# PyTorch-Lightning Documentation

***Release 0.8.1***

**William Falcon et al.**

**Jun 19, 2020**

# START HERE


1 Quick Start 1

2 Introduction Guide 7

3 Fast Performance 35

4 Callbacks 39

5 LightningModule 49

6 Loggers 81

7 Metrics 99

8 Model Hooks 135

9 Trainer 141

10 16-bit training 169

11 Computing cluster (SLURM) 171

12 Child Modules 175

13 Debugging 177

14 Experiment Logging 181

15 Experiment Reporting 187

16 Early stopping 191

17 Fast Training 193

18 Hyperparameters 195

19 Learning Rate Finder 201

20 Multi-GPU training 205

21 Multiple Datasets 217

22 Saving and loading weights 219

23 Optimization 223

24 Performance and Bottleneck Profiler 227

25 Single GPU Training 233

26 Sequential Data 235

27 Training Tricks 237

28 Transfer Learning 241

29 TPU support 245

30 Test set 249

31 Contributor Covenant Code of Conduct 251

32 Contributing 253

33 How to become a core contributor 259

34 What does this PR do? 261

35 Before submitting 263

36 Pytorch Lightning Governance | Persons of interest 265

37 Indices and tables 267

Python Module Index 535

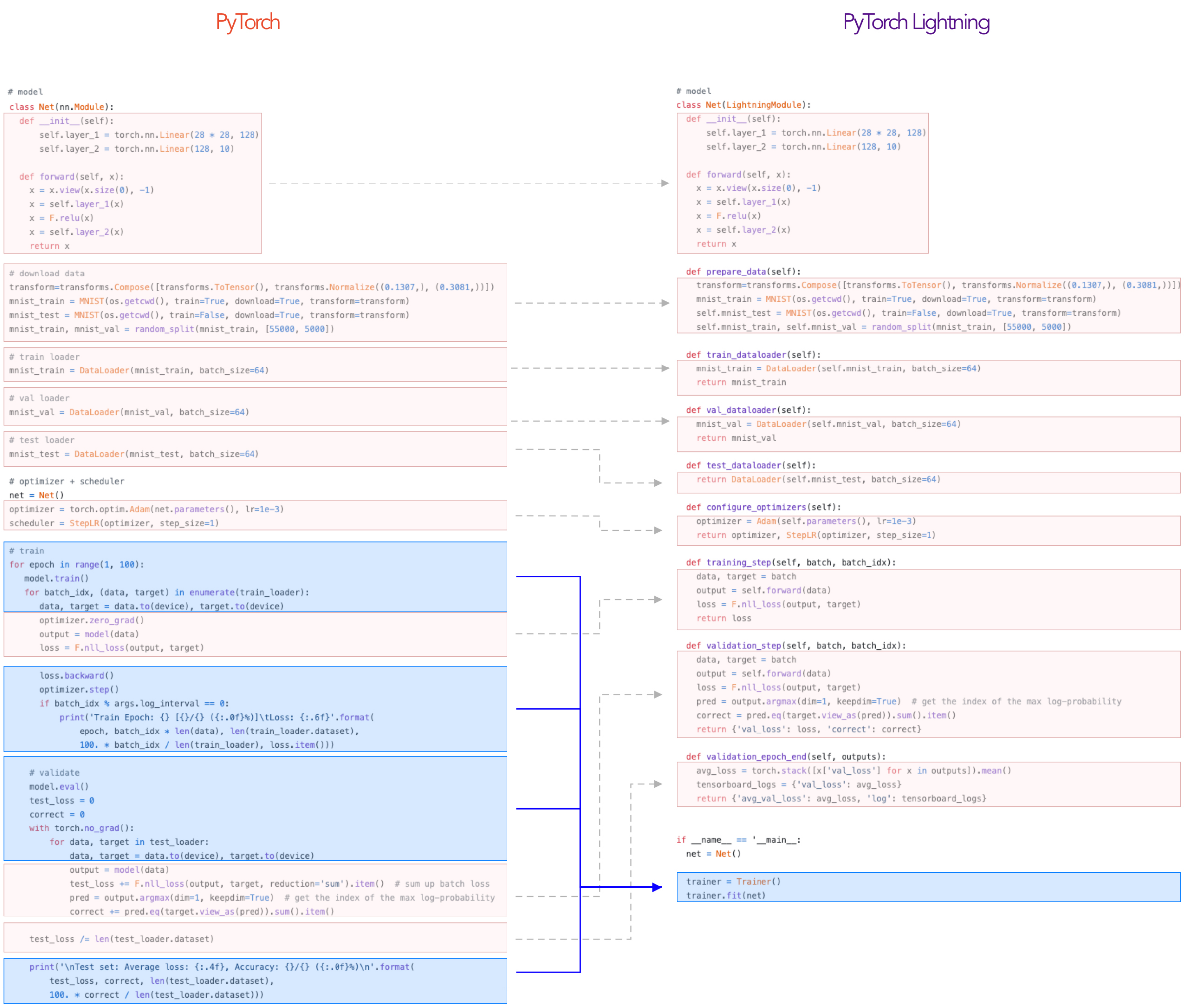
Index 537

# CHAPTER ONE

# QUICK START

PyTorch Lightning is nothing more than organized PyTorch code. Once you've organized it into a LightningModule, it automates most of the training for you.

To illustrate, here's the typical PyTorch project structure organized in a LightningModule.



## 1.1 Step 1: Define a LightningModule

```
import os

import torch
from torch.nn import functional as F
from torch.utils.data import DataLoader
from torchvision.datasets import MNIST
from torchvision import transforms
from pytorch_lightning.core.lightning import LightningModule

class LitModel(LightningModule):

    def __init__(self):
        super().__init__()
        self.l1 = torch.nn.Linear(28 * 28, 10)

    def forward(self, x):
        return torch.relu(self.l1(x.view(x.size(0), -1)))

    def training_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self(x)
        loss = F.cross_entropy(y_hat, y)
        tensorboard_logs = {'train_loss': loss}
        return {'loss': loss, 'log': tensorboard_logs}

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(), lr=0.001)

    def train_dataloader(self):
        dataset = MNIST(os.getcwd(), train=True, download=True, transform=transforms.
↪ToTensor())
        loader = DataLoader(dataset, batch_size=32, num_workers=4, shuffle=True)
        return loader
```

## 1.2 Step 2: Fit with a Trainer

```
from pytorch_lightning import Trainer

model = LitModel()

# most basic trainer, uses good defaults
trainer = Trainer(gpus=8, num_nodes=1)
trainer.fit(model)
```

Under the hood, lightning does (in high-level pseudocode):

```
model = LitModel()
train_dataloader = model.train_dataloader()
optimizer = model.configure_optimizers()

for epoch in epochs:
    train_outs = []
```

