorch_pfn_extras.onnx.scoped_anchor(**attrs)	Add anchor node to the scoped modules

pytorch_pfn_extras.onnx.annotate

pytorch_pfn_extras.onnx.**annotate**(***attrs*)

Annotation parameters to the target function.

Usage:

```
>>> class Net(nn.Module):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.conv = nn.Conv2d(1, 6, 3)
...         self.conv2 = nn.Conv2d(6, 12, 3)
...     def forward(self, x):
...         with pytorch_pfn_extras.onnx.annotate(key='value'):
...             h = self.conv(x)
...         h = self.conv2(h)
...         return h
```

Use this annotate function under with statement, then the first Conv operator will be emit with customized attributes. Customized attributes are invalid for ONNX format, so pay attention that some ONNX runtimes cannot run the output ONNX graph.

This annotation is enabled with either `pytorch_pfn_extras.onnx.export_testcase` or `pytorch_pfn_extras.onnx.export`.

Parameters
attrs (*dict*) – annotation parameters

Return type
AbstractContextManager[None]

pytorch_pfn_extras.onnx.apply_annotation

pytorch_pfn_extras.onnx.**apply_annotation**(*fn*, **args*, ***attrs*)

Annotation applier to the target function

Usage:

```
>>> class Net(nn.Module):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.conv = nn.Conv2d(1, 6, 3)
...         self.conv2 = nn.Conv2d(6, 12, 3)
...     def forward(self, x):
...         def _conv(x):
...             h = self.conv(x)
...             return torch.relu(h)
...         h = pytorch_pfn_extras.onnx.apply_annotation(
...             _conv, key='value')
...         h = self.conv2(h)
...         return h
```

Annotate into all operators emitted from the target function even if included not `nn.Module` function. On the above code, the first Conv and ReLu operator will be emit with customized attributes. Customized attributes are invlid for ONNX format, so pay attention that some ONNX runtimes cannot run the output ONNX graph.

This applier is enabled with either `pytorch_pfn_extras.onnx.export_testcase` or `pytorch_pfn_extras.onnx.export`.

Parameters

- **fn** (*func*) – the target function to be annotated, `args` is used for this function. Cannot pass `kwargs` for the function.
- **args** (*tuple*) – arguments for the target function
- **attrs** (*dict*) – annotation paramters

Return type
Any

pytorch_pfn_extras.onnx.as_output

`pytorch_pfn_extras.onnx.as_output(name, value, add_identity=True)`

Parameters

- **name** (*str*) –
- **value** (*Tensor*) –
- **add_identity** (*bool*) –

Return type
Tensor

pytorch_pfn_extras.onnx.export

`pytorch_pfn_extras.onnx.export(model, args, f, return_output=False, strip_large_tensor_data=False, large_tensor_threshold=100, chrome_tracing='', **kwargs)`

Export model into ONNX Graph.

Parameters

- **f** (IO) – A file-like object or a string file path to be written to this file.
- **return_output** (*bool*) – If True, return output values come from the model.
- **strip_large_tensor_data** (*bool*) – If True, this function will strip data of large tensors to reduce ONNX file size for benchmarking
- **large_tensor_threshold** (*int*) – If number of elements of tensor is larger than this value, the tensor is stripped when *strip_large_tensor_data* is True
- **model** (*Module*) –
- **args** (*Sequence[Any]*) –
- **chrome_tracing** (*str*) –
- **kwargs** (*Any*) –

Return type
Any

Warning: This function is not thread safe.

pytorch_pfn_extras.onnx.export_testcase

pytorch_pfn_extras.onnx.**export_testcase**(*model*, *args*, *out_dir*, *, *output_grad=False*, *metadata=True*, *model_overwrite=True*, *strip_large_tensor_data=False*, *large_tensor_threshold=100*, *return_output=False*, *user_meta=None*, *export_torch_script=False*, *export_torch_trace=False*, *export_chrome_tracing=True*, ***kwargs*)

Export model and I/O tensors of the model in protobuf format.

Parameters

- **output_grad** (*bool or Tensor*) – If True, this function will output model's gradient with names 'gradient_%d.pb'. If set Tensor, use it as gradient *input*. The gradient inputs are output as 'gradient_input_%d.pb' along with gradient.
- **metadata** (*bool*) – If True, output meta information taken from git log.
- **model_overwrite** (*bool*) – If False and model.onnx has already existed, only export input/output data as another test dataset.
- **strip_large_tensor_data** (*bool*) – If True, this function will strip data of large tensors to reduce ONNX file size for benchmarking
- **large_tensor_threshold** (*int*) – If number of elements of tensor is larger than this value, the tensor is stripped when *strip_large_tensor_data* is True
- **return_output** (*bool*) – If True, return output values come from the model.
- **export_torch_script** (*bool*) – Output model_script.pt using torch.jit.script
- **export_torch_trace** (*bool*) – Output model_trace.pt using torch.jit.trace
- **model** (*Union[Module, ScriptModule]*) –
- **args** (*Any*) –
- **out_dir** (*str*) –
- **user_meta** (*Optional[Mapping[str, Any]]*) –
- **export_chrome_tracing** (*bool*) –
- **kwargs** (*Any*) –

Return type
Any

Warning: This function is not thread safe.

Note: When exporting a model whose forward takes keyword arguments of `torch.Tensor` type, you can pass them by putting a dict as the last element of `args`. When the keyword arguments have default values, you need to explicitly include them into the dict. Also, you must explicitly specify `input_names` that are the names of both positional and keyword arguments.

pytorch_pfn_extras.onnx.grad

pytorch_pfn_extras.onnx.**grad**(*output*, *inputs*, *retain_graph=None*, *create_graph=False*, *only_inputs=True*, *allow_unused=False*)

Parameters

- **output** (*Tensor*) –
- **inputs** (*Tuple[Tensor, ...]*) –
- **retain_graph** (*Optional[bool]*) –
- **create_graph** (*bool*) –
- **only_inputs** (*bool*) –
- **allow_unused** (*bool*) –

Return type
Tuple[*Optional*[*Tensor*], ...]

pytorch_pfn_extras.onnx.is_large_tensor

pytorch_pfn_extras.onnx.**is_large_tensor**(*tensor*, *threshold*)

Parameters

- **tensor** (*TensorProto*) –
- **threshold** (*int*) –

Return type
bool

pytorch_pfn_extras.onnx.load_model

pytorch_pfn_extras.onnx.**load_model**(*f*, *format=None*, *load_external_data=True*)

Load model from ONNX file.

This is a wrapper to *onnx.load_model* that automatically falls back to *load_external_data=False* when tensors are stripped.

Parameters

- **f** (*Union[IO, str]*) – A file-like object or a string file path to be written to this file.
- **format** (*Optional[Any]*) – A reserved arg
- **load_external_data** (*bool*) – If True and the external data under the same directory of the model, load the external data

Return type
ModelProto

pytorch_pfn_extras.onnx.no_grad

pytorch_pfn_extras.onnx.**no_grad**(*fn*, **args*, ***kwargs*)

Parameters

- **fn** (*Callable[[...], Any]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

pytorch_pfn_extras.onnx.scoped_anchor

pytorch_pfn_extras.onnx.**scoped_anchor**(***attrs*)

Add anchor node to the scoped modules

Usage:

```
>>> class Net(nn.Module):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.conv = nn.Conv2d(1, 6, 3)
...         self.conv2 = nn.Conv2d(6, 12, 3)
...     def forward(self, x):
...         with pytorch_pfn_extras.onnx.scoped_anchor(key='value'):
...             h = self.conv(x)
...         h = self.conv2(h)
...         return h
```

Use this scoped anchoring under with statement, then dummy Identity nodes are added before/after the first Conv operator with customized attributes.

This anchoring is triggered by `nn.Module` applying function, cannot use this with `torch.*` functions.

This annotation is enabled with either `pytorch_pfn_extras.onnx.export_testcase` or `pytorch_pfn_extras.onnx.export`.

Parameters
attrs (*dict*) – annotation parameters

Return type
AbstractContextManager[None]

Modules

pytorch_pfn_extras.onnx.annotate(**attrs)	Annotation parameters to the target function.
pytorch_pfn_extras.onnx.export_testcase(...)	Export model and I/O tensors of the model in protobuf format.
pytorch_pfn_extras.onnx.load	
pytorch_pfn_extras.onnx.pfto_exporter	
pytorch_pfn_extras.onnx.strip_large_tensor	
pytorch_pfn_extras.onnx.symbolic_registry	
pytorch_pfn_extras.onnx.unstrip_tensor	

pytorch_pfn_extras.onnx.load

Functions

pytorch_pfn_extras.onnx.load.load_model(f[, ...])	Load model from ONNX file.

pytorch_pfn_extras.onnx.load.load_model

pytorch_pfn_extras.onnx.load.**load_model**(*f*, *format=None*, *load_external_data=True*)

Load model from ONNX file.

This is a wrapper to *onnx.load_model* that automatically falls back to *load_external_data=False* when tensors are stripped.

Parameters

- **f** (*Union[IO, str]*) – A file-like object or a string file path to be written to this file.
- **format** (*Optional[Any]*) – A reserved arg
- **load_external_data** (*bool*) – If True and the external data under the same directory of the model, load the external data

Return type

ModelProto

Classes

pytorch_pfn_extras.onnx.load.IO(*args, **kwds)	Generic base class for TextIO and BinaryIO.
pytorch_pfn_extras.onnx.load.Path(*args, ...)	PurePath subclass that can make system calls.
pytorch_pfn_extras.onnx.load.Text	alias of `str`

pytorch_pfn_extras.onnx.load.IO

class pytorch_pfn_extras.onnx.load.**IO**(**args*, ***kwds*)

Bases: `Generic`

Generic base class for TextIO and BinaryIO.

This is an abstract, generic version of the return of open().

NOTE: This does not distinguish between the different possible classes (text vs. binary, read vs. write vs. read/write, append-only, unbuffered). The TextIO and BinaryIO subclasses below capture the distinctions between text vs. binary, which is pervasive in the interface; however we currently do not offer a way to track the other distinctions in the type system.

Methods

`__init__`()
close()
fileno()
flush()
isatty()
read([n])
readable()
readline([limit])
readlines([hint])
seek(offset[, whence])
seekable()
tell()
truncate([size])
writable()
write(s)
writelines(lines)

Attributes

closed
mode
name

abstract close()

Return type
None

abstract property closed: bool

abstract fileno()

Return type
int

abstract flush()

Return type
None

abstract isatty()

Return type
bool

abstract property mode: str

abstract property name: str

abstract read(*n=-1*)

Parameters
n (*int*) –

Return type
AnyStr

abstract readable()

Return type
bool

abstract readline(*limit=-1*)

Parameters
limit (*int*) –

Return type
AnyStr

abstract readlines(*hint=-1*)

Parameters
hint (*int*) –

Return type
List

abstract seek(*offset*, *whence=0*)

Parameters

- **offset** (*int*) –
- **whence** (*int*) –

Return type
int

abstract seekable()

Return type
bool

abstract tell()

Return type
int

abstract truncate(*size=None*)

Parameters
size (*Optional[int]*) –

Return type
int

abstract writable()

Return type
bool

abstract write(*s*)

Parameters
s (*AnyStr*) –

Return type
int

abstract writelines(*lines*)

Parameters
lines (*List*) –

Return type
None

pytorch_pfn_extras.onnx.load.Path

class pytorch_pfn_extras.onnx.load.**Path**(**args*, ***kwargs*)

Bases: PurePath

PurePath subclass that can make system calls.

Path represents a filesystem path but unlike PurePath, also offers methods to do system calls on path objects. Depending on your system, instantiating a Path will return either a PosixPath or a WindowsPath object. You can also instantiate a PosixPath or WindowsPath directly, but cannot instantiate a WindowsPath on a POSIX system or vice versa.

Construct a PurePath from one or several strings and or existing PurePath objects. The strings and path objects are combined so as to yield a canonicalized path, which is incorporated into the new PurePath object.

Methods

__init__()	

absolute()	Return an absolute version of this path.
as_posix()	Return the string representation of the path with forward (/) slashes.
as_uri()	Return the path as a 'file' URI.
chmod(mode)	Change the permissions of the path, like os.chmod().
cwd()	Return a new path pointing to the current working directory (as returned by os.getcwd()).
exists()	Whether this path exists.
expanduser()	Return a new path with expanded ~ and ~user constructs (as returned by os.path.expanduser)
glob(pattern)	Iterate over this subtree and yield all existing files (of any kind, including directories) matching the given relative pattern.
group()	Return the group name of the file gid.
home()	Return a new path pointing to the user's home directory (as returned by os.path.expanduser('~')).
is_absolute()	True if the path is absolute (has both a root and, if applicable, a drive).
is_block_device()	Whether this path is a block device.
is_char_device()	Whether this path is a character device.
is_dir()	Whether this path is a directory.
is_fifo()	Whether this path is a FIFO.
is_file()	Whether this path is a regular file (also True for symlinks pointing to regular files).
is_mount()	Check if this path is a POSIX mount point
is_reserved()	Return True if the path contains one of the special names reserved by the system, if any.
is_socket()	Whether this path is a socket.
is_symlink()	Whether this path is a symbolic link.
iterdir()	Iterate over the files in this directory.
joinpath(*args)	Combine this path with one or several arguments, and return a new path representing either a subpath (if all arguments are relative paths) or a totally different path (if one of the arguments is anchored).
lchmod(mode)	Like chmod(), except if the path points to a symlink, the symlink's permissions are changed, rather than its target's.
link_to(target)	Make the target path a hard link pointing to this path.
lstat()	Like stat(), except if the path points to a symlink, the symlink's status information is returned, rather than its target's.
match(path_pattern)	Return True if this path matches the given pattern.
mkdir([mode, parents, exist_ok])	Create a new directory at this given path.
open([mode, buffering, encoding, errors, ...])	Open the file pointed by this path and return a file object, as the built-in open() function does.
owner()	Return the login name of the file owner.
read_bytes()	Open the file in bytes mode, read it, and close the file.

continues on next page

Table 29 – continued from previous page

read_text([encoding, errors])	Open the file in text mode, read it, and close the file.
relative_to(*other)	Return the relative path to another path identified by the passed arguments.
rename(target)	Rename this path to the target path.
replace(target)	Rename this path to the target path, overwriting if that path exists.
resolve([strict])	Make the path absolute, resolving all symlinks on the way and also normalizing it (for example turning slashes into backslashes under Windows).
rglob(pattern)	Recursively yield all existing files (of any kind, including directories) matching the given relative pattern, anywhere in this subtree.
rmdir()	Remove this directory.
samefile(other_path)	Return whether other_path is the same or not as this file (as returned by os.path.samefile()).
stat()	Return the result of the stat() system call on this path, like os.stat() does.
symlink_to(target[, target_is_directory])	Make this path a symlink pointing to the target path.
touch([mode, exist_ok])	Create this file with the given access mode, if it doesn't exist.
unlink([missing_ok])	Remove this file or link.
with_name(name)	Return a new path with the file name changed.
with_suffix(suffix)	Return a new path with the file suffix changed.
write_bytes(data)	Open the file in bytes mode, write to it, and close the file.
write_text(data[, encoding, errors])	Open the file in text mode, write to it, and close the file.

Attributes

anchor	The concatenation of the drive and root, or ''.
drive	The drive prefix (letter or UNC path), if any.
name	The final path component, if any.
parent	The logical parent of the path.
parents	A sequence of this path's logical parents.
parts	An object providing sequence-like access to the components in the filesystem path.
root	The root of the path, if any.
stem	The final path component, minus its last suffix.
suffix	The final component's last suffix, if any.
suffixes	A list of the final component's suffixes, if any.

absolute()

Return an absolute version of this path. This function works even if the path doesn't point to anything.

No normalization is done, i.e. all '.' and '..' will be kept along. Use resolve() to get the canonical path to a file.

chmod(*mode*)

Change the permissions of the path, like os.chmod().

classmethod cwd()

Return a new path pointing to the current working directory (as returned by os.getcwd()).

exists()

Whether this path exists.

expanduser()

Return a new path with expanded ~ and ~user constructs (as returned by os.path.expanduser)

glob(*pattern*)

Iterate over this subtree and yield all existing files (of any kind, including directories) matching the given relative pattern.

group()

Return the group name of the file gid.

classmethod home()

Return a new path pointing to the user's home directory (as returned by os.path.expanduser('~')).

is_block_device()

Whether this path is a block device.

is_char_device()

Whether this path is a character device.

is_dir()

Whether this path is a directory.

is_fifo()

Whether this path is a FIFO.

is_file()

Whether this path is a regular file (also True for symlinks pointing to regular files).

is_mount()

Check if this path is a POSIX mount point

is_socket()

Whether this path is a socket.

is_symlink()

Whether this path is a symbolic link.

iterdir()

Iterate over the files in this directory. Does not yield any result for the special paths '.' and '..'.

lchmod(*mode*)

Like chmod(), except if the path points to a symlink, the symlink's permissions are changed, rather than its target's.

link_to(*target*)

Make the target path a hard link pointing to this path.

Note this function does not make this path a hard link to *target*, despite the implication of the function and argument names. The order of arguments (target, link) is the reverse of Path.symlink_to, but matches that of os.link.

`lstat()`

Like stat(), except if the path points to a symlink, the symlink's status information is returned, rather than its target's.

`mkdir(mode=511, parents=False, exist_ok=False)`

Create a new directory at this given path.

`open(mode='r', buffering=-1, encoding=None, errors=None, newline=None)`

Open the file pointed by this path and return a file object, as the built-in open() function does.

`owner()`

Return the login name of the file owner.

`read_bytes()`

Open the file in bytes mode, read it, and close the file.

`read_text(encoding=None, errors=None)`

Open the file in text mode, read it, and close the file.

`rename(target)`

Rename this path to the target path.

The target path may be absolute or relative. Relative paths are interpreted relative to the current working directory, *not* the directory of the Path object.

Returns the new Path instance pointing to the target path.

`replace(target)`

Rename this path to the target path, overwriting if that path exists.

The target path may be absolute or relative. Relative paths are interpreted relative to the current working directory, *not* the directory of the Path object.

Returns the new Path instance pointing to the target path.

`resolve(strict=False)`

Make the path absolute, resolving all symlinks on the way and also normalizing it (for example turning slashes into backslashes under Windows).

`rglob(pattern)`

Recursively yield all existing files (of any kind, including directories) matching the given relative pattern, anywhere in this subtree.

`rmdir()`

Remove this directory. The directory must be empty.

`samefile(other_path)`

Return whether other_path is the same or not as this file (as returned by os.path.samefile()).

`stat()`

Return the result of the stat() system call on this path, like os.stat() does.

`symlink_to(target, target_is_directory=False)`

Make this path a symlink pointing to the target path. Note the order of arguments (link, target) is the reverse of os.symlink.

`touch(mode=438, exist_ok=True)`

Create this file with the given access mode, if it doesn't exist.

unlink(*missing_ok=False*)

Remove this file or link. If the path is a directory, use rmdir() instead.

write_bytes(*data*)

Open the file in bytes mode, write to it, and close the file.

write_text(*data*, *encoding=None*, *errors=None*)

Open the file in text mode, write to it, and close the file.

pytorch_pfn_extras.onnx.load.Text

pytorch_pfn_extras.onnx.load.**Text**

alias of `str`

pytorch_pfn_extras.onnx.pfto_exporter

Modules

pytorch_pfn_extras.onnx.pfto_exporter.export	

pytorch_pfn_extras.onnx.strip_large_tensor

Functions

pytorch_pfn_extras.onnx.strip_large_tensor.is_large_tensor(...)	
pytorch_pfn_extras.onnx.strip_large_tensor.reduce(...)	Apply a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value.

pytorch_pfn_extras.onnx.strip_large_tensor.is_large_tensor

pytorch_pfn_extras.onnx.strip_large_tensor.**is_large_tensor**(*tensor*, *threshold*)

Parameters

- **tensor** (*TensorProto*) –
- **threshold** (*int*) –

Return type

bool

pytorch_pfn_extras.onnx.strip_large_tensor.reduce

pytorch_pfn_extras.onnx.strip_large_tensor.**reduce**(*function*, *sequence*[, *initial*]) → value

Apply a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value. For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates ((((1+2)+3)+4)+5). If initial is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty.

pytorch_pfn_extras.onnx.symbolic_registry

Functions

pytorch_pfn_extras.onnx.symbolic_registry.cast(...)	Cast a value to a type.
pytorch_pfn_extras.onnx.symbolic_registry.get_registered_op(...)	
pytorch_pfn_extras.onnx.symbolic_registry.is_registered_op(...)	
pytorch_pfn_extras.onnx.symbolic_registry.register_op(...)	

Classes

pytorch_pfn_extras.onnx.symbolic_registry.Value	

pytorch_pfn_extras.onnx.unstrip_tensor

Functions

pytorch_pfn_extras.onnx.unstrip_tensor.unstrip(path)	Unstrip ONNX models and test data(.pb).

pytorch_pfn_extras.onnx.unstrip_tensor.unstrip

pytorch_pfn_extras.onnx.unstrip_tensor.**unstrip**(*path*, *out_path=''*)

Unstrip ONNX models and test data(.pb).

Add tensor(raw data) to the target ONNXs (and test data). Values are random following mean and variance written in meta information.

Parameters

- **path** (*str*) – The target directory path, ONNX file, or Tensor (Protobuf) file path.
- **out_path** (*str*) – Output path to be written.

Return type

None

Classes

pytorch_pfn_extras.onnx.unstrip_tensor.Path(...)	PurePath subclass that can make system calls.

pytorch_pfn_extras.onnx.unstrip_tensor.Path

class pytorch_pfn_extras.onnx.unstrip_tensor.**Path**(**args*, ***kwargs*)

Bases: PurePath

PurePath subclass that can make system calls.

Path represents a filesystem path but unlike PurePath, also offers methods to do system calls on path objects. Depending on your system, instantiating a Path will return either a PosixPath or a WindowsPath object. You can also instantiate a PosixPath or WindowsPath directly, but cannot instantiate a WindowsPath on a POSIX system or vice versa.

Construct a PurePath from one or several strings and or existing PurePath objects. The strings and path objects are combined so as to yield a canonicalized path, which is incorporated into the new PurePath object.

Methods

__init__()	
absolute()	Return an absolute version of this path.
as_posix()	Return the string representation of the path with forward (/) slashes.
as_uri()	Return the path as a 'file' URI.
chmod(mode)	Change the permissions of the path, like os.chmod().
cwd()	Return a new path pointing to the current working directory (as returned by os.getcwd()).
exists()	Whether this path exists.
expanduser()	Return a new path with expanded ~ and ~user constructs (as returned by os.path.expanduser)
glob(pattern)	Iterate over this subtree and yield all existing files (of any kind, including directories) matching the given relative pattern.
group()	Return the group name of the file gid.
home()	Return a new path pointing to the user's home directory (as returned by os.path.expanduser('~')).
is_absolute()	True if the path is absolute (has both a root and, if applicable, a drive).
is_block_device()	Whether this path is a block device.
is_char_device()	Whether this path is a character device.
is_dir()	Whether this path is a directory.
is_fifo()	Whether this path is a FIFO.
is_file()	Whether this path is a regular file (also True for symlinks pointing to regular files).
is_mount()	Check if this path is a POSIX mount point

continues on next page

Table 30 – continued from previous page

is_reserved()	Return True if the path contains one of the special names reserved by the system, if any.
is_socket()	Whether this path is a socket.
is_symlink()	Whether this path is a symbolic link.
iterdir()	Iterate over the files in this directory.
joinpath(*args)	Combine this path with one or several arguments, and return a new path representing either a subpath (if all arguments are relative paths) or a totally different path (if one of the arguments is anchored).
lchmod(mode)	Like chmod(), except if the path points to a symlink, the symlink's permissions are changed, rather than its target's.
link_to(target)	Make the target path a hard link pointing to this path.
lstat()	Like stat(), except if the path points to a symlink, the symlink's status information is returned, rather than its target's.
match(path_pattern)	Return True if this path matches the given pattern.
mkdir([mode, parents, exist_ok])	Create a new directory at this given path.
open([mode, buffering, encoding, errors, ...])	Open the file pointed by this path and return a file object, as the built-in open() function does.
owner()	Return the login name of the file owner.
read_bytes()	Open the file in bytes mode, read it, and close the file.
read_text([encoding, errors])	Open the file in text mode, read it, and close the file.
relative_to(*other)	Return the relative path to another path identified by the passed arguments.
rename(target)	Rename this path to the target path.
replace(target)	Rename this path to the target path, overwriting if that path exists.
resolve([strict])	Make the path absolute, resolving all symlinks on the way and also normalizing it (for example turning slashes into backslashes under Windows).
rglob(pattern)	Recursively yield all existing files (of any kind, including directories) matching the given relative pattern, anywhere in this subtree.
rmdir()	Remove this directory.
samefile(other_path)	Return whether other_path is the same or not as this file (as returned by os.path.samefile()).
stat()	Return the result of the stat() system call on this path, like os.stat() does.
symlink_to(target[, target_is_directory])	Make this path a symlink pointing to the target path.
touch([mode, exist_ok])	Create this file with the given access mode, if it doesn't exist.
unlink([missing_ok])	Remove this file or link.
with_name(name)	Return a new path with the file name changed.
with_suffix(suffix)	Return a new path with the file suffix changed.
write_bytes(data)	Open the file in bytes mode, write to it, and close the file.
write_text(data[, encoding, errors])	Open the file in text mode, write to it, and close the file.

Attributes

anchor	The concatenation of the drive and root, or ''.
drive	The drive prefix (letter or UNC path), if any.
name	The final path component, if any.
parent	The logical parent of the path.
parents	A sequence of this path's logical parents.
parts	An object providing sequence-like access to the components in the filesystem path.
root	The root of the path, if any.
stem	The final path component, minus its last suffix.
suffix	The final component's last suffix, if any.
suffixes	A list of the final component's suffixes, if any.

absolute()

Return an absolute version of this path. This function works even if the path doesn't point to anything.

No normalization is done, i.e. all '.' and '..' will be kept along. Use resolve() to get the canonical path to a file.

chmod(*mode*)

Change the permissions of the path, like os.chmod().

classmethod cwd()

Return a new path pointing to the current working directory (as returned by os.getcwd()).

exists()

Whether this path exists.

expanduser()

Return a new path with expanded ~ and ~user constructs (as returned by os.path.expanduser)

glob(*pattern*)

Iterate over this subtree and yield all existing files (of any kind, including directories) matching the given relative pattern.

group()

Return the group name of the file gid.

classmethod home()

Return a new path pointing to the user's home directory (as returned by os.path.expanduser('~')).

is_block_device()

Whether this path is a block device.

is_char_device()

Whether this path is a character device.

is_dir()

Whether this path is a directory.

is_fifo()

Whether this path is a FIFO.

is_file()

Whether this path is a regular file (also True for symlinks pointing to regular files).

is_mount()

Check if this path is a POSIX mount point

is_socket()

Whether this path is a socket.

is_symlink()

Whether this path is a symbolic link.

iterdir()

Iterate over the files in this directory. Does not yield any result for the special paths '.' and '..'.

lchmod(*mode*)

Like chmod(), except if the path points to a symlink, the symlink's permissions are changed, rather than its target's.

link_to(*target*)

Make the target path a hard link pointing to this path.

Note this function does not make this path a hard link to *target*, despite the implication of the function and argument names. The order of arguments (target, link) is the reverse of Path.symlink_to, but matches that of os.link.

lstat()

Like stat(), except if the path points to a symlink, the symlink's status information is returned, rather than its target's.

mkdir(*mode=511*, *parents=False*, *exist_ok=False*)

Create a new directory at this given path.

open(*mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*)

Open the file pointed by this path and return a file object, as the built-in open() function does.

owner()

Return the login name of the file owner.

read_bytes()

Open the file in bytes mode, read it, and close the file.

read_text(*encoding=None*, *errors=None*)

Open the file in text mode, read it, and close the file.

rename(*target*)

Rename this path to the target path.

The target path may be absolute or relative. Relative paths are interpreted relative to the current working directory, *not* the directory of the Path object.

Returns the new Path instance pointing to the target path.

replace(*target*)

Rename this path to the target path, overwriting if that path exists.

The target path may be absolute or relative. Relative paths are interpreted relative to the current working directory, *not* the directory of the Path object.

Returns the new Path instance pointing to the target path.

resolve(*strict=False*)

Make the path absolute, resolving all symlinks on the way and also normalizing it (for example turning slashes into backslashes under Windows).

rglob(*pattern*)

Recursively yield all existing files (of any kind, including directories) matching the given relative pattern, anywhere in this subtree.

rmdir()

Remove this directory. The directory must be empty.

samefile(*other_path*)

Return whether other_path is the same or not as this file (as returned by os.path.samefile()).

stat()

Return the result of the stat() system call on this path, like os.stat() does.

symlink_to(*target*, *target_is_directory=False*)

Make this path a symlink pointing to the target path. Note the order of arguments (link, target) is the reverse of os.symlink.

touch(*mode=438*, *exist_ok=True*)

Create this file with the given access mode, if it doesn't exist.

unlink(*missing_ok=False*)

Remove this file or link. If the path is a directory, use rmdir() instead.

write_bytes(*data*)

Open the file in bytes mode, write to it, and close the file.

write_text(*data*, *encoding=None*, *errors=None*)

Open the file in text mode, write to it, and close the file.

pytorch_pfn_extras.profiler

Functions

pytorch_pfn_extras.profiler.get_time_summary()
pytorch_pfn_extras.profiler.record(tag[, ...])

pytorch_pfn_extras.profiler.record_function(tag)
pytorch_pfn_extras.profiler.record_iterable(...)

pytorch_pfn_extras.profiler.get_time_summary

pytorch_pfn_extras.profiler.**get_time_summary**()

Return type
TimeSummary

pytorch_pfn_extras.profiler.record

pytorch_pfn_extras.profiler.**record**(*tag*, *metric=None*, *use_cuda=False*, *enable=True*, *device='cpu'*)

Parameters

- **tag** (*Optional[str]*) –
- **metric** (*Optional[str]*) –
- **use_cuda** (*bool*) –
- **enable** (*bool*) –
- **device** (*DeviceLike*) –

Return type
Generator[*_ReportNotification*, None, None]

pytorch_pfn_extras.profiler.record_function

pytorch_pfn_extras.profiler.**record_function**(*tag*, *use_cuda=False*, *enable=True*)

Parameters

- **tag** (*Optional[str]*) –
- **use_cuda** (*bool*) –
- **enable** (*bool*) –

Return type
Callable[[*Callable*[[...], *_T*]], *Callable*[[...], *_T*]]

pytorch_pfn_extras.profiler.record_iterable

pytorch_pfn_extras.profiler.**record_iterable**(*tag*, *iter*, *divide_metric=False*, *use_cuda=False*, *enable=True*)

Parameters

- **tag** (*Optional[str]*) –
- **iter** (*Iterable[_T]*) –
- **divide_metric** (*bool*) –
- **use_cuda** (*bool*) –
- **enable** (*bool*) –

Return type
Iterable[*_T*]

Classes

pytorch_pfn_extras.profiler.TimeSummary(*[, ...])	Online summarization of execution times.

pytorch_pfn_extras.profiler.TimeSummary

class pytorch_pfn_extras.profiler.**TimeSummary**(*, *max_queue_size=1000*, *auto_init=True*)

Bases: `object`

Online summarization of execution times.

TimeSummary computes the average and standard deviation of exeuction times in both cpu and gpu devices.

Parameters

- **max_queue_size** (*int*) – Length limit of the internal queues that keep reported time info until they are summarized.
- **auto_init** (*bool*) – Whether to automatically call *initialize()* when the instance is created.

Methods

__init__(*[, max_queue_size, auto_init])	
add(name, value)	
complete_report(tag, use_cuda, begin_event, ...)	
finalize()	
initialize()	Initializes the worker threads for TimeSummary.
report(tag[, use_cuda])	Context manager to automatically report execution times.
summary([clear])	
synchronize()	

__init__(*, *max_queue_size=1000*, *auto_init=True*)

Parameters

- **max_queue_size** (*int*) –
- **auto_init** (*bool*) –

Return type
None

add(*name*, *value*)

Parameters

- **name** (*str*) –
- **value** (*float*) –

Return type
None

complete_report(*tag*, *use_cuda*, *begin_event*, *begin*)

Parameters

- **tag** (*str*) –
- **use_cuda** (*bool*) –
- **begin_event** (*Optional[Event]*) –
- **begin** (*float*) –

Return type
None

finalize()

Return type
None

initialize()

Initializes the worker threads for TimeSummary.

Usually you do not have to call it for yourself. However in case you directly use `ppe.time_summary` outside of *pytorch_pfn_extras.training.extensions.ProfileReport*, you have to explicitly call `initialize()` in advance.

Return type
None

report(*tag*, *use_cuda=False*)

Context manager to automatically report execution times.

The start and completion times are obtained automatically, the user only needs to provide a tag to identify the value in the summary values.

Parameters

- **tag** (*str*) – A name to identify the section of code being profiled.
- **use_cuda** (*bool*) – Indicates if GPU time should also be profiled.

Return type
Generator[*_ReportNotification*, None, None]

summary(*clear=False*)

Parameters
clear (*bool*) –

Return type
Generator[*Tuple*[DictSummary, *Dict*[str, float]], None, None]

synchronize()

Return type
None

pytorch_pfn_extras.reporting

Functions

pytorch_pfn_extras.reporting.get_current_reporter()	Returns the current reporter object.
pytorch_pfn_extras.reporting.overload(func)	Decorator for overloaded functions/methods.
pytorch_pfn_extras.reporting.report(values)	Reports observed values with the current reporter object.
pytorch_pfn_extras.reporting.report_scope(...)	Returns a report scope with the current reporter.

pytorch_pfn_extras.reporting.get_current_reporter

pytorch_pfn_extras.reporting.**get_current_reporter**()

Returns the current reporter object.

Return type
Reporter

pytorch_pfn_extras.reporting.overload

pytorch_pfn_extras.reporting.**overload**(*func*)

Decorator for overloaded functions/methods.

In a stub file, place two or more stub definitions for the same function in a row, each decorated with @overload. For example:

@overload def utf8(value: None) -> None: … @overload def utf8(value: bytes) -> bytes: … @overload def utf8(value: str) -> bytes: …

In a non-stub file (i.e. a regular .py file), do the same but follow it with an implementation. The implementation should *not* be decorated with @overload. For example:

@overload def utf8(value: None) -> None: … @overload def utf8(value: bytes) -> bytes: … @overload def utf8(value: str) -> bytes: … def utf8(value):

implementation goes here

pytorch_pfn_extras.reporting.report

pytorch_pfn_extras.reporting.**report**(*values*, *observer=None*)

Reports observed values with the current reporter object.

Any reporter object can be set current by the `with` statement. This function calls the *Reporter.report()* method of the current reporter. If no reporter object is current, this function does nothing.

Example

The most typical example is a use within *nn.Module*. Suppose that a module is registered to the current reporter as an observer (for example, the target module of the optimizer is automatically registered to the main reporter. We can report some values from the link as follows:

```
class MyRegressor:
    def __init__(self, predictor):
        super().__init__(predictor=predictor)

    def __call__(self, x, y):
        # This chain just computes the mean absolute and squared
        # errors between the prediction and y.
        pred = self.predictor(x)
        abs_error = F.sum(abs(pred - y)) / len(x)
        loss = F.mean_squared_error(pred, y)

        # Report the mean absolute and squared errors.
        reporter.report({
            'abs_error': abs_error,
            'squared_error': loss,
        }, self)

        return loss
```

If the module is named `'main'` in the hierarchy these reported values are named `'main/abs_error'` and `'main/squared_error'`.

Parameters

- **values** (*dict*) – Dictionary of observed values.
- **observer** (*Optional[Module]*) – Observer object. Its object ID is used to retrieve the observer name, which is used as the prefix of the registration name of the observed value.

Return type
None

pytorch_pfn_extras.reporting.report_scope

pytorch_pfn_extras.reporting.**report_scope**(*observation*)

Returns a report scope with the current reporter.

This is equivalent to `get_current_reporter().scope(observation)`, except that it does not make the reporter current redundantly.

Parameters
observation (*Dict[str, Union[Tensor, ndarray, floating, float, Callable[[], float]]]*) –

Return type
Generator[None, None, None]

Classes

pytorch_pfn_extras.reporting.DictSummary()	Online summarization of a sequence of dictionaries.
pytorch_pfn_extras.reporting.Reporter()	Object to which observed values are reported.
pytorch_pfn_extras.reporting.Summary()	Online summarization of a sequence of scalars.

pytorch_pfn_extras.reporting.DictSummary

class pytorch_pfn_extras.reporting.**DictSummary**

Bases: `object`

Online summarization of a sequence of dictionaries.

`DictSummary` computes the statistics of a given set of scalars online. It only computes the statistics for scalar values and variables of scalar values in the dictionaries.

Methods

__init__()	
add(d)	Adds a dictionary of scalars.
compute_mean()	Creates a dictionary of mean values.
load_state_dict(to_load)	
make_statistics()	Creates a dictionary of statistics.
state_dict()	

__init__()

Return type
None

add(*d*)

Adds a dictionary of scalars.

Parameters
d (*dict*) – Dictionary of scalars to accumulate. Only elements of scalars, zero-dimensional arrays, and variables of zero-dimensional arrays are accumulated. When the value is a tuple, the second element is interpreted as a weight.

