pytorch-pfn-extras

Preferred Networks, Inc.

Jan 25, 2022

CONTENTS

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

CHAPTER

ONE

USER GUIDE

1.1 Trainer (techinical preview)

1.1.1 Trainer and Evaluator

Note: The Trainer/Evaluator APIs are currently under technical preview and may subject to change in the future versions.

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']),
]

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.
```

(continues on next page)

(continued from previous page)

```
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.

1.1.2 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.

1.1.3 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)
```

(continues on next page)

(continued from previous page)

```
        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
    # 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.

1.2 Extensions

1.2.1 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
```

(continues on next page)

(continued from previous page)

```
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.

1.2.2 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):
        """
        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']
```

1.2.3 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.).

1.2.4 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.

1.3 Utilities

1.3.1 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):
    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)
```

(continues on next page)

(continued from previous page)

```
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?
```


1.3.2 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'
```

(continues on next page)

(continued from previous page)

```
''')
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)
# 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'
```

(continues on next page)

(continued from previous page)

```
    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,
}

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
```

(continues on next page)

(continued from previous page)

```
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
}

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
```

(continues on next page)

(continued from previous page)

```
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
```

1.3.3 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
```

(continues on next page)

(continued from previous page)

```
└── 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__()
```

(continues on next page)

(continued from previous page)

```
        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]
) {
  %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]
  %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)
```

(continues on next page)

(continued from previous page)

```
  %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]
) 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)
```

(continues on next page)

(continued from previous page)

```
  %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)
        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]
```

(continues on next page)