(continues on next page)

(continued from previous page)

```
    for batch in train_dataloader:
        loss = model.training_step(batch)
        loss.backward()
        train_outs.append(loss.detach())

        optimizer.step()
        optimizer.zero_grad()

    # optional for logging, etc...
    model.training_epoch_end(train_outs)
```

## 1.3 Validation loop

To also add a validation loop add the following functions

```
class LitModel(LightningModule):

    def validation_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self(x)
        return {'val_loss': F.cross_entropy(y_hat, y)}

    def validation_epoch_end(self, outputs):
        avg_loss = torch.stack([x['val_loss'] for x in outputs]).mean()
        tensorboard_logs = {'val_loss': avg_loss}
        return {'val_loss': avg_loss, 'log': tensorboard_logs}

    def val_dataloader(self):
        # TODO: do a real train/val split
        dataset = MNIST(os.getcwd(), train=False, download=True, transform=transforms.
↪ToTensor())
        loader = DataLoader(dataset, batch_size=32, num_workers=4)
        return loader
```

And now the trainer will call the validation loop automatically

```
# most basic trainer, uses good defaults
trainer = Trainer(gpus=8, num_nodes=1)
trainer.fit(model)
```

Under the hood in pseudocode, lightning does the following:

```
# ...
for batch in train_dataloader:
    loss = model.training_step()
    loss.backward()
    # ...

    if validate_at_some_point:
        model.eval()
        val_outs = []
        for val_batch in model.val_dataloader:
            val_out = model.validation_step(val_batch)
            val_outs.append(val_out)
```

(continues on next page)

(continued from previous page)

```
        model.validation_epoch_end(val_outs)
        model.train()
```

The beauty of Lightning is that it handles the details of when to validate, when to call .eval(), turning off gradients, detaching graphs, making sure you don't enable shuffle for val, etc...

**Note:** Lightning removes all the million details you need to remember during research

## 1.4 Test loop

You might also need a test loop

```
class LitModel(LightningModule):

    def test_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self(x)
        return {'test_loss': F.cross_entropy(y_hat, y)}

    def test_epoch_end(self, outputs):
        avg_loss = torch.stack([x['test_loss'] for x in outputs]).mean()
        tensorboard_logs = {'test_loss': avg_loss}
        return {'avg_test_loss': avg_loss, 'log': tensorboard_logs}

    def test_dataloader(self):
        # TODO: do a real train/val split
        dataset = MNIST(os.getcwd(), train=False, download=True, transform=transforms.
↪ToTensor())
        loader = DataLoader(dataset, batch_size=32, num_workers=4)
        return loader
```

However, this time you need to specifically call test (this is done so you don't use the test set by mistake)

```
# OPTION 1:
# test after fit
trainer.fit(model)
trainer.test()

# OPTION 2:
# test after loading weights
model = LitModel.load_from_checkpoint(PATH)
trainer = Trainer(tpu_cores=1)
trainer.test()
```

Again, under the hood, lightning does the following in (pseudocode):

```
model.eval()
test_outs = []
for test_batch in model.test_dataloader:
    test_out = model.test_step(val_batch)
    test_outs.append(test_out)
```

(continues on next page)

(continued from previous page)

```
model.test_epoch_end(test_outs)
```

## 1.5 Datasets

If you don't want to define the datasets as part of the LightningModule, just pass them into fit instead.

```
# pass in datasets if you want.
train_dataloader = DataLoader(dataset, batch_size=32, num_workers=4)
val_dataloader, test_dataloader = ...

trainer = Trainer(gpus=8, num_nodes=1)
trainer.fit(model, train_dataloader, val_dataloader)

trainer.test(test_dataloader=test_dataloader)
```

The advantage of this method is the ability to reuse models for different datasets. The disadvantage is that for research it makes readability and reproducibility more difficult. This is why we recommend to define the datasets in the LightningModule if you're doing research, but use the method above for production models or for prediction tasks.

## 1.6 Why do you need Lightning?

Notice the code above has nothing about .cuda() or 16-bit or early stopping or logging, etc. . . This is where Lightning adds a ton of value.

Without changing a SINGLE line of your code, you can now do the following with the above code

```
# train on TPUs using 16 bit precision with early stopping
# using only half the training data and checking validation every quarter of a␣
↪training epoch
trainer = Trainer(
    tpu_cores=8,
    precision=16,
    early_stop_checkpoint=True,
    limit_train_batches=0.5,
    val_check_interval=0.25
)

# train on 256 GPUs
trainer = Trainer(
    gpus=8,
    num_nodes=32
)

# train on 1024 CPUs across 128 machines
trainer = Trainer(
    num_processes=8,
    num_nodes=128
)
```

And the best part is that your code is STILL just PyTorch. . . meaning you can do anything you would normally do.

```
model = LitModel()
model.eval()

y_hat = model(x)

model.anything_you_can_do_with_pytorch()
```

## 1.7 Summary

In short, by refactoring your PyTorch code:

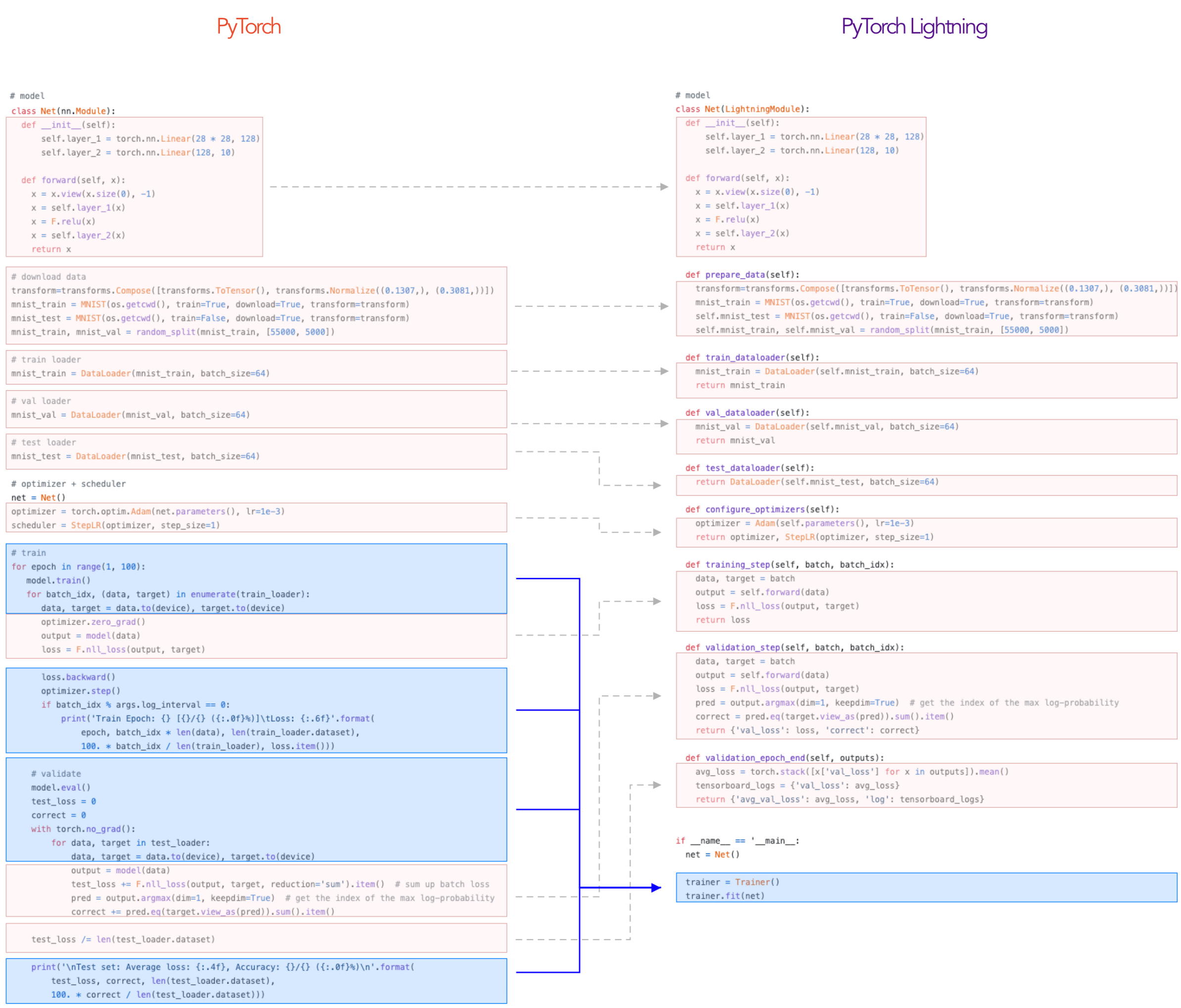
1. You STILL keep pure PyTorch.
2. You DON'T lose any flexibility.
3. You can get rid of all of your boilerplate.
4. You make your code generalizable to any hardware.
5. Your code is now readable and easier to reproduce (ie: you help with the reproducibility crisis).
6. Your LightningModule is still just a pure PyTorch module.

# CHAPTER TWO

# INTRODUCTION GUIDE

PyTorch Lightning provides a very simple template for organizing your PyTorch code. Once you've organized it into a LightningModule, it automates most of the training for you.

To illustrate, here's the typical PyTorch project structure organized in a LightningModule.




As your project grows in complexity with things like 16-bit precision, distributed training, etc... the part in blue quickly becomes onerous and starts distracting from the core research code.

## 2.1 Goal of this guide

This guide walks through the major parts of the library to help you understand what each parts does. But at the end of the day, you write the same PyTorch code… just organize it into the LightningModule template which means you keep ALL the flexibility without having to deal with any of the boilerplate code

To show how Lightning works, we'll start with an MNIST classifier. We'll end showing how to use inheritance to very quickly create an AutoEncoder.

**Note:** Any DL/ML PyTorch project fits into the Lightning structure. Here we just focus on 3 types of research to illustrate.

---

## 2.2 Installing Lightning

Lightning is trivial to install.

```
conda activate my_env
pip install pytorch-lightning
```

Or without conda environments, anywhere you can use pip.

```
pip install pytorch-lightning
```

Or with conda

```
conda install pytorch-lightning -c conda-forge
```

---

## 2.3 Lightning Philosophy

Lightning factors DL/ML code into three types:

- Research code
- Engineering code
- Non-essential code

### 2.3.1 Research code

In the MNIST generation example, the research code would be the particular system and how it's trained (ie: A GAN or VAE). In Lightning, this code is abstracted out by the *LightningModule*.

```
l1 = nn.Linear(...)
l2 = nn.Linear(...)
decoder = Decoder()

x1 = l1(x)
```

(continues on next page)

(continued from previous page)

```
x2 = l2(x2)
out = decoder(features, x)

loss = perceptual_loss(x1, x2, x) + CE(out, x)
```

### 2.3.2 Engineering code

The Engineering code is all the code related to training this system. Things such as early stopping, distribution over GPUs, 16-bit precision, etc. This is normally code that is THE SAME across most projects.

In Lightning, this code is abstracted out by the *Trainer*.

```
model.cuda(0)
x = x.cuda(0)

distributed = DistributedParallel(model)

with gpu_zero:
    download_data()

dist.barrier()
```

### 2.3.3 Non-essential code

This is code that helps the research but isn't relevant to the research code. Some examples might be: 1. Inspect gradients 2. Log to tensorboard.

In Lightning this code is abstracted out by *Callbacks*.

```
# log samples
z = Q.rsample()
generated = decoder(z)
self.experiment.log('images', generated)
```

---

## 2.4 Elements of a research project

Every research project requires the same core ingredients:

1. A model
2. Train/val/test data
3. Optimizer(s)
4. Training step computations
5. Validation step computations
6. Test step computations

### 2.4.1 The Model

The LightningModule provides the structure on how to organize these 5 ingredients.

Let's first start with the model. In this case we'll design a 3-layer neural network.

```
import torch
from torch.nn import functional as F
from torch import nn
from pytorch_lightning.core.lightning import LightningModule

class LitMNIST(LightningModule):

  def __init__(self):
    super().__init__()

    # mnist images are (1, 28, 28) (channels, width, height)
    self.layer_1 = torch.nn.Linear(28 * 28, 128)
    self.layer_2 = torch.nn.Linear(128, 256)
    self.layer_3 = torch.nn.Linear(256, 10)

  def forward(self, x):
    batch_size, channels, width, height = x.size()

    # (b, 1, 28, 28) -> (b, 1*28*28)
    x = x.view(batch_size, -1)

    # layer 1
    x = self.layer_1(x)
    x = torch.relu(x)

    # layer 2
    x = self.layer_2(x)
    x = torch.relu(x)

    # layer 3
    x = self.layer_3(x)

    # probability distribution over labels
    x = torch.log_softmax(x, dim=1)

    return x
```

Notice this is a *LightningModule* instead of a *torch.nn.Module*. A LightningModule is equivalent to a PyTorch Module except it has added functionality. However, you can use it EXACTLY the same as you would a PyTorch Module.

```
net = LitMNIST()
x = torch.Tensor(1, 1, 28, 28)
out = net(x)
```