Return type
None

compute_mean()

Creates a dictionary of mean values.

It returns a single dictionary that holds a mean value for each entry added to the summary.

Returns
Dictionary of mean values.

Return type
dict

`load_state_dict`(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

`make_statistics`()

Creates a dictionary of statistics.

It returns a single dictionary that holds mean and standard deviation values for every entry added to the summary. For an entry of name `'key'`, these values are added to the dictionary by names `'key'` and `'key.std'`, respectively.

Returns
Dictionary of statistics of all entries.

Return type
dict

`state_dict`()

Return type
Dict[str, *Any*]

pytorch_pfn_extras.reporting.Reporter

`class` pytorch_pfn_extras.reporting.**Reporter**

Bases: `object`

Object to which observed values are reported.

Reporter is used to collect values that users want to watch. The reporter object holds a mapping from value names to the actually observed values. We call this mapping *observations*.

When a value is passed to the reporter, an object called *observer* can be optionally attached. In this case, the name of the observer is added as the prefix of the value name. The observer name should be registered beforehand.

See the following example:

```
>>> from pytorch_pfn_extras.reporting import Reporter, report, report_scope
>>>
>>> reporter = Reporter()
>>> observer = object()  # it can be an arbitrary (reference) object
>>> reporter.add_observer('my_observer', observer)
>>> observation = {}
>>> with reporter.scope(observation):
...     reporter.report({'x': 1}, observer)
...
>>> observation
{'my_observer/x': 1}
```

There are also a global API to add values:

```
>>> reporter = Reporter()
>>> observation = {}
>>> with reporter:
...     with report_scope(observation):
...         report({'x': 1})
...
>>> observation
{'x': 1}
```

The most important application of Reporter is to report observed values from each link or chain in the training and validation procedures. and some extensions prepare their own Reporter object with the hierarchy of the target module registered as observers. We can use *report()* function inside any nn.Module to report the observed values (e.g., training loss, accuracy, activation statistics, etc.).

observation

Dictionary of observed values.

Methods

__init__()	

add_observer(name, observer)	Registers an observer of values.
add_observers(prefix, observers)	Registers multiple observers at once.
report(values[, observer])	Reports observed values.
scope(observation)	Creates a scope to report observed values to `observation`.

__init__()

Return type
None

add_observer(*name*, *observer*)

Registers an observer of values.

Observer defines a scope of names for observed values. Values observed with the observer are registered with names prefixed by the observer name.

Parameters

- **name** (*str*) – Name of the observer.
- **observer** (*Module*) – The observer object. Note that the reporter distinguishes the observers by their object ids (i.e., `id(owner)`), rather than the object equality.

Return type
None

add_observers(*prefix*, *observers*)

Registers multiple observers at once.

This is a convenient method to register multiple objects at once.

Parameters

- **prefix** (*str*) – Prefix of each name of observers.
- **observers** (*Sequence[Tuple[str, Module]]*) – Iterator of name and observer pairs.

Return type
None

report(*values*, *observer=None*)

Reports observed values.

The values are written with the key, prefixed by the name of the observer object if given.

Note: If a value is of type `Tensor`, the variable is copied without preserving the computational graph and the new variable object purged from the graph is stored to the observer.

Parameters

- **values** (*dict*) – Dictionary of observed values.
- **observer** (*Optional[Module]*) – Observer object. Its object ID is used to retrieve the observer name, which is used as the prefix of the registration name of the observed value.

Return type
None

scope(*observation*)

Creates a scope to report observed values to `observation`.

This is a context manager to be passed to `with` statements. In this scope, the observation dictionary is changed to the given one.

It also makes this reporter object current.

Parameters
observation (*dict*) – Observation dictionary. All observations reported inside of the `with` statement are written to this dictionary.

Return type
Generator[None, None, None]

pytorch_pfn_extras.reporting.Summary

class pytorch_pfn_extras.reporting.**Summary**

Bases: `object`

Online summarization of a sequence of scalars.

Summary computes the statistics of given scalars online.

Methods

__init__()	
add(value[, weight])	Adds a scalar value.
compute_mean()	Computes the mean.
load_state_dict(to_load)	
make_statistics()	Computes and returns the mean and standard deviation values.
state_dict()	

__init__()

Return type
None

add(*value*, *weight=1*)

Adds a scalar value.

Parameters

- **value** (*Union[Tensor, ndarray, floating, float, Callable[[], float]]*) – Scalar value to accumulate. It is either a NumPy scalar or a zero-dimensional array (on CPU or GPU).
- **weight** (*Union[Tensor, ndarray, floating, float]*) – An optional weight for the value. It is a NumPy scalar or a zero-dimensional array (on CPU or GPU). Default is 1 (integer).

Return type
None

compute_mean()

Computes the mean.

Return type
Union[*Tensor*, *ndarray*, *floating*, float]

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

make_statistics()

Computes and returns the mean and standard deviation values.

Returns
Mean and standard deviation values.

Return type
tuple

state_dict()

Return type
Dict[str, *Any*]

pytorch_pfn_extras.runtime

Classes

pytorch_pfn_extras.runtime.BaseRuntime(...)	A base class for collections of device-specific callback functions.
pytorch_pfn_extras.runtime.PyTorchRuntime(...)	A collections of callback functions for the devices that PyTorch supports by default.

pytorch_pfn_extras.runtime.BaseRuntime

class pytorch_pfn_extras.runtime.**BaseRuntime**(*device_spec*, *options*)

Bases: object

A base class for collections of device-specific callback functions.

The function attributes of this class will be called from ppe.to or ppe.handler.Handler.

ppe.runtime.runtime_registry stores the runtime classes and dispatches them by feeding the corresponding name string as an input.

Parameters

- **device_spec** (*torch.device or str*) – The device that modules and tensors are transferred to.
- **options** (*dict*) – A configuration dictionary that can be used from runtime method.

Methods

__init__(device_spec, options)	

convert_batch(args)	Transfers the given batch to the specific device.
eval_post_step(evaluator, module, batch_idx, ...)	The method called at the end of each evaluation.
eval_pre_step(evaluator, module, batch_idx, ...)	The method called at the beginning of each evaluation.
execute(code_block, batch)	Method called by the CodeBlocks API to do device dependent execution.
initialize_module(module, loader_or_batch[, ...])	Initializes the module at the beginning of training or inference.
map(func, iterable[, out_keys, device])	Method called by the user to apply function to iterable efficiently.
move_module(module)	Transfers the module to the specific device.
move_tensor(tensor)	Transfers the tensor to the specific device.
trace(event_name, arg)	Context manager for tracing PPE events in the custom device tools.
train_cleanup(module)	A method called only once when compleing a training run.
train_epoch_begin(module)	Preprocess of each epoch.
train_epoch_end(module)	Completion of each epoch.
train_post_step(trainer, module, batch_idx, ...)	Postprocess of each step.
train_pre_step(trainer, module, batch_idx, batch)	Preprocess of each step.
train_validation_begin(module)	The method called before each evaluation.
train_validation_end(module)	The method called after each evaluation.

__init__(*device_spec*, *options*)

Parameters

- **device_spec** (*Union[str, device]*) –
- **options** (*Dict[str, Any]*) –

Return type
None

convert_batch(*args*)

Transfers the given batch to the specific device.

Parameters
args (*object*) – A batch data of any type.

Returns
A batch data transferred to the specific device of the same type as input.

Return type
Any

eval_post_step(*evaluator*, *module*, *batch_idx*, *batch*, *outs*)

The method called at the end of each evaluation.

Parameters

- **evaluator** (Evaluator) – An evaluator.
- **module** (*torch.nn.Module*) – A module.

- **batch_idx** (*int*) – The batch index.
- **batch** (*list of torch.Tensor*) – The list of input tensors of this batch.
- **outs** (*Any*) – (list of torch.Tensor): The list of output tensors of this batch.

Return type
None

Returns: None

eval_pre_step(*evaluator*, *module*, *batch_idx*, *batch*)

The method called at the beginning of each evaluation.

Parameters

- **evaluator** (Evaluator) – An evaluator.
- **module** (*torch.nn.Module*) – A module.
- **batch_idx** (*int*) – The batch index.
- **batch** (*list of torch.Tensor*) – The list of input tensors of this batch.

Return type
None

Returns: None

execute(*code_block*, *batch*)

Method called by the CodeBlocks API to do device dependent execution.

Parameters

- **code_block** (CodeBlock) – The codeblock requesting execution.
- **batch** (*dict of str, torch.Tensor*) – The input tensors of this batch.

Returns
The results of executing the codeblock on this runtime.

Return type
Any

initialize_module(*module*, *loader_or_batch*, *optimizer=None*)

Initializes the module at the beginning of training or inference.

Parameters

- **module** (*torch.nn.Module*) – A module.
- **loader_or_batch** (DataLoader *or torch.Tensor*) – A data loader or a tensor.
- **optimizer** (*Optimizer or None*) – An optimizer. This argument is sometimes used to copy LR from the original optimizer to the training model.

Return type
None

Returns: None

map(*func*, *iterable*, *out_keys=None*, *device='cpu'*)

Method called by the user to apply function to iterable efficiently.

Parameters

- **func** (CodeBlock) – The function to be executed

- **iterable** (*Iterable[Any]*) – The data
- **out_keys** (*Optional[Set[str]]*) – The output keys that to be moved to the host device
- **device** (*Any*) – The torch device that contains the final outputs

Returns
The result of *func*

Return type
Iterable[*Any*]

move_module(*module*)

Transfers the module to the specific device.

Before this method is called, `ppe.to` will add this class as an new attribute ("_ppe_runtime") to the input module.

Parameters
module (*torch.nn.Module*) – A module.

Returns
A module transferred to the specific device.

Return type
Module

move_tensor(*tensor*)

Transfers the tensor to the specific device.

Parameters
tensor (*torch.Tensor*) – A tensor.

Returns
A tensor transferred to the specific device.

Return type
Tensor

classmethod trace(*event_name*, *arg*)

Context manager for tracing PPE events in the custom device tools.

Parameters

- **event_name** (*Optional[str]*) – The name of the event being traced
- **arg** (*Any*) – Custom argument for the tracer

Return type
Generator[None, None, None]

train_cleanup(*module*)

A method called only once when compleing a training run.

Parameters
module (*torch.nn.Module*) – A module.

Return type
None

Returns: None

`train_epoch_begin`(*module*)

Preprocess of each epoch.

Parameters
module (*torch.nn.Module*) – A module.

Return type
None

Returns: None

`train_epoch_end`(*module*)

Completion of each epoch.

Parameters
module (*torch.nn.Module*) – A module.

Return type
None

Returns: None

`train_post_step`(*trainer*, *module*, *batch_idx*, *batch*, *outs*)

Postprocess of each step.

This method is called at the end of every steps: the set of (typically one) iterations and an update.

Parameters

- **trainer** (Trainer) – A trainer.
- **module** (*torch.nn.Module*) – A module.
- **batch_idx** (*int*) – The batch index.
- **batch** (*list of torch.Tensor*) – The list of input tensors of this batch.
- **outs** (*Any*) – (list of torch.Tensor): The list of output tensors of this batch.

Return type
None

Returns: None

`train_pre_step`(*trainer*, *module*, *batch_idx*, *batch*)

Preprocess of each step.

This method is called at the beginning of every steps: the set of (typically one) iterations and an update.

Parameters

- **trainer** (Trainer) – A trainer.
- **module** (*torch.nn.Module*) – A module.
- **batch_idx** (*int*) – The batch index.
- **batch** (*list of torch.Tensor*) – The list of input tensors of this batch.

Return type
None

Returns: None

`train_validation_begin`(*module*)

The method called before each evaluation.

Parameters
module (*torch.nn.Module*) – A module.

Return type
None

Returns: None

`train_validation_end`(*module*)

The method called after each evaluation.

Parameters
module (*torch.nn.Module*) – A module.

Return type
None

Returns: None

pytorch_pfn_extras.runtime.PyTorchRuntime

`class` pytorch_pfn_extras.runtime.`PyTorchRuntime`(*device_spec*, *options*)

Bases: *BaseRuntime*

A collections of callback functions for the devices that PyTorch supports by default.

Parameters

- **device_spec** (*torch.device or str*) – The device.
- **options** (*dict, optional*) – The configuration options.
  - **'autocast' (bool or dict):**
    If `True`, `torch.cuda.amp.autocast` is enabled. using `{"enabled": True, "device_type": "cuda"}` as autocast options. Default is `False` which corresponds to the following options `{"enabled": False, "device_type": "cuda"}` dict type. If dict, Options to pass to `torch.autocast`. Includes `device_type`, `dtype` among others.
  - **'grad_scaler' (torch.cuda.amp.GradScaler):**
    A gradient scaler that outputs are applied to.

Methods

__init__(device_spec, options)	

convert_batch(args)	Transfers the given batch to the specific device.
eval_post_step(evaluator, module, batch_idx, ...)	The method called at the end of each evaluation.
eval_pre_step(evaluator, module, batch_idx, ...)	The method called at the beginning of each evaluation.
execute(code_block, batch)	Method called by the CodeBlocks API to do device dependent execution.
initialize_module(module, loader_or_batch[, ...])	Initializes the module at the beginning of training or inference.
map(func, iterable[, out_keys, device])	Method called by the user to apply function to iterable efficiently.
move_module(module)	Transfers the module to the specific device.
move_tensor(tensor)	Transfers the tensor to the specific device.
trace(event_name, arg)	Context manager for tracing PPE events in the custom device tools.
train_cleanup(module)	A method called only once when compleing a training run.
train_epoch_begin(module)	Preprocess of each epoch.
train_epoch_end(module)	Completion of each epoch.
train_post_step(trainer, module, batch_idx, ...)	Postprocess of each step.
train_pre_step(trainer, module, batch_idx, batch)	Preprocess of each step.
train_validation_begin(module)	The method called before each evaluation.
train_validation_end(module)	The method called after each evaluation.

__init__(*device_spec*, *options*)

Parameters

- **device_spec** (*Union[str, device]*) –
- **options** (*Dict[str, Any]*) –

Return type
None

eval_post_step(*evaluator*, *module*, *batch_idx*, *batch*, *outs*)

The method called at the end of each evaluation.

Parameters

- **evaluator** (Evaluator) – An evaluator.
- **module** (*torch.nn.Module*) – A module.
- **batch_idx** (*int*) – The batch index.
- **batch** (*list of torch.Tensor*) – The list of input tensors of this batch.
- **outs** (*Any*) – (list of torch.Tensor): The list of output tensors of this batch.

Return type
None

Returns: None

eval_pre_step(*evaluator*, *module*, *batch_idx*, *batch*)

The method called at the beginning of each evaluation.

Parameters

- **evaluator** (Evaluator) – An evaluator.
- **module** (*torch.nn.Module*) – A module.
- **batch_idx** (*int*) – The batch index.
- **batch** (*list of torch.Tensor*) – The list of input tensors of this batch.

Return type
None

Returns: None

execute(*code_block*, *batch*)

Method called by the CodeBlocks API to do device dependent execution.

Parameters

- **code_block** (CodeBlock) – The codeblock requesting execution.
- **batch** (*dict of str, torch.Tensor*) – The input tensors of this batch.

Returns
The results of executing the codeblock on this runtime.

Return type
Any

initialize_module(*module*, *loader_or_batch*, *optimizer=None*)

Initializes the module at the beginning of training or inference.

Parameters

- **module** (*torch.nn.Module*) – A module.
- **loader_or_batch** (DataLoader *or torch.Tensor*) – A data loader or a tensor.
- **optimizer** (*Optimizer or None*) – An optimizer. This argument is sometimes used to copy LR from the original optimizer to the training model.

Return type
None

Returns: None

map(*func*, *iterable*, *out_keys=None*, *device='cpu'*)

Method called by the user to apply function to iterable efficiently.

Parameters

- **func** (CodeBlock) – The function to be executed
- **iterable** (*Iterable[Any]*) – The data
- **out_keys** (*Optional[Set[str]]*) – The output keys that to be moved to the host device
- **device** (*Any*) – The torch device that contains the final outputs

Returns
The result of *func*

Return type
Iterable[*Any*]

`move_module`(*module*)

Transfers the module to the specific device.

Before this method is called, `ppe.to` will add this class as an new attribute ("_ppe_runtime") to the input module.

Parameters
module (*torch.nn.Module*) – A module.

Returns
A module transferred to the specific device.

Return type
Module

`move_tensor`(*tensor*)

Transfers the tensor to the specific device.

Parameters
tensor (*torch.Tensor*) – A tensor.

Returns
A tensor transferred to the specific device.

Return type
Tensor

`classmethod trace`(*event_name*, *arg*)

Context manager for tracing PPE events in the custom device tools.

Parameters

- **event_name** (*Optional[str]*) – The name of the event being traced
- **arg** (*Any*) – Custom argument for the tracer

Return type
Generator[None, None, None]

`train_cleanup`(*module*)

A method called only once when compleing a training run.

Parameters
module (*torch.nn.Module*) – A module.

Return type
None

Returns: None

`train_epoch_begin`(*module*)

Preprocess of each epoch.

Parameters
module (*torch.nn.Module*) – A module.

Return type
None

Returns: None

train_epoch_end(*module*)

Completion of each epoch.

Parameters
module (*torch.nn.Module*) – A module.

Return type
None

Returns: None

train_post_step(*trainer*, *module*, *batch_idx*, *batch*, *outs*)

Postprocess of each step.

This method is called at the end of every steps: the set of (typically one) iterations and an update.

Parameters

- **trainer** (Trainer) – A trainer.
- **module** (*torch.nn.Module*) – A module.
- **batch_idx** (*int*) – The batch index.
- **batch** (*list of torch.Tensor*) – The list of input tensors of this batch.
- **outs** (*Any*) – (list of torch.Tensor): The list of output tensors of this batch.

Return type
None

Returns: None

train_pre_step(*trainer*, *module*, *batch_idx*, *batch*)

Preprocess of each step.

This method is called at the beginning of every steps: the set of (typically one) iterations and an update.

Parameters

- **trainer** (Trainer) – A trainer.
- **module** (*torch.nn.Module*) – A module.
- **batch_idx** (*int*) – The batch index.
- **batch** (*list of torch.Tensor*) – The list of input tensors of this batch.

Return type
None

Returns: None

train_validation_begin(*module*)

The method called before each evaluation.

Parameters
module (*torch.nn.Module*) – A module.

Return type
None

Returns: None

`train_validation_end`(*module*)

The method called after each evaluation.

Parameters
module (*torch.nn.Module*) – A module.

Return type
None

Returns: None

pytorch_pfn_extras.testing

pytorch_pfn_extras.torchscript

Functions

pytorch_pfn_extras.torchscript.find_inplace(g)
pytorch_pfn_extras.torchscript.run_jit_pass(p, ...)

pytorch_pfn_extras.torchscript.find_inplace

pytorch_pfn_extras.torchscript.**find_inplace**(*g*)

Parameters
g (*Graph*) –

Return type
Tuple[*Graph*, *List*[*Node*]]

pytorch_pfn_extras.torchscript.run_jit_pass

pytorch_pfn_extras.torchscript.**run_jit_pass**(*p*, *g*, **args*, ***kwargs*)

Parameters

- **p** (*Callable*) –
- **g** (*Graph*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

pytorch_pfn_extras.training

Functions

pytorch_pfn_extras.training.make_extension([...])	Decorator to make given function into an extension.

pytorch_pfn_extras.training.make_extension

pytorch_pfn_extras.training.**make_extension**(*trigger=(1, 'iteration')*, *default_name=None*, *priority=100*, *finalizer=<function <lambda>>*, *initializer=<function <lambda>>*, *on_error=<function <lambda>>*)

Decorator to make given function into an extension.

This decorator just adds some attributes to a given function. The value of the attributes are given by the arguments of this decorator.

See *Extension* for details of extensions. Most of the default values of arguments also follow those for this class.

Parameters

- **trigger** (*TriggerLike*) – Default trigger of the extension.
- **default_name** (*Optional[str]*) – Default name of the extension. The name of a given function is used by default.
- **priority** (*int*) – Default priority of the extension.
- **finalizer** (*ExtensionLike*) – Finalizer function of this extension. It is called at the end of the training loop.
- **initializer** (*ExtensionLike*) – Initializer function of this extension. It is called at the beginning of the training loop.
- **on_error** (*Callable[[ExtensionsManagerProtocol, Exception, TracebackType], None]*) – Error handler callback function of this extension. It is called after an error is raised during the training loop.

Return type

Callable[[ExtensionLike], ExtensionLike]

Classes

pytorch_pfn_extras.training.AccuracyMetric(...)	A metric for an evaluator to report accuracy.
pytorch_pfn_extras.training.DistributedEvaluator(...)	
pytorch_pfn_extras.training.Evaluator(...[, ...])	
pytorch_pfn_extras.training.Extension()	Base class of extensions.
pytorch_pfn_extras.training.ExtensionEntry(...)	Extension and options.
pytorch_pfn_extras.training.ExtensionsManager(...)	Manages the extensions and the current status.
pytorch_pfn_extras.training.ExtensionsManagerProtocol(...)	
pytorch_pfn_extras.training.IgniteExtensionsManager(...)	Manages extensions and the current status in Ignite training loop.
pytorch_pfn_extras.training.StateObjectProtocol(...)	
pytorch_pfn_extras.training.Trainer(handler, ...)	

pytorch_pfn_extras.training.AccuracyMetric

class pytorch_pfn_extras.training.**AccuracyMetric**(*label_key*, *output_key*)

Bases: `object`

A metric for an evaluator to report accuracy.

Parameters

- **label_key** (*str*) – The key name of label.
- **output_key** (*str*) – The key name of prediction.

Methods

__init__(label_key, output_key)	

__call__(*batch*, *out*)

Call self as a function.

Parameters

- **batch** (*Dict[str, Tensor]*) –
- **out** (*Dict[str, Tensor]*) –

Return type

Dict[str, *Any*]

__init__(*label_key*, *output_key*)

Parameters

- **label_key** (*str*) –
- **output_key** (*str*) –

Return type
None

pytorch_pfn_extras.training.DistributedEvaluator

class pytorch_pfn_extras.training.**DistributedEvaluator**(*handler*, *models*, *, *progress_bar=False*, *metrics=None*, *profile=None*)

Bases: *Evaluator*

Methods

__init__(handler, models, *[, progress_bar, ...])	
run(loader, *[, eval_len])	Executes the evaluation loop.

Parameters

- **handler** (BaseHandler) –
- **models** (*Union[Module, Mapping[str, Module]]*) –
- **progress_bar** (*bool*) –
- **metrics** (*Optional[Sequence[MetricType]]*) –
- **profile** (*Optional[profile]*) –

__init__(*handler*, *models*, *, *progress_bar=False*, *metrics=None*, *profile=None*)

Parameters

- **handler** (BaseHandler) –
- **models** (*Union[Module, Mapping[str, Module]]*) –
- **progress_bar** (*bool*) –
- **metrics** (*Optional[Sequence[MetricType]]*) –
- **profile** (*Optional[profile]*) –

pytorch_pfn_extras.training.Evaluator

class pytorch_pfn_extras.training.**Evaluator**(*handler*, *models*, *, *progress_bar=False*, *metrics=None*, *profile=None*)

Bases: object

Methods

__init__(handler, models, *[, progress_bar, ...])	
run(loader, *[, eval_len])	Executes the evaluation loop.

Parameters

- **handler** (BaseHandler) –
- **models** (*Union[Module, Mapping[str, Module]]*) –
- **progress_bar** (*bool*) –
- **metrics** (*Optional[Sequence[MetricType]]*) –
- **profile** (*Optional[profile]*) –

__init__(*handler*, *models*, *, *progress_bar=False*, *metrics=None*, *profile=None*)

Parameters

- **handler** (BaseHandler) –
- **models** (*Union[Module, Mapping[str, Module]]*) –
- **progress_bar** (*bool*) –
- **metrics** (*Optional[Sequence[MetricType]]*) –
- **profile** (*Optional[profile]*) –

run(*loader*, *, *eval_len=None*)

Executes the evaluation loop.

Parameters

- **loader** (*torch.utils.data.DataLoader*) – A data loader for evaluation.
- **eval_len** (*int, optional*) – The number of iterations per one evaluation epoch.

Return type

None

pytorch_pfn_extras.training.Extension

class pytorch_pfn_extras.training.**Extension**

Bases: `object`

Base class of extensions.

An extension is a callable object that takes the manager object as the argument. It also provides some default configurations as its attributes, e.g. the default trigger and the default priority. This class provides a set of typical default values for these attributes.

There are three ways to define users' own extensions: inheriting this class, decorating closures by *make_extension()*, or using any callable including lambda functions as extensions. Decorator can slightly reduce the overhead and is much easier to use, while this class provides more flexibility (for example, it can have methods to configure the behavior). Using a lambda function allows one-line coding for simple purposes, but users have to specify the configurations as arguments to `ExtensionsManager.extend()`. For a callable not inheriting this class, the default configurations of this class are used unless the user explicitly specifies them in `ExtensionsManager.extend()` method.

trigger

Default value of trigger for this extension. It is set to `(1, 'iteration')` by default.

Type
TriggerLike

priority

Default priority of the extension. It is set to `PRIORITY_READER` by default.

Type
int

~Extension.name

Name of the extension. It is set to `None` by default. This value will be overwritten when registering an extension to a manager. See `pytorch_pfn_extras.ExtensionsManager.extend()` for details.

Methods

`__init__`()	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
Any

property default_name: str

Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

finalize(*manager*)

Finalizes the extension.

This method is called at the end of the training loop.

Parameters
manager (ExtensionsManagerProtocol) –

Return type
None

initialize(*manager*)

Initializes up the manager state.

This method is called before entering the training loop. An extension modifying the state of ExtensionsManager can override this method to initialize it.

When the manager has been restored from a snapshot, this method has to recover an appropriate part of the state of the manager.

Parameters
manager (ExtensionsManager) – Manager object to call this extension.

Return type
None

is_async = False

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

name: Optional[str] = None

needs_model_state = False

on_error(*manager*, *exc*, *tb*)

Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **manager** (ExtensionsManager) –
- **extension.** (*Manager object to call this*) –
- **exc** (*Exception*) – arbitrary exception thrown during update loop.
- **tb** (*traceback*) – traceback object of the exception

Return type
None

priority: int = 100

state_dict()

Serializes the extension state.

It is called when a manager that owns this extension is serialized. It serializes nothing by default.

Return type
Dict[str, *Any*]

trigger: TriggerLike = (1, 'iteration')

pytorch_pfn_extras.training.ExtensionEntry

class pytorch_pfn_extras.training.**ExtensionEntry**(*extension*, *, *name=None*, *priority=None*, *trigger=None*, *call_before_training=False*)

Bases: object

Extension and options. When name, priority, or trigger is not specified, it is copied from the attributes of the given extension.

Parameters

- **extension** (*ExtensionLike*) – An extension.
- **name** (*Optional[str]*) – Name of extension.
- **priority** (*Optional[int]*) – Invocation priority of the extension.

- **trigger** (*Optional[TriggerLike]*) – Trigger object that determines when to invoke the extension.
- **call_before_training** (*bool*) – Flag to call extension before training.

See also:

`pytorch_pfn_extras.training.ExtensionsManager.extend()`

Methods

`__init__`(extension, *[, name, priority, ...])
`load_state_dict`(to_load)
`state_dict`()

__init__(*extension*, *, *name=None*, *priority=None*, *trigger=None*, *call_before_training=False*)

Parameters

- **extension** (*ExtensionLike*) –
- **name** (*Optional[str]*) –
- **priority** (*Optional[int]*) –
- **trigger** (*Optional[TriggerLike]*) –
- **call_before_training** (*bool*) –

Return type
None

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

state_dict()

Return type
Dict[str, *Any*]

pytorch_pfn_extras.training.ExtensionsManager

class pytorch_pfn_extras.training.**ExtensionsManager**(*models*, *optimizers*, *max_epochs*, *, *iters_per_epoch*, *extensions=None*, *out_dir='result'*, *stop_trigger=None*, *writer=None*, *transform_model=<function ExtensionsManager.<lambda>>*, *enable_profile=False*, *state_objects={}*)

Bases: _BaseExtensionsManager

Manages the extensions and the current status.

Parameters

- **models** (dict or *torch.nn.Module*) – Map of string to Module or an actual Module
- **optimizers** (dict or *torch.Optimizer*) – Map of string to Optimizer or an actual Optimizer.
- **max_epochs** (*int*) – Number of epochs in the whole training loop. Ignored if *stop_trigger* is passed as a kwarg.
- **iters_per_epoch** (*int*) – Number of iterations in one epoch.
- **extensions** (*list or None*) – List of Extentions to be used.
- **out_dir** (*str*) – Output directory (default: result).
- **stop_trigger** (*trigger object, optional*) – to determine wether training has concluded. The default is an interval trigger set to *max_epochs*
- **writer** (*writing.Writer object*) – Writer that can be used by extensions to write data to custom filesystems.
- **enable_profile** (*bool*) – Flag to enable/disable profiling of iterations. Default is *False*.
- **transform_model** (*Callable[[str, Module], Module]*) –
- **state_objects** (*Dict[str, StateObjectProtocol]*) –

Methods

__init__(models, optimizers, max_epochs, *, ...)	
extend(extension[, name, trigger, priority, ...])	Registers an extension to the manager.
finalize()	
get_extension(name)	Returns the extension of a given name.
load_state_dict(to_load)	
needs_model_state([iteration])	
needs_state_this_iteration()	
run_extensions()	
run_iteration(*[, step_optimizers])	Context manager to run an iteration.
start_extensions()	
state_dict()	

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
optimizers
out
raw_models
stop_trigger
updater

__init__(*models*, *optimizers*, *max_epochs*, ***, *iters_per_epoch*, *extensions=None*, *out_dir='result'*, *stop_trigger=None*, *writer=None*, *transform_model=<function ExtensionsManager.<lambda>>*, *enable_profile=False*, *state_objects={}*)

Parameters

- **models** (*Union[Module, Dict[str, Module]]*) –
- **optimizers** (*Union[Optimizer, Dict[str, Optimizer]]*) –
- **max_epochs** (*int*) –
- **iters_per_epoch** (*int*) –
- **extensions** (*Optional[Sequence[extension_module.ExtensionLike]]*) –
- **out_dir** (*str*) –
- **stop_trigger** (*trigger_module.TriggerLike*) –
- **writer** (*Optional[Writer]*) –
- **transform_model** (*Callable[[str, Module], Module]*) –
- **enable_profile** (*bool*) –
- **state_objects** (*Dict[str, StateObjectProtocol]*) –

Return type
None

finalize()

Return type
None

run_iteration(*, *step_optimizers=None*)

Context manager to run an iteration.

This manager can additionally run a step in the specified optimizers names.

Parameters

- **step_optimizers** (*list or None*) – names of the optimizers
- **step** (*to call zero_grad and*) –

Return type

Generator[None, None, None]

pytorch_pfn_extras.training.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: Protocol

Methods

__init__(*args, **kwargs)
get_extension(name)

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.IgniteExtensionsManager

class pytorch_pfn_extras.training.**IgniteExtensionsManager**(*engine*, *models*, *optimizers*, *max_epochs*, *, *extensions=None*, *out_dir='result'*, *writer=None*, *enable_profile=False*, *state_objects={}*)

Bases: _BaseExtensionsManager

Manages extensions and the current status in Ignite training loop.

Parameters

- **engine** (*ignite.engine.Engine*) – Ignite trainer engine
- **models** (*dict or torch.nn.Module*) – Map of string to Module or an actual Module
- **optimizers** (*dict or torch.Optimizer*) – Map of string to Optimizer or an actual Optimizer.
- **max_epochs** (*int*) – Number of epochs in the whole training loop.
- **extensions** (*list or None*) – List of Extentions to be used.
- **out_dir** (*str*) – Output directory (default: result).
- **writer** (*writing.Writer object*) – Writer that can be used by extensions to write data to custom filesystems.
- **enable_profile** (*bool*) – Flag to enable/disable profiling of iterations. Default is *False*.
- **state_objects** (*Dict[str, StateObjectProtocol]*) –

Methods

__init__(engine, models, optimizers, ...[, ...])	
extend(extension[, name, trigger, priority, ...])	Registers an extension to the manager.
get_extension(name)	Returns the extension of a given name.
load_state_dict(to_load)	
needs_model_state([iteration])	
needs_state_this_iteration()	
run_extensions()	
set_ignite_handlers()	
start_extensions()	
state_dict()	

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
optimizers
out
raw_models
stop_trigger
updater

__init__(*engine*, *models*, *optimizers*, *max_epochs*, ***, *extensions=None*, *out_dir='result'*, *writer=None*, *enable_profile=False*, *state_objects={}*)

Parameters

- **engine** (*ignite.engine.Engine*) –
- **models** (*Union[Module, Mapping[str, Module]]*) –
- **optimizers** (*Union[Optimizer, Mapping[str, Optimizer]]*) –
- **max_epochs** (*int*) –
- **extensions** (*Optional[Sequence[extension_module.ExtensionLike]]*) –
- **out_dir** (*str*) –
- **writer** (*Optional[Writer]*) –
- **enable_profile** (*bool*) –
- **state_objects** (*Dict[str, StateObjectProtocol]*) –

Return type
None

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

set_ignite_handlers()

Return type
None

state_dict()

Return type
Dict[str, *Any*]

pytorch_pfn_extras.training.StateObjectProtocol

class pytorch_pfn_extras.training.**StateObjectProtocol**(**args*, ***kwargs*)

Bases: Protocol

Methods

__init__(*args, **kwargs)
load_state_dict(state_dict)
state_dict()

__init__(**args*, ***kwargs*)

`load_state_dict`(*state_dict*)

Parameters
`state_dict` (*Dict[str, Any]*) –

Return type
None

`state_dict`()

Return type
Dict[str, *Any*]

pytorch_pfn_extras.training.Trainer

`class` pytorch_pfn_extras.training.`Trainer`(*handler*, *, *evaluator*, *models*, *profile=None*, ***kwargs*)

Bases: `object`

Methods

__init__(handler, *, evaluator, models[, ...])	
extend(extension[, name, trigger, priority, ...])	
get_optimizer(name)	
is_epoch_last_iter(idx)	
load_state_dict(to_load)	
run(train_loader[, val_loader, train_len, ...])	Executes the training loop.
set_optimizer(name, optimizer)	
state_dict()	

Attributes

epoch
epoch_detail
evaluator
is_before_training
iteration
manager
models
optimizers
stop_trigger

Parameters

- **handler** (*handler_module.BaseHandler*) –
- **evaluator** (*Optional[Union[Evaluator, Tuple[Evaluator, Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]], Mapping[str, Union[Evaluator, Tuple[Evaluator, Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]]]]]*) –
- **models** (*Union[Module, Mapping[str, Module]]*) –
- **profile** (*Optional[profile]*) –
- **kwargs** (*Any*) –

__init__(*handler*, ***, *evaluator*, *models*, *profile=None*, ***kwargs*)

Parameters

- **handler** (*handler_module.BaseHandler*) –
- **evaluator** (*Optional[Union[Evaluator, Tuple[Evaluator, Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]], Mapping[str, Union[Evaluator, Tuple[Evaluator, Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]]]]]*) –
- **models** (*Union[Module, Mapping[str, Module]]*) –
- **profile** (*Optional[profile]*) –
- **kwargs** (*Any*) –

property epoch: int

property epoch_detail: float

property evaluator: Optional[Evaluator]

extend(*extension*, *name=None*, *trigger=None*, *priority=None*, *, *call_before_training=False*, **kwargs*)

Parameters

- **extension** (*Union[extension.ExtensionLike, ExtensionEntry]*) –
- **name** (*Optional[str]*) –
- **trigger** (*TriggerLike*) –
- **priority** (*Optional[int]*) –
- **call_before_training** (*bool*) –
- **kwargs** (*Any*) –

Return type
None

get_optimizer(*name*)

Parameters
name (*str*) –

Return type
Optimizer

property is_before_training: bool

is_epoch_last_iter(*idx*)

Parameters
idx (*int*) –

Return type
bool

property iteration: int

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

property manager: ExtensionsManager

property models: Mapping[str, Module]

property optimizers: Mapping[str, Optimizer]

run(*train_loader*, *val_loader=None*, *, *train_len=None*, *eval_len=None*)

Executes the training loop.

Parameters

- **train_loader** (*torch.utils.data.DataLoader*) – A data loader for training.

- **val_loader** (*torch.utils.data.DataLoader, optional*) – A data loader passed to `Evaluator.run()`.
- **train_len** (*int, optional*) – The number of iterations per one training epoch. The default value is inferred from the size of training data loader.
- **eval_len** (*int, optional*) – The number of iterations per one evaluation epoch, passed to `Evaluator.run()`

Return type
None

See also:

- *pytorch_pfn_extras.training._evaluator.Evaluator()*

set_optimizer(*name*, *optimizer*)

Parameters

- **name** (*str*) –
- **optimizer** (*Optimizer*) –

Return type
None

state_dict()

Return type
Dict[str, *Any*]

property stop_trigger: *Trigger*

Modules

pytorch_pfn_extras.training.extension

pytorch_pfn_extras.training.extensions

pytorch_pfn_extras.training.manager

pytorch_pfn_extras.training.metrics

pytorch_pfn_extras.training.trigger

pytorch_pfn_extras.training.triggers

pytorch_pfn_extras.training.extension

Functions

pytorch_pfn_extras.training.extension.make_extension([...])	Decorator to make given function into an extension.

pytorch_pfn_extras.training.extension.make_extension

pytorch_pfn_extras.training.extension.**make_extension**(*trigger=(1, 'iteration')*, *default_name=None*, *priority=100*, *finalizer=<function <lambda>>*, *initializer=<function <lambda>>*, *on_error=<function <lambda>>*)

Decorator to make given function into an extension.