(continued from previous page)

```
) 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__()
        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`.

1.3.4 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 Training Loop

2.1.1 Trainer (techincal preview)

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])	
handler.BaseHandler(logic, options, *args, ...)	
handler.Handler(logic, entry_runtime, options)	
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.

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*)

Creates a trainer object.

Parameters

- **models** (*Union[torch.nn.modules.module.Module, Mapping[str, torch.nn.modules.module.Module]]*) – Map of string to Module or an actual Module.
- **optimizers** (*Union[torch.optim.optimizer.Optimizer, Mapping[str, torch.optim.optimizer.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[training.Extension]]*) – 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[pytorch_pfn_extras.handler._logic.Logic]*) – A logic object. If *None* is given, an logic object is instantiated from the default logic class.
- **transform_model** (*Callable[[str, torch.nn.modules.module.Module], torch.nn.modules.module.Module]*) – A function to transform a model structure, often used to unwrap the a module from DDP module.
- **handler_class** (*Optional[Type[pytorch_pfn_extras.handler._handler.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.

Return type Trainer

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*)

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[torch.nn.modules.module.Module, Mapping[str, torch.nn.modules.module.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[pytorch_pfn_extras.handler._logic.Logic]*) – A logic object. If *None* is given, an logic object is instantiated from the default logic class.
- **handler_class** (*Optional[Type[pytorch_pfn_extras.handler._handler.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.

Return type *Evaluator*

pytorch_pfn_extras.handler.BaseLogic

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

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

__init__(*options=None*)

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

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.

pytorch_pfn_extras.handler.Logic

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

Parameters

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

Return type None

__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):** If `True`, `torch.cuda.amp.autocast` is enabled. Default is `False`.
  - **`'grad_scaler'` (torch.cuda.amp.GradScaler):** A gradient scaler that outputs are applied to.

Return type None

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.

pytorch_pfn_extras.handler.BaseHandler

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

Parameters

- **logic** (pytorch_pfn_extras.handler._logic.BaseLogic) –
- **options** (*Dict[str, Any]*) –
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type None

__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

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_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.

pytorch_pfn_extras.handler.Handler

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

Parameters

- **logic** (pytorch_pfn_extras.handler._logic.BaseLogic) –
- **entry_runtime** (BaseRuntime) –
- **options** (*Dict[str, Any]*) –

Return type None

__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

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_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.

pytorch_pfn_extras.runtime.BaseRuntime

class pytorch_pfn_extras.runtime.**BaseRuntime**(*device_spec*, *options*)

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.

Return type None

__init__(*device_spec*, *options*)

Parameters

- **device_spec** (*Union[str, torch.device]*) –
- **options** (*Dict[str, Any]*) –

Return type None

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.
`move_module`(module)	Transfers the module to the specific device.
`move_tensor`(tensor)	Transfers the tensor to the specific device.
`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.

pytorch_pfn_extras.runtime.PyTorchRuntime

class pytorch_pfn_extras.runtime.**PyTorchRuntime**(*device_spec*, *options*)

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):** If `True`, `torch.cuda.amp.autocast` is enabled. Default is `False`.
  - **'grad_scaler' (torch.cuda.amp.GradScaler):** A gradient scaler that outputs are applied to.

Return type None

__init__(*device_spec*, *options*)

Parameters

- **device_spec** (*Union[str, torch.device]*) –
- **options** (*Dict[str, Any]*) –

Return type None

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.
`move_module`(module)	Transfers the module to the specific device.
`move_tensor`(tensor)	Transfers the tensor to the specific device.
`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.

2.1.2 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.

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*)

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, torch.nn.modules.module.Module], torch.nn.modules.module.Module]*) –

Return type None

__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*)

Parameters

- **models** (*Union[torch.nn.modules.module.Module, Dict[str, torch.nn.modules.module.Module]]*) –
- **optimizers** (*Union[torch.optim.optimizer.Optimizer, Dict[str, torch.optim.optimizer.Optimizer]]*) –
- **max_epochs** (*int*) –
- **iters_per_epoch** (*int*) –
- **extensions** (*Optional[Sequence[extension_module.ExtensionLike]]*) –
- **out_dir** (*str*) –
- **stop_trigger** (*trigger_module.TriggerLike*) –
- **writer** (*Optional[pytorch_pfn_extras.writing._writer_base.Writer]*) –
- **transform_model** (*Callable[[str, torch.nn.modules.module.Module], torch.nn.modules.module.Module]*) –
- **enable_profile** (*bool*) –

Return type None

Methods

__init__(models, optimizers, max_epochs, *, …)	
complete_iteration(*[, observation, …])	Context manager to complete deferred iterations.
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()	

continues on next page

Table 9 – continued from previous page

run_extensions(*[, completed, only_iterations])	
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

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*)