Out:

```
torch.Size([1, 10])
```

### 2.4.2 Data

The Lightning Module organizes your dataloaders and data processing as well. Here's the PyTorch code for loading MNIST

```
from torch.utils.data import DataLoader, random_split
from torchvision.datasets import MNIST
import os
from torchvision import datasets, transforms

# transforms
# prepare transforms standard to MNIST
transform=transforms.Compose([transforms.ToTensor(),
                              transforms.Normalize((0.1307,), (0.3081,))])

# data
mnist_train = MNIST(os.getcwd(), train=True, download=True)
mnist_train = DataLoader(mnist_train, batch_size=64)
```

When using PyTorch Lightning, we use the exact same code except we organize it into the LightningModule

```
from torch.utils.data import DataLoader, random_split
from torchvision.datasets import MNIST
import os
from torchvision import datasets, transforms

class LitMNIST(LightningModule):

    def train_dataloader(self):
        transform=transforms.Compose([transforms.ToTensor(),
                                      transforms.Normalize((0.1307,), (0.3081,))])
        mnist_train = MNIST(os.getcwd(), train=True, download=False,
                            transform=transform)
        return DataLoader(mnist_train, batch_size=64)
```

Notice the code is exactly the same, except now the training dataloading has been organized by the LightningModule under the *train_dataloader* method. This is great because if you run into a project that uses Lightning and want to figure out how they prepare their training data you can just look in the *train_dataloader* method.

Usually though, we want to separate the things that write to disk in data-processing from things like transforms which happen in memory. This is only relevant in multi-GPU or TPU training.

```
class LitMNIST(LightningModule):

    def prepare_data(self):
        # download only (not called on every GPU, just the root GPU per node)
        MNIST(os.getcwd(), train=True, download=True)

    def train_dataloader(self):
        # no download, just transform
        transform=transforms.Compose([transforms.ToTensor(),
                                      transforms.Normalize((0.1307,), (0.3081,))])
        mnist_train = MNIST(os.getcwd(), train=True, download=False,
                            transform=transform)
        return DataLoader(mnist_train, batch_size=64)
```

Doing it in the *prepare_data* method ensures that when you have multiple GPUs you won't overwrite the data. This is a contrived example but it gets more complicated with things like NLP or Imagenet.

*prepare_data* gets called on the *LOCAL_RANK=0* GPU per node. If your nodes share a file system, set *Trainer(prepare_data_per_node=False)* and it will be code from node=0, gpu=0 only.

In general fill these methods with the following:

```
class LitMNIST(LightningModule):

    def prepare_data(self):
        # stuff here is done once at the very beginning of training
        # before any distributed training starts

        # download stuff
        # save to disk
        # etc...
        ...

    def train_dataloader(self):
        # data transforms
        # dataset creation
        # return a DataLoader
        ...
```

### 2.4.3 Models defined by data

Sometimes a model needs to know about the data to be built (ie: number of classes or vocab size). In this case we recommend the following:

1. use *prepare_data* to download and process the dataset.
2. use *setup* to do splits, and build your model internals

Example:

```
class LitMNIST(LightningModule):

    def __init__(self):
        self.l1 = None

    def prepare_data(self):
        download_data()
        tokenize()

    def setup(self, step):
        # step is either 'fit' or 'test' 90% of the time not relevant
        data = load_data()
        num_classes = data.classes
        self.l1 = nn.Linear(..., num_classes)
```

### 2.4.4 Optimizer

Next we choose what optimizer to use for training our system. In PyTorch we do it as follows:

```
from torch.optim import Adam
optimizer = Adam(LitMNIST().parameters(), lr=1e-3)
```

In Lightning we do the same but organize it under the configure_optimizers method.

```
class LitMNIST(LightningModule):

    def configure_optimizers(self):
        return Adam(self.parameters(), lr=1e-3)
```

**Note:** The LightningModule itself has the parameters, so pass in self.parameters()

However, if you have multiple optimizers use the matching parameters

```
class LitMNIST(LightningModule):

    def configure_optimizers(self):
        return Adam(self.generator(), lr=1e-3), Adam(self.discriminator(), lr=1e-3)
```

### 2.4.5 Training step

The training step is what happens inside the training loop.

```
for epoch in epochs:
    for batch in data:
        # TRAINING STEP
        # ....
        # TRAINING STEP
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

In the case of MNIST we do the following

```
for epoch in epochs:
    for batch in data:
        # TRAINING STEP START
        x, y = batch
        logits = model(x)
        loss = F.nll_loss(logits, y)
        # TRAINING STEP END

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

In Lightning, everything that is in the training step gets organized under the *training_step* function in the Lightning-Module

```python
class LitMNIST(LightningModule):

    def training_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = F.nll_loss(logits, y)
        return {'loss': loss}
        # return loss (also works)
```

Again, this is the same PyTorch code except that it has been organized by the LightningModule. This code is not restricted which means it can be as complicated as a full seq-2-seq, RL loop, GAN, etc...

---

## 2.5 Training

So far we defined 4 key ingredients in pure PyTorch but organized the code inside the LightningModule.

1. Model.
2. Training data.
3. Optimizer.
4. What happens in the training loop.

For clarity, we'll recall that the full LightningModule now looks like this.

```python
class LitMNIST(LightningModule):
    def __init__(self):
        super().__init__()
        self.layer_1 = torch.nn.Linear(28 * 28, 128)
        self.layer_2 = torch.nn.Linear(128, 256)
        self.layer_3 = torch.nn.Linear(256, 10)

    def forward(self, x):
        batch_size, channels, width, height = x.size()
        x = x.view(batch_size, -1)
        x = self.layer_1(x)
        x = torch.relu(x)
        x = self.layer_2(x)
        x = torch.relu(x)
        x = self.layer_3(x)
        x = torch.log_softmax(x, dim=1)
        return x

    def train_dataloader(self):
        transform=transforms.Compose([transforms.ToTensor(),
                                      transforms.Normalize((0.1307,), (0.3081,))])
        mnist_train = MNIST(os.getcwd(), train=True, download=False,
↪transform=transform)
        return DataLoader(mnist_train, batch_size=64)

    def configure_optimizers(self):
        return Adam(self.parameters(), lr=1e-3)

    def training_step(self, batch, batch_idx):
```

(continues on next page)

(continued from previous page)

```
        x, y = batch
        logits = self(x)
        loss = F.nll_loss(logits, y)

        # add logging
        logs = {'loss': loss}
        return {'loss': loss, 'log': logs}
```

Again, this is the same PyTorch code, except that it's organized by the LightningModule. This organization now lets us train this model

### 2.5.1 Train on CPU

```
from pytorch_lightning import Trainer

model = LitMNIST()
trainer = Trainer()
trainer.fit(model)
```

You should see the following weights summary and progress bar

```
  | Name    | Type   | Params
--------------------------------
0 | layer_1 | Linear | 100 K
1 | layer_2 | Linear | 33 K
2 | layer_3 | Linear | 2 K
Epoch 1: 27%  250/938 [00:08<00:22, 30.10it/s, loss=0.353, v_num=2]
```

### 2.5.2 Logging

When we added the *log* key in the return dictionary it went into the built in tensorboard logger. But you could have also logged by calling:

```
def training_step(self, batch, batch_idx):
    # ...
    loss = ...
    self.logger.summary.scalar('loss', loss)
```

Which will generate automatic tensorboard logs.