This decorator just adds some attributes to a given function. The value of the attributes are given by the arguments of this decorator.

See *Extension* for details of extensions. Most of the default values of arguments also follow those for this class.

Parameters

- **trigger** (*TriggerLike*) – Default trigger of the extension.
- **default_name** (*Optional[str]*) – Default name of the extension. The name of a given function is used by default.
- **priority** (*int*) – Default priority of the extension.
- **finalizer** (*ExtensionLike*) – Finalizer function of this extension. It is called at the end of the training loop.
- **initializer** (*ExtensionLike*) – Initializer function of this extension. It is called at the beginning of the training loop.
- **on_error** (*Callable[[ExtensionsManagerProtocol, Exception, TracebackType], None]*) – Error handler callback function of this extension. It is called after an error is raised during the training loop.

Return type

Callable[[ExtensionLike], ExtensionLike]

Classes

pytorch_pfn_extras.training.extension.Extension()	Base class of extensions.
pytorch_pfn_extras.training.extension.ExtensionEntry(...)	Extension and options.
pytorch_pfn_extras.training.extension.ExtensionsManagerProtocol(...)	

pytorch_pfn_extras.training.extension.Extension

class pytorch_pfn_extras.training.extension.**Extension**

Bases: `object`

Base class of extensions.

An extension is a callable object that takes the manager object as the argument. It also provides some default configurations as its attributes, e.g. the default trigger and the default priority. This class provides a set of typical default values for these attributes.

There are three ways to define users' own extensions: inheriting this class, decorating closures by *make_extension()*, or using any callable including lambda functions as extensions. Decorator can slightly reduce the overhead and is much easier to use, while this class provides more flexibility (for example, it can have methods to configure the behavior). Using a lambda function allows one-line coding for simple purposes, but users have to specify the configurations as arguments to `ExtensionsManager.extend()`. For a callable not inheriting this class, the default configurations of this class are used unless the user explicitly specifies them in `ExtensionsManager.extend()` method.

trigger

Default value of trigger for this extension. It is set to `(1, 'iteration')` by default.

Type
TriggerLike

priority

Default priority of the extension. It is set to `PRIORITY_READER` by default.

Type
int

~Extension.name

Name of the extension. It is set to `None` by default. This value will be overwritten when registering an extension to a manager. See `pytorch_pfn_extras.ExtensionsManager.extend()` for details.

Methods

`__init__`()	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
Any

property default_name: str

Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

finalize(*manager*)

Finalizes the extension.

This method is called at the end of the training loop.

Parameters
manager (ExtensionsManagerProtocol) –

Return type
None

initialize(*manager*)

Initializes up the manager state.

This method is called before entering the training loop. An extension modifying the state of ExtensionsManager can override this method to initialize it.

When the manager has been restored from a snapshot, this method has to recover an appropriate part of the state of the manager.

Parameters
manager (ExtensionsManager) – Manager object to call this extension.

Return type
None

`is_async = False`

`load_state_dict`(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

`name: Optional[str] = None`

`needs_model_state = False`

`on_error`(*manager*, *exc*, *tb*)

Handles the error raised during training before finalization.

This method is called when an exception is thrown during the training loop, before finalize. An extension that needs different error handling from finalize, can override this method to handle errors.

Parameters

- **manager** (ExtensionsManager) –
- **extension.** (*Manager object to call this*) –
- **exc** (*Exception*) – arbitrary exception thrown during update loop.
- **tb** (*traceback*) – traceback object of the exception

Return type
None

`priority: int = 100`

`state_dict`()

Serializes the extension state.

It is called when a manager that owns this extension is serialized. It serializes nothing by default.

Return type
Dict[str, *Any*]

`trigger: TriggerLike = (1, 'iteration')`

pytorch_pfn_extras.training.extension.ExtensionEntry

`class` pytorch_pfn_extras.training.extension.**ExtensionEntry**(*extension*, *, *name=None*, *priority=None*, *trigger=None*, *call_before_training=False*)

Bases: `object`

Extension and options. When name, priority, or trigger is not specified, it is copied from the attributes of the given `extension`.

Parameters

- **extension** (*ExtensionLike*) – An extension.
- **name** (*Optional[str]*) – Name of extension.
- **priority** (*Optional[int]*) – Invocation priority of the extension.

- **trigger** (*Optional[TriggerLike]*) – Trigger object that determines when to invoke the extension.
- **call_before_training** (*bool*) – Flag to call extension before training.

See also:

`pytorch_pfn_extras.training.ExtensionsManager.extend()`

Methods

__init__(extension, *[, name, priority, ...])
load_state_dict(to_load)
state_dict()

__init__(*extension*, *, *name=None*, *priority=None*, *trigger=None*, *call_before_training=False*)

Parameters

- **extension** (*ExtensionLike*) –
- **name** (*Optional[str]*) –
- **priority** (*Optional[int]*) –
- **trigger** (*Optional[TriggerLike]*) –
- **call_before_training** (*bool*) –

Return type
None

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

state_dict()

Return type
Dict[str, *Any*]

pytorch_pfn_extras.training.extension.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.extension.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: Protocol

Methods

__init__(*args, **kwargs)
get_extension(name)

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.extensions

Functions

pytorch_pfn_extras.training.extensions.observe_lr(...)	Returns an extension to record the learning rate.
pytorch_pfn_extras.training.extensions.observe_value(...)	Returns an extension to continuously record a value.
pytorch_pfn_extras.training.extensions.snapshot([...])	Returns a trainer extension to take snapshots of the trainer.
pytorch_pfn_extras.training.extensions.snapshot_object(...)	Returns an extension to take snapshots of a given object.

pytorch_pfn_extras.training.extensions.observe_lr

pytorch_pfn_extras.training.extensions.**observe_lr**(*optimizer*, *param_group=0*, *observation_key='lr'*)

Returns an extension to record the learning rate.

Parameters

- **optimizer** (*Optimizer*) – Optimizer whose learning rate is recorded.
- **param_group** (*int*) – Param group of the optimizer to observe
- **observation_key** (*str*) – Key of observation to record.

Returns

The extension function.

Return type

Any

This extension is triggered each epoch by default. To change this, use the `trigger` argument with the `ExtensionsManager.extend()` method.

pytorch_pfn_extras.training.extensions.observe_value

pytorch_pfn_extras.training.extensions.**observe_value**(*observation_key*, *target_func*)

Returns an extension to continuously record a value.

Parameters

- **observation_key** (*str*) – Key of observation to record.
- **target_func** (*function*) – Function that returns the value to record. It must take one argument: :class:~pytorch_pfn_extras.training.ExtensionsManager object.

Returns
The extension function.

Return type
Callable[[ExtensionsManagerProtocol], None]

This extension is triggered each epoch by default. To change this, use the `trigger` argument with the `ExtensionsManager.extend()` method.

pytorch_pfn_extras.training.extensions.snapshot

pytorch_pfn_extras.training.extensions.**snapshot**(*savefun=None*, *filename='snapshot_iter_{.iteration}'*, ***, *target=None*, *condition=None*, *writer=None*, *snapshot_on_error=False*, *n_retains=-1*, *autoload=False*, *saver_rank=None*)

Returns a trainer extension to take snapshots of the trainer.

This extension serializes the manager object and saves it to the output directory. It is used to support resuming the training loop from the saved state.

This extension is called once per epoch by default. To take a snapshot at a different interval, a trigger object specifying the required interval can be passed along with this extension to the *extend()* method of the manager.

The default priority is -100, which is lower than that of most built-in extensions.

Parameters

- **savefun** (*Optional[Any]*) – Function to save the manager. It takes two arguments: the output file path and the manager object. It is `torch.save()` by default. If `writer` is specified, this argument must be `None`.
- **filename** (*str*) – Name of the file into which the manager is serialized. It can be a format string, where the manager object is passed to the `str.format()` method.
- **target** (*Optional[Any]*) – Object to serialize. If it is not specified, it will be the manager object.
- **condition** (*Optional[Any]*) – Condition object. It must be a callable object that returns boolean without any arguments. If it returns `True`, the snapshot will be done. If not, it will be skipped. The default is a function that always returns `True`.
- **writer** (*Optional[Writer]*) – Writer object. It must be a callable object. See below for the list of built-in writers. If `savefun` is other than `None`, this argument must be `None`. In that case, a *SimpleWriter* object instantiated with specified `savefun` argument will be used.
- **snapshot_on_error** (*bool*) – Whether to take a snapshot in case training loop has been failed.

- **n_retains** (*int*) – Number of snapshot files to retain through the cleanup. Must be a positive integer for any cleanup to take place. Automatic deletion of old snapshots only works when the filename is string.
- **autoload** (*bool*) – With this enabled, the extension automatically finds the latest snapshot and loads the data to the target. Automatic loading only works when the filename is a string. It is assumed that snapshots are generated by `torch.save()` .
- **saver_rank** (*int*) – If defined, the snapshot will be taken by only one rank when running in distributed mode and restored by all.

Returns
Snapshot extension object.

Return type
_Snapshot

Using asynchronous writers

By specifying `writer` argument, writing operations can be made asynchronous, hiding I/O overhead of snapshots.

```
>>> from pytorch_pfn_extras.training import extensions
>>> from pytorch_pfn_extras import writing
>>> writer = writing.ProcessWriter()
>>> manager.extend(extensions.snapshot(writer=writer), trigger=(1, 'epoch'))
```

To change the format, you can pass a saving function as `savefun` argument of the writer.

```
>>> from pytorch_pfn_extras.training import extensions
>>> from pytorch_pfn_extras import writing
>>> writer = writing.ProcessWriter(
...     savefun=torch.save)
>>> manager.extend(extensions.snapshot(writer=writer), trigger=(1, 'epoch'))
```

This is the list of built-in snapshot writers.

- *pytorch_pfn_extras.writing.SimpleWriter*
- *pytorch_pfn_extras.writing.ThreadWriter*
- *pytorch_pfn_extras.writing.ProcessWriter*
- *pytorch_pfn_extras.writing.ThreadQueueWriter*
- *pytorch_pfn_extras.writing.ProcessQueueWriter*

See also:

- *pytorch_pfn_extras.training.extensions.snapshot_object()*

pytorch_pfn_extras.training.extensions.snapshot_object

pytorch_pfn_extras.training.extensions.**snapshot_object**(*target*, *filename*, *savefun=None*, ***, *condition=None*, *writer=None*, *snapshot_on_error=False*, *n_retains=-1*, *autoload=False*)

Returns an extension to take snapshots of a given object.

This extension serializes the given object and saves it to the output directory.

This extension is called once per epoch by default. To take a snapshot at a different interval, a trigger object specifying the required interval can be passed along with this extension to the *extend()* method of the manager.

The default priority is lower than that of most built-in extensions.

Parameters

- **target** (*Any*) – Object to serialize.
- **filename** (*str*) – Name of the file into which the object is serialized. It can be a format string, where the manager object is passed to the `str.format()` method. For example, `'snapshot_{.iteration}'` is converted to `'snapshot_10000'` at the 10,000th iteration.
- **savefun** (*Optional[Any]*) – Function to save the object. It takes two arguments: the output file path and the object to serialize.
- **condition** – Condition object. It must be a callable object that returns boolean without any arguments. If it returns `True`, the snapshot will be done. If not, it will be skipped. The default is a function that always returns `True`.
- **writer** – Writer object. It must be a callable object. See below for the list of built-in writers. If `savefun` is other than `None`, this argument must be `None`. In that case, a *SimpleWriter* object instantiated with specified `savefun` argument will be used.
- **snapshot_on_error** (*bool*) – Whether to take a snapshot in case training loop has failed.
- **n_retains** (*int*) – Number of snapshot files to retain through the cleanup. Must be a positive integer for any cleanup to take place. Automatic deletion of old snapshots only works when the filename is string.
- **autoload** (*bool*) – With this enabled, the extension automatically finds the latest snapshot and loads the data to the target. Automatic loading only works when the filename is a string.
- **saver_rank** (*int*) – If defined, the snapshot will be taken by only one rank when running in distributed mode and restored by all.
- **kwargs** (*Any*) –

Returns

Snapshot extension object.

Return type

_Snapshot

See also:

- *pytorch_pfn_extras.training.extensions.snapshot()*

Classes

pytorch_pfn_extras.training.extensions.BestValue(...)	Extension traces the best value of a specific key in the observation.
pytorch_pfn_extras.training.extensions.DistributedEvaluator(...)	An extension to evaluate models on a validation set in a distributed training setup.
pytorch_pfn_extras.training.extensions.Evaluator(...)	An extension to evaluate models on a validation set.
pytorch_pfn_extras.training.extensions.FailOnNonNumber(*)	An extension to raise RuntimeError if parameters and its gradients contain NaN or Inf.
pytorch_pfn_extras.training.extensions.IgniteEvaluator(...)	
pytorch_pfn_extras.training.extensions.LRScheduler(...)	Trainer extension to adjust the learning rate using PyTorch's learning rate scheduler.
pytorch_pfn_extras.training.extensions.LogReport([...])	An extension to output the accumulated results to a log file.
pytorch_pfn_extras.training.extensions.MaxValue(key)	Extension traces the maximum value of a specific key in the observation.
pytorch_pfn_extras.training.extensions.MicroAverage(...)	Calculates micro-average ratio.
pytorch_pfn_extras.training.extensions.MinValue(key)	Extension traces the maximum value of a specific key in the observation.
pytorch_pfn_extras.training.extensions.ParameterStatistics(links)	An extension to report parameter statistics.
pytorch_pfn_extras.training.extensions.PlotReport(y_keys)	An extension to output plots.
pytorch_pfn_extras.training.extensions.PrintReport([...])	An extension to print the accumulated results.
pytorch_pfn_extras.training.extensions.PrintReportCLI	alias of *PrintReport*
pytorch_pfn_extras.training.extensions.ProfileReport([...])	Writes the profile results to a file.
pytorch_pfn_extras.training.extensions.ProgressBar([...])	An extension to print a progress bar and recent training status.
pytorch_pfn_extras.training.extensions.ProgressBarCLI	alias of *ProgressBar*
pytorch_pfn_extras.training.extensions.Slack(channel)	An extension to communicate with Slack.
pytorch_pfn_extras.training.extensions.SlackWebhook(url)	An extension to communicate with Slack using Incoming Webhook.
pytorch_pfn_extras.training.extensions.VariableStatisticsPlot(targets)	An extension to plot statistics for `Tensors`.

pytorch_pfn_extras.training.extensions.BestValue

class pytorch_pfn_extras.training.extensions.**BestValue**(*key*, *compare*, *trigger=(1, 'epoch')*)

Bases: *Extension*

Extension traces the best value of a specific key in the observation.

Parameters

- **key** (*str*) – Key of value.
- **compare** (*callable*) – Compare function which takes current best value and new value and returns whether new value is better than current best.
- **trigger** (*TriggerLike*) – Trigger that decides the comparison interval between current best value and new value. This must be a tuple in the form of `<int>, 'epoch'` or `<int>, 'iteration'` which is passed to `BestValueTrigger`.

Methods

__init__(key, compare[, trigger])	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

best_epoch	Returns the epoch count that the current best value is observed.
best_iteration	Returns the iteration count that the current best value is observed.
best_value	Returns the current best value.
default_name	
is_async	
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
None

__init__(*key*, *compare*, *trigger=(1, 'epoch')*)

Parameters

- **key** (*str*) –
- **compare** (*Callable[[float, float], bool]*) –
- **trigger** (*TriggerLike*) –

Return type
None

property best_epoch: int

Returns the epoch count that the current best value is observed.

If no value has been observed yet, it raises a RuntimError.

property best_iteration: int

Returns the iteration count that the current best value is observed.

If no value has been observed yet, it raises a RuntimError.

property best_value: float

Returns the current best value.

If no value has been observed yet, it raises a RuntimError.

default_name = 'best_value'

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

state_dict()

Serializes the extension state.

It is called when a manager that owns this extension is serialized. It serializes nothing by default.

Return type
Dict[str, *Any*]

pytorch_pfn_extras.training.extensions.DistributedEvaluator

class pytorch_pfn_extras.training.extensions.**DistributedEvaluator**(*self*, *iterator*, *target*, *eval_func=None*, ***, *progress_bar=False*)

Bases: *Evaluator*

An extension to evaluate models on a validation set in a distributed training setup.

In case torch.distributed is used to parallelize training iterations, it is efficient to also run evaluation in parallel by splitting the validation set to each worker process and conduct evaluation separately followed by aggregation of results of each worker, which can be achieved by :class:~`DistributedEvaluator`.

This extension basically behaves similarly to *Evaluator*, but adds an aggregation step in *Evaluator.evaluate()*. A summary of evaluation (`DictSummary`) in each worker process is collected in "all-gather" manner and then accumulated. Therefore all the worker processes must attend the evaluation, i.e., make sure all the processes have a *Evaluator* extension object configured in the `ExtensionManager` with the same trigger. All the worker process will get identical evaluation result returned by *Evaluator.evaluate()* and reported to an observation.

It is necessary to pass a DataLoader with an appropripate sampler which properly splits the validation dataset to each MPI worker process. PyTorch DistributedSampler implements this, but it allows sampler repetition in order to make the number of samples assigned to each process identical. For evaluation purpose it distorts the evaluation result, hence it is recommended to use `DistributedValidationSampler` instead.

Methods

__init__(iterator, target[, eval_hook, ...])	
`add_metric`(metric_fn)	Adds a custom metric to the evaluator.
`eval_func`(*args, **kwargs)	
`evaluate`()	Evaluates the model and returns a result dictionary.
`finalize`(manager)	Finalizes the extension.
`get_all_iterators`()	Returns a dictionary of all iterators.
`get_all_targets`()	Returns a dictionary of all target links.
`get_iterator`(name)	Returns the iterator of the given name.
`get_target`(name)	Returns the target link of the given name.
`initialize`(manager)	Initializes up the manager state.
`load_state_dict`(to_load)	
`on_error`(manager, exc, tb)	Handles the error raised during training before finalization.
`state_dict`()	Serializes the extension state.

Attributes

default_name
is_async
name
needs_model_state
priority
trigger

Parameters

- **iterator** (*Union[DataLoader[Any], Dict[str, DataLoader[Any]]]*) –
- **target** (*Union[Module, Dict[str, Module]]*) –
- **eval_hook** (*Optional[Callable[[Evaluator], None]]*) –
- **eval_func** (*Optional[Callable[[...], Any]]*) –
- **kwargs** (*Any*) –

__init__(*iterator*, *target*, *eval_hook=None*, *eval_func=None*, ***kwargs*)

Parameters

- **iterator** (*Union[DataLoader[Any], Dict[str, DataLoader[Any]]]*) –
- **target** (*Union[Module, Dict[str, Module]]*) –
- **eval_hook** (*Optional[Callable[[Evaluator], None]]*) –
- **eval_func** (*Optional[Callable[[...], Any]]*) –
- **kwargs** (*Any*) –

Return type
None

pytorch_pfn_extras.training.extensions.Evaluator

class pytorch_pfn_extras.training.extensions.**Evaluator**(*self*, *iterator*, *target*, *eval_func=None*, ***, *progress_bar=False*)

Bases: *Extension*

An extension to evaluate models on a validation set.

This extension evaluates the current models by a given evaluation function. It creates a `Reporter` object to store values observed in the evaluation function on each iteration. The report for all iterations are aggregated to `DictSummary`. The collected mean values are further reported to the reporter object of the manager, where the name of each observation is prefixed by the evaluator name. See `Reporter` for details in naming rules of the reports.

Evaluator has a structure to customize similar to that of `StandardUpdater`. The main differences are:

- There are no optimizers in an evaluator. Instead, it holds links to evaluate.
- An evaluation loop function is used instead of an update function.
- Preparation routine can be customized, which is called before each evaluation. It can be used, e.g., to initialize the state of stateful recurrent networks.

There are two ways to modify the evaluation behavior besides setting a custom evaluation function. One is by setting a custom evaluation loop via the `eval_func` argument. The other is by inheriting this class and overriding the *evaluate()* method. In latter case, users have to create and handle a reporter object manually. Users also have to copy the iterators before using them, in order to reuse them at the next time of evaluation. In both cases, the functions are called in testing mode

This extension is called at the end of each epoch by default.

Parameters

- **iterator** (*Union[DataLoader[Any], Dict[str, DataLoader[Any]]]*) – Dataset iterator for the validation dataset. It can also be a dictionary of iterators. If this is just an iterator, the iterator is registered by the name `'main'`.
- **target** (*Union[Module, Dict[str, Module]]*) – torch.nn.Module object or a dictionary of links to evaluate. If this is just a layer object, the link is registered by the name `'main'`.
- **eval_func** (*Optional[Callable[[...], Any]]*) – Evaluation function called at each iteration. The target link to evaluate as a callable is used by default.
- **progress_bar** – Boolean flag to show a progress bar while training, which is similar to `ProgressBar`. (default: `False`)
- **metrics** – List of callables that are called every batch to calculate metrics such as accuracy, roc_auc or others The signature of the callable is: *def metric_fn(batch, output, last_iteration)* (default: `[]`)
- **eval_hook** (*Optional[Callable[[Evaluator], None]]*) –
- **kwargs** (*Any*) –

Warning: The argument `progress_bar` is experimental. The interface can change in the future.

eval_hook

Function to prepare for each evaluation process.

eval_func

Evaluation function called at each iteration.

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type

Any

Methods

__init__(iterator, target[, eval_hook, ...])	
add_metric(metric_fn)	Adds a custom metric to the evaluator.
eval_func(*args, **kwargs)	
evaluate()	Evaluates the model and returns a result dictionary.
finalize(manager)	Finalizes the extension.
get_all_iterators()	Returns a dictionary of all iterators.
get_all_targets()	Returns a dictionary of all target links.
get_iterator(name)	Returns the iterator of the given name.
get_target(name)	Returns the target link of the given name.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

default_name
is_async
name
needs_model_state
priority
trigger

__call__(*manager=None*)

Executes the evaluator extension.

Unlike usual extensions, this extension can be executed without passing a manager object. This extension reports the performance on validation dataset using the `report()` function. Thus, users can use this extension independently from any manager by manually configuring a `Reporter` object.

Parameters
manager (`ExtensionsManager`) – Manager object that invokes this extension.

Returns
Result dictionary that contains mean statistics of values reported by the evaluation function.

Return type
dict

__init__(*iterator*, *target*, *eval_hook=None*, *eval_func=None*, ***kwargs*)

Parameters

- **iterator** (*Union[DataLoader[Any], Dict[str, DataLoader[Any]]]*) –
- **target** (*Union[Module, Dict[str, Module]]*) –
- **eval_hook** (*Optional[Callable[[Evaluator], None]]*) –
- **eval_func** (*Optional[Callable[[...], Any]]*) –
- **kwargs** (*Any*) –

Return type
None

add_metric(*metric_fn*)

Adds a custom metric to the evaluator.

The metric is a callable that is executed every batch with the following signature: *def metric_fn(batch, output, last_iteration)*

Batch is the input batch passed to the model. Output is the result of evaluating batch, last_iteration is a boolean flag that indicates if its the last batch in the evaluation.

Parameters
metric_fn (*Callable[[Any, Any, Any], None]*) –

Return type
None

default_name = 'validation'

eval_func(**args*, ***kwargs*)

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

evaluate()

Evaluates the model and returns a result dictionary.

This method runs the evaluation loop over the validation dataset. It accumulates the reported values to `DictSummary` and returns a dictionary whose values are means computed by the summary.

Users can override this method to customize the evaluation routine.

Returns
Result dictionary. This dictionary is further reported via `report()` without specifying any observer.

Return type
dict

get_all_iterators()

Returns a dictionary of all iterators.

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

`get_all_targets`()

Returns a dictionary of all target links.

Return type
Dict[str, *Module*]

`get_iterator`(*name*)

Returns the iterator of the given name.

Parameters
name (*str*) –

Return type
DataLoader[*Any*]

`get_target`(*name*)

Returns the target link of the given name.

Parameters
name (*str*) –

Return type
Module

`priority: int = 300`

`trigger: TriggerLike = (1, 'epoch')`

pytorch_pfn_extras.training.extensions.FailOnNonNumber

`class` pytorch_pfn_extras.training.extensions.**FailOnNonNumber**(*, *check_grad=True*)

Bases: *Extension*

An extension to raise RuntimeError if parameters and its gradients contain NaN or Inf.

Although parameters including non-number such as NaN and Inf are unnecessary in most cases the training loop will continue to compute even if the parameters in a given optimizer diverge. This extension is aimed to reduce unnecessary computations by throwing `RuntimeError` if the parameters contain NaN or Inf.

Parameters
check_grad (*bool*) – Set to False to skip checking gradients.

Methods

__init__(*[, check_grad])	

`finalize`(manager)	Finalizes the extension.
`initialize`(manager)	Initializes up the manager state.
`load_state_dict`(to_load)	
`on_error`(manager, exc, tb)	Handles the error raised during training before finalization.
`state_dict`()	Serializes the extension state.

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
None

__init__(*, *check_grad=True*)

Parameters
check_grad (*bool*) –

needs_model_state = True

pytorch_pfn_extras.training.extensions.IgniteEvaluator

class pytorch_pfn_extras.training.extensions.**IgniteEvaluator**(*evaluator*, *iterator*, *target*, ***kwargs*)

Bases: *Evaluator*

Methods

__init__(evaluator, iterator, target, **kwargs)	
add_metric(metric_fn)	Adds a custom metric to the evaluator.
eval_func(*args, **kwargs)	
evaluate()	Evaluates the model and returns a result dictionary.
finalize(manager)	Finalizes the extension.
get_all_iterators()	Returns a dictionary of all iterators.
get_all_targets()	Returns a dictionary of all target links.
get_iterator(name)	Returns the iterator of the given name.
get_target(name)	Returns the target link of the given name.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
set_evaluator_handlers()	
state_dict()	Serializes the extension state.

Attributes

default_name
is_async
name
needs_model_state
priority
trigger

Parameters

- **evaluator** (*Engine*) –
- **iterator** (*Union[DataLoader[Any], Dict[str, DataLoader[Any]]]*) –
- **target** (*Union[Module, Dict[str, Module]]*) –
- **kwargs** (*Any*) –

__init__(*evaluator*, *iterator*, *target*, ***kwargs*)

Parameters

- **evaluator** (*Engine*) –
- **iterator** (*Union[DataLoader[Any], Dict[str, DataLoader[Any]]]*) –

- **target** (*Union[Module, Dict[str, Module]]*) –
- **kwargs** (*Any*) –

evaluate()

Evaluates the model and returns a result dictionary.

This method runs the evaluation loop over the validation dataset. It accumulates the reported values to `DictSummary` and returns a dictionary whose values are means computed by the summary.

Users can override this method to customize the evaluation routine.

Returns
Result dictionary. This dictionary is further reported via `report()` without specifying any observer.

Return type
dict

set_evaluator_handlers()

Return type
None

pytorch_pfn_extras.training.extensions.LRScheduler

class pytorch_pfn_extras.training.extensions.**LRScheduler**(*scheduler*, ***, *stepper=<function _default_stepper>*, *trigger=(1, 'epoch')*, *wait_for_first_optimizer_step=False*, *is_async=True*)

Bases: *Extension*

Trainer extension to adjust the learning rate using PyTorch's learning rate scheduler.

This extension calls *step()* method of the given LR scheduler. (*torch.option.lr_scheduler.**). When using *ReduceLROnPlateau*, the latest reported *val/loss* value will be used. This behavior can be customized by passing a custom *stepper* function.

Parameters

- **scheduler** (*_LRScheduler or ReduceLROnPlateau*) – Any instance of *torch.optim.lr_scheduler.**.
- **stepper** (*callable*) – Function that performs the step on the scheduler.
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) – Frequency to call this extension.
- **wait_for_first_optimizer_step** (*bool*) – Wait until *optimizer.step()* is called before invoking *scheduler.step()*. This can address the issue where *optimizer.step()* is not called from the first iteration when using GradScaler.
- **is_async** (*bool*) –

Methods

__init__(scheduler, *[, stepper, trigger, ...])	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(state)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.
step_by_value(key)	

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
None

__init__(*scheduler*, *, *stepper=<function _default_stepper>*, *trigger=(1, 'epoch')*, *wait_for_first_optimizer_step=False*, *is_async=True*)

Parameters

- **scheduler** (*Any*) –
- **stepper** (*Any*) –
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –
- **wait_for_first_optimizer_step** (*bool*) –
- **is_async** (*bool*) –

Return type
None

`load_state_dict`(*state*)

Parameters
state (*Dict[str, Any]*) –

Return type
None

`state_dict`()

Serializes the extension state.

It is called when a manager that owns this extension is serialized. It serializes nothing by default.

Return type
Dict[str, *Any*]

`static step_by_value`(*key*)

Parameters
key (*Optional[str]*) –

Return type
Any

pytorch_pfn_extras.training.extensions.LogReport

`class` pytorch_pfn_extras.training.extensions.`LogReport`(*keys=None*, *trigger=(1, 'epoch')*, *postprocess=None*, *filename=None*, *append=False*, *format=None*, ***kwargs*)

Bases: *Extension*

An extension to output the accumulated results to a log file.

This extension accumulates the observations of the manager to `DictSummary` at a regular interval specified by a supplied trigger, and writes them into a log file in JSON format.

There are two triggers to handle this extension. One is the trigger to invoke this extension, which is used to handle the timing of accumulating the results. It is set to `1, 'iteration'` by default. The other is the trigger to determine when to emit the result. When this trigger returns True, this extension appends the summary of accumulated values to the list of past summaries, and writes the list to the log file. Then, this extension makes a new fresh summary object which is used until the next time that the trigger fires.

It also adds some entries to each result dictionary.

- `'epoch'` and `'iteration'` are the epoch and iteration counts at the output, respectively.
- `'elapsed_time'` is the elapsed time in seconds since the training begins. The value is taken from `ExtensionsManager.elapsed_time`.

Parameters

- **keys** (*iterable of strs*) – Keys of values to accumulate. If this is None, all the values are accumulated and output to the log file.
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) – Trigger that decides when to aggregate the result and output the values. This is distinct from the trigger of this extension itself. If it

is a tuple in the form `<int>, 'epoch'` or `<int>, 'iteration'`, it is passed to `IntervalTrigger`.

- **postprocess** (*Optional[Callable[[Mapping[str, Any]], None]]*) – Callback to postprocess the result dictionaries. Each result dictionary is passed to this callback on the output. This callback can modify the result dictionaries, which are used to output to the log file.
- **filename** (*str*) – Name of the log file under the output directory. It can be a format string: the last result dictionary is passed for the formatting. For example, users can use '{iteration}' to separate the log files for different iterations. (default: `'log'`)
- **append** (*bool, optional*) – If the file is JSON Lines or YAML, contents will be appended instead of rewritting the file every call.
- **format** (*str, optional*) – accepted values are *'json'*, *'json-lines'* and *'yaml'*.
- **writer** (*writer object, optional*) – must be callable. object to dump the log to. If specified, it needs to have a correct *savefun* defined. The writer can override the save location in the *pytorch_pfn_extras.training.ExtensionsManager* object
- **kwargs** (*Any*) –

Note: Enabling *append=True* reduces size of snapshots (and thus reduces the time needed to take snapshots). Note that extensions relying on the logs from past iterations may behave differently; for example, when resuming from a snapshot, PrintReport will not show logs of iterations already done.

Methods

__init__([keys, trigger, postprocess, ...])	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.
to_dataframe()	

Attributes

default_name	Default name of the extension.
is_async	
log	The current list of observation dictionaries.
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
None

__init__(*keys=None*, *trigger=(1, 'epoch')*, *postprocess=None*, *filename=None*, *append=False*, *format=None*, ***kwargs*)

Parameters

- **keys** (*Optional[Iterable[str]]*) –
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –
- **postprocess** (*Optional[Callable[[Mapping[str, Any]], None]]*) –
- **filename** (*Optional[str]*) –
- **append** (*bool*) –
- **format** (*Optional[str]*) –
- **kwargs** (*Any*) –

finalize(*manager*)

Finalizes the extension.

This method is called at the end of the training loop.

Parameters
manager (ExtensionsManagerProtocol) –

Return type
None

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

property log: List[Mapping[str, Any]]

The current list of observation dictionaries.

state_dict()

Serializes the extension state.

It is called when a manager that owns this extension is serialized. It serializes nothing by default.

Return type
Dict[str, *Any*]

to_dataframe()

Return type
pandas.DataFrame

pytorch_pfn_extras.training.extensions.MaxValue

class pytorch_pfn_extras.training.extensions.**MaxValue**(*key*, *trigger=(1, 'epoch')*)

Bases: *BestValue*

Extension traces the maximum value of a specific key in the observation.

Parameters

- **key** (*str*) – Key of value.
- **trigger** (*TriggerLike*) – Trigger that decides the comparison interval between current maximum value and new value. This must be a tuple in the form of `<int>, 'epoch'` or `<int>, 'iteration'` which is passed to `BestValueTrigger`.

Methods

__init__(key[, trigger])	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

best_epoch	Returns the epoch count that the current best value is observed.
best_iteration	Returns the iteration count that the current best value is observed.
best_value	Returns the current best value.
default_name	
is_async	
name	
needs_model_state	
priority	
trigger	

__init__(*key*, *trigger=(1, 'epoch')*)

Parameters

- **key** (*str*) –
- **trigger** (*TriggerLike*) –

default_name = 'max_value'

pytorch_pfn_extras.training.extensions.MicroAverage

class pytorch_pfn_extras.training.extensions.**MicroAverage**(*numerator_key*, *denominator_key*, *result_key*, *trigger=(1, 'epoch')*)

Bases: *Extension*

Calculates micro-average ratio.

Give N batches and values $\{n_1, \ldots, n_N\}$ and $\{d_1, \ldots, d_N\}$, this extension calculates micro-average of these ratio defined as:

$$\frac{\sum_i^N n_i}{\sum_i^N d_i}.$$

A user usually uses the number of examples which a system correctly predict as n_i and the number of total examples in i-th batch as d_i. This value is called macro-average of precision.

Note that macro-average is defined as:

$$\frac{1}{N}\sum_i^N (n_i/d_i),$$

It is same to the micro-average when each mini-batch has the same d_i.

You need to report numerator value (the number of correct examples) and denominator value (the number of examples) in your model.

```
>>> class MyModel(torch.nn.Module):
...     def __call__(self, x, y):
...         loss = torch.nn.CrossEntropyLoss(x, y)
...         correct = (x.data.argmax(axis=1) == y.data).sum()
...         total = len(y.data)
...         reporting.report({'correct': correct, 'total': total}, self)
...         return loss
```

And then, make an extension with corresponding reporting keys and register it.

```
>>> ext = extensions.MicroAverage(
...     'main/correct', 'main/total', 'main/accuracy')
```

Parameters

- **numerator_key** (*str*) – Key string of obserbation storing a numerator value.
- **denominator_key** (*str*) – Key string of obserbation storing a denominator value.
- **result_key** (*str*) – Key string of obserbation to store a result.
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) – Trigger that decides when to calcurate average. This is distinct from the trigger of this extension itself. If it is a tuple in the form `<int>, 'epoch'` or `<int>, 'iteration'`, it is passed to `IntervalTrigger`.

Methods

__init__(numerator_key, denominator_key, ...)	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
None

__init__(*numerator_key*, *denominator_key*, *result_key*, *trigger=(1, 'epoch')*)

Parameters

- **numerator_key** (*str*) –
- **denominator_key** (*str*) –
- **result_key** (*str*) –
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –

Return type
None

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

priority: int = 200

state_dict()

Serializes the extension state.

It is called when a manager that owns this extension is serialized. It serializes nothing by default.

Return type
Dict[str, *Any*]

pytorch_pfn_extras.training.extensions.MinValue

class pytorch_pfn_extras.training.extensions.**MinValue**(*key*, *trigger=(1, 'epoch')*)

Bases: *BestValue*

Extension traces the maximum value of a specific key in the observation.

Parameters

- **key** (*str*) – Key of value.
- **trigger** (*TriggerLike*) – Trigger that decides the comparison interval between current maximum value and new value. This must be a tuple in the form of `<int>, 'epoch'` or `<int>, 'iteration'` which is passed to `BestValueTrigger`.

Methods

__init__(key[, trigger])	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

best_epoch	Returns the epoch count that the current best value is observed.
best_iteration	Returns the iteration count that the current best value is observed.
best_value	Returns the current best value.
default_name	
is_async	
name	
needs_model_state	
priority	
trigger	

__init__(*key*, *trigger=(1, 'epoch')*)

Parameters

- **key** (*str*) –
- **trigger** (*TriggerLike*) –

default_name = 'min_value'

pytorch_pfn_extras.training.extensions.ParameterStatistics

class pytorch_pfn_extras.training.extensions.**ParameterStatistics**(*links*, *statistics='default'*, *report_params=True*, *report_grads=True*, *prefix=None*, *trigger=(1, 'epoch')*, *skip_nan_params=False*)

Bases: *Extension*

An extension to report parameter statistics.

Statistics are collected and reported for a given `Module` or an iterable of `Modules`. If a link contains child modules, the statistics are reported separately for each child.