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*.

Return type None

__init__(*engine*, *models*, *optimizers*, *max_epochs*, *, *extensions=None*, *out_dir='result'*, *writer=None*, *enable_profile=False*)

Parameters

- **engine** (*ignite.engine.Engine*) –
- **models** (*Union[torch.nn.modules.module.Module, Mapping[str, torch.nn.modules.module.Module]]*) –
- **optimizers** (*Union[torch.optim.optimizer.Optimizer, Mapping[str, torch.optim.optimizer.Optimizer]]*) –
- **max_epochs** (*int*) –
- **extensions** (*Optional[Sequence[extension_module.ExtensionLike]]*) –
- **out_dir** (*str*) –
- **writer** (*Optional[pytorch_pfn_extras.writing._writer_base.Writer]*) –
- **enable_profile** (*bool*) –

Return type None

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`(*[, completed, only_iterations])	
`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

2.1.3 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.

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** (*Callable[[], None]*) – 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[[pytorch_pfn_extras.training._manager_protocol.ExtensionsManagerProtocol, Exception, types.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]

pytorch_pfn_extras.training.extension.Extension

class pytorch_pfn_extras.training.extension.**Extension**

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.

__init__()

Methods

__init__()	
finalize()	Finalizes the extension.
initialize(manager)	Initializes up the manager state.
load_state_dict(to_load)	

continues on next page

Table 14 – continued from previous page

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	

pytorch_pfn_extras.training.extension.ExtensionEntry

class pytorch_pfn_extras.training.extension.**ExtensionEntry**(*extension*, *, *name=None*, *priority=None*, *trigger=None*, *call_before_training=False*)

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.

Return type None

See also:

`pytorch_pfn_extras.training.ExtensionsManager.extend()`

__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

Methods

__init__(extension, *[, name, priority, ...])
load_state_dict(to_load)
state_dict()

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.MicroAverage(...[, trigger])	Calculates micro-average ratio.
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.VariableStatisticsPlot(targets)	An extension to plot statistics for Tensors.

pytorch_pfn_extras.training.extensions.Evaluator

class pytorch_pfn_extras.training.extensions.**Evaluator**(*self*, *iterator*, *target*, *eval_func=None*, ***, *progress_bar=False*)

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[torch.utils.data.dataloader.DataLoader[Any], Dict[str, torch.utils.data.dataloader.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[torch.nn.modules.module.Module, Dict[str, torch.nn.modules.module.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*) –

Return type None

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.

__init__(*iterator*, *target*, *eval_hook=None*, *eval_func=None*, ***kwargs*)

Parameters

- **iterator** (*Union[torch.utils.data.dataloader.DataLoader[Any], Dict[str, torch.utils.data.dataloader.DataLoader[Any]]]*) –
- **target** (*Union[torch.nn.modules.module.Module, Dict[str, torch.nn.modules.module.Module]]*) –
- **eval_hook** (*Optional[Callable[[pytorch_pfn_extras.training.extensions.evaluator.Evaluator], None]]*) –
- **eval_func** (*Optional[Callable[[...], Any]]*) –
- **kwargs** (*Any*) –

Return type None

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()	Finalizes the evaluator object.
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

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*)

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[pytorch_pfn_extras.training._trigger_util.Trigger, Callable[[pytorch_pfn_extras.training._manager_protocol.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.

__init__(*keys=None*, *trigger=(1, 'epoch')*, *postprocess=None*, *filename=None*, *append=False*, *format=None*, ***kwargs*)

Parameters

- **keys** (*Optional[Iterable[str]]*) –
- **trigger** (*Optional[Union[pytorch_pfn_extras.training._trigger_util.Trigger, Callable[[pytorch_pfn_extras.training._manager_protocol.ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –
- **postprocess** (*Optional[Callable[[Mapping[str, Any]], None]]*) –
- **filename** (*Optional[str]*) –

- **append** (*bool*) –
- **format** (*Optional[str]*) –
- **kwargs** (*Any*) –

Methods

__init__([keys, trigger, postprocess, ...])	
finalize()	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	

pytorch_pfn_extras.training.extensions.MicroAverage

class pytorch_pfn_extras.training.extensions.**MicroAverage**(*numerator_key*, *denominator_key*, *result_key*, *trigger=(1, 'epoch')*)

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[pytorch_pfn_extras.training._trigger_util.Trigger, Callable[[pytorch_pfn_extras.training._manager_protocol.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`.

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[pytorch_pfn_extras.training._trigger_util.Trigger, Callable[[pytorch_pfn_extras.training._manager_protocol.ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –

Return type None

Methods

`__init__`(numerator_key, denominator_key, ...)	

`finalize`()	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`	

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[[pytorch_pfn_extras.training._manager_protocol.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.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*)

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 ~torch.nn.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[pytorch_pfn_extras.training._trigger_util.Trigger, Callable[[pytorch_pfn_extras.training._manager_protocol.*

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))`)

__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[pytorch_pfn_extras.training._trigger_util.Trigger, Callable[[pytorch_pfn_extras.training._manager_protocol.ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –
- **skip_nan_params** (*bool*) –

Methods

__init__(links[, statistics, report_params, ...])	
finalize()	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

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*)