But you can also use any of the *number of other loggers* we support.

### 2.5.3 GPU training

But the beauty is all the magic you can do with the trainer flags. For instance, to run this model on a GPU:

```
model = LitMNIST()
trainer = Trainer(gpus=1)
trainer.fit(model)
```

```
[31] # Start tensorboard.
     %load_ext tensorboard
     %tensorboard --logdir lightning_logs/
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```


```
INFO:root:GPU available: True, used: True
INFO:root:VISIBLE GPUS: 0
INFO:root:
  | Name    | Type   | Params
-------------------------------
0 | layer_1 | Linear | 100 K
1 | layer_2 | Linear | 33 K
2 | layer_3 | Linear | 2 K
```

Epoch 1: 53% 500/938 [00:07<00:07, 61.53it/s, loss=0.206, v_num=3]

### 2.5.4 Multi-GPU training

Or you can also train on multiple GPUs.

```
model = LitMNIST()
trainer = Trainer(gpus=8)
trainer.fit(model)
```

Or multiple nodes

```
# (32 GPUs)
model = LitMNIST()
trainer = Trainer(gpus=8, num_nodes=4, distributed_backend='ddp')
trainer.fit(model)
```

Refer to the *distributed computing guide for more details*.

### 2.5.5 TPUs

Did you know you can use PyTorch on TPUs? It's very hard to do, but we've worked with the xla team to use their awesome library to get this to work out of the box!

Let's train on Colab (full demo available here)

First, change the runtime to TPU (and reinstall lightning).

Next, install the required xla library (adds support for PyTorch on TPUs)

> !curl https://raw.githubusercontent.com/pytorch/xla/master/contrib/scripts/env-setup.py -o pytorch-xla-env-setup.py !python pytorch-xla-env-setup.py –version nightly –apt-packages libomp5 libopenblas-dev

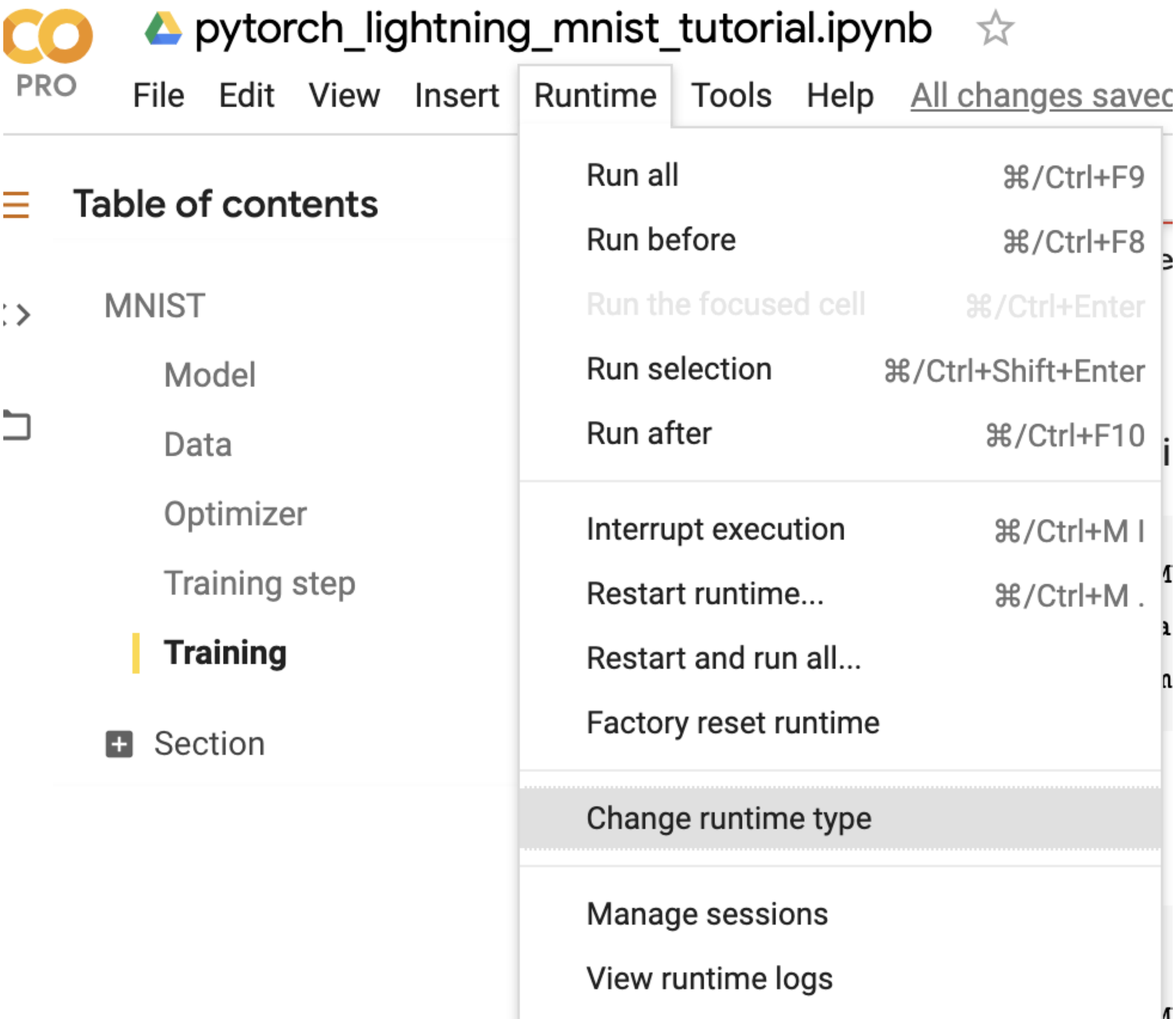
In distributed training (multiple GPUs and multiple TPU cores) each GPU or TPU core will run a copy of this program. This means that without taking any care you will download the dataset N times which will cause all sorts of issues.

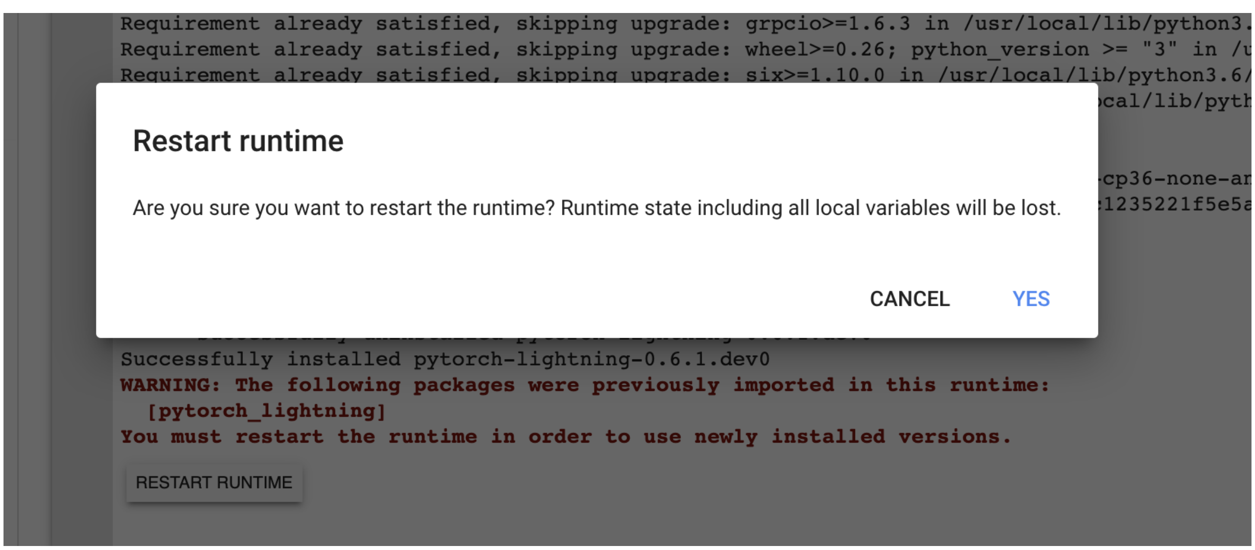
To solve this problem, move the download code to the *prepare_data* method in the LightningModule. In this method we do all the preparation we need to do once (instead of on every gpu).

*prepare_data* can be called in two ways, once per node or only on the root node (*Trainer(prepare_data_per_node=False)*).