Any function that takes a one-dimensional `torch.Tensor` and outputs a single or multiple real numbers can be registered to handle the collection of statistics, e.g. `numpy.ndarray.mean()`.

The keys of reported statistics follow the convention of link name followed by parameter name, attribute name and function name, e.g. `VGG16Layers/conv1_1/W/data/mean`. They are prepended with an optional prefix and appended with integer indices if the statistics generating function return multiple values.

Parameters

- **links** (*instance or iterable of Module*) – Module(s) containing the parameters to observe. The link is expected to have a `name` attribute which is used as a part of the report key.
- **statistics** (*dict or 'default'*) – Dictionary with function name to function mappings. The name is a string and is used as a part of the report key. The function is responsible for generating the statistics. If the special value `'default'` is specified, the default statistics functions will be used.
- **report_params** (*bool*) – If `True`, report statistics for parameter values such as weights and biases.
- **report_grads** (*bool*) – If `True`, report statistics for parameter gradients.
- **prefix** (*str*) – Optional prefix to prepend to the report keys.
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) – Trigger that decides when to aggregate the results and report the values.
- **skip_nan_params** (*bool*) – If `True`, statistics are not computed for parameters including NaNs and a single NaN value is immediately reported instead. Otherwise, this extension will simply try to compute the statistics without performing any checks for NaNs.

Note: The default statistic functions are as follows:

- `'mean'` (`xp.mean(x)`)
- `'std'` (`xp.std(x)`)
- `'min'` (`xp.min(x)`)
- `'max'` (`xp.max(x)`)
- `'zeros'` (`xp.count_nonzero(x == 0)`)
- `'percentile'` (`xp.percentile(x, (0.13, 2.28, 15.87, 50, 84.13, 97.72, 99.87))`)

Methods

__init__(links[, statistics, report_params, ...])	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
register_statistics(name, function)	Register a function to compute a certain statistic.
state_dict()	Serializes the extension state.

Attributes

default_name
default_statistics
is_async
name
needs_model_state
priority
report_key_template
trigger

__call__(*manager*)

Execute the statistics extension.

Collect statistics for the current state of parameters.

Note that this method will merely update its statistic summary, unless the internal trigger is fired. If the trigger is fired, the summary will also be reported and then reset for the next accumulation.

Parameters
manager (ExtensionsManager) – Associated manager that invoked this extension.

Return type
None

__init__(*links*, *statistics='default'*, *report_params=True*, *report_grads=True*, *prefix=None*, *trigger=(1, 'epoch')*, *skip_nan_params=False*)

Parameters

- **links** (*Any*) –
- **statistics** (*Any*) –

- **report_params** (*bool*) –
- **report_grads** (*bool*) –
- **prefix** (*Optional[str]*) –
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –
- **skip_nan_params** (*bool*) –

default_name = 'parameter_statistics'

default_statistics = {'max': <function <lambda>>, 'mean': <function <lambda>>, 'min': <function <lambda>>, 'std': <function <lambda>>, 'zeros': <function <lambda>>}

priority: int = 300

register_statistics(*name*, *function*)

Register a function to compute a certain statistic.

The registered function will be called each time the extension runs and the results will be included in the report.

Parameters

- **name** (*str*) – Name of the statistic.
- **function** (*Any*) – Function to generate the statistic. Any function that takes a one-dimensional `numpy.ndarray` or a `cupy.ndarray` and outputs a single or multiple real numbers is allowed.

Return type
None

report_key_template = '{prefix}{param_name}/{attr_name}/{function_name}'

pytorch_pfn_extras.training.extensions.PlotReport

class pytorch_pfn_extras.training.extensions.**PlotReport**(*y_keys*, *x_key='iteration'*, *trigger=(1, 'epoch')*, *postprocess=None*, *filename='plot.png'*, *marker='x'*, *grid=True*)

Bases: *Extension*

An extension to output plots.

This extension accumulates the observations of the manager to *DictSummary* at a regular interval specified by a supplied trigger, and plot a graph with using them.

There are two triggers to handle this extension. One is the trigger to invoke this extension, which is used to handle the timing of accumulating the results. It is set to `1, 'iteration'` by default. The other is the trigger to determine when to emit the result. When this trigger returns True, this extension appends the summary of accumulated values to the list of past summaries, and writes the list to the log file. Then, this extension makes a new fresh summary object which is used until the next time that the trigger fires.

It also adds `'epoch'` and `'iteration'` entries to each result dictionary, which are the epoch and iteration counts at the output.

Warning: If your environment needs to specify a backend of matplotlib explicitly, please call `matplotlib.use` before calling `manager.run_iteration`. For example:

```
import matplotlib
matplotlib.use('Agg')

manager.extend(
    extensions.PlotReport(['main/loss', 'validation/main/loss'],
                          'epoch', filename='loss.png'))
with manager.run_iteration():
    pass
```

Then, once one of instances of this extension is called, `matplotlib.use` will have no effect.

For the details, please see here: https://matplotlib.org/faq/usage_faq.html#what-is-a-backend

Parameters

- **y_keys** (*iterable of strs*) – Keys of values regarded as y. If this is `None`, nothing is output to the graph.
- **x_key** (*str*) – Keys of values regarded as x. The default value is 'iteration'.
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) – Trigger that decides when to aggregate the result and output the values. This is distinct from the trigger of this extension itself. If it is a tuple in the form `<int>, 'epoch'` or `<int>, 'iteration'`, it is passed to `IntervalTrigger`.
- **postprocess** (*Any*) – Callback to postprocess the result dictionaries. Figure object, Axes object, and all plot data are passed to this callback in this order. This callback can modify the figure.
- **filename** (*str*) – Name of the figure file under the output directory. It can be a format string. For historical reasons `file_name` is also accepted as an alias of this argument.
- **marker** (*str*) – The marker used to plot the graph. Default is `'x'`. If `None` is given, it draws with no markers.
- **grid** (*bool*) – If `True`, set the axis grid on. The default value is `True`.
- **writer** (*writer object, optional*) – must be callable. object to dump the log to. If specified, it needs to have a correct *savefun* defined. The writer can override the save location in the *pytorch_pfn_extras.training.ExtensionsManager* object
- **kwargs** (*Any*) –

Methods

__init__(y_keys[, x_key, trigger, ...])	
available()	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
None

__init__(*y_keys*, *x_key='iteration'*, *trigger=(1, 'epoch')*, *postprocess=None*, *filename=None*, *marker='x'*, *grid=True*, ***kwargs*)

Parameters

- **y_keys** (*Union[Iterable[str], str]*) –
- **x_key** (*str*) –
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –
- **postprocess** (*Optional[Any]*) –
- **filename** (*Optional[str]*) –

- **marker** (*str*) –
- **grid** (*bool*) –
- **kwargs** (*Any*) –

static available()

Return type
bool

finalize(*manager*)

Finalizes the extension.

This method is called at the end of the training loop.

Parameters
manager (ExtensionsManagerProtocol) –

Return type
None

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

state_dict()

Serializes the extension state.

It is called when a manager that owns this extension is serialized. It serializes nothing by default.

Return type
Dict[str, *Any*]

pytorch_pfn_extras.training.extensions.PrintReport

class pytorch_pfn_extras.training.extensions.**PrintReport**(*entries=None*, *log_report='LogReport'*, *out=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Bases: *Extension*

An extension to print the accumulated results.

This extension uses the log accumulated by a *LogReport* extension to print specified entries of the log in a human-readable format.

Parameters

- **entries** (*list of str or None*) – List of keys of observations to print. If *None* is passed, automatically infer keys from reported dict.
- **log_report** (*str or* LogReport) – Log report to accumulate the observations. This is either the name of a LogReport extensions registered to the manager, or a LogReport instance to use internally.
- **out** (IO*[Any]*) – Stream to print the bar. Standard output is used by default.

Methods

__init__([entries, log_report, out])	
finalize(manager)	Finalizes the extension.
get_log_report(manager)	
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
None

__init__(*entries=None*, *log_report='LogReport'*, *out=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Parameters

- **entries** (*Optional[Sequence[str]]*) –
- **log_report** (*Union[str, LogReport]*) –
- **out** (*IO[Any]*) –

Return type
None

`get_log_report`(*manager*)

Parameters
manager (ExtensionsManagerProtocol) –

Return type
LogReport

`initialize`(*manager*)

Initializes up the manager state.

This method is called before entering the training loop. An extension modifying the state of `ExtensionsManager` can override this method to initialize it.

When the manager has been restored from a snapshot, this method has to recover an appropriate part of the state of the manager.

Parameters
manager (ExtensionsManager) – Manager object to call this extension.

Return type
None

`load_state_dict`(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

`state_dict`()

Serializes the extension state.

It is called when a manager that owns this extension is serialized. It serializes nothing by default.

Return type
Dict[str, *Any*]

pytorch_pfn_extras.training.extensions.PrintReportCLI

pytorch_pfn_extras.training.extensions.**PrintReportCLI**

alias of *PrintReport*

pytorch_pfn_extras.training.extensions.ProfileReport

class pytorch_pfn_extras.training.extensions.**ProfileReport**(*store_keys=None*, *report_keys=None*, *trigger=(1, 'epoch')*, *filename=None*, *append=False*, *format=None*, ***kwargs*)

Bases: *Extension*

Writes the profile results to a file.

Times are reported by using the *pytorch_pfn_extras.profiler.TimeSummary.report()* context manager.

Parameters

- **store_keys** (*iterable of strs*) – Keys of values to write to the profiler report file.
- **report_keys** (*iterable of strs*) – Keys of values that will be reported.
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) – Trigger that decides when to aggregate the result and output the values. This is distinct from the trigger of this extension itself. If it is a tuple in the form `<int>, 'epoch'` or `<int>, 'iteration'`, it is passed to `IntervalTrigger`.
- **filename** (*str*) – Name of the log file under the output directory. It can be a format string: the last result dictionary is passed for the formatting. For example, users can use '{iteration}' to separate the log files for different iterations. If the log name is None, it does not output the log to any file.
- **append** (*bool, optionsl*) – If the file is JSON Lines or YAML, contents will be appended instead of rewriting the file every call.
- **format** (*str, optional*) – accepted values are *'json'*, *'json-lines'* and *'yaml'*.
- **writer** (*writer object, optional*) – must be callable. object to dump the log to. If specified, it needs to have a correct *savefun* defined. The writer can override the save location in the *pytorch_pfn_extras.training.ExtensionsManager* object
- **entries** (*list*) – list of str
- **kwargs** (*Any*) –

Returns
header string templates (str): template string for print values.

Return type
header (str)

Methods

__init__([store_keys, report_keys, trigger, ...])	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
None

__init__(*store_keys=None*, *report_keys=None*, *trigger=(1, 'epoch')*, *filename=None*, *append=False*, *format=None*, ***kwargs*)

Parameters

- **store_keys** (*Optional[Iterable[str]]*) –
- **report_keys** (*Optional[Iterable[str]]*) –
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –
- **filename** (*Optional[str]*) –
- **append** (*bool*) –
- **format** (*Optional[str]*) –
- **kwargs** (*Any*) –

finalize(*manager*)

Finalizes the extension.

This method is called at the end of the training loop.

Parameters
manager (ExtensionsManagerProtocol) –

Return type
None

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

`state_dict()`

Serializes the extension state.

It is called when a manager that owns this extension is serialized. It serializes nothing by default.

Return type
Dict[str, *Any*]

pytorch_pfn_extras.training.extensions.ProgressBar

`class pytorch_pfn_extras.training.extensions.ProgressBar(`*training_length=None*, *update_interval=100*, *bar_length=50*, *out=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*`)`

Bases: *Extension*

An extension to print a progress bar and recent training status.

This extension prints a progress bar at every call. It watches the current iteration and epoch to print the bar.

Parameters

- **training_length** (*tuple or None*) – Length of whole training. It consists of an integer and either `'epoch'` or `'iteration'`. If this value is omitted and the stop trigger of the manager is `IntervalTrigger`, this extension uses its attributes to determine the length of the training.
- **update_interval** (*int*) – Number of iterations to skip printing the progress bar.
- **bar_length** (*int*) – Length of the progress bar in characters.
- **out** (*Any*) – Stream to print the bar. Standard output is used by default.

Methods

__init__([training_length, update_interval, ...])	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
None

__init__(*training_length=None*, *update_interval=100*, *bar_length=50*, *out=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Parameters

- **training_length** (*Optional[Any]*) –
- **update_interval** (*int*) –
- **bar_length** (*int*) –
- **out** (*Any*) –

finalize(*manager*)

Finalizes the extension.

This method is called at the end of the training loop.

Parameters
manager (ExtensionsManagerProtocol) –

Return type
None

pytorch_pfn_extras.training.extensions.ProgressBarCLI

pytorch_pfn_extras.training.extensions.**ProgressBarCLI**

alias of *ProgressBar*

pytorch_pfn_extras.training.extensions.Slack

class pytorch_pfn_extras.training.extensions.**Slack**(*channel*, *msg=None*, ***, *start_msg='{default}'*, *end_msg='{default}'*, *error_msg='{default}'*, *thread=True*, *filenames=None*, *upload_trigger=None*, *context=None*, *token=None*)

Bases: `_SlackBase`

An extension to communicate with Slack.

Example

```
>>> ppe.training.extensions.Slack(
...     channel="experiment-progress",
...     msg="Epoch #{manager.epoch}: loss = {val/loss}",
...     end_msg="{default} \n <@username> Check out the result!",
...
...     # Upload files at the end of the experiment.
...     filenames=["result/statistics.png"],
...     upload_trigger=(max_epoch, 'epoch'),
... )
```

This extension posts a message when:

- `start_msg`: The training has started
- `msg`: The extension is triggered, usually at the end of each epoch
- `end_msg`: The training has finished
- `error_msg`: An exception has raised during the training

These messages can be specified as a format string, a callable that returns a string, or None to disable posting on that event.

When using a format string, the following variables are available for use:

- `manager`: an ExtensionsManager object
- `default`: the default message string
- `context`: an arbitrary object passed to this extension
- `error`: an Exception object (for `error_msg` only)
- All reported values (`manager.observations`)

When using a callable, it should take *(ExtensionsManager, context)* or *(ExtensionsManager, Exception, context)* (for `error_msg`) and return a string.

This extension can upload files along with the message when triggered. `filenames` can be a list of filenames (the same formatting rule as `msg` apply), or a callable taking (ExtensionsManager, context) and returning a list of filenames.

To use this extension, you must create a Slack app, then specify the token via an environment variable `SLACK_BOT_TOKEN` or `token` option.

Parameters

- **channel** (*str*) – The channel where messages and files will be sent. This can be a channel name or a channel ID.
- **msg** (*str, callable, or None*) – A message to be sent when triggered. It can be a string to be formatted using `.format` or a callable that returns a string.
- **start_msg** (*str, callable, or None*) – A message to be sent at the beginning of the experiment.
- **end_msg** (*str, callable, or None*) – A message to be sent at the completion of the experiment.
- **error_msg** (*str, callable, or None*) – A message to be sent when an exception is raised during the experiment.
- **thread** (*bool*) – When True, subsequent messages will be posted as a thread of the original message. Default is `True`.
- **filenames** (*list of str or callable*) – A list of files that will be uploaded. These are string templates that can take values in the same way as `msg`, or a callable that returns a list of filenames.
- **upload_trigger** (*trigger or None*) – Used to upload files at certain events. If not specified, files will be uploaded in every call.
- **context** (*Any*) – Any arbitrary user object you will need when generating a message.
- **token** (*str*) – Slack bot token. If `None`, the environment variable `SLACK_BOT_TOKEN` will be used. Optional, default is `None`.

Methods

__init__(channel[, msg, start_msg, end_msg, ...])	
default_end_msg(context)	
default_error_msg(exc, context)	
default_msg(context)	
default_start_msg(context)	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__init__(*channel*, *msg=None*, ***, *start_msg='{default}'*, *end_msg='{default}'*, *error_msg='{default}'*, *thread=True*, *filenames=None*, *upload_trigger=None*, *context=None*, *token=None*)

Parameters

- **channel** (*str*) –
- **msg** (*Optional[Union[str, Callable[[ExtensionsManagerProtocol, Any], str]]]*) –
- **start_msg** (*Optional[Union[str, Callable[[ExtensionsManagerProtocol, Any], str]]]*) –
- **end_msg** (*Optional[Union[str, Callable[[ExtensionsManagerProtocol, Any], str]]]*) –
- **error_msg** (*Optional[Union[str, Callable[[ExtensionsManagerProtocol, Any, Exception], str]]]*) –
- **thread** (*bool*) –
- **filenames** (*Optional[Union[Sequence[str], Callable[[ExtensionsManagerProtocol, Any], Sequence[str]]]]*) –
- **upload_trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –
- **context** (*Optional[Any]*) –
- **token** (*Optional[str]*) –

Return type
None

trigger: Optional[Union[*Trigger*, Callable[[*ExtensionsManagerProtocol*], bool], Tuple[float, str]]] = (1, 'epoch')

pytorch_pfn_extras.training.extensions.SlackWebhook

class pytorch_pfn_extras.training.extensions.**SlackWebhook**(*url*, *msg=None*, ***, *start_msg='{default}'*, *end_msg='{default}'*, *error_msg='{default}'*, *context=None*)

Bases: `_SlackBase`

An extension to communicate with Slack using Incoming Webhook.

Example

```
>>> ppe.training.extensions.SlackWebhook(
...     url="https://hooks.slack.com/services/Txxxxx.....",
...     msg="Epoch #{manager.epoch}: loss = {val/loss}",
...     end_msg="{default} \n <@username> Check out the result!",
... )
```

This extension posts a message when:

- `start_msg`: The training has started
- `msg`: The extension is triggered, usually at the end of each epoch
- `end_msg`: The training has finished
- `error_msg`: An exception has raised during the training

These messages can be specified as a format string, a callable that returns a string, or None to disable posting on that event.

When using a format string, the following variables are available for use:

- `manager`: an ExtensionsManager object
- `default`: the default message string
- `context`: an arbitrary object passed to this extension
- `error`: an Exception object (for `error_msg` only)
- All reported values (`manager.observations`)

When using a callable, it should take *(ExtensionsManager, context)* or *(ExtensionsManager, Exception, context)* (for `error_msg`) and return a string.

Parameters

- **url** (*str*) – Incoming webhook URL to send messages.
- **msg** (*str, callable, or None*) – A message to be sent when triggered. It can be a string to be formatted using `.format` or a callable that returns a string.
- **start_msg** (*str, callable, or None*) – A message to be sent at the beginning of the experiment.
- **end_msg** (*str, callable, or None*) – A message to be sent at the completion of the experiment.
- **error_msg** (*str, callable, or None*) – A message to be sent when an exception is raised during the experiment.
- **context** (*object*) – Any arbitrary user object you will need when generating a message.

Methods

`__init__`(url[, msg, start_msg, end_msg, ...])	
`default_end_msg`(context)	
`default_error_msg`(exc, context)	
`default_msg`(context)	
`default_start_msg`(context)	
`finalize`(manager)	Finalizes the extension.
`initialize`(manager)	Initializes up the manager state.
`load_state_dict`(to_load)	
`on_error`(manager, exc, tb)	Handles the error raised during training before finalization.
`state_dict`()	Serializes the extension state.

Attributes

`default_name`	Default name of the extension.
`is_async`	
`name`	
`needs_model_state`	
`priority`	
`trigger`	

__init__(*url*, *msg=None*, ***, *start_msg='{default}'*, *end_msg='{default}'*, *error_msg='{default}'*, *context=None*)

Parameters

- **url** (*str*) –
- **msg** (*Optional[Union[str, Callable[[ExtensionsManagerProtocol, Any], str]]]*) –
- **start_msg** (*Optional[Union[str, Callable[[ExtensionsManagerProtocol, Any], str]]]*) –
- **end_msg** (*Optional[Union[str, Callable[[ExtensionsManagerProtocol, Any], str]]]*) –
- **error_msg** (*Optional[Union[str, Callable[[ExtensionsManagerProtocol, Any, Exception], str]]]*) –
- **context** (*Optional[Any]*) –

Return type
None

pytorch_pfn_extras.training.extensions.VariableStatisticsPlot

class pytorch_pfn_extras.training.extensions.**VariableStatisticsPlot**(*targets*, *max_sample_size=1000*, *report_data=True*, *report_grad=True*, *plot_mean=True*, *plot_std=True*, *percentile_sigmas=(0, 0.13, 2.28, 15.87, 50, 84.13, 97.72, 99.87, 100)*, *trigger=(1, 'epoch')*, *filename='statistics.png'*, *figsize=None*, *marker=None*, *grid=True*)

Bases: *Extension*

An extension to plot statistics for Tensors.

This extension collects statistics for a single torch.Tensor, a list of torch.Tensors or similarly a single or a list of torch.nn.Modules containing one or more torch.Tensors. In case multiple torch.Tensors are found, the means are computed. The collected statistics are plotted and saved as an image in the directory specified by the Manager.

Statistics include mean, standard deviation and percentiles.

This extension uses reservoir sampling to preserve memory, using a fixed size running sample. This means that collected items in the sample are discarded uniformly at random when the number of items becomes larger than the maximum sample size, but each item is expected to occur in the sample with equal probability.

:param targets (torch.Tensor: or list of either): Parameters for which statistics are collected. :param torch.nn.Module: or list of either): Parameters for which statistics are collected. :param max_sample_size: Maximum number of running samples. :type max_sample_size: int :param report_data: If True, data (e.g. weights) statistics are plotted. If

False, they are neither computed nor plotted.

Parameters

- **report_grad** (*bool*) – If True, gradient statistics are plotted. If False, they are neither computed nor plotted.
- **plot_mean** (*bool*) – If True, means are plotted. If False, they are neither computed nor plotted.
- **plot_std** (*bool*) – If True, standard deviations are plotted. If False, they are neither computed nor plotted.
- **percentile_sigmas** (*float or tuple of floats*) – Percentiles to plot in the range $[0, 100]$.
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) – Trigger that decides when to save the plots as an image. This is distinct from the trigger of this extension itself. If it is a tuple in the form <int>, 'epoch' or <int>, 'iteration', it is passed to IntervalTrigger.

- **filename** (*str*) – Name of the output image file under the output directory. For historical reasons `file_name` is also accepted as an alias of this argument.
- **figsize** (*tuple of int*) – Matlotlib `figsize` argument that specifies the size of the output image.
- **marker** (*str*) – Matplotlib `marker` argument that specified the marker style of the plots.
- **grid** (*bool*) – Matplotlib `grid` argument that specifies whether grids are rendered in in the plots or not.
- **writer** (*writer object, optional*) – must be callable. object to dump the log to. If specified, it needs to have a correct *savefun* defined. The writer can override the save location in the *pytorch_pfn_extras.training.ExtensionsManager* object
- **targets** (*Any*) –
- **max_sample_size** (*int*) –
- **report_data** (*bool*) –
- **kwargs** (*Any*) –

Methods

__init__(targets[, max_sample_size, ...])	
available()	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
save_plot_using_module(plt, manager)	
state_dict()	Serializes the extension state.

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
None

__init__(*targets*, *max_sample_size=1000*, *report_data=True*, *report_grad=True*, *plot_mean=True*, *plot_std=True*, *percentile_sigmas=(0, 0.13, 2.28, 15.87, 50, 84.13, 97.72, 99.87, 100)*, *trigger=(1, 'epoch')*, *filename=None*, *figsize=None*, *marker=None*, *grid=True*, ***kwargs*)

Parameters

- **targets** (*Any*) –
- **max_sample_size** (*int*) –
- **report_data** (*bool*) –
- **report_grad** (*bool*) –
- **plot_mean** (*bool*) –
- **plot_std** (*bool*) –
- **percentile_sigmas** (*Union[float, Tuple[float, ...]]*) –
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –
- **filename** (*Optional[str]*) –
- **figsize** (*Optional[Tuple[int, ...]]*) –
- **marker** (*Optional[str]*) –
- **grid** (*bool*) –
- **kwargs** (*Any*) –

static available()

Return type
bool

finalize(*manager*)

Finalizes the extension.

This method is called at the end of the training loop.

Parameters
manager (ExtensionsManagerProtocol) –

Return type
None

save_plot_using_module(*plt*, *manager*)

Parameters

- **plt** (*Any*) –

- **manager** (ExtensionsManagerProtocol) –

Return type
None

Modules

pytorch_pfn_extras.training.extensions.best_value
pytorch_pfn_extras.training.extensions.evaluator
pytorch_pfn_extras.training.extensions.fail_on_non_number
pytorch_pfn_extras.training.extensions.log_report
pytorch_pfn_extras.training.extensions.lr_scheduler
pytorch_pfn_extras.training.extensions.micro_average
pytorch_pfn_extras.training.extensions.parameter_statistics
pytorch_pfn_extras.training.extensions.plot_report
pytorch_pfn_extras.training.extensions.print_report
pytorch_pfn_extras.training.extensions.profile_report
pytorch_pfn_extras.training.extensions.progress_bar
pytorch_pfn_extras.training.extensions.slack
pytorch_pfn_extras.training.extensions.snapshot_writers
pytorch_pfn_extras.training.extensions.util
pytorch_pfn_extras.training.extensions.value_observation
pytorch_pfn_extras.training.extensions.variable_statistics_plot

pytorch_pfn_extras.training.extensions.best_value

Classes

pytorch_pfn_extras.training.extensions.best_value.BestValue(...)	Extension traces the best value of a specific key in the observation.
pytorch_pfn_extras.training.extensions.best_value.ExtensionsManagerProtocol(...)	
pytorch_pfn_extras.training.extensions.best_value.MaxValue(key)	Extension traces the maximum value of a specific key in the observation.
pytorch_pfn_extras.training.extensions.best_value.MinValue(key)	Extension traces the maximum value of a specific key in the observation.

pytorch_pfn_extras.training.extensions.best_value.BestValue

class pytorch_pfn_extras.training.extensions.best_value.**BestValue**(*key*, *compare*, *trigger=(1, 'epoch')*)

Bases: *Extension*

Extension traces the best value of a specific key in the observation.

Parameters

- **key** (*str*) – Key of value.
- **compare** (*callable*) – Compare function which takes current best value and new value and returns whether new value is better than current best.
- **trigger** (*TriggerLike*) – Trigger that decides the comparison interval between current best value and new value. This must be a tuple in the form of `<int>, 'epoch'` or `<int>, 'iteration'` which is passed to `BestValueTrigger`.

Methods

__init__(key, compare[, trigger])	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

best_epoch	Returns the epoch count that the current best value is observed.
best_iteration	Returns the iteration count that the current best value is observed.
best_value	Returns the current best value.
default_name	
is_async	
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
None

__init__(*key*, *compare*, *trigger=(1, 'epoch')*)

Parameters

- **key** (*str*) –
- **compare** (*Callable[[float, float], bool]*) –
- **trigger** (*TriggerLike*) –

Return type
None

property best_epoch: int

Returns the epoch count that the current best value is observed.

If no value has been observed yet, it raises a RuntimError.

property best_iteration: int

Returns the iteration count that the current best value is observed.

If no value has been observed yet, it raises a RuntimError.

property best_value: float

Returns the current best value.

If no value has been observed yet, it raises a RuntimError.

default_name = 'best_value'

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

state_dict()

Serializes the extension state.

It is called when a manager that owns this extension is serialized. It serializes nothing by default.

Return type
Dict[str, *Any*]

pytorch_pfn_extras.training.extensions.best_value.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.extensions.best_value.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: Protocol

Methods

__init__(*args, **kwargs)
get_extension(name)

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.extensions.best_value.MaxValue

class pytorch_pfn_extras.training.extensions.best_value.**MaxValue**(*key*, *trigger=(1, 'epoch')*)

Bases: *BestValue*

Extension traces the maximum value of a specific key in the observation.

Parameters

- **key** (*str*) – Key of value.
- **trigger** (*TriggerLike*) – Trigger that decides the comparison interval between current maximum value and new value. This must be a tuple in the form of `<int>, 'epoch'` or `<int>, 'iteration'` which is passed to `BestValueTrigger`.

Methods

__init__(key[, trigger])	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

best_epoch	Returns the epoch count that the current best value is observed.
best_iteration	Returns the iteration count that the current best value is observed.
best_value	Returns the current best value.
default_name	
is_async	
name	
needs_model_state	
priority	
trigger	

__init__(*key*, *trigger=(1, 'epoch')*)

Parameters

- **key** (*str*) –
- **trigger** (*TriggerLike*) –

default_name = 'max_value'

pytorch_pfn_extras.training.extensions.best_value.MinValue

class pytorch_pfn_extras.training.extensions.best_value.**MinValue**(*key*, *trigger=(1, 'epoch')*)

Bases: *BestValue*

Extension traces the maximum value of a specific key in the observation.

Parameters

- **key** (*str*) – Key of value.
- **trigger** (*TriggerLike*) – Trigger that decides the comparison interval between current maximum value and new value. This must be a tuple in the form of `<int>, 'epoch'` or `<int>, 'iteration'` which is passed to `BestValueTrigger`.

Methods

__init__(key[, trigger])	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

best_epoch	Returns the epoch count that the current best value is observed.
best_iteration	Returns the iteration count that the current best value is observed.
best_value	Returns the current best value.
default_name	
is_async	
name	
needs_model_state	
priority	
trigger	

__init__(*key*, *trigger=(1, 'epoch')*)

Parameters

- **key** (*str*) –
- **trigger** (*TriggerLike*) –

default_name = 'min_value'

pytorch_pfn_extras.training.extensions.evaluator

Classes

pytorch_pfn_extras.training.extensions.evaluator.DistributedEvaluator(...)	An extension to evaluate models on a validation set in a distributed training setup.
pytorch_pfn_extras.training.extensions.evaluator.Evaluator(...)	An extension to evaluate models on a validation set.
pytorch_pfn_extras.training.extensions.evaluator.ExtensionsManagerProtocol(...)	
pytorch_pfn_extras.training.extensions.evaluator.IgniteEvaluator(...)	
pytorch_pfn_extras.training.extensions.evaluator.IterationStatus(size)	
pytorch_pfn_extras.training.extensions.evaluator.TextIO(...)	Typed version of the return of open() in text mode.

pytorch_pfn_extras.training.extensions.evaluator.DistributedEvaluator

class pytorch_pfn_extras.training.extensions.evaluator.**DistributedEvaluator**(*self*, *iterator*, *target*, *eval_func=None*, ***, *progress_bar=False*)

Bases: *Evaluator*

An extension to evaluate models on a validation set in a distributed training setup.

In case torch.distributed is used to parallelize training iterations, it is efficient to also run evaluation in parallel by splitting the validation set to each worker process and conduct evaluation separately followed by aggregation of results of each worker, which can be achieved by :class:~`DistributedEvaluator`.

This extension basically behaves similarly to *Evaluator*, but adds an aggregation step in *Evaluator.evaluate()*. A summary of evaluation (`DictSummary`) in each worker process is collected in "all-gather" manner and then accumulated. Therefore all the worker processes must attend the evaluation, i.e., make sure all the processes have a *Evaluator* extension object configured in the `ExtensionManager` with the same trigger. All the worker process will get identical evaluation result returned by *Evaluator.evaluate()* and reported to an observation.

It is necessary to pass a DataLoader with an appropripate sampler which properly splits the validation dataset to each MPI worker process. PyTorch DistributedSampler implements this, but it allows sampler repetition in order to make the number of samples assigned to each process identical. For evaluation purpose it distorts the evaluation result, hence it is recommended to use `DistributedValidationSampler` instead.

Methods

__init__(iterator, target[, eval_hook, ...])	
add_metric(metric_fn)	Adds a custom metric to the evaluator.
eval_func(*args, **kwargs)	
evaluate()	Evaluates the model and returns a result dictionary.
finalize(manager)	Finalizes the extension.
get_all_iterators()	Returns a dictionary of all iterators.
get_all_targets()	Returns a dictionary of all target links.
get_iterator(name)	Returns the iterator of the given name.
get_target(name)	Returns the target link of the given name.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

default_name
is_async
name
needs_model_state
priority
trigger

Parameters

- **iterator** (*Union[DataLoader[Any], Dict[str, DataLoader[Any]]]*) –
- **target** (*Union[Module, Dict[str, Module]]*) –
- **eval_hook** (*Optional[Callable[[Evaluator], None]]*) –
- **eval_func** (*Optional[Callable[[...], Any]]*) –
- **kwargs** (*Any*) –

__init__(*iterator*, *target*, *eval_hook=None*, *eval_func=None*, ***kwargs*)

Parameters

- **iterator** (*Union[DataLoader[Any], Dict[str, DataLoader[Any]]]*) –
- **target** (*Union[Module, Dict[str, Module]]*) –

- **eval_hook** (*Optional[Callable[[Evaluator], None]]*) –
- **eval_func** (*Optional[Callable[...], Any]]*) –
- **kwargs** (*Any*) –

Return type
None

pytorch_pfn_extras.training.extensions.evaluator.Evaluator

class pytorch_pfn_extras.training.extensions.evaluator.**Evaluator**(*self*, *iterator*, *target*, *eval_func=None*, ***, *progress_bar=False*)

Bases: *Extension*

An extension to evaluate models on a validation set.

This extension evaluates the current models by a given evaluation function. It creates a `Reporter` object to store values observed in the evaluation function on each iteration. The report for all iterations are aggregated to `DictSummary`. The collected mean values are further reported to the reporter object of the manager, where the name of each observation is prefixed by the evaluator name. See `Reporter` for details in naming rules of the reports.

Evaluator has a structure to customize similar to that of `StandardUpdater`. The main differences are:

- There are no optimizers in an evaluator. Instead, it holds links to evaluate.
- An evaluation loop function is used instead of an update function.
- Preparation routine can be customized, which is called before each evaluation. It can be used, e.g., to initialize the state of stateful recurrent networks.

There are two ways to modify the evaluation behavior besides setting a custom evaluation function. One is by setting a custom evaluation loop via the `eval_func` argument. The other is by inheriting this class and overriding the *evaluate()* method. In latter case, users have to create and handle a reporter object manually. Users also have to copy the iterators before using them, in order to reuse them at the next time of evaluation. In both cases, the functions are called in testing mode

This extension is called at the end of each epoch by default.

Parameters

- **iterator** (*Union[DataLoader[Any], Dict[str, DataLoader[Any]]]*) – Dataset iterator for the validation dataset. It can also be a dictionary of iterators. If this is just an iterator, the iterator is registered by the name `'main'`.
- **target** (*Union[Module, Dict[str, Module]]*) – torch.nn.Module object or a dictionary of links to evaluate. If this is just a layer object, the link is registered by the name `'main'`.
- **eval_func** (*Optional[Callable[...], Any]]*) – Evaluation function called at each iteration. The target link to evaluate as a callable is used by default.
- **progress_bar** – Boolean flag to show a progress bar while training, which is similar to `ProgressBar`. (default: `False`)
- **metrics** – List of callables that are called every batch to calculate metrics such as accuracy, roc_auc or others The signature of the callable is: *def metric_fn(batch, output, last_iteration)* (default: `[]`)
- **eval_hook** (*Optional[Callable[[Evaluator], None]]*) –

- **kwargs** (*Any*) –

Warning: The argument `progress_bar` is experimental. The interface can change in the future.

eval_hook

Function to prepare for each evaluation process.

eval_func

Evaluation function called at each iteration.

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type

Any

Methods

__init__(iterator, target[, eval_hook, ...])	
add_metric(metric_fn)	Adds a custom metric to the evaluator.
eval_func(*args, **kwargs)	
evaluate()	Evaluates the model and returns a result dictionary.
finalize(manager)	Finalizes the extension.
get_all_iterators()	Returns a dictionary of all iterators.
get_all_targets()	Returns a dictionary of all target links.
get_iterator(name)	Returns the iterator of the given name.
get_target(name)	Returns the target link of the given name.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

default_name
is_async
name
needs_model_state
priority
trigger

__call__(*manager=None*)

Executes the evaluator extension.

Unlike usual extensions, this extension can be executed without passing a manager object. This extension reports the performance on validation dataset using the `report()` function. Thus, users can use this extension independently from any manager by manually configuring a `Reporter` object.

Parameters
manager (ExtensionsManager) – Manager object that invokes this extension.

Returns
Result dictionary that contains mean statistics of values reported by the evaluation function.

Return type
dict

__init__(*iterator*, *target*, *eval_hook=None*, *eval_func=None*, ***kwargs*)

Parameters

- **iterator** (*Union[DataLoader[Any], Dict[str, DataLoader[Any]]]*) –
- **target** (*Union[Module, Dict[str, Module]]*) –
- **eval_hook** (*Optional[Callable[[Evaluator], None]]*) –
- **eval_func** (*Optional[Callable[[...], Any]]*) –
- **kwargs** (*Any*) –

Return type
None

add_metric(*metric_fn*)

Adds a custom metric to the evaluator.

The metric is a callable that is executed every batch with the following signature: *def metric_fn(batch, output, last_iteration)*

Batch is the input batch passed to the model. Output is the result of evaluating batch, last_iteration is a boolean flag that indicates if its the last batch in the evaluation.

Parameters
metric_fn (*Callable[[Any, Any, Any], None]*) –

Return type
None

`default_name = 'validation'`

`eval_func`(**args*, ***kwargs*)

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

`evaluate`()

Evaluates the model and returns a result dictionary.

This method runs the evaluation loop over the validation dataset. It accumulates the reported values to `DictSummary` and returns a dictionary whose values are means computed by the summary.