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[pytorch_pfn_extras.training._trigger_util.Trigger, Callable[[pytorch_pfn_extras.training._manager_protocol.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*) –

__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[pytorch_pfn_extras.training._trigger_util.Trigger, Callable[[pytorch_pfn_extras.training._manager_protocol.ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –
- **postprocess** (*Optional[Any]*) –
- **filename** (*Optional[str]*) –
- **marker** (*str*) –
- **grid** (*bool*) –
- **kwargs** (*Any*) –

Methods

`__init__`(y_keys[, x_key, trigger, ...])	
`available`()	
`finalize`()	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`	

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'>*)

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.

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, pytorch_pfn_extras.training.extensions.log_report.LogReport]*) –
- **out** (*IO[Any]*) –

Return type None

Methods

__init__([entries, log_report, out])	
finalize()	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	

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'>*)

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.

__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*) –

Methods

__init__([training_length, update_interval, ...])	

finalize()	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	

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*)

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[pytorch_pfn_extras.training._trigger_util.Trigger, Callable[[pytorch_pfn_extras.training._manager_protocol.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)

__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[pytorch_pfn_extras.training._trigger_util.Trigger, Callable[[pytorch_pfn_extras.training._manager_protocol.ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –
- **filename** (*Optional[str]*) –
- **append** (*bool*) –
- **format** (*Optional[str]*) –
- **kwargs** (*Any*) –

Methods

__init__([store_keys, report_keys, trigger, ...])	
finalize()	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	

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[pytorch_pfn_extras.writing._writer_base.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 pytorch_pfn_extras.training.extensions._snapshot._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.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*)

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[pytorch_pfn_extras.training._trigger_util.Trigger, Callable[[pytorch_pfn_extras.training._manager_protocol.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*) –

__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[pytorch_pfn_extras.training._trigger_util.Trigger, Callable[[pytorch_pfn_extras.training._manager_protocol.ExtensionsManagerProtocol], bool], Tuple[float, str]]]*) –
- **filename** (*Optional[str]*) –
- **figsize** (*Optional[Tuple[int, ...]]*) –
- **marker** (*Optional[str]*) –
- **grid** (*bool*) –
- **kwargs** (*Any*) –

Methods

__init__(targets[, max_sample_size, ...])	
available()	
finalize()	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	

2.1.4 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.

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')*)

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

Return type None

__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

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`()	

pytorch_pfn_extras.training.triggers.IntervalTrigger

class pytorch_pfn_extras.training.triggers.**IntervalTrigger**(*period*, *unit*)

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'`.

__init__(*period*, *unit*)

Parameters

- **period** (*float*) –
- **unit** (*UnitLiteral*) –

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`()	

pytorch_pfn_extras.training.triggers.ManualScheduleTrigger

class pytorch_pfn_extras.training.triggers.**ManualScheduleTrigger**(*points*, *unit*)

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'`.

__init__(*points*, *unit*)

Parameters

- **points** (*Union[float, Sequence[float]]*) –
- **unit** (*UnitLiteral*) –

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`()	

pytorch_pfn_extras.training.triggers.BestValueTrigger

class pytorch_pfn_extras.training.triggers.**BestValueTrigger**(*key*, *compare*, *trigger=(1, 'epoch')*)
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`.

Return type None

__init__(*key*, *compare*, *trigger=(1, 'epoch')*)

Parameters

- **key** (*str*) –
- **compare** (*Callable[[float, float], bool]*) –
- **trigger** (*TriggerLike*) –

Return type None

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()	

pytorch_pfn_extras.training.triggers.MaxValueTrigger

class pytorch_pfn_extras.training.triggers.**MaxValueTrigger**(*key*, *trigger=(1, 'epoch')*)
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`.

__init__(*key*, *trigger=(1, 'epoch')*)

Parameters

- **key** (*str*) –
- **trigger** (*TriggerLike*) –

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()	

pytorch_pfn_extras.training.triggers.MinValueTrigger

class pytorch_pfn_extras.training.triggers.**MinValueTrigger**(*key*, *trigger=(1, 'epoch')*)

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`.

__init__(*key*, *trigger=(1, 'epoch')*)

Parameters

- **key** (*str*) –
- **trigger** (*TriggerLike*) –

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()	

pytorch_pfn_extras.training.triggers.OnceTrigger

class pytorch_pfn_extras.training.triggers.**OnceTrigger**(*call_on_resume=False*)

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.

Return type None

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.

__init__(*call_on_resume=False*)

Parameters **call_on_resume** (*bool*) –

Return type None

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

pytorch_pfn_extras.training.triggers.TimeTrigger

class pytorch_pfn_extras.training.triggers.**TimeTrigger**(*period*)

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.

Return type None

__init__(*period*)

Parameters **period** (*float*) –

Return type None

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()	

2.1.5 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.

pytorch_pfn_extras.reporting.Reporter

class pytorch_pfn_extras.reporting.**Reporter**

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):
```