```
class LitMNIST(LightningModule):
    def prepare_data(self):
        # download only
        MNIST(os.getcwd(), train=True, download=True, transform=transforms.ToTensor())
        MNIST(os.getcwd(), train=False, download=True, transform=transforms.
↪ToTensor())

    def setup(self, stage):
        # transform
        transform=transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.
↪1307,), (0.3081,))])
        MNIST(os.getcwd(), train=True, download=False, transform=transform)
        MNIST(os.getcwd(), train=False, download=False, transform=transform)

        # train/val split
        mnist_train, mnist_val = random_split(mnist_train, [55000, 5000])

        # assign to use in dataloaders
```

(continues on next page)


pytorch_lightning_mnist_tutorial.ipynb
PRO
File Edit View Insert Runtime Tools Help All changes saved
Table of contents
MNIST
Model
Data
Optimizer
Training step
Training
Section
Run all ⌘/Ctrl+F9
Run before ⌘/Ctrl+F8
Run the focused cell ⌘/Ctrl+Enter
Run selection ⌘/Ctrl+Shift+Enter
Run after ⌘/Ctrl+F10
Interrupt execution ⌘/Ctrl+M I
Restart runtime... ⌘/Ctrl+M .
Restart and run all...
Factory reset runtime
Change runtime type
Manage sessions
View runtime logs


Requirement already satisfied, skipping upgrade: grpcio>=1.6.3 in /usr/local/lib/python3.
Requirement already satisfied, skipping upgrade: wheel>=0.26; python_version >= "3" in /u
Requirement already satisfied, skipping upgrade: six>=1.10.0 in /usr/local/lib/python3.6/
Restart runtime
Are you sure you want to restart the runtime? Runtime state including all local variables will be lost.
CANCEL
YES
Successfully installed pytorch-lightning-0.6.1.dev0
WARNING: The following packages were previously imported in this runtime:
[pytorch_lightning]
You must restart the runtime in order to use newly installed versions.
RESTART RUNTIME

(continued from previous page)

```
        self.train_dataset = mnist_train
        self.val_dataset = mnist_val
        self.test_dataset = mnist_test

    def train_dataloader(self):
        return DataLoader(self.train_dataset, batch_size=64)

    def val_dataloader(self):
        return DataLoader(self.val_dataset, batch_size=64)

    def test_dataloader(self):
        return DataLoader(self.test_dataset, batch_size=64)
```

The *prepare_data* method is also a good place to do any data processing that needs to be done only once (ie: download or tokenize, etc. . . ).

**Note:** Lightning inserts the correct DistributedSampler for distributed training. No need to add yourself!

Now we can train the LightningModule on a TPU without doing anything else!

```
model = LitMNIST()
trainer = Trainer(tpu_cores=8)
trainer.fit(model)
```

You'll now see the TPU cores booting up.

```
INFO:root:training on 8 TPU cores
INFO:root:INIT TPU local core: 0, global rank: 0
INFO:root:INIT TPU local core: 3, global rank: 3
INFO:root:INIT TPU local core: 1, global rank: 1
```

Notice the epoch is MUCH faster!

```
INFO:root:
  | Name    | Type   | Params
-------------------------------
0 | layer_1 | Linear | 100 K
1 | layer_2 | Linear | 33 K
2 | layer_3 | Linear | 2 K
Using downloaded and verified file: /content/MNIST/raw/train-images-idx3-ubyte.gz
Extracting /content/MNIST/raw/train-images-idx3-ubyte.gz to /content/MNIST/raw
Using downloaded and verified file: /content/MNIST/raw/train-labels-idx1-ubyte.gz
Extracting /content/MNIST/raw/train-labels-idx1-ubyte.gz to /content/MNIST/raw
Using downloaded and verified file: /content/MNIST/raw/t10k-images-idx3-ubyte.gz
Extracting /content/MNIST/raw/t10k-images-idx3-ubyte.gz to /content/MNIST/raw
Using downloaded and verified file: /content/MNIST/raw/t10k-labels-idx1-ubyte.gz
Extracting /content/MNIST/raw/t10k-labels-idx1-ubyte.gz to /content/MNIST/raw
Processing...
Done!
Epoch 6: 42%   50/118 [00:00<00:01, 62.22it/s, loss=0.067, v_num=10]
```

## 2.6 Hyperparameters

Lightning has utilities to interact seamlessly with the command line ArgumentParser and plays well with the hyperparameter optimization framework of your choice.

---

### 2.6.1 ArgumentParser

Lightning is designed to augment a lot of the functionality of the built-in Python ArgumentParser

```
from argparse import ArgumentParser
parser = ArgumentParser()
parser.add_argument('--layer_1_dim', type=int, default=128)
args = parser.parse_args()
```

This allows you to call your program like so:

```
python trainer.py --layer_1_dim 64
```

---

### 2.6.2 Argparser Best Practices

It is best practice to layer your arguments in three sections.

1. Trainer args (gpus, num_nodes, etc. . . )
2. Model specific arguments (layer_dim, num_layers, learning_rate, etc. . . )
3. Program arguments (data_path, cluster_email, etc. . . )

We can do this as follows. First, in your LightningModule, define the arguments specific to that module. Remember that data splits or data paths may also be specific to a module (ie: if your project has a model that trains on Imagenet and another on CIFAR-10).