Users can override this method to customize the evaluation routine.

Returns
Result dictionary. This dictionary is further reported via `report()` without specifying any observer.

Return type
dict

`get_all_iterators`()

Returns a dictionary of all iterators.

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

`get_all_targets`()

Returns a dictionary of all target links.

Return type
Dict[str, *Module*]

`get_iterator`(*name*)

Returns the iterator of the given name.

Parameters
name (*str*) –

Return type
DataLoader[*Any*]

`get_target`(*name*)

Returns the target link of the given name.

Parameters
name (*str*) –

Return type
Module

`priority: int = 300`

`trigger: TriggerLike = (1, 'epoch')`

pytorch_pfn_extras.training.extensions.evaluator.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.extensions.evaluator.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: Protocol

Methods

__init__(*args, **kwargs)
get_extension(name)

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

`get_extension`(*name*)

Parameters
name (*str*) –

Return type
Extension

`property is_before_training: bool`

`property iteration: int`

`property models: Mapping[str, Module]`

`property observation: reporting.Observation`

`property optimizers: Mapping[str, Optimizer]`

`property out: str`

`property raw_models: Mapping[str, Module]`

`property reporter: reporting.Reporter`

`property stop_trigger: bool`

`property writer: Optional[writing.Writer]`

pytorch_pfn_extras.training.extensions.evaluator.IgniteEvaluator

`class` pytorch_pfn_extras.training.extensions.evaluator.`IgniteEvaluator`(*evaluator*, *iterator*, *target*, ***kwargs*)

Bases: *Evaluator*

Methods

__init__(evaluator, iterator, target, **kwargs)	
add_metric(metric_fn)	Adds a custom metric to the evaluator.
eval_func(*args, **kwargs)	
evaluate()	Evaluates the model and returns a result dictionary.
finalize(manager)	Finalizes the extension.
get_all_iterators()	Returns a dictionary of all iterators.
get_all_targets()	Returns a dictionary of all target links.
get_iterator(name)	Returns the iterator of the given name.
get_target(name)	Returns the target link of the given name.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
set_evaluator_handlers()	
state_dict()	Serializes the extension state.

Attributes

default_name
is_async
name
needs_model_state
priority
trigger

Parameters

- **evaluator** (*Engine*) –
- **iterator** (*Union[DataLoader[Any], Dict[str, DataLoader[Any]]]*) –
- **target** (*Union[Module, Dict[str, Module]]*) –
- **kwargs** (*Any*) –

__init__(*evaluator*, *iterator*, *target*, ***kwargs*)

Parameters

- **evaluator** (*Engine*) –
- **iterator** (*Union[DataLoader[Any], Dict[str, DataLoader[Any]]]*) –
- **target** (*Union[Module, Dict[str, Module]]*) –
- **kwargs** (*Any*) –

evaluate()

Evaluates the model and returns a result dictionary.

This method runs the evaluation loop over the validation dataset. It accumulates the reported values to `DictSummary` and returns a dictionary whose values are means computed by the summary.

Users can override this method to customize the evaluation routine.

Returns
Result dictionary. This dictionary is further reported via `report()` without specifying any observer.

Return type
dict

set_evaluator_handlers()

Return type
None

pytorch_pfn_extras.training.extensions.evaluator.IterationStatus

class pytorch_pfn_extras.training.extensions.evaluator.**IterationStatus**(*size*)

Bases: object

Methods

__init__(size)

Attributes

epoch_detail

Parameters
size (*int*) –

__init__(*size*)

Parameters
size (*int*) –

Return type
None

property **epoch_detail: float**

pytorch_pfn_extras.training.extensions.evaluator.TextIO

class pytorch_pfn_extras.training.extensions.evaluator.**TextIO**(**args*, ***kwds*)

Bases: *IO*[str]

Typed version of the return of open() in text mode.

Methods

`__init__`()
`close`()
`fileno`()
`flush`()
`isatty`()
`read`([n])
`readable`()
`readline`([limit])
`readlines`([hint])
`seek`(offset[, whence])
`seekable`()
`tell`()
`truncate`([size])
`writable`()
`write`(s)
`writelines`(lines)

Attributes

buffer
closed
encoding
errors
line_buffering
mode
name
newlines

abstract property buffer: BinaryIO

abstract property encoding: str

abstract property errors: Optional[str]

abstract property line_buffering: bool

abstract property newlines: Any

pytorch_pfn_extras.training.extensions.fail_on_non_number

Classes

pytorch_pfn_extras.training.extensions.fail_on_non_number.ExtensionsManagerProtocol(...)	
pytorch_pfn_extras.training.extensions.fail_on_non_number.FailOnNonNumber(*)	An extension to raise RuntimeError if parameters and its gradients contain NaN or Inf.

pytorch_pfn_extras.training.extensions.fail_on_non_number.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.extensions.fail_on_non_number.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: Protocol

Methods

__init__(*args, **kwargs)
get_extension(name)

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.extensions.fail_on_non_number.FailOnNonNumber

class pytorch_pfn_extras.training.extensions.fail_on_non_number.**FailOnNonNumber**(*, *check_grad=True*)

Bases: *Extension*

An extension to raise RuntimeError if parameters and its gradients contain NaN or Inf.

Although parameters including non-number such as NaN and Inf are unnecessary in most cases the training loop will continue to compute even if the parameters in a given optimizer diverge. This extension is aimed to reduce unnecessary computations by throwing `RuntimeError` if the parameters contain NaN or Inf.

Parameters

check_grad (*bool*) – Set to False to skip checking gradients.

Methods

__init__(*[, check_grad])	
`finalize`(manager)	Finalizes the extension.
`initialize`(manager)	Initializes up the manager state.
`load_state_dict`(to_load)	
`on_error`(manager, exc, tb)	Handles the error raised during training before finalization.
`state_dict`()	Serializes the extension state.

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
None

__init__(*, *check_grad=True*)

Parameters
check_grad (*bool*) –

needs_model_state = True

pytorch_pfn_extras.training.extensions.log_report

Classes

pytorch_pfn_extras.training.extensions.log_report.ExtensionsManagerProtocol(...)	
pytorch_pfn_extras.training.extensions.log_report.LogReport([...])	An extension to output the accumulated results to a log file.
pytorch_pfn_extras.training.extensions.log_report.LogWriterSaveFunc(...)	

pytorch_pfn_extras.training.extensions.log_report.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.extensions.log_report.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: Protocol

Methods

__init__(*args, **kwargs)
get_extension(name)

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

`get_extension`(*name*)

Parameters
name (*str*) –

Return type
Extension

`property is_before_training: bool`

`property iteration: int`

`property models: Mapping[str, Module]`

`property observation: reporting.Observation`

`property optimizers: Mapping[str, Optimizer]`

`property out: str`

`property raw_models: Mapping[str, Module]`

`property reporter: reporting.Reporter`

`property stop_trigger: bool`

`property writer: Optional[writing.Writer]`

pytorch_pfn_extras.training.extensions.log_report.LogReport

`class` pytorch_pfn_extras.training.extensions.log_report.`LogReport`(*keys=None*, *trigger=(1, 'epoch')*, *postprocess=None*, *filename=None*, *append=False*, *format=None*, ***kwargs*)

Bases: *Extension*

An extension to output the accumulated results to a log file.

This extension accumulates the observations of the manager to `DictSummary` at a regular interval specified by a supplied trigger, and writes them into a log file in JSON format.

There are two triggers to handle this extension. One is the trigger to invoke this extension, which is used to handle the timing of accumulating the results. It is set to `1, 'iteration'` by default. The other is the trigger to determine when to emit the result. When this trigger returns True, this extension appends the summary of accumulated values to the list of past summaries, and writes the list to the log file. Then, this extension makes a new fresh summary object which is used until the next time that the trigger fires.

It also adds some entries to each result dictionary.

- `'epoch'` and `'iteration'` are the epoch and iteration counts at the output, respectively.
- `'elapsed_time'` is the elapsed time in seconds since the training begins. The value is taken from `ExtensionsManager.elapsed_time`.

Parameters

- **keys** (*iterable of strs*) – Keys of values to accumulate. If this is None, all the values are accumulated and output to the log file.

- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) – Trigger that decides when to aggregate the result and output the values. This is distinct from the trigger of this extension itself. If it is a tuple in the form `<int>, 'epoch'` or `<int>, 'iteration'`, it is passed to `IntervalTrigger`.
- **postprocess** (*Optional[Callable[[Mapping[str, Any]], None]]*) – Callback to postprocess the result dictionaries. Each result dictionary is passed to this callback on the output. This callback can modify the result dictionaries, which are used to output to the log file.
- **filename** (*str*) – Name of the log file under the output directory. It can be a format string: the last result dictionary is passed for the formatting. For example, users can use '{iteration}' to separate the log files for different iterations. (default: `'log'`)
- **append** (*bool, optional*) – If the file is JSON Lines or YAML, contents will be appended instead of rewritting the file every call.
- **format** (*str, optional*) – accepted values are *'json'*, *'json-lines'* and *'yaml'*.
- **writer** (*writer object, optional*) – must be callable. object to dump the log to. If specified, it needs to have a correct *savefun* defined. The writer can override the save location in the *pytorch_pfn_extras.training.ExtensionsManager* object
- **kwargs** (*Any*) –

Note: Enabling *append=True* reduces size of snapshots (and thus reduces the time needed to take snapshots). Note that extensions relying on the logs from past iterations may behave differently; for example, when resuming from a snapshot, PrintReport will not show logs of iterations already done.

Methods

__init__([keys, trigger, postprocess, ...])	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.
to_dataframe()	

Attributes

default_name	Default name of the extension.
is_async	
log	The current list of observation dictionaries.
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
None

__init__(*keys=None*, *trigger=(1, 'epoch')*, *postprocess=None*, *filename=None*, *append=False*, *format=None*, ***kwargs*)

Parameters

- **keys** (*Optional[Iterable[str]]*) –
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –
- **postprocess** (*Optional[Callable[[Mapping[str, Any]], None]]*) –
- **filename** (*Optional[str]*) –
- **append** (*bool*) –
- **format** (*Optional[str]*) –
- **kwargs** (*Any*) –

finalize(*manager*)

Finalizes the extension.

This method is called at the end of the training loop.

Parameters
manager (ExtensionsManagerProtocol) –

Return type
None

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

property log: List[Mapping[str, Any]]

The current list of observation dictionaries.

state_dict()

Serializes the extension state.

It is called when a manager that owns this extension is serialized. It serializes nothing by default.

Return type
Dict[str, *Any*]

to_dataframe()

Return type
pandas.DataFrame

pytorch_pfn_extras.training.extensions.log_report.LogWriterSaveFunc

class pytorch_pfn_extras.training.extensions.log_report.**LogWriterSaveFunc**(*format*, *append*)

Bases: object

Methods

__init__(format, append)

Parameters

- **format** (*str*) –
- **append** (*bool*) –

__call__(*target*, *file_o*)

Call self as a function.

Parameters

- **target** (*Dict[str, Any]*) –
- **file_o** (*Any*) –

Return type
None

__init__(*format*, *append*)

Parameters

- **format** (*str*) –
- **append** (*bool*) –

Return type
None

pytorch_pfn_extras.training.extensions.lr_scheduler

Classes

pytorch_pfn_extras.training.extensions.lr_scheduler.ExtensionsManagerProtocol(...)	
pytorch_pfn_extras.training.extensions.lr_scheduler.LRScheduler(...)	Trainer extension to adjust the learning rate using PyTorch's learning rate scheduler.
pytorch_pfn_extras.training.extensions.lr_scheduler.ReduceLROnPlateau(...)	Reduce learning rate when a metric has stopped improving.

pytorch_pfn_extras.training.extensions.lr_scheduler.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.extensions.lr_scheduler.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: `Protocol`

Methods

__init__(*args, **kwargs)	
get_extension(name)	

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.extensions.lr_scheduler.LRScheduler

class pytorch_pfn_extras.training.extensions.lr_scheduler.**LRScheduler**(*scheduler*, *, *stepper=<function _default_stepper>*, *trigger=(1, 'epoch')*, *wait_for_first_optimizer_step=False*, *is_async=True*)

Bases: *Extension*

Trainer extension to adjust the learning rate using PyTorch's learning rate scheduler.

This extension calls *step()* method of the given LR scheduler. (*torch.option.lr_scheduler.**). When using *ReduceLROnPlateau*, the latest reported *val/loss* value will be used. This behavior can be customized by passing a custom *stepper* function.

Parameters

- **scheduler** (*_LRScheduler or* ReduceLROnPlateau) – Any instance of *torch.optim.lr_scheduler.**.
- **stepper** (*callable*) – Function that performs the step on the scheduler.
- **trigger** (*Optional[Union[*Trigger*, Callable[[*ExtensionsManagerProtocol*], bool], Tuple[float, str]]]*) – Frequency to call this extension.
- **wait_for_first_optimizer_step** (*bool*) – Wait until *optimizer.step()* is called before invoking *scheduler.step()*. This can address the issue where *optimizer.step()* is not called from the first iteration when using GradScaler.
- **is_async** (*bool*) –

Methods

__init__(scheduler, *[, stepper, trigger, ...])	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(state)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.
step_by_value(key)	

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
None

__init__(*scheduler*, *, *stepper=<function _default_stepper>*, *trigger=(1, 'epoch')*, *wait_for_first_optimizer_step=False*, *is_async=True*)

Parameters

- **scheduler** (*Any*) –
- **stepper** (*Any*) –
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –
- **wait_for_first_optimizer_step** (*bool*) –
- **is_async** (*bool*) –

Return type
None

load_state_dict(*state*)

Parameters
state (*Dict[str, Any]*) –

Return type
None

state_dict()

Serializes the extension state.

It is called when a manager that owns this extension is serialized. It serializes nothing by default.

Return type
Dict[str, *Any*]

static step_by_value(*key*)

Parameters
key (*Optional[str]*) –

Return type
Any

pytorch_pfn_extras.training.extensions.lr_scheduler.ReduceLROnPlateau

class pytorch_pfn_extras.training.extensions.lr_scheduler.**ReduceLROnPlateau**(*optimizer*, *mode='min'*, *factor=0.1*, *patience=10*, *threshold=0.0001*, *threshold_mode='rel'*, *cooldown=0*, *min_lr=0*, *eps=1e-08*, *verbose=False*)

Bases: object

Reduce learning rate when a metric has stopped improving. Models often benefit from reducing the learning rate by a factor of 2-10 once learning stagnates. This scheduler reads a metrics quantity and if no improvement is seen for a 'patience' number of epochs, the learning rate is reduced.

Parameters

- **optimizer** (*Optimizer*) – Wrapped optimizer.
- **mode** (*str*) – One of *min*, *max*. In *min* mode, lr will be reduced when the quantity monitored has stopped decreasing; in *max* mode it will be reduced when the quantity monitored has stopped increasing. Default: 'min'.
- **factor** (*float*) – Factor by which the learning rate will be reduced. new_lr = lr * factor. Default: 0.1.
- **patience** (*int*) – Number of epochs with no improvement after which learning rate will be reduced. For example, if *patience* = *2*, then we will ignore the first 2 epochs with no improvement, and will only decrease the LR after the 3rd epoch if the loss still hasn't improved then. Default: 10.
- **threshold** (*float*) – Threshold for measuring the new optimum, to only focus on significant changes. Default: 1e-4.
- **threshold_mode** (*str*) – One of *rel*, *abs*. In *rel* mode, dynamic_threshold = best * (1 + threshold) in 'max' mode or best * (1 - threshold) in *min* mode. In *abs* mode, dynamic_threshold = best + threshold in *max* mode or best - threshold in *min* mode. Default: 'rel'.
- **cooldown** (*int*) – Number of epochs to wait before resuming normal operation after lr has been reduced. Default: 0.
- **min_lr** (*float or list*) – A scalar or a list of scalars. A lower bound on the learning rate of all param groups or each group respectively. Default: 0.

- **eps** (*float*) – Minimal decay applied to lr. If the difference between new and old lr is smaller than eps, the update is ignored. Default: 1e-8.
- **verbose** (*bool*) – If `True`, prints a message to stdout for each update. Default: `False`.

Example

```
>>> # xdoctest: +SKIP
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
>>> scheduler = ReduceLROnPlateau(optimizer, 'min')
>>> for epoch in range(10):
>>>     train(...)
>>>     val_loss = validate(...)
>>>     # Note that step should be called after validate()
>>>     scheduler.step(val_loss)
```

Methods

__init__(optimizer[, mode, factor, ...])
is_better(a, best)
load_state_dict(state_dict)
state_dict()
step(metrics[, epoch])

Attributes

in_cooldown

__init__(*optimizer*, *mode='min'*, *factor=0.1*, *patience=10*, *threshold=0.0001*, *threshold_mode='rel'*, *cooldown=0*, *min_lr=0*, *eps=1e-08*, *verbose=False*)

property in_cooldown

is_better(*a*, *best*)

load_state_dict(*state_dict*)

state_dict()

step(*metrics*, *epoch=None*)

pytorch_pfn_extras.training.extensions.micro_average

Classes

pytorch_pfn_extras.training.extensions.micro_average.ExtensionsManagerProtocol(...)	
pytorch_pfn_extras.training.extensions.micro_average.MicroAverage(...)	Calculates micro-average ratio.

pytorch_pfn_extras.training.extensions.micro_average.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.extensions.micro_average.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: Protocol

Methods

__init__(*args, **kwargs)
get_extension(name)

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.extensions.micro_average.MicroAverage

class pytorch_pfn_extras.training.extensions.micro_average.**MicroAverage**(*numerator_key*, *denominator_key*, *result_key*, *trigger=(1, 'epoch')*)

Bases: *Extension*

Calculates micro-average ratio.

Give N batches and values $\{n_1, \ldots, n_N\}$ and $\{d_1, \ldots, d_N\}$, this extension calculates micro-average of these ratio defined as:

$$\frac{\sum_i^N n_i}{\sum_i^N d_i}.$$

A user usually uses the number of examples which a system correctly predict as n_i and the number of total examples in i-th batch as d_i. This value is called macro-average of precision.

Note that macro-average is defined as:

$$\frac{1}{N}\sum_i^N (n_i/d_i),$$

It is same to the micro-average when each mini-batch has the same d_i.

You need to report numerator value (the number of correct examples) and denominator value (the number of examples) in your model.

```
>>> class MyModel(torch.nn.Module):
...     def __call__(self, x, y):
...         loss = torch.nn.CrossEntropyLoss(x, y)
...         correct = (x.data.argmax(axis=1) == y.data).sum()
...         total = len(y.data)
...         reporting.report({'correct': correct, 'total': total}, self)
...         return loss
```

And then, make an extension with corresponding reporting keys and register it.

```
>>> ext = extensions.MicroAverage(
...     'main/correct', 'main/total', 'main/accuracy')
```

Parameters

- **numerator_key** (*str*) – Key string of obserbation storing a numerator value.
- **denominator_key** (*str*) – Key string of obserbation storing a denominator value.

- **result_key** (*str*) – Key string of obserbation to store a result.
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) – Trigger that decides when to calcurate average. This is distinct from the trigger of this extension itself. If it is a tuple in the form `<int>, 'epoch'` or `<int>, 'iteration'`, it is passed to `IntervalTrigger`.

Methods

__init__(numerator_key, denominator_key, ...)	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
None

__init__(*numerator_key*, *denominator_key*, *result_key*, *trigger=(1, 'epoch')*)

Parameters

- **numerator_key** (*str*) –
- **denominator_key** (*str*) –
- **result_key** (*str*) –

- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –

Return type
None

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

priority: int = 200

state_dict()

Serializes the extension state.

It is called when a manager that owns this extension is serialized. It serializes nothing by default.

Return type
Dict[str, *Any*]

pytorch_pfn_extras.training.extensions.parameter_statistics

Classes

pytorch_pfn_extras.training.extensions.parameter_statistics.ExtensionsManagerProtocol(...)	
pytorch_pfn_extras.training.extensions.parameter_statistics.ParameterStatistics(links)	An extension to report parameter statistics.

pytorch_pfn_extras.training.extensions.parameter_statistics.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.extensions.parameter_statistics.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: Protocol

Methods

__init__(*args, **kwargs)	
get_extension(name)	

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.extensions.parameter_statistics.ParameterStatistics

class pytorch_pfn_extras.training.extensions.parameter_statistics.**ParameterStatistics**(*links*, *statistics='default'*, *report_params=True*, *report_grads=True*, *prefix=None*, *trigger=(1, 'epoch')*, *skip_nan_params=Fal*

Bases: *Extension*

An extension to report parameter statistics.

Statistics are collected and reported for a given `Module` or an iterable of `Module`s. If a link contains child modules, the statistics are reported separately for each child.

Any function that takes a one-dimensional `torch.Tensor` and outputs a single or multiple real numbers can be registered to handle the collection of statistics, e.g. `numpy.ndarray.mean()`.

The keys of reported statistics follow the convention of link name followed by parameter name, attribute name and function name, e.g. `VGG16Layers/conv1_1/W/data/mean`. They are prepended with an optional prefix and appended with integer indices if the statistics generating function return multiple values.

Parameters

- **links** (*instance or iterable of Module*) – Module(s) containing the parameters to observe. The link is expected to have a `name` attribute which is used as a part of the report key.
- **statistics** (*dict or 'default'*) – Dictionary with function name to function mappings. The name is a string and is used as a part of the report key. The function is responsible for generating the statistics. If the special value `'default'` is specified, the default statistics functions will be used.
- **report_params** (*bool*) – If `True`, report statistics for parameter values such as weights and biases.
- **report_grads** (*bool*) – If `True`, report statistics for parameter gradients.
- **prefix** (*str*) – Optional prefix to prepend to the report keys.

- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) – Trigger that decides when to aggregate the results and report the values.
- **skip_nan_params** (*bool*) – If `True`, statistics are not computed for parameters including NaNs and a single NaN value is immediately reported instead. Otherwise, this extension will simply try to compute the statistics without performing any checks for NaNs.

Note: The default statistic functions are as follows:

- `'mean'` (`xp.mean(x)`)
- `'std'` (`xp.std(x)`)
- `'min'` (`xp.min(x)`)
- `'max'` (`xp.max(x)`)
- `'zeros'` (`xp.count_nonzero(x == 0)`)
- `'percentile'` (`xp.percentile(x, (0.13, 2.28, 15.87, 50, 84.13, 97.72, 99.87))`)

Methods

__init__(links[, statistics, report_params, ...])

finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
register_statistics(name, function)	Register a function to compute a certain statistic.
state_dict()	Serializes the extension state.

Attributes

default_name
default_statistics
is_async
name
needs_model_state
priority
report_key_template
trigger

__call__(*manager*)

Execute the statistics extension.

Collect statistics for the current state of parameters.

Note that this method will merely update its statistic summary, unless the internal trigger is fired. If the trigger is fired, the summary will also be reported and then reset for the next accumulation.

Parameters
manager (ExtensionsManager) – Associated manager that invoked this extension.

Return type
None

__init__(*links*, *statistics='default'*, *report_params=True*, *report_grads=True*, *prefix=None*, *trigger=(1, 'epoch')*, *skip_nan_params=False*)

Parameters

- **links** (*Any*) –
- **statistics** (*Any*) –
- **report_params** (*bool*) –
- **report_grads** (*bool*) –
- **prefix** (*Optional[str]*) –
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –
- **skip_nan_params** (*bool*) –

default_name = 'parameter_statistics'

default_statistics = {'max': <function <lambda>>, 'mean': <function <lambda>>, 'min': <function <lambda>>, 'std': <function <lambda>>, 'zeros': <function <lambda>>}

priority: int = 300

register_statistics(*name*, *function*)

Register a function to compute a certain statistic.

The registered function will be called each time the extension runs and the results will be included in the report.

Parameters

- **name** (*str*) – Name of the statistic.
- **function** (*Any*) – Function to generate the statistic. Any function that takes a one-dimensional `numpy.ndarray` or a `cupy.ndarray` and outputs a single or multiple real numbers is allowed.

Return type
None

report_key_template = '{prefix}{param_name}/{attr_name}/{function_name}'

pytorch_pfn_extras.training.extensions.plot_report

Functions

pytorch_pfn_extras.training.extensions.plot_report.matplotlib_savefun(...)	

pytorch_pfn_extras.training.extensions.plot_report.matplotlib_savefun

pytorch_pfn_extras.training.extensions.plot_report.**matplotlib_savefun**(*target*, *file_o*)

Parameters

- **target** (*Tuple[Any, Any, Any]*) –
- **file_o** (*Any*) –

Return type
None

Classes

pytorch_pfn_extras.training.extensions.plot_report.ExtensionsManagerProtocol(...)	
pytorch_pfn_extras.training.extensions.plot_report.PlotReport(y_keys)	An extension to output plots.

pytorch_pfn_extras.training.extensions.plot_report.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.extensions.plot_report.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: Protocol

Methods

__init__(*args, **kwargs)	
get_extension(name)	

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.extensions.plot_report.PlotReport

class pytorch_pfn_extras.training.extensions.plot_report.**PlotReport**(*y_keys*, *x_key='iteration'*, *trigger=(1, 'epoch')*, *postprocess=None*, *filename='plot.png'*, *marker='x'*, *grid=True*)

Bases: *Extension*

An extension to output plots.

This extension accumulates the observations of the manager to *DictSummary* at a regular interval specified by a supplied trigger, and plot a graph with using them.

There are two triggers to handle this extension. One is the trigger to invoke this extension, which is used to handle the timing of accumulating the results. It is set to `1, 'iteration'` by default. The other is the trigger to determine when to emit the result. When this trigger returns True, this extension appends the summary of accumulated values to the list of past summaries, and writes the list to the log file. Then, this extension makes a new fresh summary object which is used until the next time that the trigger fires.

It also adds `'epoch'` and `'iteration'` entries to each result dictionary, which are the epoch and iteration counts at the output.

Warning: If your environment needs to specify a backend of matplotlib explicitly, please call `matplotlib.use` before calling `manager.run_iteration`. For example:

```
import matplotlib
matplotlib.use('Agg')

manager.extend(
    extensions.PlotReport(['main/loss', 'validation/main/loss'],
                          'epoch', filename='loss.png'))
with manager.run_iteration():
    pass
```

Then, once one of instances of this extension is called, `matplotlib.use` will have no effect.

For the details, please see here: https://matplotlib.org/faq/usage_faq.html#what-is-a-backend

Parameters

- **y_keys** (*iterable of strs*) – Keys of values regarded as y. If this is `None`, nothing is output to the graph.
- **x_key** (*str*) – Keys of values regarded as x. The default value is 'iteration'.

- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) – Trigger that decides when to aggregate the result and output the values. This is distinct from the trigger of this extension itself. If it is a tuple in the form `<int>, 'epoch'` or `<int>, 'iteration'`, it is passed to `IntervalTrigger`.
- **postprocess** (*Any*) – Callback to postprocess the result dictionaries. Figure object, Axes object, and all plot data are passed to this callback in this order. This callback can modify the figure.
- **filename** (*str*) – Name of the figure file under the output directory. It can be a format string. For historical reasons `file_name` is also accepted as an alias of this argument.
- **marker** (*str*) – The marker used to plot the graph. Default is `'x'`. If `None` is given, it draws with no markers.
- **grid** (*bool*) – If `True`, set the axis grid on. The default value is `True`.
- **writer** (*writer object, optional*) – must be callable. object to dump the log to. If specified, it needs to have a correct *savefun* defined. The writer can override the save location in the *pytorch_pfn_extras.training.ExtensionsManager* object
- **kwargs** (*Any*) –

Methods

__init__(y_keys[, x_key, trigger, ...])	
available()	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
None

__init__(*y_keys*, *x_key='iteration'*, *trigger=(1, 'epoch')*, *postprocess=None*, *filename=None*, *marker='x'*, *grid=True*, ***kwargs*)

Parameters

- **y_keys** (*Union[Iterable[str], str]*) –
- **x_key** (*str*) –
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –
- **postprocess** (*Optional[Any]*) –
- **filename** (*Optional[str]*) –
- **marker** (*str*) –
- **grid** (*bool*) –
- **kwargs** (*Any*) –

static available()

Return type
bool

finalize(*manager*)

Finalizes the extension.

This method is called at the end of the training loop.

Parameters
manager (ExtensionsManagerProtocol) –

Return type
None

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

state_dict()

Serializes the extension state.

It is called when a manager that owns this extension is serialized. It serializes nothing by default.

Return type
Dict[str, *Any*]

pytorch_pfn_extras.training.extensions.print_report

Functions

pytorch_pfn_extras.training.extensions.print_report.create_header_and_templates(entries)	Construct header and templates from *entries*
pytorch_pfn_extras.training.extensions.print_report.deepcopy(x)	Deep copy operation on arbitrary Python objects.
pytorch_pfn_extras.training.extensions.print_report.filter_and_sort_entries(...)	

pytorch_pfn_extras.training.extensions.print_report.create_header_and_templates

pytorch_pfn_extras.training.extensions.print_report.**create_header_and_templates**(*entries*)

Construct header and templates from *entries*

Parameters
entries (*list*) – list of str

Returns
header string templates (str): template string for print values.

Return type
header (str)

pytorch_pfn_extras.training.extensions.print_report.deepcopy

pytorch_pfn_extras.training.extensions.print_report.**deepcopy**(*x*, *memo=None*, *_nil=[]*)

Deep copy operation on arbitrary Python objects.

See the module's __doc__ string for more info.

pytorch_pfn_extras.training.extensions.print_report.filter_and_sort_entries

pytorch_pfn_extras.training.extensions.print_report.**filter_and_sort_entries**(*all_entries*, *unit='epoch'*)

Parameters

- **all_entries** (*List[str]*) –
- **unit** (*str*) –

Return type
List[str]

Classes

pytorch_pfn_extras.training.extensions.print_report.ExtensionsManagerProtocol(...)	
pytorch_pfn_extras.training.extensions.print_report.IO(...)	Generic base class for TextIO and BinaryIO.
pytorch_pfn_extras.training.extensions.print_report.PrintReport([...])	An extension to print the accumulated results.

pytorch_pfn_extras.training.extensions.print_report.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.extensions.print_report.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: Protocol

Methods

__init__(*args, **kwargs)
get_extension(name)

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.extensions.print_report.IO

class pytorch_pfn_extras.training.extensions.print_report.IO(**args*, ***kwds*)

Bases: Generic

Generic base class for TextIO and BinaryIO.

This is an abstract, generic version of the return of open().

NOTE: This does not distinguish between the different possible classes (text vs. binary, read vs. write vs. read/write, append-only, unbuffered). The TextIO and BinaryIO subclasses below capture the distinctions between text vs. binary, which is pervasive in the interface; however we currently do not offer a way to track the other distinctions in the type system.

Methods

`__init__`()
close()
fileno()
flush()
isatty()
read([n])
readable()
readline([limit])
readlines([hint])
seek(offset[, whence])
seekable()
tell()
truncate([size])
writable()
write(s)
writelines(lines)

Attributes

closed
mode
name

abstract close()

Return type
None

abstract property closed: bool

abstract fileno()

Return type
int

abstract flush()

Return type
None

abstract isatty()

Return type
bool

abstract property mode: str

abstract property name: str

abstract read(*n=-1*)

Parameters
n (*int*) –

Return type
AnyStr

abstract readable()

Return type
bool

abstract readline(*limit=-1*)

Parameters
limit (*int*) –

Return type
AnyStr

abstract readlines(*hint=-1*)

Parameters
hint (*int*) –

Return type
List

abstract seek(*offset*, *whence=0*)

Parameters

- **offset** (*int*) –
- **whence** (*int*) –

Return type
int

abstract seekable()

Return type
bool

abstract tell()

Return type
int

abstract truncate(*size=None*)

Parameters
size (*Optional[int]*) –

Return type
int

abstract writable()

Return type
bool

abstract write(*s*)

Parameters
s (*AnyStr*) –

Return type
int

abstract writelines(*lines*)

Parameters
lines (*List*) –

Return type
None

pytorch_pfn_extras.training.extensions.print_report.PrintReport

class pytorch_pfn_extras.training.extensions.print_report.**PrintReport**(*entries=None*, *log_report='LogReport'*, *out=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Bases: *Extension*

An extension to print the accumulated results.

This extension uses the log accumulated by a `LogReport` extension to print specified entries of the log in a human-readable format.

Parameters

- **entries** (*list of str or None*) – List of keys of observations to print. If *None* is passed, automatically infer keys from reported dict.
- **log_report** (*str or LogReport*) – Log report to accumulate the observations. This is either the name of a LogReport extensions registered to the manager, or a LogReport instance to use internally.
- **out** (*IO[Any]*) – Stream to print the bar. Standard output is used by default.

Methods

__init__([entries, log_report, out])	
finalize(manager)	Finalizes the extension.
get_log_report(manager)	
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
None

__init__(*entries=None*, *log_report='LogReport'*, *out=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Parameters

- **entries** (*Optional[Sequence[str]]*) –
- **log_report** (*Union[str, LogReport]*) –
- **out** (*IO[Any]*) –

Return type
None

`get_log_report`(*manager*)

Parameters
manager (ExtensionsManagerProtocol) –

Return type
LogReport

`initialize`(*manager*)

Initializes up the manager state.

This method is called before entering the training loop. An extension modifying the state of `ExtensionsManager` can override this method to initialize it.

When the manager has been restored from a snapshot, this method has to recover an appropriate part of the state of the manager.

Parameters
manager (ExtensionsManager) – Manager object to call this extension.

Return type
None

`load_state_dict`(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

`state_dict`()

Serializes the extension state.

It is called when a manager that owns this extension is serialized. It serializes nothing by default.

Return type
Dict[str, *Any*]

pytorch_pfn_extras.training.extensions.profile_report

Functions

pytorch_pfn_extras.training.extensions.profile_report.get_time_summary()

pytorch_pfn_extras.training.extensions.profile_report.get_time_summary

pytorch_pfn_extras.training.extensions.profile_report.**get_time_summary**()

Return type
TimeSummary

Classes

pytorch_pfn_extras.training.extensions.profile_report.ExtensionsManagerProtocol(...)	
pytorch_pfn_extras.training.extensions.profile_report.OrderedDict	Dictionary that remembers insertion order
pytorch_pfn_extras.training.extensions.profile_report.ProfileReport([...])	Writes the profile results to a file.

pytorch_pfn_extras.training.extensions.profile_report.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.extensions.profile_report.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: Protocol

Methods

__init__(*args, **kwargs)
get_extension(name)

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.extensions.profile_report.OrderedDict

class pytorch_pfn_extras.training.extensions.profile_report.**OrderedDict**

Bases: dict

Dictionary that remembers insertion order

Methods

__init__(*args, **kwargs)	
clear()	
copy()	
fromkeys([value])	Create a new ordered dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
move_to_end(key[, last])	Move an existing element to the end (or beginning if last is false).
pop(k[,d])	value.
popitem([last])	Remove and return a (key, value) pair from the dictionary.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

__init__(**args*, ***kwargs*)

clear() → None. Remove all items from od.

copy() → a shallow copy of od

fromkeys(*value=None*)

Create a new ordered dictionary with keys from iterable and values set to value.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

move_to_end(*key*, *last=True*)

Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem(*last=True*)

Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault(*key*, *default=None*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

pytorch_pfn_extras.training.extensions.profile_report.ProfileReport

class pytorch_pfn_extras.training.extensions.profile_report.**ProfileReport**(*store_keys=None*, *report_keys=None*, *trigger=(1, 'epoch')*, *filename=None*, *append=False*, *format=None*, ***kwargs*)

Bases: *Extension*

Writes the profile results to a file.

Times are reported by using the *pytorch_pfn_extras.profiler.TimeSummary.report()* context manager.

Parameters

- **store_keys** (*iterable of strs*) – Keys of values to write to the profiler report file.
- **report_keys** (*iterable of strs*) – Keys of values that will be reported.
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) – Trigger that decides when to aggregate the result and output the values. This is distinct from the trigger of this extension itself. If it is a tuple in the form `<int>, 'epoch'` or `<int>, 'iteration'`, it is passed to `IntervalTrigger`.

- **filename** (*str*) – Name of the log file under the output directory. It can be a format string: the last result dictionary is passed for the formatting. For example, users can use '{iteration}' to separate the log files for different iterations. If the log name is None, it does not output the log to any file.
- **append** (*bool, optionsl*) – If the file is JSON Lines or YAML, contents will be appended instead of rewriting the file every call.
- **format** (*str, optional*) – accepted values are *'json'*, *'json-lines'* and *'yaml'*.
- **writer** (*writer object, optional*) – must be callable. object to dump the log to. If specified, it needs to have a correct *savefun* defined. The writer can override the save location in the *pytorch_pfn_extras.training.ExtensionsManager* object
- **entries** (*list*) – list of str
- **kwargs** (*Any*) –

Returns
header string templates (str): template string for print values.

Return type
header (str)

Methods

__init__([store_keys, report_keys, trigger, ...])	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
None

__init__(*store_keys=None*, *report_keys=None*, *trigger=(1, 'epoch')*, *filename=None*, *append=False*, *format=None*, ***kwargs*)

Parameters

- **store_keys** (*Optional[Iterable[str]]*) –
- **report_keys** (*Optional[Iterable[str]]*) –
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –
- **filename** (*Optional[str]*) –
- **append** (*bool*) –
- **format** (*Optional[str]*) –
- **kwargs** (*Any*) –

finalize(*manager*)

Finalizes the extension.