(continues on next page)

(continued from previous page)

```
...         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.).

Return type None

observation
Dictionary of observed values.

__init__()

Return type None

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.

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)
```

(continues on next page)

(continued from previous page)

```
        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[torch.nn.modules.module.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[torch.Tensor, numpy.ndarray, numpy.floating, float, Callable[[], float]]]*) –

Return type Generator[None, None, None]

2.1.6 Logging

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 logging.Logger

2.1.7 Profiler

profiler.TimeSummary.report(tag[, use_cuda])	Context manager to automatically report execution times.

pytorch_pfn_extras.profiler.TimeSummary.report

TimeSummary.**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[pytorch_pfn_extras.profiler._time_summary._ReportNotification, None, None]

2.2 Distributed Training

nn.parallel.DistributedDataParallel(module)	Module for distributed data parallelism
distributed.initialize_ompi_environment(*[, ...])	Initialize *torch.distributed* environments with values taken from OpenMPI.

2.2.1 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*)

Module for distributed data parallelism

This class synchronizes the gradients and the buffers after backward computations.

Parameters

- **module** (*torch.nn.modules.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[torch._C._distributed_c10d.ProcessGroup]*) – Process group used for broadcasting and reducing. (default:

torch.distributed.group.WORLD)

- **reduce_function** (*Optional[Callable[[Sequence[torch.Tensor], Optional[torch._C._distributed_c10d.ProcessGroup]], None]]*) – All-reduce function
- **broadcast_function** (*Optional[Callable[[Sequence[torch.Tensor], Optional[torch._C._distributed_c10d.ProcessGroup]], None]]*) – Broadcast function
- **kwargs** (*Any*) –

Return type None

__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** (*torch.nn.modules.module.Module*) –
- **broadcast_buffers** (*bool*) –
- **negotiate_grads** (*bool*) –
- **process_group** (*Optional[torch._C._distributed_c10d.ProcessGroup]*) –
- **reduce_function** (*Optional[Callable[[Sequence[torch.Tensor], Optional[torch._C._distributed_c10d.ProcessGroup]], None]]*) –
- **broadcast_function** (*Optional[Callable[[Sequence[torch.Tensor], Optional[torch._C._distributed_c10d.ProcessGroup]], None]]*) –
- **kwargs** (*Any*) –

Return type None

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.
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 51 – continued from previous page

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.
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)	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)	Registers a backward hook on the module.
register_parameter(name, param)	Adds a parameter to the module.
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 a whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device)	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])	Sets gradients of all model parameters to zero.

Attributes

`T_destination`	alias of TypeVar('T_destination', bound=`Mapping`[`str`, `torch.Tensor`])
`dump_patches`	This allows better BC support for `load_state_dict()`.

2.2.2 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]

2.3 Check Pointing

utils.checkpoint

2.3.1 pytorch_pfn_extras.utils.checkpoint

Functions

`checkpoint`(function, *args, **kwargs)

2.4 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.

2.4.1 pytorch_pfn_extras.nn.Ensure

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

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[torch.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.

__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[torch.dtype]*) –
- **broadcastable** (*bool*) –
- **can_cast** (*bool*) –

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.
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.
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.
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)	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)	Registers a backward hook on the module.
register_parameter(name, param)	Adds a parameter to the module.

continues on next page

Table 56 – 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([destination, prefix, keep_vars])	Returns a dictionary containing a whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device)	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])	Sets gradients of all model parameters to zero.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Mapping[str, torch.Tensor])
dump_patches	This allows better BC support for load_state_dict().

2.4.2 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[torch.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** (*torch.Tensor*) –

Return type None

2.4.3 pytorch_pfn_extras.nn.LazyLinear

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

Linear module with lazy weight initialization.

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

Parameters

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

Return type None

__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

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.
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.

continues on next page

Table 58 – continued from previous page

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.
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.
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)	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)	Registers a backward hook on the module.
register_parameter(name, param)	Adds a parameter to the module.
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)	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])	Sets gradients of all model parameters to zero.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Mapping[str, torch.Tensor])
dump_patches	This allows better BC support for load_state_dict().
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

2.4.4 pytorch_pfn_extras.nn.LazyConv1d

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

Conv1d module with lazy weight initialization.

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

Parameters

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

Return type None

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

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

Parameters

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

Return type None

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.

continues on next page

Table 60 – continued from previous page

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.
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.
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)	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)	Registers a backward hook on the module.
register_parameter(name, param)	Adds a parameter to the module.
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.

continues on next page

Table 60 – continued from previous page

to_empty(*, device)	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])	Sets gradients of all model parameters to zero.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Mapping[str, torch.Tensor])
dump_patches	This allows better BC support for load_state_dict().
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

2.4.5 pytorch_pfn_extras.nn.LazyConv2d

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

Conv2d module with lazy weight initialization.

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

Parameters

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

Return type None

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

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

Parameters

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

Return type None

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.
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.
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.
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)	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)	Registers a backward hook on the module.
register_parameter(name, param)	Adds a parameter to the module.

continues on next page

Table 62 – continued from previous page

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)	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])	Sets gradients of all model parameters to zero.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Mapping[str, torch.Tensor])
dump_patches	This allows better BC support for load_state_dict().
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

2.4.6 pytorch_pfn_extras.nn.LazyConv3d

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

Conv3d module with lazy weight initialization.

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

Parameters

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

Return type None

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

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

Parameters

- **self** (*Any*) –
- **in_channels** (*Optional[int]*) –

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

Return type None

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.
`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.
`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.
`parameters`([recurse])	Returns an iterator over module parameters.
`register_backward_hook`(hook)	Registers a backward hook on the module.

continues on next page

Table 64 – continued from previous page

register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook)	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)	Registers a backward hook on the module.
register_parameter(name, param)	Adds a parameter to the module.
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)	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])	Sets gradients of all model parameters to zero.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Mapping[str, torch.Tensor])
dump_patches	This allows better BC support for load_state_dict().
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

2.4.7 pytorch_pfn_extras.nn.LazyBatchNorm1d

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

BatchNorm1d module with lazy weight initialization.

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

Parameters

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

Return type None

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

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

Parameters

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

Return type None

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.
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.
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.

continues on next page

Table 66 – continued from previous page

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)	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)	Registers a backward hook on the module.
register_parameter(name, param)	Adds a parameter to the module.
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)	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])	Sets gradients of all model parameters to zero.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Mapping[str, torch.Tensor])
dump_patches	This allows better BC support for load_state_dict().
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

2.4.8 pytorch_pfn_extras.nn.LazyBatchNorm2d

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

BatchNorm2d module with lazy weight initialization.

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

Parameters

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

Return type None

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

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

Parameters

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

Return type None

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.
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.

continues on next page

Table 68 – continued from previous page

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.
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.
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)	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)	Registers a backward hook on the module.
register_parameter(name, param)	Adds a parameter to the module.
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)	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])	Sets gradients of all model parameters to zero.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Mapping[str, torch.Tensor])
dump_patches	This allows better BC support for load_state_dict().
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

2.4.9 pytorch_pfn_extras.nn.LazyBatchNorm3d

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

BatchNorm3d module with lazy weight initialization.

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

Parameters

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

Return type None

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

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

Parameters

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

Return type None

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.
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 70 – 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.
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.
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)	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)	Registers a backward hook on the module.
register_parameter(name, param)	Adds a parameter to the module.
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.

continues on next page

Table 70 – continued from previous page

to_empty(*, device)	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])	Sets gradients of all model parameters to zero.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Mapping[str, torch.Tensor])
dump_patches	This allows better BC support for load_state_dict().
lazy_buffer_names	
lazy_parameter_names	
lazy_parmeters_determined	Returns if all lazy parameters are determined.

2.5 ONNX

2.5.1 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.

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*, ***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** (*torch.nn.modules.module.Module*) –
- **args** (*Sequence[Any]*) –
- **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*, ***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** (*torch.nn.modules.module.Module*) –
- **args** (*Any*) –
- **out_dir** (*str*) –
- **user_meta** (*Optional[Mapping[str, Any]]*) –
- **kwargs** (*Any*) –

Return type Any

Warning: This function is not thread safe.

2.5.2 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.

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)
```