```
class LitModel(LightningModule):

    @staticmethod
    def add_model_specific_args(parent_parser):
        parser = ArgumentParser(parents=[parent_parser], add_help=False)
        parser.add_argument('--encoder_layers', type=int, default=12)
        parser.add_argument('--data_path', type=str, default='/some/path')
        return parser
```

Now in your main trainer file, add the Trainer args, the program args, and add the model args

```
# ----------------
# trainer_main.py
# ----------------
from argparse import ArgumentParser
parser = ArgumentParser()

# add PROGRAM level args
parser.add_argument('--conda_env', type=str, default='some_name')
```

(continues on next page)

(continued from previous page)

```
parser.add_argument('--notification_email', type=str, default='will@email.com')

# add model specific args
parser = LitModel.add_model_specific_args(parser)

# add all the available trainer options to argparse
# ie: now --gpus --num_nodes ... --fast_dev_run all work in the cli
parser = Trainer.add_argparse_args(parser)

args = parser.parse_args()
```

Now you can call run your program like so

```
python trainer_main.py --gpus 2 --num_nodes 2 --conda_env 'my_env' --encoder_layers 12
```

Finally, make sure to start the training like so:

```
# init the trainer like this
trainer = Trainer.from_argparse_args(args, early_stopping_callback=...)

# NOT like this
trainer = Trainer(gpus=hparams.gpus, ...)

# init the model with Namespace directly
model = LitModel(args)

# or init the model with all the key-value pairs
dict_args = vars(args)
model = LitModel(**dict_args)
```

---

### 2.6.3 LightningModule hyperparameters

Often times we train many versions of a model. You might share that model or come back to it a few months later at which point it is very useful to know how that model was trained (ie: what learning_rate, neural network, etc. . . ).

Lightning has a few ways of saving that information for you in checkpoints and yaml files. The goal here is to improve readability and reproducibility

1. The first way is to ask lightning to save the values anything in the __init__ for you to the checkpoint. This also makes those values available via *self.hparams*.

```
class LitMNIST(LightningModule):

    def __init__(self, layer_1_dim=128, learning_rate=1e-2, **kwargs):
        super().__init__()
        # call this to save (layer_1_dim=128, learning_rate=1e-4) to the checkpoint
        self.save_hyperparameters()

        # equivalent
        self.save_hyperparameters(['layer_1_dim', 'learning_rate'])

        # this now works
        self.hparams.layer_1_dim
```

2. Sometimes your init might have objects or other parameters you might not want to save. In that case, choose only a few

```
class LitMNIST(LightningModule):

    def __init__(self, loss_fx, generator_network, layer_1_dim=128 **kwargs):
        super().__init__()
        self.layer_1_dim = layer_1_dim
        self.loss_fx = loss_fx

        # call this to save (layer_1_dim=128) to the checkpoint
        self.save_hyperparameters(['layer_1_dim'])

# to load specify the other args
model = LitMNIST.load_from_checkpoint(PATH, loss_fx=torch.nn.SomeOtherLoss, generator_
↪network=MyGenerator())
```

3. Assign to *self.hparams*. Anything assigned to *self.hparams* will also be saved automatically

```
# using a argparse.Namespace
class LitMNIST(LightningModule):

    def __init__(self, hparams, *args, **kwargs):
        super().__init__()
        self.hparams = hparams

        self.layer_1 = torch.nn.Linear(28 * 28, self.hparams.layer_1_dim)
        self.layer_2 = torch.nn.Linear(self.hparams.layer_1_dim, self.hparams.layer_2_
↪dim)
        self.layer_3 = torch.nn.Linear(self.hparams.layer_2_dim, 10)

    def train_dataloader(self):
        return DataLoader(mnist_train, batch_size=self.hparams.batch_size)
```

4. You can also save full objects such as *dict* or *Namespace* to the checkpoint.

```
# using a argparse.Namespace
class LitMNIST(LightningModule):

    def __init__(self, conf, *args, **kwargs):
        super().__init__()
        self.hparams = conf

        # equivalent
        self.save_hyperparameters(conf)

        self.layer_1 = torch.nn.Linear(28 * 28, self.hparams.layer_1_dim)
        self.layer_2 = torch.nn.Linear(self.hparams.layer_1_dim, self.hparams.layer_2_
↪dim)
        self.layer_3 = torch.nn.Linear(self.hparams.layer_2_dim, 10)

conf = OmegaConf.create(...)
model = LitMNIST(conf)

# this works
model.hparams.anything
```

### 2.6.4 Trainer args

To recap, add ALL possible trainer flags to the argparser and init the Trainer this way

```
parser = ArgumentParser()
parser = Trainer.add_argparse_args(parser)
hparams = parser.parse_args()

trainer = Trainer.from_argparse_args(hparams)

# or if you need to pass in callbacks
trainer = Trainer.from_argparse_args(hparams, checkpoint_callback=..., callbacks=[...
↪])
```

---

### 2.6.5 Multiple Lightning Modules

We often have multiple Lightning Modules where each one has different arguments. Instead of polluting the main.py file, the LightningModule lets you define arguments for each one.

```
class LitMNIST(LightningModule):

    def __init__(self, layer_1_dim, **kwargs):
        super().__init__()
        self.layer_1 = torch.nn.Linear(28 * 28, layer_1_dim)

    @staticmethod
    def add_model_specific_args(parent_parser):
        parser = ArgumentParser(parents=[parent_parser], add_help=False)
        parser.add_argument('--layer_1_dim', type=int, default=128)
        return parser
```

```
class GoodGAN(LightningModule):

    def __init__(self, encoder_layers, **kwargs):
        super().__init__()
        self.encoder = Encoder(layers=encoder_layers)

    @staticmethod
    def add_model_specific_args(parent_parser):
        parser = ArgumentParser(parents=[parent_parser], add_help=False)
        parser.add_argument('--encoder_layers', type=int, default=12)
        return parser
```

Now we can allow each model to inject the arguments it needs in the `main.py`

```
def main(args):
    dict_args = vars(args)

    # pick model
    if args.model_name == 'gan':
        model = GoodGAN(**dict_args)
    elif args.model_name == 'mnist':
        model = LitMNIST(**dict_args)
```

(continues on next page)

(continued from previous page)

```
    trainer = Trainer.from_argparse_args(args)
    trainer.fit(model)

if __name__ == '__main__':
    parser = ArgumentParser()
    parser = Trainer.add_argparse_args(parser)