This method is called at the end of the training loop.

Parameters
manager (ExtensionsManagerProtocol) –

Return type
None

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

state_dict()

Serializes the extension state.

It is called when a manager that owns this extension is serialized. It serializes nothing by default.

Return type
Dict[str, *Any*]

pytorch_pfn_extras.training.extensions.progress_bar

Classes

pytorch_pfn_extras.training.extensions.progress_bar.ExtensionsManagerProtocol(...)	
pytorch_pfn_extras.training.extensions.progress_bar.ProgressBar([...])	An extension to print a progress bar and recent training status.

pytorch_pfn_extras.training.extensions.progress_bar.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.extensions.progress_bar.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: Protocol

Methods

__init__(*args, **kwargs)
get_extension(name)

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.extensions.progress_bar.ProgressBar

class pytorch_pfn_extras.training.extensions.progress_bar.**ProgressBar**(*training_length=None*, *update_interval=100*, *bar_length=50*, *out=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Bases: *Extension*

An extension to print a progress bar and recent training status.

This extension prints a progress bar at every call. It watches the current iteration and epoch to print the bar.

Parameters

- **training_length** (*tuple or None*) – Length of whole training. It consists of an integer and either `'epoch'` or `'iteration'`. If this value is omitted and the stop trigger of the manager is `IntervalTrigger`, this extension uses its attributes to determine the length of the training.
- **update_interval** (*int*) – Number of iterations to skip printing the progress bar.
- **bar_length** (*int*) – Length of the progress bar in characters.
- **out** (*Any*) – Stream to print the bar. Standard output is used by default.

Methods

__init__([training_length, update_interval, ...])	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
None

__init__(*training_length=None*, *update_interval=100*, *bar_length=50*, *out=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Parameters

- **training_length** (*Optional[Any]*) –
- **update_interval** (*int*) –
- **bar_length** (*int*) –
- **out** (*Any*) –

finalize(*manager*)

Finalizes the extension.

This method is called at the end of the training loop.

Parameters
manager (ExtensionsManagerProtocol) –

Return type
None

pytorch_pfn_extras.training.extensions.slack

Classes

pytorch_pfn_extras.training.extensions.slack.ExtensionsManagerProtocol(...)	
pytorch_pfn_extras.training.extensions.slack.Slack(channel)	An extension to communicate with Slack.
pytorch_pfn_extras.training.extensions.slack.SlackWebhook(url)	An extension to communicate with Slack using Incoming Webhook.

pytorch_pfn_extras.training.extensions.slack.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.extensions.slack.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: Protocol

Methods

__init__(*args, **kwargs)
get_extension(name)

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.extensions.slack.Slack

class pytorch_pfn_extras.training.extensions.slack.**Slack**(*channel*, *msg=None*, *, *start_msg='{default}'*, *end_msg='{default}'*, *error_msg='{default}'*, *thread=True*, *filenames=None*, *upload_trigger=None*, *context=None*, *token=None*)

Bases: _SlackBase

An extension to communicate with Slack.

Example

```
>>> ppe.training.extensions.Slack(
...     channel="experiment-progress",
...     msg="Epoch #{manager.epoch}: loss = {val/loss}",
...     end_msg="{default} \n <@username> Check out the result!",
...
...     # Upload files at the end of the experiment.
...     filenames=["result/statistics.png"],
...     upload_trigger=(max_epoch, 'epoch'),
... )
```

This extension posts a message when:

- `start_msg`: The training has started
- `msg`: The extension is triggered, usually at the end of each epoch
- `end_msg`: The training has finished
- `error_msg`: An exception has raised during the training

These messages can be specified as a format string, a callable that returns a string, or None to disable posting on that event.

When using a format string, the following variables are available for use:

- `manager`: an ExtensionsManager object
- `default`: the default message string
- `context`: an arbitrary object passed to this extension
- `error`: an Exception object (for `error_msg` only)
- All reported values (`manager.observations`)

When using a callable, it should take *(ExtensionsManager, context)* or *(ExtensionsManager, Exception, context)* (for `error_msg`) and return a string.

This extension can upload files along with the message when triggered. `filenames` can be a list of filenames (the same formatting rule as `msg` apply), or a callable taking (ExtensionsManager, context) and returning a list of filenames.

To use this extension, you must create a Slack app, then specify the token via an environment variable `SLACK_BOT_TOKEN` or `token` option.

Parameters

- **channel** (*str*) – The channel where messages and files will be sent. This can be a channel name or a channel ID.
- **msg** (*str, callable, or None*) – A message to be sent when triggered. It can be a string to be formatted using `.format` or a callable that returns a string.
- **start_msg** (*str, callable, or None*) – A message to be sent at the beginning of the experiment.
- **end_msg** (*str, callable, or None*) – A message to be sent at the completion of the experiment.
- **error_msg** (*str, callable, or None*) – A message to be sent when an exception is raised during the experiment.
- **thread** (*bool*) – When True, subsequent messages will be posted as a thread of the original message. Default is `True`.
- **filenames** (*list of str or callable*) – A list of files that will be uploaded. These are string templates that can take values in the same way as `msg`, or a callable that returns a list of filenames.
- **upload_trigger** (*trigger or None*) – Used to upload files at certain events. If not specified, files will be uploaded in every call.
- **context** (*Any*) – Any arbitrary user object you will need when generating a message.
- **token** (*str*) – Slack bot token. If `None`, the environment variable `SLACK_BOT_TOKEN` will be used. Optional, default is `None`.

Methods

__init__(channel[, msg, start_msg, end_msg, ...])	
default_end_msg(context)	
default_error_msg(exc, context)	
default_msg(context)	
default_start_msg(context)	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__init__(*channel*, *msg=None*, ***, *start_msg='{default}'*, *end_msg='{default}'*, *error_msg='{default}'*, *thread=True*, *filenames=None*, *upload_trigger=None*, *context=None*, *token=None*)

Parameters

- **channel** (*str*) –
- **msg** (*Optional[Union[str, Callable[[ExtensionsManagerProtocol, Any], str]]]*) –
- **start_msg** (*Optional[Union[str, Callable[[ExtensionsManagerProtocol, Any], str]]]*) –
- **end_msg** (*Optional[Union[str, Callable[[ExtensionsManagerProtocol, Any], str]]]*) –
- **error_msg** (*Optional[Union[str, Callable[[ExtensionsManagerProtocol, Any, Exception], str]]]*) –
- **thread** (*bool*) –

- **filenames** (*Optional[Union[Sequence[str], Callable[[ExtensionsManagerProtocol, Any], Sequence[str]]]]*) –
- **upload_trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –
- **context** (*Optional[Any]*) –
- **token** (*Optional[str]*) –

Return type
None

trigger: Optional[Union[*Trigger*, Callable[[*ExtensionsManagerProtocol*], bool], Tuple[float, str]]] = (1, 'epoch')

pytorch_pfn_extras.training.extensions.slack.SlackWebhook

class pytorch_pfn_extras.training.extensions.slack.**SlackWebhook**(*url*, *msg=None*, ***, *start_msg='{default}'*, *end_msg='{default}'*, *error_msg='{default}'*, *context=None*)

Bases: _SlackBase

An extension to communicate with Slack using Incoming Webhook.

Example

```
>>> ppe.training.extensions.SlackWebhook(
...     url="https://hooks.slack.com/services/Txxxxx.....",
...     msg="Epoch #{manager.epoch}: loss = {val/loss}",
...     end_msg="{default} \n <@username> Check out the result!",
... )
```

This extension posts a message when:

- `start_msg`: The training has started
- `msg`: The extension is triggered, usually at the end of each epoch
- `end_msg`: The training has finished
- `error_msg`: An exception has raised during the training

These messages can be specified as a format string, a callable that returns a string, or None to disable posting on that event.

When using a format string, the following variables are available for use:

- `manager`: an ExtensionsManager object
- `default`: the default message string
- `context`: an arbitrary object passed to this extension
- `error`: an Exception object (for `error_msg` only)
- All reported values (`manager.observations`)

When using a callable, it should take *(ExtensionsManager, context)* or *(ExtensionsManager, Exception, context)* (for `error_msg`) and return a string.

Parameters

- **url** (*str*) – Incoming webhook URL to send messages.
- **msg** (*str, callable, or None*) – A message to be sent when triggered. It can be a string to be formatted using `.format` or a callable that returns a string.
- **start_msg** (*str, callable, or None*) – A message to be sent at the beginning of the experiment.
- **end_msg** (*str, callable, or None*) – A message to be sent at the completion of the experiment.
- **error_msg** (*str, callable, or None*) – A message to be sent when an exception is raised during the experiment.
- **context** (*object*) – Any arbitrary user object you will need when generating a message.

Methods

__init__(url[, msg, start_msg, end_msg, ...])	
default_end_msg(context)	
default_error_msg(exc, context)	
default_msg(context)	
default_start_msg(context)	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
state_dict()	Serializes the extension state.

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__init__(*url*, *msg=None*, ***, *start_msg='{default}'*, *end_msg='{default}'*, *error_msg='{default}'*, *context=None*)

Parameters

- **url** (*str*) –
- **msg** (*Optional[Union[str, Callable[[ExtensionsManagerProtocol, Any], str]]]*) –
- **start_msg** (*Optional[Union[str, Callable[[ExtensionsManagerProtocol, Any], str]]]*) –
- **end_msg** (*Optional[Union[str, Callable[[ExtensionsManagerProtocol, Any], str]]]*) –
- **error_msg** (*Optional[Union[str, Callable[[ExtensionsManagerProtocol, Any, Exception], str]]]*) –
- **context** (*Optional[Any]*) –

Return type
None

pytorch_pfn_extras.training.extensions.snapshot_writers

Classes

pytorch_pfn_extras.training.extensions.snapshot_writers.ProcessQueueWriter([...])	Snapshot writer that uses process queue.
pytorch_pfn_extras.training.extensions.snapshot_writers.ProcessWriter([...])	Snapshot writer that uses a separate process.
pytorch_pfn_extras.training.extensions.snapshot_writers.QueueWriter([...])	Base class of queue snapshot writers.
pytorch_pfn_extras.training.extensions.snapshot_writers.SimpleWriter([...])	The most simple snapshot writer.
pytorch_pfn_extras.training.extensions.snapshot_writers.StandardWriter([...])	Base class of snapshot writers which use thread or process.
pytorch_pfn_extras.training.extensions.snapshot_writers.TensorBoardWriter([...])	Writer that sends statistics to TensorBoard.
pytorch_pfn_extras.training.extensions.snapshot_writers.ThreadQueueWriter([...])	Snapshot writer that uses a thread queue.
pytorch_pfn_extras.training.extensions.snapshot_writers.ThreadWriter([...])	Snapshot writer that uses a separate thread.
pytorch_pfn_extras.training.extensions.snapshot_writers.Writer([...])	Base class of snapshot writers.

pytorch_pfn_extras.training.extensions.snapshot_writers.ProcessQueueWriter

class pytorch_pfn_extras.training.extensions.snapshot_writers.**ProcessQueueWriter**(*savefun=<function save>*, *fs=None*, *out_dir=''*, *task=None*)

Bases: *QueueWriter*[Process]

Snapshot writer that uses process queue.

This class creates a process and a queue by `multiprocessing` module. The process will be a consumer of this queue, and the main process will be a producer of this queue.

Note: Forking a new process from MPI process might be danger. Consider using *ThreadQueueWriter* instead of `ProcessQueueWriter` if you are using MPI.

See also:

- *pytorch_pfn_extras.training.extensions.snapshot()*

Methods

__init__([savefun, fs, out_dir, task])	
consume(q)	
create_consumer(q)	
create_queue()	
create_task(savefun)	
finalize()	Finalizes the writer.
initialize(out_dir)	
save(filename, out_dir, target, savefun, ...)	

__init__(*savefun=<function save>*, *fs=None*, *out_dir=''*, *task=None*)

Parameters

- **savefun** (*Callable[[...], None]*) –
- **fs** (*Optional[Any]*) –
- **out_dir** (*str*) –
- **task** (*Optional[Callable[[...], None]]*) –

Return type
None

create_consumer(*q*)

Parameters
q (*queue.Queue[_QueUnit]*) –

Return type
Process

create_queue()

Return type
queue.Queue[_QueUnit]

pytorch_pfn_extras.training.extensions.snapshot_writers.ProcessWriter

class pytorch_pfn_extras.training.extensions.snapshot_writers.**ProcessWriter**(*savefun=<function save>*, *fs=None*, *out_dir=''*, ***kwds*)

Bases: *StandardWriter*[Process]

Snapshot writer that uses a separate process.

This class creates a new process that invokes the actual saving function.

Note: Forking a new process from a MPI process might be danger. Consider using *ThreadWriter* instead of ProcessWriter if you are using MPI.

See also:

- *pytorch_pfn_extras.training.extensions.snapshot()*

Methods

__init__([savefun, fs, out_dir])	
create_worker(filename, out_dir, target, *)	Creates a worker for the snapshot.
finalize()	Finalizes the writer.
initialize(out_dir)	
save(filename, out_dir, target, savefun, ...)	

__init__(*savefun=<function save>*, *fs=None*, *out_dir=''*, ***kwds*)

Parameters
- **savefun** (*Callable[[...], None]*) –
- **fs** (*Optional[Any]*) –
- **out_dir** (*str*) –
- **kwds** (*Any*) –

Return type
None

create_worker(*filename*, *out_dir*, *target*, *, *savefun=None*, *append=False*, **savefun_kwargs*)

Creates a worker for the snapshot.

This method creates a thread or a process to take a snapshot. The created worker must have `start()` and `join()` methods. If the worker has an `exitcode` attribute (e.g., `multiprocessing.Process`), the value will be tested.

Parameters

- **filename** (*str*) –
- **out_dir** (*str*) –
- **target** (*Union[Sequence[Any], Mapping[str, Any]]*) –
- **savefun** (*Optional[Callable[[...], None]]*) –
- **append** (*bool*) –
- **savefun_kwargs** (*Any*) –

Return type
Process

pytorch_pfn_extras.training.extensions.snapshot_writers.QueueWriter

class pytorch_pfn_extras.training.extensions.snapshot_writers.**QueueWriter**(*savefun=<function save>*, *fs=None*, *out_dir=''*, *task=None*)

Bases: *Writer*, Generic[_Worker]

Base class of queue snapshot writers.

This class is a base class of snapshot writers that use a queue. A Queue is created when this class is constructed, and every time when `__call__` is invoked, a snapshot task is put into the queue.

Parameters

- **savefun** – Callable object which is passed to the *create_task()* if the task is `None`. It takes three arguments: the output file path, the serialized dictionary object, and the optional keyword arguments.
- **fs** – FileSystem abstracting interface to implement all the operations. optional, defaults to None
- **out_dir** – str. Specifies the directory this writer will use. It takes precedence over the one specified in *__call__* optional, defaults to `''`
- **task** – Callable object. Its `__call__` must have a same interface to `Writer.__call__`. This object is directly put into the queue.

See also:

- *pytorch_pfn_extras.training.extensions.snapshot()*

Methods

__init__([savefun, fs, out_dir, task])	
consume(q)	
create_consumer(q)	
create_queue()	
create_task(savefun)	
finalize()	Finalizes the writer.
initialize(out_dir)	
save(filename, out_dir, target, savefun, ...)	

__call__(*filename*, *out_dir*, *target*, *, *savefun=None*, *append=False*)

Does the actual writing to the file.

This method is invoked by a Snapshot object every time it takes a snapshot.

Parameters

- **filename** (*str*) – Name of the file into which the serialized target is saved. It is a concrete file name, i.e. not a pre-formatted template string.
- **out_dir** (*str*) – Output directory. Corresponds to :py:attr:`ExtensionsManager.out <pytorch_pfn_extras.training.ExtensionsManager.out>`.
- **target** (*dict*) – Serialized object which will be saved.
- **savefun** (*callable*) – A callable that accepts a two positional arguments (an object to be serialized, file path) like *torch.save*.
- **append** (*bool*) – Mode used to open the file. True to use the append mode, False to use the write mode (truncates the file if it already exists).

Return type
None

__init__(*savefun=<function save>*, *fs=None*, *out_dir=''*, *task=None*)

Parameters

- **savefun** (*Callable[[...], None]*) –
- **fs** (*Optional[Any]*) –
- **out_dir** (*str*) –
- **task** (*Optional[Callable[[...], None]]*) –

Return type
None

consume(*q*)

Parameters
q (*queue.Queue[_QueUnit]*) –

Return type
None

`create_consumer`(*q*)

Parameters
q (*queue.Queue[_QueUnit]*) –

Return type
_Worker

`create_queue`()

Return type
queue.Queue[_QueUnit]

`create_task`(*savefun*)

Parameters
savefun (*Callable[[...], None]*) –

Return type
Callable[[. . .], None]

`finalize`()

Finalizes the writer.

Calling this method on already-finalized Writer does nothing.

Return type
None

pytorch_pfn_extras.training.extensions.snapshot_writers.SimpleWriter

class pytorch_pfn_extras.training.extensions.snapshot_writers.**SimpleWriter**(*savefun=<function save>*, *fs=None*, *out_dir=''*, ***kwds*)

Bases: *Writer*

The most simple snapshot writer.

This class just passes the arguments to the actual saving function.

Parameters

- **savefun** (*Callable[[...], None]*) – Callable object. It takes three arguments: the output file path, the serialized dictionary object, and the optional keyword arguments.
- **fs** (*Any*) – FileSystem abstracting interface to implement all the operations. optional, defaults to None
- **out_dir** (*str*) – str. Specifies the directory this writer will use. It takes precedence over the one specified in *__call__* optional, defaults to `''`
- **kwds** (*Any*) – Keyword arguments for the `savefun`.

See also:

- *pytorch_pfn_extras.training.extensions.snapshot()*

Methods

__init__([savefun, fs, out_dir])	
finalize()	Finalizes the writer.
initialize(out_dir)	
save(filename, out_dir, target, savefun, ...)	

__call__(*filename*, *out_dir*, *target*, ***, *savefun=None*, *append=False*)

Does the actual writing to the file.

This method is invoked by a Snapshot object every time it takes a snapshot.

Parameters

- **filename** (*str*) – Name of the file into which the serialized target is saved. It is a concrete file name, i.e. not a pre-formatted template string.
- **out_dir** (*str*) – Output directory. Corresponds to :py:attr:`ExtensionsManager.out <pytorch_pfn_extras.training.ExtensionsManager.out>`.
- **target** (*dict*) – Serialized object which will be saved.
- **savefun** (*callable*) – A callable that accepts a two positional arguments (an object to be serialized, file path) like *torch.save*.
- **append** (*bool*) – Mode used to open the file. True to use the append mode, False to use the write mode (truncates the file if it already exists).

Return type
None

__init__(*savefun=<function save>*, *fs=None*, *out_dir=''*, ***kwds*)

Parameters

- **savefun** (*Callable[[...], None]*) –
- **fs** (*Optional[Any]*) –
- **out_dir** (*str*) –
- **kwds** (*Any*) –

Return type
None

pytorch_pfn_extras.training.extensions.snapshot_writers.StandardWriter

class pytorch_pfn_extras.training.extensions.snapshot_writers.**StandardWriter**(*savefun=<function save>*, *fs=None*, *out_dir=''*, ***kwds*)

Bases: *Writer*, Generic[_Worker]

Base class of snapshot writers which use thread or process.

This class creates a new thread or a process every time when `__call__` is invoked.

Parameters

- **savefun** – Callable object. It takes three arguments: the output file path, the serialized dictionary object, and the optional keyword arguments.
- **fs** – FileSystem abstracting interface to implement all the operations. optional, defaults to None
- **out_dir** – str. Specifies the directory this writer will use. It takes precedence over the one specified in *__call__* optional, defaults to `''`
- **kwds** – Keyword arguments for the `savefun`.

See also:

- *pytorch_pfn_extras.training.extensions.snapshot()*

Methods

__init__([savefun, fs, out_dir])	
create_worker(filename, out_dir, target, *)	Creates a worker for the snapshot.
finalize()	Finalizes the writer.
initialize(out_dir)	
save(filename, out_dir, target, savefun, ...)	

__call__(*filename*, *out_dir*, *target*, *, *savefun=None*, *append=False*)

Does the actual writing to the file.

This method is invoked by a `Snapshot` object every time it takes a snapshot.

Parameters

- **filename** (*str*) – Name of the file into which the serialized target is saved. It is a concrete file name, i.e. not a pre-formatted template string.
- **out_dir** (*str*) – Output directory. Corresponds to :py:attr:`ExtensionsManager.out <pytorch_pfn_extras.training.ExtensionsManager.out>`.
- **target** (*dict*) – Serialized object which will be saved.
- **savefun** (*callable*) – A callable that accepts a two positional arguments (an object to be serialized, file path) like *torch.save*.

- **append** (*bool*) – Mode used to open the file. True to use the append mode, False to use the write mode (truncates the file if it already exists).

Return type
None

__init__(*savefun=<function save>*, *fs=None*, *out_dir=''*, ***kwds*)

Parameters

- **savefun** (*Callable[[...], None]*) –
- **fs** (*Optional[Any]*) –
- **out_dir** (*str*) –
- **kwds** (*Any*) –

Return type
None

create_worker(*filename*, *out_dir*, *target*, ***, *savefun=None*, *append=False*, ***savefun_kwargs*)

Creates a worker for the snapshot.

This method creates a thread or a process to take a snapshot. The created worker must have `start()` and `join()` methods. If the worker has an `exitcode` attribute (e.g., `multiprocessing.Process`), the value will be tested.

Parameters

- **filename** (*str*) –
- **out_dir** (*str*) –
- **target** (*Union[Sequence[Any], Mapping[str, Any]]*) –
- **savefun** (*Optional[Callable[[...], None]]*) –
- **append** (*bool*) –
- **savefun_kwargs** (*Any*) –

Return type
_Worker

finalize()

Finalizes the writer.

Calling this method on already-finalized Writer does nothing.

Return type
None

pytorch_pfn_extras.training.extensions.snapshot_writers.TensorBoardWriter

class pytorch_pfn_extras.training.extensions.snapshot_writers.**TensorBoardWriter**(*savefun=None*, *fs=None*, *out_dir=''*, *stats=None*, ***kwds*)

Bases: `object`

Writer that sends statistics to TensorBoard.

This class contains a *torch.utils.tensorboard.SummaryWriter* object that is used to send the collected statistics to TensorBoard. A list of stats can be specified to report only the desired ones.

Parameters

- **savefun** (*Optional[Callable[[...], None]]*) – Ignored.
- **fs** (*Any*) – Ignored.
- **out_dir** (*str*) – Passed as `log_dir` argument to SummaryWriter.
- **stats** (*list*) – List of statistic keys.
- **kwds** (*Any*) – Passed as an additional arguments to SummaryWriter.

Methods

__init__([savefun, fs, out_dir, stats])	
finalize()	

__call__(*filename*, *out_dir*, *target*, ***, *savefun=None*, *append=False*)

Sends the statistics to the TensorBoard.

Parameters

- **filename** (*str*) – Ignored.
- **out_dir** (*str*) – Ignored.
- **target** (*dict or list*) – The statistics of the iteration. If given as a list, only the last element (assumed to be a dict containing the latest iteration statistics) is reported.
- **savefun** (*Optional[Callable[[...], None]]*) – Ignored.
- **append** (*bool*) – Ignored.

Return type
None

__init__(*savefun=None*, *fs=None*, *out_dir=''*, *stats=None*, ***kwds*)

Parameters

- **savefun** (*Optional[Callable[[...], None]]*) –
- **fs** (*Optional[Any]*) –
- **out_dir** (*str*) –
- **stats** (*Optional[KeysView[str]]*) –
- **kwds** (*Any*) –

Return type
None

finalize()

Return type
None

pytorch_pfn_extras.training.extensions.snapshot_writers.ThreadQueueWriter

class pytorch_pfn_extras.training.extensions.snapshot_writers.**ThreadQueueWriter**(*savefun=<function save>*, *fs=None*, *out_dir=''*, *task=None*)

Bases: *QueueWriter*[Thread]

Snapshot writer that uses a thread queue.

This class creates a thread and a queue by `threading` and `queue` modules respectively. The thread will be a consumer of the queue, and the main thread will be a producer of the queue.

See also:

- *pytorch_pfn_extras.training.extensions.snapshot()*

Methods

__init__([savefun, fs, out_dir, task])	
consume(q)	
create_consumer(q)	
create_queue()	
create_task(savefun)	
finalize()	Finalizes the writer.
initialize(out_dir)	
save(filename, out_dir, target, savefun, ...)	

__init__(*savefun=<function save>*, *fs=None*, *out_dir=''*, *task=None*)

Parameters

- **savefun** (*Callable[[...], None]*) –
- **fs** (*Optional[Any]*) –
- **out_dir** (*str*) –
- **task** (*Optional[Callable[[...], None]]*) –

Return type
None

create_consumer(*q*)

Parameters
q (*queue.Queue[_QueUnit]*) –

Return type
Thread

`create_queue`()

Return type
queue.Queue[_QueUnit]

pytorch_pfn_extras.training.extensions.snapshot_writers.ThreadWriter

class pytorch_pfn_extras.training.extensions.snapshot_writers.**ThreadWriter**(*savefun=<function save>*, *fs=None*, *out_dir=''*, ***kwds*)

Bases: *StandardWriter*[Thread]

Snapshot writer that uses a separate thread.

This class creates a new thread that invokes the actual saving function.

See also:

- *pytorch_pfn_extras.training.extensions.snapshot()*

Methods

__init__([savefun, fs, out_dir])	
create_worker(filename, out_dir, target, *)	Creates a worker for the snapshot.
finalize()	Finalizes the writer.
initialize(out_dir)	
save(filename, out_dir, target, savefun, ...)	

__init__(*savefun=<function save>*, *fs=None*, *out_dir=''*, ***kwds*)

Parameters

- **savefun** (*Callable[[...], None]*) –
- **fs** (*Optional[Any]*) –
- **out_dir** (*str*) –
- **kwds** (*Any*) –

Return type
None

create_worker(*filename*, *out_dir*, *target*, ***, *savefun=None*, *append=False*, ***savefun_kwargs*)

Creates a worker for the snapshot.

This method creates a thread or a process to take a snapshot. The created worker must have `start()` and `join()` methods. If the worker has an `exitcode` attribute (e.g., `multiprocessing.Process`), the value will be tested.

Parameters

- **filename** (*str*) –
- **out_dir** (*str*) –
- **target** (*Union[Sequence[Any], Mapping[str, Any]]*) –
- **savefun** (*Optional[Callable[[...], None]]*) –
- **append** (*bool*) –
- **savefun_kwargs** (*Any*) –

Return type
Thread

pytorch_pfn_extras.training.extensions.snapshot_writers.Writer

class pytorch_pfn_extras.training.extensions.snapshot_writers.**Writer**(*fs=None*, *out_dir=''*)

Bases: `object`

Base class of snapshot writers.

`Snapshot` invokes `__call__` of this class every time when taking a snapshot. This class determines how the actual saving function will be invoked.

Note: This extension first writes the serialized object to a temporary file and then rename it to the target file name. Thus, if the program stops right before the renaming, the temporary file might be left in the output directory.

See also:

- *pytorch_pfn_extras.training.extensions.snapshot()*

Methods

__init__([fs, out_dir])	
finalize()	Finalizes the writer.
initialize(out_dir)	
save(filename, out_dir, target, savefun, ...)	

Parameters

- **fs** (*Any*) –
- **out_dir** (*str*) –

__call__(*filename*, *out_dir*, *target*, ***, *savefun=None*, *append=False*)

Does the actual writing to the file.

This method is invoked by a `Snapshot` object every time it takes a snapshot.

Parameters

- **filename** (*str*) – Name of the file into which the serialized target is saved. It is a concrete file name, i.e. not a pre-formatted template string.
- **out_dir** (*str*) – Output directory. Corresponds to :py:attr:`ExtensionsManager.out <pytorch_pfn_extras.training.ExtensionsManager.out>`.
- **target** (*dict*) – Serialized object which will be saved.
- **savefun** (*callable*) – A callable that accepts a two positional arguments (an object to be serialized, file path) like *torch.save*.
- **append** (*bool*) – Mode used to open the file. True to use the append mode, False to use the write mode (truncates the file if it already exists).

Return type
None

__init__(*fs=None*, *out_dir=''*)

Parameters
- **fs** (*Optional[Any]*) –
- **out_dir** (*str*) –

Return type
None

finalize()

Finalizes the writer.

Calling this method on already-finalized Writer does nothing.

Return type
None

initialize(*out_dir*)

Parameters
out_dir (*str*) –

Return type
None

save(*filename*, *out_dir*, *target*, *savefun*, *append*, ***savefun_kwargs*)

Parameters
- **filename** (*str*) –
- **out_dir** (*str*) –
- **target** (*Union[Sequence[Any], Mapping[str, Any]]*) –
- **savefun** (*Callable[[...], None]*) –
- **append** (*bool*) –
- **savefun_kwargs** (*Any*) –

Return type
None

pytorch_pfn_extras.training.extensions.util

Classes

pytorch_pfn_extras.training.extensions.util.ExtensionsManagerProtocol(...)	
pytorch_pfn_extras.training.extensions.util.ProgressBar([out])	
pytorch_pfn_extras.training.extensions.util.TextIO(...)	Typed version of the return of open() in text mode.

pytorch_pfn_extras.training.extensions.util.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.extensions.util.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: Protocol

Methods

__init__(*args, **kwargs)
get_extension(name)

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.extensions.util.ProgressBar

class pytorch_pfn_extras.training.extensions.util.**ProgressBar**(*out=None*)

Bases: object

Methods

__init__([out])	
close()	
erase_console()	
flush()	
get_lines()	
move_cursor_up(n)	
update([manager])	
update_speed(iteration, epoch_detail)	

Parameters
out (*Optional[TextIO]*) –

__init__(*out=None*)

Parameters
out (*Optional[TextIO]*) –

Return type
None

close()

Return type
None

erase_console()

Return type
None

flush()

Return type
None

get_lines()

Return type
Sequence[str]

move_cursor_up(*n*)

Parameters
n (*int*) –

Return type
None

update(*manager=None*)

Parameters
manager (*Optional[ExtensionsManagerProtocol]*) –

Return type
None

update_speed(*iteration*, *epoch_detail*)

Parameters

- **iteration** (*int*) –
- **epoch_detail** (*float*) –

Return type
Tuple[float, float]

pytorch_pfn_extras.training.extensions.util.TextIO

class pytorch_pfn_extras.training.extensions.util.**TextIO**(**args*, ***kwds*)

Bases: *IO*[str]

Typed version of the return of open() in text mode.

Methods

`__init__`()
`close`()
`fileno`()
`flush`()
`isatty`()
`read`([n])
`readable`()
`readline`([limit])
`readlines`([hint])
`seek`(offset[, whence])
`seekable`()
`tell`()
`truncate`([size])
`writable`()
`write`(s)
`writelines`(lines)

Attributes

buffer
closed
encoding
errors
line_buffering
mode
name
newlines

abstract property buffer: BinaryIO

abstract property encoding: str

abstract property errors: Optional[str]

abstract property line_buffering: bool

abstract property newlines: Any

pytorch_pfn_extras.training.extensions.value_observation

Functions

pytorch_pfn_extras.training.extensions.value_observation.observe_lr(...)	Returns an extension to record the learning rate.
pytorch_pfn_extras.training.extensions.value_observation.observe_value(...)	Returns an extension to continuously record a value.

pytorch_pfn_extras.training.extensions.value_observation.observe_lr

pytorch_pfn_extras.training.extensions.value_observation.**observe_lr**(*optimizer*, *param_group=0*, *observation_key='lr'*)

Returns an extension to record the learning rate.

Parameters

- **optimizer** (*Optimizer*) – Optimizer whose learning rate is recorded.
- **param_group** (*int*) – Param group of the optimizer to observe
- **observation_key** (*str*) – Key of observation to record.

Returns
The extension function.

Return type
Any

This extension is triggered each epoch by default. To change this, use the `trigger` argument with the `ExtensionsManager.extend()` method.

pytorch_pfn_extras.training.extensions.value_observation.observe_value

pytorch_pfn_extras.training.extensions.value_observation.**observe_value**(*observation_key*, *target_func*)

Returns an extension to continuously record a value.

Parameters

- **observation_key** (*str*) – Key of observation to record.
- **target_func** (*function*) – Function that returns the value to record. It must take one argument: :class:~pytorch_pfn_extras.training.ExtensionsManager object.

Returns
The extension function.

Return type
Callable[[ExtensionsManagerProtocol], None]

This extension is triggered each epoch by default. To change this, use the `trigger` argument with the `ExtensionsManager.extend()` method.

Classes

pytorch_pfn_extras.training.extensions.value_observation.ExtensionsManagerProtocol(...)

pytorch_pfn_extras.training.extensions.value_observation.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.extensions.value_observation.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: `Protocol`

Methods

__init__(*args, **kwargs)

get_extension(name)

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.extensions.variable_statistics_plot

Functions

pytorch_pfn_extras.training.extensions.variable_statistics_plot.matplotlib_savefun(...)
pytorch_pfn_extras.training.extensions.variable_statistics_plot.percentile(a, ...)

pytorch_pfn_extras.training.extensions.variable_statistics_plot.matplotlib_savefun

pytorch_pfn_extras.training.extensions.variable_statistics_plot.**matplotlib_savefun**(*target*, *file_o*)

Parameters

- **target** (*Tuple[Any, Any]*) –
- **file_o** (*Any*) –

Return type
None

pytorch_pfn_extras.training.extensions.variable_statistics_plot.percentile

pytorch_pfn_extras.training.extensions.variable_statistics_plot.**percentile**(*a*, *q*, *axis*)

Parameters

- **a** (*Tensor*) –
- **q** (*Union[float, Tuple[float, ...]]*) –
- **axis** (*int*) –

Return type
Any

Classes

pytorch_pfn_extras.training.extensions.variable_statistics_plot.ExtensionsManagerProtocol(...)	
pytorch_pfn_extras.training.extensions.variable_statistics_plot.Reservoir(...)	Reservoir sample with a fixed sized buffer.
pytorch_pfn_extras.training.extensions.variable_statistics_plot.Statistician(...)	Helper to compute basic NumPy-like statistics.
pytorch_pfn_extras.training.extensions.variable_statistics_plot.VariableStatisticsPlot(targets)	An extension to plot statistics for `Tensor`s.

pytorch_pfn_extras.training.extensions.variable_statistics_plot.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.extensions.variable_statistics_plot.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: `Protocol`

Methods

__init__(*args, **kwargs)
get_extension(name)

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.extensions.variable_statistics_plot.Reservoir

class pytorch_pfn_extras.training.extensions.variable_statistics_plot.**Reservoir**(*size*, *data_shape*, *dtype=<class 'numpy.float32'>*)

Bases: object

Reservoir sample with a fixed sized buffer.

Methods

__init__(size, data_shape[, dtype])	
add(x[, idx])	
get_data()	

Parameters

- **size** (*int*) –
- **data_shape** (*Tuple[int, ...]*) –
- **dtype** (*Any*) –

__init__(*size*, *data_shape*, *dtype=<class 'numpy.float32'>*)

Parameters

- **size** (*int*) –
- **data_shape** (*Tuple[int, ...]*) –
- **dtype** (*Any*) –

Return type
None

add(*x*, *idx=None*)

Parameters

- **x** (*Any*) –
- **idx** (*Optional[Any]*) –

Return type
None

`get_data`()

Return type
Tuple[*Any*, *Any*]

pytorch_pfn_extras.training.extensions.variable_statistics_plot.Statistician

class pytorch_pfn_extras.training.extensions.variable_statistics_plot.**Statistician**(*collect_mean*, *collect_std*, *percentile_sigmas*)

Bases: `object`

Helper to compute basic NumPy-like statistics.

Methods

__init__(collect_mean, collect_std, ...)	
collect(x, axis)	

Parameters
- **collect_mean** (*bool*) –
- **collect_std** (*bool*) –
- **percentile_sigmas** (*Union[float, Tuple[float, ...]]*) –

__call__(*x*, *axis=0*, *dtype=None*)
Call self as a function.

Parameters
- **x** (*Any*) –
- **axis** (*Any*) –
- **dtype** (*Optional[Any]*) –

Return type
Dict[str, *Any*]

__init__(*collect_mean*, *collect_std*, *percentile_sigmas*)

Parameters
- **collect_mean** (*bool*) –
- **collect_std** (*bool*) –
- **percentile_sigmas** (*Union[float, Tuple[float, ...]]*) –

Return type
None

`collect`(*x*, *axis*)

Parameters

- **x** (*Any*) –
- **axis** (*int*) –

Return type
Dict[str, *Any*]

pytorch_pfn_extras.training.extensions.variable_statistics_plot.VariableStatisticsPlot

`class` pytorch_pfn_extras.training.extensions.variable_statistics_plot.`VariableStatisticsPlot`(*targets*, *max_sample_re-port_data=Tr re-port_grad=Tr plot_mean=T plot_std=Tru per-centile_sigma 0.13, 2.28, 15.87, 50, 84.13, 97.72, 99.87, 100)*, *trig-ger=(1, 'epoch')*, *file-name='statist fig-size=None*, *marker=None grid=True*)

Bases: *Extension*

An extension to plot statistics for `Tensor`s.