(continues on next page)

(continued from previous page)

```
...             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.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]

2.6 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	

2.6.1 pytorch_pfn_extras.dataset.SharedDataset

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

Dataset that caches the load samples in shared memory

Args

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

Methods

__init__(sm_size[, cache_type])
cache_item(idx, x)
is_cached(idx)
register_datapipe_as_function(function_name, ...)
register_function(function_name, function)

Attributes

functions

2.6.2 pytorch_pfn_extras.dataset.TabularDataset

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

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)]
```

__init__()

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.
register_datapipe_as_function(function_name, ...)	
register_function(function_name, function)	
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

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

2.6.3 pytorch_pfn_extras.dataset.ItemNotFoundException

exception pytorch_pfn_extras.dataset.**ItemNotFoundException**

2.7 Config

config.Config(config[, types])

2.7.1 pytorch_pfn_extras.config.Config

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

Parameters

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

Return type None

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

Parameters

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

Return type None

Methods

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

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

2.7.2 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]

2.7.3 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 *pytorch_pfn_extras.config.Config*

2.8 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.

2.8.1 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 torch.Tensor

2.8.2 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** (*torch.Tensor*) –

Return type Any

2.8.3 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, torch.Tensor]*) –

Return type Any

2.8.4 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** (*torch.dtype*) – PyTorch's dtype object.

Returns NumPy type object.

Return type Any

2.8.5 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 torch.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.

2.8.6 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[torch.cuda.streams.Stream]*) –

Return type Generator[None, None, None]

2.8.7 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

2.8.8 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

PYTHON MODULE INDEX

p

INDEX

Symbols

A

P

R

S

T

U

V