This extension collects statistics for a single `torch.Tensor`, a list of `torch.Tensor`s or similarly a single or a list of `torch.nn.Module`s containing one or more `torch.Tensor`s. In case multiple `torch.Tensor`s are found, the means are computed. The collected statistics are plotted and saved as an image in the directory specified by the `Manager`.

Statistics include mean, standard deviation and percentiles.

This extension uses reservoir sampling to preserve memory, using a fixed size running sample. This means that collected items in the sample are discarded uniformly at random when the number of items becomes larger than the maximum sample size, but each item is expected to occur in the sample with equal probability.

:param targets (`torch.Tensor`: or list of either): Parameters for which statistics are collected. :param `torch.nn.Module`: or list of either): Parameters for which statistics are collected. :param max_sample_size: Maximum number of running samples. :type max_sample_size: int :param report_data: If `True`, data (e.g. weights) statistics are plotted. If

`False`, they are neither computed nor plotted.

Parameters

- **report_grad** (*bool*) – If `True`, gradient statistics are plotted. If `False`, they are neither computed nor plotted.
- **plot_mean** (*bool*) – If `True`, means are plotted. If `False`, they are neither computed nor plotted.
- **plot_std** (*bool*) – If `True`, standard deviations are plotted. If `False`, they are neither computed nor plotted.
- **percentile_sigmas** (*float or tuple of floats*) – Percentiles to plot in the range $[0, 100]$.
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) – Trigger that decides when to save the plots as an image. This is distinct from the trigger of this extension itself. If it is a tuple in the form `<int>, 'epoch'` or `<int>, 'iteration'`, it is passed to `IntervalTrigger`.
- **filename** (*str*) – Name of the output image file under the output directory. For historical reasons `file_name` is also accepted as an alias of this argument.
- **figsize** (*tuple of int*) – Matlotlib `figsize` argument that specifies the size of the output image.
- **marker** (*str*) – Matplotlib `marker` argument that specified the marker style of the plots.
- **grid** (*bool*) – Matplotlib `grid` argument that specifies whether grids are rendered in in the plots or not.
- **writer** (*writer object, optional*) – must be callable. object to dump the log to. If specified, it needs to have a correct *savefun* defined. The writer can override the save location in the *pytorch_pfn_extras.training.ExtensionsManager* object
- **targets** (*Any*) –
- **max_sample_size** (*int*) –
- **report_data** (*bool*) –
- **kwargs** (*Any*) –

Methods

__init__(targets[, max_sample_size, ...])	
available()	
finalize(manager)	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	
on_error(manager, exc, tb)	Handles the error raised during training before finalization.
save_plot_using_module(plt, manager)	
state_dict()	Serializes the extension state.

Attributes

default_name	Default name of the extension.
is_async	
name	
needs_model_state	
priority	
trigger	

__call__(*manager*)

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

Parameters
manager (ExtensionsManager) – Manager object to call this operator.

Return type
None

__init__(*targets*, *max_sample_size=1000*, *report_data=True*, *report_grad=True*, *plot_mean=True*, *plot_std=True*, *percentile_sigmas=(0, 0.13, 2.28, 15.87, 50, 84.13, 97.72, 99.87, 100)*, *trigger=(1, 'epoch')*, *filename=None*, *figsize=None*, *marker=None*, *grid=True*, ***kwargs*)

Parameters

- **targets** (*Any*) –
- **max_sample_size** (*int*) –
- **report_data** (*bool*) –
- **report_grad** (*bool*) –

- **plot_mean** (*bool*) –
- **plot_std** (*bool*) –
- **percentile_sigmas** (*Union[float, Tuple[float, ...]]*) –
- **trigger** (*Optional[Union[Trigger, Callable[[ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –
- **filename** (*Optional[str]*) –
- **figsize** (*Optional[Tuple[int, ...]]*) –
- **marker** (*Optional[str]*) –
- **grid** (*bool*) –
- **kwargs** (*Any*) –

static available()

Return type
bool

finalize(*manager*)

Finalizes the extension.

This method is called at the end of the training loop.

Parameters
manager (ExtensionsManagerProtocol) –

Return type
None

save_plot_using_module(*plt*, *manager*)

Parameters

- **plt** (*Any*) –
- **manager** (ExtensionsManagerProtocol) –

Return type
None

pytorch_pfn_extras.training.manager

Functions

pytorch_pfn_extras.training.manager. default_transform_model(n, x)
pytorch_pfn_extras.training.manager. record(tag)

Classes

pytorch_pfn_extras.training.manager.ExtensionsManager(...)	Manages the extensions and the current status.
pytorch_pfn_extras.training.manager.IgniteExtensionsManager(...)	Manages extensions and the current status in Ignite training loop.
pytorch_pfn_extras.training.manager.StateObjectProtocol(...)	

pytorch_pfn_extras.training.metrics

Classes

pytorch_pfn_extras.training.metrics.AccuracyMetric(...)	A metric for an evaluator to report accuracy.

pytorch_pfn_extras.training.metrics.AccuracyMetric

class pytorch_pfn_extras.training.metrics.**AccuracyMetric**(*label_key*, *output_key*)

Bases: object

A metric for an evaluator to report accuracy.

Parameters

- **label_key** (*str*) – The key name of label.
- **output_key** (*str*) – The key name of prediction.

Methods

__init__(label_key, output_key)	

__call__(*batch*, *out*)

Call self as a function.

Parameters

- **batch** (*Dict[str, Tensor]*) –
- **out** (*Dict[str, Tensor]*) –

Return type

Dict[str, *Any*]

__init__(*label_key*, *output_key*)

Parameters

- **label_key** (*str*) –
- **output_key** (*str*) –

Return type
None

pytorch_pfn_extras.training.trigger

Functions

pytorch_pfn_extras.training.trigger.get_trigger(trigger)	Gets a trigger object.

pytorch_pfn_extras.training.trigger.get_trigger

pytorch_pfn_extras.training.trigger.**get_trigger**(*trigger*)

Gets a trigger object.

Trigger object is a callable that accepts a *ExtensionsManager* object as an argument and returns a boolean value. When it returns True, various kinds of events can occur depending on the context in which the trigger is used. For example, if the trigger is passed to the `extend()` method of a manager, then the registered extension is invoked only when the trigger returns True.

This function returns a trigger object based on the argument. If `trigger` is already a callable, it just returns the trigger. If `trigger` is `None`, it returns a trigger that never fires. Otherwise, it creates a `IntervalTrigger`.

Parameters
trigger (*Optional[Union[*Trigger*, Callable[[*ExtensionsManagerProtocol*], bool], Tuple[float, str]]]*) – Trigger object. It can be either an already built trigger object (i.e., a callable object that accepts a manager object and returns a bool value), or a tuple. In latter case, the tuple is passed to `IntervalTrigger`.

Returns
`trigger` if it is a callable, otherwise a `IntervalTrigger` object made from `trigger`.

Return type
Trigger

Classes

pytorch_pfn_extras.training.trigger.IntervalTrigger(...)	Trigger based on a fixed interval.
pytorch_pfn_extras.training.trigger.Trigger()	Base class for triggers.

pytorch_pfn_extras.training.trigger.IntervalTrigger

class pytorch_pfn_extras.training.trigger.**IntervalTrigger**(*period*, *unit*)

Bases: *Trigger*

Trigger based on a fixed interval.

This trigger accepts iterations divided by a given interval. There are two ways to specify the interval: per iterations and epochs. *Iteration* means the number of updates, while *epoch* means the number of sweeps over the training dataset. Fractional values are allowed if the interval is a number of epochs; the trigger uses the *iteration* and *epoch_detail* attributes defined by the manager.

For the description of triggers see `get_trigger()`.

Parameters

- **period** (*int or float*) – Length of the interval. Must be an integer if unit is `'iteration'`.
- **unit** (*str*) – Unit of the length specified by `period`. It must be either `'iteration'` or `'epoch'`.

Methods

__init__(period, unit)	
get_training_length()	
load_state_dict(state)	
may_fire(iteration, epoch_length)	Flags if the trigger may fire at the current iteration
state_dict()	

__call__(*manager*)

Decides whether the extension should be called on this iteration.

Parameters

manager (*ExtensionsManager*) – Manager object that this trigger is associated with. The iteration related information in this manager is used to determine if the trigger should fire.

Returns

True if the corresponding extension should be invoked in this iteration.

Return type

bool

__init__(*period*, *unit*)

Parameters

- **period** (*float*) –
- **unit** (*UnitLiteral*) –

`get_training_length`()

Return type
Tuple[float, str]

`may_fire`(*iteration*, *epoch_length*)

Flags if the trigger may fire at the current iteration

This must not alter the trigger state

Parameters

- **iteration** (*int*) –
- **epoch_length** (*int*) –

Return type
bool

pytorch_pfn_extras.training.trigger.Trigger

`class` pytorch_pfn_extras.training.trigger.`Trigger`

Bases: `object`

Base class for triggers.

Methods

`__init__`()	
load_state_dict(state)	
may_fire(iteration, epoch_len)	Flags if the trigger may fire at the current iteration
state_dict()	

`__call__`(*manager*)

Call self as a function.

Parameters
manager (ExtensionsManagerProtocol) –

Return type
bool

`load_state_dict`(*state*)

Parameters
state (*Dict[str, Any]*) –

Return type
None

`may_fire`(*iteration*, *epoch_len*)

Flags if the trigger may fire at the current iteration

This must not alter the trigger state

Parameters

- **iteration** (*int*) –
- **epoch_len** (*int*) –

Return type
bool

`state_dict()`

Return type
Dict[str, *Any*]

pytorch_pfn_extras.training.triggers

Classes

pytorch_pfn_extras.training.triggers.BestValueTrigger(...)	Trigger invoked when specific value becomes best.
pytorch_pfn_extras.training.triggers.EarlyStoppingTrigger(self)	Trigger for Early Stopping
pytorch_pfn_extras.training.triggers.IntervalTrigger(...)	Trigger based on a fixed interval.
pytorch_pfn_extras.training.triggers.ManualScheduleTrigger(...)	Trigger invoked at specified point(s) of iterations or epochs.
pytorch_pfn_extras.training.triggers.MaxValueTrigger(key)	Trigger invoked when specific value becomes maximum.
pytorch_pfn_extras.training.triggers.MinValueTrigger(key)	Trigger invoked when specific value becomes minimum.
pytorch_pfn_extras.training.triggers.OnceTrigger([...])	Trigger based on the starting point of the iteration.
pytorch_pfn_extras.training.triggers.TimeTrigger(period)	Trigger based on a fixed time interval.

pytorch_pfn_extras.training.triggers.BestValueTrigger

class pytorch_pfn_extras.training.triggers.**BestValueTrigger**(*key*, *compare*, *trigger=(1, 'epoch')*)

Bases: *Trigger*

Trigger invoked when specific value becomes best.

Parameters

- **key** (*str*) – Key of value.
- **compare** (*callable*) – Compare function which takes current best value and new value and returns whether new value is better than current best.
- **trigger** (*TriggerLike*) – Trigger that decides the comparison interval between current best value and new value. This must be a tuple in the form of `<int>, 'epoch'` or `<int>, 'iteration'` which is passed to `IntervalTrigger`.

Methods

__init__(key, compare[, trigger])	
load_state_dict(to_load)	
may_fire(iteration, epoch_length)	Flags if the trigger may fire at the current iteration
state_dict()	

__call__(*manager*)

Decides whether the extension should be called on this iteration.

Parameters
manager (ExtensionsManager) – Manager object that this trigger is associated with. The observation of this manager is used to determine if the trigger should fire.

Returns
True if the corresponding extension should be invoked in this iteration.

Return type
bool

__init__(*key*, *compare*, *trigger=(1, 'epoch')*)

Parameters

- **key** (*str*) –
- **compare** (*Callable[[float, float], bool]*) –
- **trigger** (*TriggerLike*) –

Return type
None

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

may_fire(*iteration*, *epoch_length*)

Flags if the trigger may fire at the current iteration

This must not alter the trigger state

Parameters

- **iteration** (*int*) –
- **epoch_length** (*int*) –

Return type
bool

state_dict()

Return type
Dict[str, *Any*]

pytorch_pfn_extras.training.triggers.EarlyStoppingTrigger

class pytorch_pfn_extras.training.triggers.**EarlyStoppingTrigger**(*self*, *check_trigger=(1, 'epoch')*, *monitor='main/loss'*, *patience=3*, *mode='auto'*, *verbose=False*, *max_trigger=(100, 'epoch')*)

Bases: *Trigger*

Trigger for Early Stopping

This trigger works as follows. Within each *check interval* defined by the `check_trigger` argument, it monitors and accumulates the reported value at each iteration. At the end of each interval, it computes the mean of the accumulated values and compares it to the previous ones to maintain the *best* value. When it finds that the best value is not updated for some periods (defined by `patience`), this trigger fires.

Parameters

- **monitor** (*str*) – The metric you want to monitor
- **check_trigger** (*TriggerLike*) – Trigger that decides the comparison interval between current best value and new value. This must be a tuple in the form of `<int>, 'epoch'` or `<int>, 'iteration'` which is passed to `IntervalTrigger`.
- **patience** (*int*) – Counts to let the trigger be patient. The trigger will not fire until the condition is met for successive `patience` checks.
- **mode** (*str*) – `'max'`, `'min'`, or `'auto'`. It is used to determine how to compare the monitored values.
- **verbose** (*bool*) – Enable verbose output. If verbose is true, you can get more information
- **max_trigger** (*Tuple[int, UnitLiteral]*) – Upper bound of the number of training loops

Methods

__init__([check_trigger, monitor, patience, ...])	
get_training_length()	
load_state_dict(state)	
may_fire(iteration, epoch_length)	Flags if the trigger may fire at the current iteration
state_dict()	

__call__(*manager*)

Decides whether the training loop should be stopped.

Parameters

manager (ExtensionsManager) – Manager object that this trigger is associated with. The `observation` of this manager is used to determine if the trigger should fire.

Returns

`True` if the training loop should be stopped.

Return type
bool

__init__(*check_trigger=(1, 'epoch')*, *monitor='main/loss'*, *patience=3*, *mode='auto'*, *verbose=False*, *max_trigger=(100, 'epoch')*)

Parameters

- **check_trigger** (*TriggerLike*) –
- **monitor** (*str*) –
- **patience** (*int*) –
- **mode** (*str*) –
- **verbose** (*bool*) –
- **max_trigger** (*Tuple[int, UnitLiteral]*) –

Return type
None

get_training_length()

Return type
Tuple[float, str]

may_fire(*iteration*, *epoch_length*)

Flags if the trigger may fire at the current iteration

This must not alter the trigger state

Parameters

- **iteration** (*int*) –
- **epoch_length** (*int*) –

Return type
bool

pytorch_pfn_extras.training.triggers.IntervalTrigger

class pytorch_pfn_extras.training.triggers.**IntervalTrigger**(*period*, *unit*)

Bases: *Trigger*

Trigger based on a fixed interval.

This trigger accepts iterations divided by a given interval. There are two ways to specify the interval: per iterations and epochs. *Iteration* means the number of updates, while *epoch* means the number of sweeps over the training dataset. Fractional values are allowed if the interval is a number of epochs; the trigger uses the *iteration* and *epoch_detail* attributes defined by the manager.

For the description of triggers see `get_trigger()`.

Parameters

- **period** (*int or float*) – Length of the interval. Must be an integer if unit is `'iteration'`.
- **unit** (*str*) – Unit of the length specified by `period`. It must be either `'iteration'` or `'epoch'`.

Methods

__init__(period, unit)	
get_training_length()	
load_state_dict(state)	
may_fire(iteration, epoch_length)	Flags if the trigger may fire at the current iteration
state_dict()	

__call__(*manager*)

Decides whether the extension should be called on this iteration.

Parameters
manager (ExtensionsManager) – Manager object that this trigger is associated with. The iteration related information in this manager is used to determine if the trigger should fire.

Returns
True if the corresponding extension should be invoked in this iteration.

Return type
bool

__init__(*period*, *unit*)

Parameters

- **period** (*float*) –
- **unit** (*UnitLiteral*) –

get_training_length()

Return type
Tuple[float, str]

may_fire(*iteration*, *epoch_length*)

Flags if the trigger may fire at the current iteration

This must not alter the trigger state

Parameters

- **iteration** (*int*) –
- **epoch_length** (*int*) –

Return type
bool

pytorch_pfn_extras.training.triggers.ManualScheduleTrigger

class pytorch_pfn_extras.training.triggers.**ManualScheduleTrigger**(*points*, *unit*)

Bases: *Trigger*

Trigger invoked at specified point(s) of iterations or epochs.

This trigger accepts iterations or epochs indicated by given point(s). There are two ways to specify the point(s): iteration and epoch. `iteration` means the number of updates, while `epoch` means the number of sweeps over the training dataset. Fractional values are allowed if the point is a number of epochs; the trigger uses the `iteration` and `epoch_detail` attributes defined by the manager.

Parameters

- **points** (*int, float, or list of int or float*) – time of the trigger. Must be an integer or list of integer if unit is `'iteration'`.
- **unit** (*str*) – Unit of the time specified by `points`. It must be either `'iteration'` or `'epoch'`.

Methods

__init__(points, unit)	
load_state_dict(state)	
may_fire(iteration, epoch_length)	Flags if the trigger may fire at the current iteration
state_dict()	

__call__(*manager*)

Decides whether the extension should be called on this iteration.

Parameters
manager (ExtensionsManager) – Manager object that this trigger is associated with. The iteration information in this manager is used to determine if the trigger should fire.

Returns
True if the corresponding extension should be invoked in this iteration.

Return type
bool

__init__(*points*, *unit*)

Parameters

- **points** (*Union[float, Sequence[float]]*) –
- **unit** (*UnitLiteral*) –

may_fire(*iteration*, *epoch_length*)

Flags if the trigger may fire at the current iteration

This must not alter the trigger state

Parameters

- **iteration** (*int*) –

- **epoch_length** (*int*) –

Return type
bool

pytorch_pfn_extras.training.triggers.MaxValueTrigger

class pytorch_pfn_extras.training.triggers.**MaxValueTrigger**(*key*, *trigger=(1, 'epoch')*)

Bases: *BestValueTrigger*

Trigger invoked when specific value becomes maximum.

For example you can use this trigger to take snapshot on the epoch the validation accuracy is maximum.

Parameters

- **key** (*str*) – Key of value. The trigger fires when the value associated with this key becomes maximum.
- **trigger** (*TriggerLike*) – Trigger that decides the comparison interval between current best value and new value. This must be a tuple in the form of `<int>, 'epoch'` or `<int>, 'iteration'` which is passed to `IntervalTrigger`.

Methods

__init__(key[, trigger])	
load_state_dict(to_load)	
may_fire(iteration, epoch_length)	Flags if the trigger may fire at the current iteration
state_dict()	

__init__(*key*, *trigger=(1, 'epoch')*)

Parameters

- **key** (*str*) –
- **trigger** (*TriggerLike*) –

pytorch_pfn_extras.training.triggers.MinValueTrigger

class pytorch_pfn_extras.training.triggers.**MinValueTrigger**(*key*, *trigger=(1, 'epoch')*)

Bases: *BestValueTrigger*

Trigger invoked when specific value becomes minimum.

For example you can use this trigger to take snapshot on the epoch the validation loss is minimum.

Parameters

- **key** (*str*) – Key of value. The trigger fires when the value associated with this key becomes minimum.

- **trigger** (*TriggerLike*) – Trigger that decides the comparison interval between current best value and new value. This must be a tuple in the form of `<int>, 'epoch'` or `<int>, 'iteration'` which is passed to `IntervalTrigger`.

Methods

__init__(key[, trigger])	
load_state_dict(to_load)	
may_fire(iteration, epoch_length)	Flags if the trigger may fire at the current iteration
state_dict()	

__init__(*key*, *trigger=(1, 'epoch')*)

Parameters

- **key** (*str*) –
- **trigger** (*TriggerLike*) –

pytorch_pfn_extras.training.triggers.OnceTrigger

class pytorch_pfn_extras.training.triggers.**OnceTrigger**(*call_on_resume=False*)

Bases: *Trigger*

Trigger based on the starting point of the iteration.

This trigger accepts only once at starting point of the iteration. There are two ways to specify the starting point: only starting point in whole iteration or called again when training resumed.

Parameters

call_on_resume (*bool*) – Whether the extension is called again or not when restored from a snapshot. It is set to `False` by default.

finished

Flag that indicates whether or not this trigger will

Type

bool

fire in the future. This flag is used to determine if the extension

should be initialized after resume.

Methods

__init__([call_on_resume])	
load_state_dict(to_load)	
may_fire(iteration, epoch_length)	Flags if the trigger may fire at the current iteration
state_dict()	

Attributes

finished

__call__(*manager*)

Call self as a function.

Parameters
manager (ExtensionsManagerProtocol) –

Return type
bool

__init__(*call_on_resume=False*)

Parameters
call_on_resume (*bool*) –

Return type
None

property finished: bool

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

may_fire(*iteration*, *epoch_length*)

Flags if the trigger may fire at the current iteration

This must not alter the trigger state

Parameters

- **iteration** (*int*) –
- **epoch_length** (*int*) –

Return type
bool

`state_dict`()

Return type
Dict[str, *Any*]

pytorch_pfn_extras.training.triggers.TimeTrigger

`class` pytorch_pfn_extras.training.triggers.`TimeTrigger`(*period*)

Bases: *Trigger*

Trigger based on a fixed time interval.

This trigger accepts iterations with a given interval time.

Parameters
period (*float*) – Interval time. It is given in seconds.

Methods

__init__(period)	
load_state_dict(to_load)	
may_fire(iteration, epoch_len)	Flags if the trigger may fire at the current iteration
state_dict()	

`__call__`(*manager*)

Call self as a function.

Parameters
manager (ExtensionsManagerProtocol) –

Return type
bool

`__init__`(*period*)

Parameters
period (*float*) –

Return type
None

`load_state_dict`(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

`state_dict`()

Return type
Dict[str, *Any*]

Modules

pytorch_pfn_extras.training.triggers.early_stopping_trigger
pytorch_pfn_extras.training.triggers.interval_trigger
pytorch_pfn_extras.training.triggers.manual_schedule_trigger
pytorch_pfn_extras.training.triggers.minmax_value_trigger
pytorch_pfn_extras.training.triggers.once_trigger
pytorch_pfn_extras.training.triggers.time_trigger

pytorch_pfn_extras.training.triggers.early_stopping_trigger

Classes

pytorch_pfn_extras.training.triggers.early_stopping_trigger.EarlyStoppingTrigger(self)	Trigger for Early Stopping
pytorch_pfn_extras.training.triggers.early_stopping_trigger.ExtensionsManagerProtocol(...)	

pytorch_pfn_extras.training.triggers.early_stopping_trigger.EarlyStoppingTrigger

class pytorch_pfn_extras.training.triggers.early_stopping_trigger.**EarlyStoppingTrigger**(*self*, *check_trigger=(1, 'epoch')*, *monitor='main/loss'*, *patience=3*, *mode='auto'*, *verbose=False*, *max_trigger=(100, 'epoch')*)

Bases: *Trigger*

Trigger for Early Stopping

This trigger works as follows. Within each *check interval* defined by the `check_trigger` argument, it monitors and accumulates the reported value at each iteration. At the end of each interval, it computes the mean of the accumulated values and compares it to the previous ones to maintain the *best* value. When it finds that the best value is not updated for some periods (defined by `patience`), this trigger fires.

Parameters

- **monitor** (*str*) – The metric you want to monitor
- **check_trigger** (*TriggerLike*) – Trigger that decides the comparison interval between current best value and new value. This must be a tuple in the form of `<int>, 'epoch'` or `<int>, 'iteration'` which is passed to `IntervalTrigger`.
- **patience** (*int*) – Counts to let the trigger be patient. The trigger will not fire until the condition is met for successive `patience` checks.
- **mode** (*str*) – `'max'`, `'min'`, or `'auto'`. It is used to determine how to compare the monitored values.
- **verbose** (*bool*) – Enable verbose output. If verbose is true, you can get more information
- **max_trigger** (*Tuple[int, UnitLiteral]*) – Upper bound of the number of training loops

Methods

__init__([check_trigger, monitor, patience, ...])	
get_training_length()	
load_state_dict(state)	
may_fire(iteration, epoch_length)	Flags if the trigger may fire at the current iteration
state_dict()	

__call__(*manager*)

Decides whether the training loop should be stopped.

Parameters
manager (ExtensionsManager) – Manager object that this trigger is associated with. The `observation` of this manager is used to determine if the trigger should fire.

Returns
`True` if the training loop should be stopped.

Return type
bool

__init__(*check_trigger=(1, 'epoch')*, *monitor='main/loss'*, *patience=3*, *mode='auto'*, *verbose=False*, *max_trigger=(100, 'epoch')*)

Parameters

- **check_trigger** (*TriggerLike*) –
- **monitor** (*str*) –
- **patience** (*int*) –
- **mode** (*str*) –
- **verbose** (*bool*) –
- **max_trigger** (*Tuple[int, UnitLiteral]*) –

Return type
None

`get_training_length`()

Return type
Tuple[float, str]

`may_fire`(*iteration*, *epoch_length*)

Flags if the trigger may fire at the current iteration

This must not alter the trigger state

Parameters

- **iteration** (*int*) –
- **epoch_length** (*int*) –

Return type
bool

pytorch_pfn_extras.training.triggers.early_stopping_trigger.ExtensionsManagerProtocol

`class` pytorch_pfn_extras.training.triggers.early_stopping_trigger.`ExtensionsManagerProtocol`(**args*, ***kwargs*)

Bases: Protocol

Methods

__init__(*args, **kwargs)
get_extension(name)

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.triggers.interval_trigger

Classes

pytorch_pfn_extras.training.triggers.interval_trigger.ExtensionsManagerProtocol(...)	
pytorch_pfn_extras.training.triggers.interval_trigger.IntervalTrigger(...)	Trigger based on a fixed interval.

pytorch_pfn_extras.training.triggers.interval_trigger.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.triggers.interval_trigger.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: Protocol

Methods

__init__(*args, **kwargs)	
get_extension(name)	

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.triggers.interval_trigger.IntervalTrigger

class pytorch_pfn_extras.training.triggers.interval_trigger.**IntervalTrigger**(*period*, *unit*)

Bases: *Trigger*

Trigger based on a fixed interval.

This trigger accepts iterations divided by a given interval. There are two ways to specify the interval: per iterations and epochs. *Iteration* means the number of updates, while *epoch* means the number of sweeps over the training dataset. Fractional values are allowed if the interval is a number of epochs; the trigger uses the *iteration* and *epoch_detail* attributes defined by the manager.

For the description of triggers see `get_trigger()`.

Parameters

- **period** (*int or float*) – Length of the interval. Must be an integer if unit is `'iteration'`.
- **unit** (*str*) – Unit of the length specified by `period`. It must be either `'iteration'` or `'epoch'`.

Methods

__init__(period, unit)	
get_training_length()	
load_state_dict(state)	
may_fire(iteration, epoch_length)	Flags if the trigger may fire at the current iteration
state_dict()	

__call__(*manager*)

Decides whether the extension should be called on this iteration.

Parameters

manager (ExtensionsManager) – Manager object that this trigger is associated with. The iteration related information in this manager is used to determine if the trigger should fire.

Returns

True if the corresponding extension should be invoked in this iteration.

Return type
bool

__init__(*period*, *unit*)

Parameters

- **period** (*float*) –
- **unit** (*UnitLiteral*) –

get_training_length()

Return type
Tuple[float, str]

may_fire(*iteration*, *epoch_length*)

Flags if the trigger may fire at the current iteration

This must not alter the trigger state

Parameters

- **iteration** (*int*) –
- **epoch_length** (*int*) –

Return type
bool

pytorch_pfn_extras.training.triggers.manual_schedule_trigger

Classes

pytorch_pfn_extras.training.triggers.manual_schedule_trigger.ExtensionsManagerProtocol(...)	
pytorch_pfn_extras.training.triggers.manual_schedule_trigger.ManualScheduleTrigger(...)	Trigger invoked at specified point(s) of iterations or epochs.

pytorch_pfn_extras.training.triggers.manual_schedule_trigger.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.triggers.manual_schedule_trigger.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: Protocol

Methods

__init__(*args, **kwargs)
get_extension(name)

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.triggers.manual_schedule_trigger.ManualScheduleTrigger

class pytorch_pfn_extras.training.triggers.manual_schedule_trigger.**ManualScheduleTrigger**(*points*, *unit*)

Bases: *Trigger*

Trigger invoked at specified point(s) of iterations or epochs.

This trigger accepts iterations or epochs indicated by given point(s). There are two ways to specify the point(s): iteration and epoch. `iteration` means the number of updates, while `epoch` means the number of sweeps over the training dataset. Fractional values are allowed if the point is a number of epochs; the trigger uses the `iteration` and `epoch_detail` attributes defined by the manager.

Parameters

- **points** (*int, float, or list of int or float*) – time of the trigger. Must be an integer or list of integer if unit is `'iteration'`.
- **unit** (*str*) – Unit of the time specified by `points`. It must be either `'iteration'` or `'epoch'`.

Methods

__init__(points, unit)	
load_state_dict(state)	
may_fire(iteration, epoch_length)	Flags if the trigger may fire at the current iteration
state_dict()	

__call__(*manager*)

Decides whether the extension should be called on this iteration.

Parameters
manager (ExtensionsManager) – Manager object that this trigger is associated with. The iteration information in this manager is used to determine if the trigger should fire.

Returns
True if the corresponding extension should be invoked in this iteration.

Return type
bool

__init__(*points*, *unit*)

Parameters

- **points** (*Union[float, Sequence[float]]*) –
- **unit** (*UnitLiteral*) –

may_fire(*iteration*, *epoch_length*)

Flags if the trigger may fire at the current iteration

This must not alter the trigger state

Parameters

- **iteration** (*int*) –
- **epoch_length** (*int*) –

Return type
bool

pytorch_pfn_extras.training.triggers.minmax_value_trigger

Classes

pytorch_pfn_extras.training.triggers.minmax_value_trigger.BestValueTrigger(...)	Trigger invoked when specific value becomes best.
pytorch_pfn_extras.training.triggers.minmax_value_trigger.ExtensionsManagerProtocol(...)	
pytorch_pfn_extras.training.triggers.minmax_value_trigger.MaxValueTrigger(key)	Trigger invoked when specific value becomes maximum.
pytorch_pfn_extras.training.triggers.minmax_value_trigger.MinValueTrigger(key)	Trigger invoked when specific value becomes minimum.

pytorch_pfn_extras.training.triggers.minmax_value_trigger.BestValueTrigger

class pytorch_pfn_extras.training.triggers.minmax_value_trigger.**BestValueTrigger**(*key*, *compare*, *trigger=(1, 'epoch')*)

Bases: *Trigger*

Trigger invoked when specific value becomes best.

Parameters

- **key** (*str*) – Key of value.
- **compare** (*callable*) – Compare function which takes current best value and new value and returns whether new value is better than current best.
- **trigger** (*TriggerLike*) – Trigger that decides the comparison interval between current best value and new value. This must be a tuple in the form of `<int>, 'epoch'` or `<int>, 'iteration'` which is passed to `IntervalTrigger`.

Methods

__init__(key, compare[, trigger])	
load_state_dict(to_load)	
may_fire(iteration, epoch_length)	Flags if the trigger may fire at the current iteration
state_dict()	

__call__(*manager*)

Decides whether the extension should be called on this iteration.

Parameters
manager (ExtensionsManager) – Manager object that this trigger is associated with. The `observation` of this manager is used to determine if the trigger should fire.

Returns
`True` if the corresponding extension should be invoked in this iteration.

Return type
bool

__init__(*key*, *compare*, *trigger=(1, 'epoch')*)

Parameters

- **key** (*str*) –
- **compare** (*Callable[[float, float], bool]*) –
- **trigger** (*TriggerLike*) –

Return type
None

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

may_fire(*iteration*, *epoch_length*)

Flags if the trigger may fire at the current iteration

This must not alter the trigger state

Parameters

- **iteration** (*int*) –
- **epoch_length** (*int*) –

Return type
bool

`state_dict`()

Return type
Dict[str, *Any*]

pytorch_pfn_extras.training.triggers.minmax_value_trigger.ExtensionsManagerProtocol

`class` pytorch_pfn_extras.training.triggers.minmax_value_trigger.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: `Protocol`

Methods

__init__(*args, **kwargs)
get_extension(name)

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.triggers.minmax_value_trigger.MaxValueTrigger

class pytorch_pfn_extras.training.triggers.minmax_value_trigger.**MaxValueTrigger**(*key*, *trigger=(1, 'epoch')*)

Bases: *BestValueTrigger*

Trigger invoked when specific value becomes maximum.

For example you can use this trigger to take snapshot on the epoch the validation accuracy is maximum.

Parameters

- **key** (*str*) – Key of value. The trigger fires when the value associated with this key becomes maximum.
- **trigger** (*TriggerLike*) – Trigger that decides the comparison interval between current best value and new value. This must be a tuple in the form of `<int>, 'epoch'` or `<int>, 'iteration'` which is passed to `IntervalTrigger`.

Methods

__init__(key[, trigger])	
load_state_dict(to_load)	
may_fire(iteration, epoch_length)	Flags if the trigger may fire at the current iteration
state_dict()	

__init__(*key*, *trigger=(1, 'epoch')*)

Parameters

- **key** (*str*) –
- **trigger** (*TriggerLike*) –

pytorch_pfn_extras.training.triggers.minmax_value_trigger.MinValueTrigger

class pytorch_pfn_extras.training.triggers.minmax_value_trigger.**MinValueTrigger**(*key*, *trigger=(1, 'epoch')*)

Bases: *BestValueTrigger*

Trigger invoked when specific value becomes minimum.

For example you can use this trigger to take snapshot on the epoch the validation loss is minimum.

Parameters

- **key** (*str*) – Key of value. The trigger fires when the value associated with this key becomes minimum.
- **trigger** (*TriggerLike*) – Trigger that decides the comparison interval between current best value and new value. This must be a tuple in the form of `<int>, 'epoch'` or `<int>, 'iteration'` which is passed to `IntervalTrigger`.

Methods

__init__(key[, trigger])	
load_state_dict(to_load)	
may_fire(iteration, epoch_length)	Flags if the trigger may fire at the current iteration
state_dict()	

__init__(*key*, *trigger=(1, 'epoch')*)

Parameters

- **key** (*str*) –
- **trigger** (*TriggerLike*) –

pytorch_pfn_extras.training.triggers.once_trigger

Classes

pytorch_pfn_extras.training.triggers.once_trigger.ExtensionsManagerProtocol(...)	
pytorch_pfn_extras.training.triggers.once_trigger.OnceTrigger([...])	Trigger based on the starting point of the iteration.

pytorch_pfn_extras.training.triggers.once_trigger.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.triggers.once_trigger.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: Protocol

Methods

__init__(*args, **kwargs)
get_extension(name)

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.triggers.once_trigger.OnceTrigger

class pytorch_pfn_extras.training.triggers.once_trigger.**OnceTrigger**(*call_on_resume=False*)

Bases: *Trigger*

Trigger based on the starting point of the iteration.

This trigger accepts only once at starting point of the iteration. There are two ways to specify the starting point: only starting point in whole iteration or called again when training resumed.

Parameters
call_on_resume (*bool*) – Whether the extension is called again or not when restored from a snapshot. It is set to `False` by default.

finished

Flag that indicates whether or not this trigger will

Type
bool

fire in the future. This flag is used to determine if the extension

should be initialized after resume.

Methods

__init__([call_on_resume])	
load_state_dict(to_load)	
may_fire(iteration, epoch_length)	Flags if the trigger may fire at the current iteration
state_dict()	

Attributes

finished

__call__(*manager*)

Call self as a function.

Parameters
manager (ExtensionsManagerProtocol) –

Return type
bool

__init__(*call_on_resume=False*)

Parameters
call_on_resume (*bool*) –

Return type
None

property finished: bool

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

may_fire(*iteration*, *epoch_length*)

Flags if the trigger may fire at the current iteration

This must not alter the trigger state

Parameters

- **iteration** (*int*) –
- **epoch_length** (*int*) –

Return type
bool

state_dict()

Return type
Dict[str, *Any*]

pytorch_pfn_extras.training.triggers.time_trigger

Classes

pytorch_pfn_extras.training.triggers.time_trigger.ExtensionsManagerProtocol(...)	
pytorch_pfn_extras.training.triggers.time_trigger.TimeTrigger(period)	Trigger based on a fixed time interval.

pytorch_pfn_extras.training.triggers.time_trigger.ExtensionsManagerProtocol

class pytorch_pfn_extras.training.triggers.time_trigger.**ExtensionsManagerProtocol**(**args*, ***kwargs*)

Bases: Protocol

Methods

__init__(*args, **kwargs)
get_extension(name)

Attributes

elapsed_time
epoch
epoch_detail
is_before_training
iteration
models
observation
optimizers
out
raw_models
reporter
stop_trigger
writer

__init__(**args*, ***kwargs*)

property elapsed_time: float

property epoch: int

property epoch_detail: float

get_extension(*name*)

Parameters
name (*str*) –

Return type
Extension

property is_before_training: bool

property iteration: int

property models: Mapping[str, Module]

property observation: reporting.Observation

property optimizers: Mapping[str, Optimizer]

property out: str

property raw_models: Mapping[str, Module]

property reporter: *reporting.Reporter*

property stop_trigger: bool

property writer: Optional[*writing.Writer*]

pytorch_pfn_extras.training.triggers.time_trigger.TimeTrigger

class pytorch_pfn_extras.training.triggers.time_trigger.**TimeTrigger**(*period*)

Bases: *Trigger*

Trigger based on a fixed time interval.

This trigger accepts iterations with a given interval time.

Parameters
period (*float*) – Interval time. It is given in seconds.

Methods

__init__(period)	
load_state_dict(to_load)	
may_fire(iteration, epoch_len)	Flags if the trigger may fire at the current iteration
state_dict()	

__call__(*manager*)

Call self as a function.

Parameters
manager (ExtensionsManagerProtocol) –

Return type
bool

__init__(*period*)

Parameters
period (*float*) –

Return type
None

load_state_dict(*to_load*)

Parameters
to_load (*Dict[str, Any]*) –

Return type
None

state_dict()

Return type
Dict[str, *Any*]

pytorch_pfn_extras.utils

Modules

pytorch_pfn_extras.utils.checkpoint
pytorch_pfn_extras.utils.comparer

pytorch_pfn_extras.utils.checkpoint

Functions

pytorch_pfn_extras.utils.checkpoint.checkpoint(...)

pytorch_pfn_extras.utils.checkpoint.checkpoint

pytorch_pfn_extras.utils.checkpoint.**checkpoint**(*function*, **args*, ***kwargs*)

Parameters

- **function** (*Module*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

pytorch_pfn_extras.utils.comparer

Functions

pytorch_pfn_extras.utils.comparer.get_default_comparer([...])	Creates default comparer function.
pytorch_pfn_extras.utils.comparer.intermediate_value(...)	

pytorch_pfn_extras.utils.comparer.get_default_comparer

pytorch_pfn_extras.utils.comparer.**get_default_comparer**(*rtol=0.0001*, *atol=0*, *equal_nan=True*)

Creates default comparer function.

The created function will compare the outputs by using *torch.testing.assert_allclose* with specified options.

Parameters

- **rtol** (*float*) – Relative tolerance.
- **atol** (*float*) – Absolute tolerance.
- **equal_nan** (*bool*) – If True, NaNs will be ignored.

Return type
Callable[[str, str, str, *Any*, *Any*], None]

pytorch_pfn_extras.utils.comparer.intermediate_value

pytorch_pfn_extras.utils.comparer.**intermediate_value**(*name*, *value*)

Parameters

- **name** (*str*) –
- **value** (*Tensor*) –

Return type
None

Classes

pytorch_pfn_extras.utils.comparer.Comparer(*)	A class for comparison of iteration outputs and model parameters.
pytorch_pfn_extras.utils.comparer.ModelComparer(engines)	A class for comparison of iteration model parameters.
pytorch_pfn_extras.utils.comparer.OutputsComparer(engines)	A class for comparison of iteration outputs.

pytorch_pfn_extras.utils.comparer.Comparer

class pytorch_pfn_extras.utils.comparer.**Comparer**(*, *trigger=None*, *compare_fn=<function get_default_comparer.<locals>.compare_fn>*, *concurrency=None*, *outputs=True*, *params=False*, *baseline=None*)

Bases: object

A class for comparison of iteration outputs and model parameters.

This class is mainly used to compare results between different devices.

Parameters

- **trigger** (Trigger) – Trigger object that determines when to compare values.

- **compare_fn** (*function*) – Comparison function. Default is `get_default_comparer()`.
- **concurrency** (*int, optional*) – The upper bound limit on the number of workers that run concurrently. If `None`, inferred from the size of `engines`.
- **outputs** (*tuple of str or bool*) – A set of keys of output dict to compare.
- **params** (*tuple of str or bool*) – A set of keys of model parameters to compare.
- **baseline** (*str, optional*) – The baseline engine that is assumed to be correct.

Examples

```
>>> trainer_cpu = ppe.engine.create_trainer(
        model, optimizer, 1, device='cpu')
>>> trainer_gpu = ppe.engine.create_trainer(
        model, optimizer, 1, device='cuda:0')
>>> comp = ppe.utils.comparer.Comparer()
>>> comp.add_engine("cpu", engine_cpu, train_1, eval_1)
>>> comp.add_engine("gpu", engine_gpu, train_2, eval_2)
>>> comp.compare()
```

Methods

__init__(*[, trigger, compare_fn, ...])	A class for comparison of iteration outputs and model parameters.
add_dump(name, dir)	Add an engine to compare variables.
add_engine(name, engine, *args, **kwargs)	Add an engine to compare variables.
compare()	Compares outputs.
dump(engine, dir, *args, **kwargs)	Add an engine to compare variables.

__init__(*, *trigger=None*, *compare_fn=<function get_default_comparer.<locals>.compare_fn>*, *concurrency=None*, *outputs=True*, *params=False*, *baseline=None*)

A class for comparison of iteration outputs and model parameters.

This class is mainly used to compare results between different devices.

Parameters

- **trigger** (`Trigger`) – Trigger object that determines when to compare values.
- **compare_fn** (*function*) – Comparison function. Default is `get_default_comparer()`.
- **concurrency** (*int, optional*) – The upper bound limit on the number of workers that run concurrently. If `None`, inferred from the size of `engines`.
- **outputs** (*tuple of str or bool*) – A set of keys of output dict to compare.
- **params** (*tuple of str or bool*) – A set of keys of model parameters to compare.
- **baseline** (*str, optional*) – The baseline engine that is assumed to be correct.

Return type

None

Examples

```
>>> trainer_cpu = ppe.engine.create_trainer(
        model, optimizer, 1, device='cpu')
>>> trainer_gpu = ppe.engine.create_trainer(
        model, optimizer, 1, device='cuda:0')
>>> comp = ppe.utils.comparer.Comparer()
>>> comp.add_engine("cpu", engine_cpu, train_1, eval_1)
>>> comp.add_engine("gpu", engine_gpu, train_2, eval_2)
>>> comp.compare()
```

add_dump(*name*, *dir*)

Add an engine to compare variables.

Parameters

- **name** (*str*) – The name of dump.
- **dir** (*str*) – The directory that the results are saved to.

Return type
None

add_engine(*name*, *engine*, **args*, ***kwargs*)

Add an engine to compare variables.

Parameters

- **name** (*str*) – Engine name.
- **engine** (Trainer *or* Evaluator) – An engine to compare variables.
- ****kwargs** (**args and*) – Arguments passed to `engine.run`.
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

compare()

Compares outputs.

Return type
None

dump(*engine*, *dir*, **args*, ***kwargs*)

Add an engine to compare variables.

Parameters

- **engine** (Trainer *or* Evaluator) – An engine to compare variables.
- **dir** (*str*) – Name of the directory that the results are saved to.
- ****kwargs** (**args and*) – Arguments passed to `engine.run`.
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
None

pytorch_pfn_extras.utils.comparer.ModelComparer

class pytorch_pfn_extras.utils.comparer.**ModelComparer**(*engines*, *to_compare_keys=None*, *, *compare_fn=<function get_default_comparer.<locals>.compare_fn>*, *concurrency=None*)

Bases: `_ComparerBase`

A class for comparison of iteration model parameters.

This class is mainly used to compare results between different devices.

Parameters

- **engines** (*dict of Engines*) – Trainers or Evaluators to compare outputs.
- **to_compare_keys** (*tuple of str, optional*) – A set of keys of model parameters to compare.
- **compare_fn** (*function*) – Comparison function. Default is `get_default_comparer()`.
- **concurrency** (*int, optional*) – The upper bound limit on the number of workers that run concurrently. If `None`, inferred from the size of `engines`.

Examples

```
>>> trainer_cpu = ppe.engine.create_trainer(
        model, optimizer, 1, device='cpu')
>>> trainer_gpu = ppe.engine.create_trainer(
        model, optimizer, 1, device='cuda:0')
>>> comp = ppe.utils.comparer.ModelComparer(
        {"cpu": trainer_cpu, "gpu": trainer_gpu})
>>> comp.compare({"cpu": loader, "gpu": loader}])
```

Methods

__init__(engines[, to_compare_keys, ...])	A class for comparison of iteration model parameters.
compare(loaders[, n_iters])	Compares outputs.
compare_targets(name, engine, batch_idx, target)	
run_engine(engine, loaders)	

__init__(*engines*, *to_compare_keys=None*, *, *compare_fn=<function get_default_comparer.<locals>.compare_fn>*, *concurrency=None*)

A class for comparison of iteration model parameters.

This class is mainly used to compare results between different devices.

Parameters

- **engines** (*dict of Engines*) – Trainers or Evaluators to compare outputs.

- **to_compare_keys** (*tuple of str, optional*) – A set of keys of model parameters to compare.
- **compare_fn** (*function*) – Comparison function. Default is `get_default_comparer()`.
- **concurrency** (*int, optional*) – The upper bound limit on the number of workers that run concurrently. If `None`, inferred from the size of `engines`.

Examples

```
>>> trainer_cpu = ppe.engine.create_trainer(
        model, optimizer, 1, device='cpu')
>>> trainer_gpu = ppe.engine.create_trainer(
        model, optimizer, 1, device='cuda:0')
>>> comp = ppe.utils.comparer.ModelComparer(
        {"cpu": trainer_cpu, "gpu": trainer_gpu})
>>> comp.compare({"cpu": loader, "gpu": loader}])
```

pytorch_pfn_extras.utils.comparer.OutputsComparer

class pytorch_pfn_extras.utils.comparer.**OutputsComparer**(*engines*, *to_compare_keys=None*, ***, *compare_fn=<function get_default_comparer.<locals>.compare_fn>*, *concurrency=None*)

Bases: `_ComparerBase`

A class for comparison of iteration outputs.

This class is mainly used to compare results between different devices.

Parameters

- **engines** (*dict of Engines*) – Trainers or Evaluators to compare outputs.
- **to_compare_keys** (*tuple of str, optional*) – A set of keys of output dict to compare.
- **compare_fn** (*function*) – Comparison function. Default is `get_default_comparer()`.
- **concurrency** (*int, optional*) – The upper bound limit on the number of workers that run concurrently. If `None`, inferred from the size of `engines`.

Examples

```
>>> trainer_cpu = ppe.engine.create_trainer(
        model, optimizer, 1, device='cpu')
>>> trainer_gpu = ppe.engine.create_trainer(
        model, optimizer, 1, device='cuda:0')
>>> comp = ppe.utils.comparer.OutputsComparer(
        {"cpu": trainer_cpu, "gpu": trainer_gpu})
>>> comp.compare({"cpu": loader, "gpu": loader}])
```

Methods

`__init__`(engines[, to_compare_keys, ...])	A class for comparison of iteration outputs.
`compare`(loaders[, n_iters])	Compares outputs.
`compare_targets`(name, engine, batch_idx, target)	
`run_engine`(engine, loaders)	

__init__(*engines*, *to_compare_keys=None*, ***, *compare_fn=<function get_default_comparer.<locals>.compare_fn>*, *concurrency=None*)

A class for comparison of iteration outputs.

This class is mainly used to compare results between different devices.

Parameters

- **engines** (*dict of Engines*) – Trainers or Evaluators to compare outputs.
- **to_compare_keys** (*tuple of str, optional*) – A set of keys of output dict to compare.
- **compare_fn** (*function*) – Comparison function. Default is `get_default_comparer()`.
- **concurrency** (*int, optional*) – The upper bound limit on the number of workers that run concurrently. If `None`, inferred from the size of `engines`.

Return type
None

Examples

```
>>> trainer_cpu = ppe.engine.create_trainer(
        model, optimizer, 1, device='cpu')
>>> trainer_gpu = ppe.engine.create_trainer(
        model, optimizer, 1, device='cuda:0')
>>> comp = ppe.utils.comparer.OutputsComparer(
        {"cpu": trainer_cpu, "gpu": trainer_gpu})
>>> comp.compare({"cpu": loader, "gpu": loader}])
```

pytorch_pfn_extras.writing

Classes

pytorch_pfn_extras.writing.ProcessQueueWriter([...])	Snapshot writer that uses process queue.
pytorch_pfn_extras.writing.ProcessWriter([...])	Snapshot writer that uses a separate process.
pytorch_pfn_extras.writing.QueueWriter([...])	Base class of queue snapshot writers.
pytorch_pfn_extras.writing.SimpleWriter([...])	The most simple snapshot writer.
pytorch_pfn_extras.writing.StandardWriter([...])	Base class of snapshot writers which use thread or process.
pytorch_pfn_extras.writing.TensorBoardWriter([...])	Writer that sends statistics to TensorBoard.
pytorch_pfn_extras.writing.ThreadQueueWriter([...])	Snapshot writer that uses a thread queue.
pytorch_pfn_extras.writing.ThreadWriter([...])	Snapshot writer that uses a separate thread.
pytorch_pfn_extras.writing.Writer([fs, out_dir])	Base class of snapshot writers.

pytorch_pfn_extras.writing.ProcessQueueWriter

class pytorch_pfn_extras.writing.**ProcessQueueWriter**(*savefun=<function save>*, *fs=None*, *out_dir=''*, *task=None*)

Bases: *QueueWriter*[Process]

Snapshot writer that uses process queue.

This class creates a process and a queue by `multiprocessing` module. The process will be a consumer of this queue, and the main process will be a producer of this queue.

Note: Forking a new process from MPI process might be danger. Consider using *ThreadQueueWriter* instead of `ProcessQueueWriter` if you are using MPI.

See also:

- *pytorch_pfn_extras.training.extensions.snapshot()*

Methods

__init__([savefun, fs, out_dir, task])	
consume(q)	
create_consumer(q)	
create_queue()	
create_task(savefun)	
finalize()	Finalizes the writer.
initialize(out_dir)	
save(filename, out_dir, target, savefun, ...)	

__init__(*savefun=<function save>*, *fs=None*, *out_dir=''*, *task=None*)

Parameters

- **savefun** (*Callable[[...], None]*) –
- **fs** (*Optional[Any]*) –
- **out_dir** (*str*) –
- **task** (*Optional[Callable[[...], None]]*) –

Return type
None

create_consumer(*q*)

Parameters
q (*queue.Queue[_QueUnit]*) –

Return type
Process

create_queue()

Return type
queue.Queue[_QueUnit]

pytorch_pfn_extras.writing.ProcessWriter

class pytorch_pfn_extras.writing.**ProcessWriter**(*savefun=<function save>*, *fs=None*, *out_dir=''*, ***kwds*)

Bases: *StandardWriter*[Process]

Snapshot writer that uses a separate process.

This class creates a new process that invokes the actual saving function.

Note: Forking a new process from a MPI process might be danger. Consider using *ThreadWriter* instead of `ProcessWriter` if you are using MPI.

See also:

- *pytorch_pfn_extras.training.extensions.snapshot()*

Methods

__init__([savefun, fs, out_dir])	
create_worker(filename, out_dir, target, *)	Creates a worker for the snapshot.
`finalize`()	Finalizes the writer.
`initialize`(out_dir)	
`save`(filename, out_dir, target, savefun, ...)	

__init__(*savefun=<function save>*, *fs=None*, *out_dir=''*, ***kwds*)

Parameters

- **savefun** (*Callable[[...], None]*) –
- **fs** (*Optional[Any]*) –
- **out_dir** (*str*) –
- **kwds** (*Any*) –

Return type
None

create_worker(*filename*, *out_dir*, *target*, ***, *savefun=None*, *append=False*, ***savefun_kwargs*)

Creates a worker for the snapshot.

This method creates a thread or a process to take a snapshot. The created worker must have `start()` and `join()` methods. If the worker has an `exitcode` attribute (e.g., `multiprocessing.Process`), the value will be tested.

Parameters

- **filename** (*str*) –
- **out_dir** (*str*) –
- **target** (*Union[Sequence[Any], Mapping[str, Any]]*) –
- **savefun** (*Optional[Callable[[...], None]]*) –
- **append** (*bool*) –
- **savefun_kwargs** (*Any*) –

Return type
Process

pytorch_pfn_extras.writing.QueueWriter

class pytorch_pfn_extras.writing.**QueueWriter**(*savefun=<function save>*, *fs=None*, *out_dir=''*, *task=None*)

Bases: *Writer*, Generic[_Worker]

Base class of queue snapshot writers.

This class is a base class of snapshot writers that use a queue. A Queue is created when this class is constructed, and every time when __call__ is invoked, a snapshot task is put into the queue.

Parameters

- **savefun** – Callable object which is passed to the *create_task()* if the task is None. It takes three arguments: the output file path, the serialized dictionary object, and the optional keyword arguments.
- **fs** – FileSystem abstracting interface to implement all the operations. optional, defaults to None
- **out_dir** – str. Specifies the directory this writer will use. It takes precedence over the one specified in *__call__* optional, defaults to ''
- **task** – Callable object. Its __call__ must have a same interface to Writer.__call__. This object is directly put into the queue.

See also:

- *pytorch_pfn_extras.training.extensions.snapshot()*

Methods

__init__([savefun, fs, out_dir, task])	
consume(q)	
create_consumer(q)	
create_queue()	
create_task(savefun)	
finalize()	Finalizes the writer.
initialize(out_dir)	
save(filename, out_dir, target, savefun, ...)	

__call__(*filename*, *out_dir*, *target*, *, *savefun=None*, *append=False*)

Does the actual writing to the file.

This method is invoked by a Snapshot object every time it takes a snapshot.

Parameters

- **filename** (*str*) – Name of the file into which the serialized target is saved. It is a concrete file name, i.e. not a pre-formatted template string.

- **out_dir** (*str*) – Output directory. Corresponds to :py:attr:`ExtensionsManager.out <pytorch_pfn_extras.training.ExtensionsManager.out>`.
- **target** (*dict*) – Serialized object which will be saved.
- **savefun** (*callable*) – A callable that accepts a two positional arguments (an object to be serialized, file path) like *torch.save*.
- **append** (*bool*) – Mode used to open the file. True to use the append mode, False to use the write mode (truncates the file if it already exists).

Return type
None

__init__(*savefun=<function save>*, *fs=None*, *out_dir=''*, *task=None*)

Parameters
- **savefun** (*Callable[[...], None]*) –
- **fs** (*Optional[Any]*) –
- **out_dir** (*str*) –
- **task** (*Optional[Callable[[...], None]]*) –

Return type
None

consume(*q*)

Parameters
q (*queue.Queue[_QueUnit]*) –

Return type
None

create_consumer(*q*)

Parameters
q (*queue.Queue[_QueUnit]*) –

Return type
_Worker

create_queue()

Return type
queue.Queue[_QueUnit]

create_task(*savefun*)

Parameters
savefun (*Callable[[...], None]*) –

Return type
Callable[[...], None]

finalize()

Finalizes the writer.

Calling this method on already-finalized Writer does nothing.

Return type
None

pytorch_pfn_extras.writing.SimpleWriter

class pytorch_pfn_extras.writing.**SimpleWriter**(*savefun=<function save>*, *fs=None*, *out_dir=''*, ***kwds*)

Bases: *Writer*

The most simple snapshot writer.

This class just passes the arguments to the actual saving function.

Parameters

- **savefun** (*Callable[[...], None]*) – Callable object. It takes three arguments: the output file path, the serialized dictionary object, and the optional keyword arguments.
- **fs** (*Any*) – FileSystem abstracting interface to implement all the operations. optional, defaults to None
- **out_dir** (*str*) – str. Specifies the directory this writer will use. It takes precedence over the one specified in *__call__* optional, defaults to `''`
- **kwds** (*Any*) – Keyword arguments for the `savefun`.

See also:

- *pytorch_pfn_extras.training.extensions.snapshot()*

Methods

__init__([savefun, fs, out_dir])	
`finalize`()	Finalizes the writer.
`initialize`(out_dir)	
`save`(filename, out_dir, target, savefun, ...)	

__call__(*filename*, *out_dir*, *target*, ***, *savefun=None*, *append=False*)

Does the actual writing to the file.

This method is invoked by a `Snapshot` object every time it takes a snapshot.

Parameters

- **filename** (*str*) – Name of the file into which the serialized target is saved. It is a concrete file name, i.e. not a pre-formatted template string.
- **out_dir** (*str*) – Output directory. Corresponds to :py:attr:`ExtensionsManager.out <pytorch_pfn_extras.training.ExtensionsManager.out>`.
- **target** (*dict*) – Serialized object which will be saved.
- **savefun** (*callable*) – A callable that accepts a two positional arguments (an object to be serialized, file path) like *torch.save*.
- **append** (*bool*) – Mode used to open the file. True to use the append mode, False to use the write mode (truncates the file if it already exists).

Return type

None

__init__(*savefun=<function save>*, *fs=None*, *out_dir=''*, ***kwds*)

Parameters

- **savefun** (*Callable[[...], None]*) –
- **fs** (*Optional[Any]*) –
- **out_dir** (*str*) –
- **kwds** (*Any*) –

Return type
None

pytorch_pfn_extras.writing.StandardWriter

class pytorch_pfn_extras.writing.**StandardWriter**(*savefun=<function save>*, *fs=None*, *out_dir=''*, ***kwds*)

Bases: *Writer*, Generic[_Worker]

Base class of snapshot writers which use thread or process.

This class creates a new thread or a process every time when __call__ is invoked.

Parameters

- **savefun** – Callable object. It takes three arguments: the output file path, the serialized dictionary object, and the optional keyword arguments.
- **fs** – FileSystem abstracting interface to implement all the operations. optional, defaults to None
- **out_dir** – str. Specifies the directory this writer will use. It takes precedence over the one specified in *__call__* optional, defaults to ''
- **kwds** – Keyword arguments for the savefun.

See also:

- *pytorch_pfn_extras.training.extensions.snapshot()*

Methods

__init__([savefun, fs, out_dir])	
create_worker(filename, out_dir, target, *)	Creates a worker for the snapshot.
finalize()	Finalizes the writer.
initialize(out_dir)	
save(filename, out_dir, target, savefun, ...)	

__call__(*filename*, *out_dir*, *target*, ***, *savefun=None*, *append=False*)

Does the actual writing to the file.

This method is invoked by a Snapshot object every time it takes a snapshot.

Parameters

- **filename** (*str*) – Name of the file into which the serialized target is saved. It is a concrete file name, i.e. not a pre-formatted template string.
- **out_dir** (*str*) – Output directory. Corresponds to :py:attr:`ExtensionsManager.out <pytorch_pfn_extras.training.ExtensionsManager.out>`.
- **target** (*dict*) – Serialized object which will be saved.
- **savefun** (*callable*) – A callable that accepts a two positional arguments (an object to be serialized, file path) like *torch.save*.
- **append** (*bool*) – Mode used to open the file. True to use the append mode, False to use the write mode (truncates the file if it already exists).

Return type
None

__init__(*savefun=<function save>*, *fs=None*, *out_dir=''*, ***kwds*)

Parameters

- **savefun** (*Callable[[...], None]*) –
- **fs** (*Optional[Any]*) –
- **out_dir** (*str*) –
- **kwds** (*Any*) –

Return type
None

create_worker(*filename*, *out_dir*, *target*, ***, *savefun=None*, *append=False*, ***savefun_kwargs*)

Creates a worker for the snapshot.

This method creates a thread or a process to take a snapshot. The created worker must have `start()` and `join()` methods. If the worker has an `exitcode` attribute (e.g., `multiprocessing.Process`), the value will be tested.

Parameters

- **filename** (*str*) –
- **out_dir** (*str*) –
- **target** (*Union[Sequence[Any], Mapping[str, Any]]*) –
- **savefun** (*Optional[Callable[[...], None]]*) –
- **append** (*bool*) –
- **savefun_kwargs** (*Any*) –

Return type
_Worker

finalize()

Finalizes the writer.

Calling this method on already-finalized Writer does nothing.

Return type
None

pytorch_pfn_extras.writing.TensorBoardWriter

class pytorch_pfn_extras.writing.**TensorBoardWriter**(*savefun=None*, *fs=None*, *out_dir=''*, *stats=None*, ***kwds*)

Bases: `object`

Writer that sends statistics to TensorBoard.

This class contains a *torch.utils.tensorboard.SummaryWriter* object that is used to send the collected statistics to TensorBoard. A list of stats can be specified to report only the desired ones.

Parameters

- **savefun** (*Optional[Callable[[...], None]]*) – Ignored.
- **fs** (*Any*) – Ignored.
- **out_dir** (*str*) – Passed as `log_dir` argument to SummaryWriter.
- **stats** (*list*) – List of statistic keys.
- **kwds** (*Any*) – Passed as an additional arguments to SummaryWriter.

Methods

__init__([savefun, fs, out_dir, stats])	
finalize()	

__call__(*filename*, *out_dir*, *target*, ***, *savefun=None*, *append=False*)

Sends the statistics to the TensorBoard.

Parameters

- **filename** (*str*) – Ignored.
- **out_dir** (*str*) – Ignored.
- **target** (*dict or list*) – The statistics of the iteration. If given as a list, only the last element (assumed to be a dict containing the latest iteration statistics) is reported.
- **savefun** (*Optional[Callable[[...], None]]*) – Ignored.
- **append** (*bool*) – Ignored.

Return type
None

__init__(*savefun=None*, *fs=None*, *out_dir=''*, *stats=None*, ***kwds*)

Parameters

- **savefun** (*Optional[Callable[[...], None]]*) –
- **fs** (*Optional[Any]*) –
- **out_dir** (*str*) –
- **stats** (*Optional[KeysView[str]]*) –
- **kwds** (*Any*) –

Return type
None

`finalize`()

Return type
None

pytorch_pfn_extras.writing.ThreadQueueWriter

`class` pytorch_pfn_extras.writing.`ThreadQueueWriter`(*savefun=<function save>*, *fs=None*, *out_dir=''*, *task=None*)

Bases: *QueueWriter*[Thread]

Snapshot writer that uses a thread queue.

This class creates a thread and a queue by `threading` and `queue` modules respectively. The thread will be a consumer of the queue, and the main thread will be a producer of the queue.

See also:

- *pytorch_pfn_extras.training.extensions.snapshot()*

Methods

__init__([savefun, fs, out_dir, task])	
consume(q)	
create_consumer(q)	
create_queue()	
create_task(savefun)	
finalize()	Finalizes the writer.
initialize(out_dir)	
save(filename, out_dir, target, savefun, ...)	

`__init__`(*savefun=<function save>*, *fs=None*, *out_dir=''*, *task=None*)

Parameters

- **savefun** (*Callable[[...], None]*) –
- **fs** (*Optional[Any]*) –
- **out_dir** (*str*) –
- **task** (*Optional[Callable[[...], None]]*) –

Return type
None

create_consumer(*q*)

Parameters
q (*queue.Queue[_QueUnit]*) –

Return type
Thread

create_queue()

Return type
queue.Queue[_QueUnit]

pytorch_pfn_extras.writing.ThreadWriter

class pytorch_pfn_extras.writing.**ThreadWriter**(*savefun=<function save>*, *fs=None*, *out_dir=''*, ***kwds*)

Bases: *StandardWriter*[Thread]

Snapshot writer that uses a separate thread.

This class creates a new thread that invokes the actual saving function.

See also:

- *pytorch_pfn_extras.training.extensions.snapshot()*

Methods

__init__([savefun, fs, out_dir])	
create_worker(filename, out_dir, target, *)	Creates a worker for the snapshot.
finalize()	Finalizes the writer.
initialize(out_dir)	
save(filename, out_dir, target, savefun, ...)	

__init__(*savefun=<function save>*, *fs=None*, *out_dir=''*, ***kwds*)

Parameters
- **savefun** (*Callable[[...], None]*) –
- **fs** (*Optional[Any]*) –
- **out_dir** (*str*) –
- **kwds** (*Any*) –

Return type
None

create_worker(*filename*, *out_dir*, *target*, ***, *savefun=None*, *append=False*, ***savefun_kwargs*)

Creates a worker for the snapshot.

This method creates a thread or a process to take a snapshot. The created worker must have `start()` and `join()` methods. If the worker has an `exitcode` attribute (e.g., `multiprocessing.Process`), the value will be tested.

Parameters

- **filename** (*str*) –
- **out_dir** (*str*) –
- **target** (*Union[Sequence[Any], Mapping[str, Any]]*) –
- **savefun** (*Optional[Callable[[...], None]]*) –
- **append** (*bool*) –
- **savefun_kwargs** (*Any*) –

Return type
Thread

pytorch_pfn_extras.writing.Writer

class pytorch_pfn_extras.writing.**Writer**(*fs=None*, *out_dir=''*)

Bases: `object`

Base class of snapshot writers.

`Snapshot` invokes `__call__` of this class every time when taking a snapshot. This class determines how the actual saving function will be invoked.

Note: This extension first writes the serialized object to a temporary file and then rename it to the target file name. Thus, if the program stops right before the renaming, the temporary file might be left in the output directory.

See also:

- *pytorch_pfn_extras.training.extensions.snapshot()*

Methods

__init__([fs, out_dir])	
finalize()	Finalizes the writer.
initialize(out_dir)	
save(filename, out_dir, target, savefun, ...)	

Parameters

- **fs** (*Any*) –
- **out_dir** (*str*) –

__call__(*filename*, *out_dir*, *target*, ***, *savefun=None*, *append=False*)

Does the actual writing to the file.

This method is invoked by a `Snapshot` object every time it takes a snapshot.

Parameters

- **filename** (*str*) – Name of the file into which the serialized target is saved. It is a concrete file name, i.e. not a pre-formatted template string.
- **out_dir** (*str*) – Output directory. Corresponds to :py:attr:`ExtensionsManager.out <pytorch_pfn_extras.training.ExtensionsManager.out>`.
- **target** (*dict*) – Serialized object which will be saved.
- **savefun** (*callable*) – A callable that accepts a two positional arguments (an object to be serialized, file path) like *torch.save*.
- **append** (*bool*) – Mode used to open the file. True to use the append mode, False to use the write mode (truncates the file if it already exists).

Return type
None

__init__(*fs=None*, *out_dir=''*)

Parameters
- **fs** (*Optional[Any]*) –
- **out_dir** (*str*) –

Return type
None

finalize()

Finalizes the writer.

Calling this method on already-finalized Writer does nothing.

Return type
None

initialize(*out_dir*)

Parameters
out_dir (*str*) –

Return type
None

save(*filename*, *out_dir*, *target*, *savefun*, *append*, ***savefun_kwargs*)

Parameters
- **filename** (*str*) –
- **out_dir** (*str*) –
- **target** (*Union[Sequence[Any], Mapping[str, Any]]*) –
- **savefun** (*Callable[[...], None]*) –
- **append** (*bool*) –
- **savefun_kwargs** (*Any*) –

Return type
None

2.2 Training Loop

2.2.1 Trainer

engine.create_trainer(models, optimizers, ...)	Creates a trainer object.
engine.create_evaluator(models, *[, ...])	Creates an evaluator object.
handler.BaseLogic([options])	
handler.Logic([model_name, options])	A set of methods that defines the training logic.
handler.BaseHandler(logic, options, *args, ...)	Base class of Handler.
handler.Handler(logic, entry_runtime, options)	A set of callback functions to perform device-specific operations.
runtime.BaseRuntime(device_spec, options)	A base class for collections of device-specific callback functions.
runtime.PyTorchRuntime(device_spec, options)	A collections of callback functions for the devices that PyTorch supports by default.

2.2.2 Extensions Manager

training.ExtensionsManager(models, ...[, ...])	Manages the extensions and the current status.
training.IgniteExtensionsManager(engine, ...)	Manages extensions and the current status in Ignite training loop.

2.2.3 Extensions

training.extension.make_extension([trigger, ...])	Decorator to make given function into an extension.
training.extension.Extension()	Base class of extensions.
training.extension.ExtensionEntry(extension, *)	Extension and options.

training.extensions.BestValue(key, compare)	Extension traces the best value of a specific key in the observation.
training.extensions.Evaluator(self, ...[, ...])	An extension to evaluate models on a validation set.
training.extensions.LogReport([keys, ...])	An extension to output the accumulated results to a log file.
training.extensions.MaxValue(key[, trigger])	Extension traces the maximum value of a specific key in the observation.
training.extensions.MicroAverage(...[, trigger])	Calculates micro-average ratio.
training.extensions.MinValue(key[, trigger])	Extension traces the maximum value of a specific key in the observation.
training.extensions.observe_lr(optimizer[, ...])	Returns an extension to record the learning rate.
training.extensions.observe_value(...)	Returns an extension to continuously record a value.
training.extensions.ParameterStatistics(links)	An extension to report parameter statistics.
training.extensions.PlotReport(y_keys[, ...])	An extension to output plots.
training.extensions.PrintReport([entries, ...])	An extension to print the accumulated results.
training.extensions.ProgressBar([...])	An extension to print a progress bar and recent training status.
training.extensions.ProfileReport([...])	Writes the profile results to a file.
training.extensions.snapshot([savefun, ...])	Returns a trainer extension to take snapshots of the trainer.
training.extensions.Slack(channel[, msg, ...])	An extension to communicate with Slack.
training.extensions.SlackWebhook(url[, msg, ...])	An extension to communicate with Slack using Incoming Webhook.
training.extensions.VariableStatisticsPlot(targets)	An extension to plot statistics for `Tensor`s.

2.2.4 Triggers

training.triggers.EarlyStoppingTrigger(self)	Trigger for Early Stopping
training.triggers.IntervalTrigger(period, unit)	Trigger based on a fixed interval.
training.triggers.ManualScheduleTrigger(...)	Trigger invoked at specified point(s) of iterations or epochs.
training.triggers.BestValueTrigger(key, compare)	Trigger invoked when specific value becomes best.
training.triggers.MaxValueTrigger(key[, trigger])	Trigger invoked when specific value becomes maximum.
training.triggers.MinValueTrigger(key[, trigger])	Trigger invoked when specific value becomes minimum.
training.triggers.OnceTrigger([call_on_resume])	Trigger based on the starting point of the iteration.
training.triggers.TimeTrigger(period)	Trigger based on a fixed time interval.

2.2.5 Reporting

reporting.Reporter()	Object to which observed values are reported.
reporting.report(values[, observer])	Reports observed values with the current reporter object.
reporting.report_scope(observation)	Returns a report scope with the current reporter.

2.2.6 Logging

logging.get_logger(name)	Returns a child logger to be used by applications.

2.2.7 Profiler

profiler.TimeSummary.report(tag[, use_cuda])	Context manager to automatically report execution times.

2.3 Distributed Training

nn.parallel.DistributedDataParallel(module)	Module for distributed data parallelism
distributed.initialize_ompi_environment(*[, ...])	Initialize *torch.distributed* environments with values taken from OpenMPI.

2.4 Check Pointing

utils.checkpoint	

2.5 Lazy Modules

nn.Ensure(*[, shape, dtype, broadcastable, ...])	Module to check the shape of a tensor.
nn.ensure(tensor[, shape, dtype, ...])	Checks the shape and type of a tensor.
nn.LazyLinear(in_features, *args, **kwargs)	Linear module with lazy weight initialization.
nn.LazyConv1d(in_channels, *args, **kwargs)	Conv1d module with lazy weight initialization.
nn.LazyConv2d(in_channels, *args, **kwargs)	Conv2d module with lazy weight initialization.
nn.LazyConv3d(in_channels, *args, **kwargs)	Conv3d module with lazy weight initialization.
nn.LazyBatchNorm1d(num_features, *args, **kwargs)	BatchNorm1d module with lazy weight initialization.
nn.LazyBatchNorm2d(num_features, *args, **kwargs)	BatchNorm2d module with lazy weight initialization.
nn.LazyBatchNorm3d(num_features, *args, **kwargs)	BatchNorm3d module with lazy weight initialization.

2.6 ONNX

2.6.1 Export

onnx.export(model, args, f[, return_output, ...])	Export model into ONNX Graph.
onnx.export_testcase(model, args, out_dir, *)	Export model and I/O tensors of the model in protobuf format.

2.6.2 Annotation

onnx.annotate(**attrs)	Annotation parameters to the target function.
onnx.apply_annotation(fn, *args, **attrs)	Annotation applier to the target function
onnx.scoped_anchor(**attrs)	Add anchor node to the scoped modules
onnx.export(model, args, f[, return_output, ...])	Export model into ONNX Graph.
onnx.export_testcase(model, args, out_dir, *)	Export model and I/O tensors of the model in protobuf format.

2.7 Datasets

dataset.SharedDataset(sm_size[, cache_type])	Dataset that caches the load samples in shared memory
dataset.TabularDataset(*args, **kwds)	An abstract class that represents tabular dataset.
dataset.ItemNotFoundException	

2.8 Config

config.Config(config[, types])	

config_types.optuna_types(trial)	
config_types.load_path_with_optuna_types(...)	

2.9 NumPy/CuPy Compatibility

from_ndarray(ndarray)	Creates a *torch.Tensor* from a *numpy.ndarray* or *cupy.ndarray*.
as_ndarray(tensor)	Creates a *numpy.ndarray* or *cupy.ndarray* from *torch.Tensor*.
get_xp(obj)	Returns a module of ndarray implementation (*numpy* or *cupy*) for the given *obj*.
as_numpy_dtype(torch_dtype)	Returns NumPy dtype for the given PyTorch dtype.
from_numpy_dtype(numpy_dtype)	Returns PyTorch dtype for the given NumPy dtype.

cuda.stream(stream)	Context-manager that selects a given stream.
cuda.use_torch_mempool_in_cupy()	Use the PyTorch memory pool in CuPy.
cuda.use_default_mempool_in_cupy()	Use the default memory pool in CuPy.

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

M

N

O

P

Q

R

S

T

U

V

W