    # figure out which model to use
    parser.add_argument('--model_name', type=str, default='gan', help='gan or mnist')

    # THIS LINE IS KEY TO PULL THE MODEL NAME
    temp_args, _ = parser.parse_known_args()

    # let the model add what it wants
    if temp_args.model_name == 'gan':
        parser = GoodGAN.add_model_specific_args(parser)
    elif temp_args.model_name == 'mnist':
        parser = LitMNIST.add_model_specific_args(parser)

    args = parser.parse_args()

    # train
    main(args)
```

and now we can train MNIST or the GAN using the command line interface!

```
$ python main.py --model_name gan --encoder_layers 24
$ python main.py --model_name mnist --layer_1_dim 128
```

---

### 2.6.6 Hyperparameter Optimization

Lightning is fully compatible with the hyperparameter optimization libraries! Here are some useful ones:

- Hydra
- Optuna

---

## 2.7 Validating

For most cases, we stop training the model when the performance on a validation split of the data reaches a minimum.

Just like the *training_step*, we can define a *validation_step* to check whatever metrics we care about, generate samples or add more to our logs.

```
for epoch in epochs:
    for batch in data:
        # ...
        # train

    # validate
```

(continues on next page)

(continued from previous page)

```
    outputs = []
    for batch in val_data:
        x, y = batch                        # validation_step
        y_hat = model(x)                    # validation_step
        loss = loss(y_hat, x)               # validation_step
        outputs.append({'val_loss': loss})  # validation_step

    full_loss = outputs.mean()              # validation_epoch_end
```

Since the *validation_step* processes a single batch, in Lightning we also have a *validation_epoch_end* method which allows you to compute statistics on the full dataset after an epoch of validation data and not just the batch.

In addition, we define a *val_dataloader* method which tells the trainer what data to use for validation. Notice we split the train split of MNIST into train, validation. We also have to make sure to do the sample split in the *train_dataloader* method.

```
class LitMNIST(LightningModule):
    def validation_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = F.nll_loss(logits, y)
        return {'val_loss': loss}

    def validation_epoch_end(self, outputs):
        avg_loss = torch.stack([x['val_loss'] for x in outputs]).mean()
        tensorboard_logs = {'val_loss': avg_loss}
        return {'val_loss': avg_loss, 'log': tensorboard_logs}

    def val_dataloader(self):
        transform=transforms.Compose([transforms.ToTensor(),
                                      transforms.Normalize((0.1307,), (0.3081,))])
        mnist_train = MNIST(os.getcwd(), train=True, download=False,
                            transform=transform)
        _, mnist_val = random_split(mnist_train, [55000, 5000])
        mnist_val = DataLoader(mnist_val, batch_size=64)
        return mnist_val
```

Again, we've just organized the regular PyTorch code into two steps, the *validation_step* method which operates on a single batch and the *validation_epoch_end* method to compute statistics on all batches.

If you have these methods defined, Lightning will call them automatically. Now we can train while checking the validation set.

```
from pytorch_lightning import Trainer

model = LitMNIST()
trainer = Trainer(tpu_cores=8)
trainer.fit(model)
```

You may have noticed the words *Validation sanity check* logged. This is because Lightning runs 5 batches of validation before starting to train. This is a kind of unit test to make sure that if you have a bug in the validation loop, you won't need to potentially wait a full epoch to find out.

**Note:** Lightning disables gradients, puts model in eval mode and does everything needed for validation.

## 2.8 Testing

Once our research is done and we're about to publish or deploy a model, we normally want to figure out how it will generalize in the "real world." For this, we use a held-out split of the data for testing.

Just like the validation loop, we define exactly the same steps for testing:

- test_step
- test_epoch_end
- test_dataloader

```
class LitMNIST(LightningModule):
    def test_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = F.nll_loss(logits, y)
        return {'val_loss': loss}

    def test_epoch_end(self, outputs):
        avg_loss = torch.stack([x['val_loss'] for x in outputs]).mean()
        tensorboard_logs = {'val_loss': avg_loss}
        return {'val_loss': avg_loss, 'log': tensorboard_logs}

    def test_dataloader(self):
        transform=transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.
↪1307,), (0.3081,))])
        mnist_train = MNIST(os.getcwd(), train=False, download=False,␣
↪transform=transform)
        _, mnist_val = random_split(mnist_train, [55000, 5000])
        mnist_val = DataLoader(mnist_val, batch_size=64)
        return mnist_val
```

However, to make sure the test set isn't used inadvertently, Lightning has a separate API to run tests. Once you train your model simply call *.test()*.

```
from pytorch_lightning import Trainer

model = LitMNIST()
trainer = Trainer(tpu_cores=8)
trainer.fit(model)

# run test set
trainer.test()
```

Out:

```
--------------------------------------------------------------
TEST RESULTS
{'test_loss': tensor(1.1703, device='cuda:0')}
--------------------------------------------------------------
```

You can also run the test from a saved lightning model

```
model = LitMNIST.load_from_checkpoint(PATH)
trainer = Trainer(tpu_cores=8)
trainer.test(model)
```

**Note:** Lightning disables gradients, puts model in eval mode and does everything needed for testing.

**Warning:** .test() is not stable yet on TPUs. We're working on getting around the multiprocessing challenges.

## 2.9 Predicting

Again, a LightningModule is exactly the same as a PyTorch module. This means you can load it and use it for prediction.

```
model = LitMNIST.load_from_checkpoint(PATH)
x = torch.Tensor(1, 1, 28, 28)
out = model(x)
```

On the surface, it looks like *forward* and *training_step* are similar. Generally, we want to make sure that what we want the model to do is what happens in the *forward*. whereas the *training_step* likely calls forward from within it.

```
class MNISTClassifier(LightningModule):

    def forward(self, x):
        batch_size, channels, width, height = x.size()
        x = x.view(batch_size, -1)
        x = self.layer_1(x)
        x = torch.relu(x)
        x = self.layer_2(x)
        x = torch.relu(x)
        x = self.layer_3(x)
        x = torch.log_softmax(x, dim=1)
        return x

    def training_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = F.nll_loss(logits, y)
        return loss
```

```
model = MNISTClassifier()
x = mnist_image()
logits = model(x)
```

In this case, we've set this LightningModel to predict logits. But we could also have it predict feature maps:

```
class MNISTRepresentator(LightningModule):

    def forward(self, x):
        batch_size, channels, width, height = x.size()
        x = x.view(batch_size, -1)
        x = self.layer_1(x)
        x1 = torch.relu(x)
        x = self.layer_2(x1)
```

(continues on next page)

(continued from previous page)

```
        x2 = torch.relu(x)
        x3 = self.layer_3(x2)
        return [x, x1, x2, x3]

    def training_step(self, batch, batch_idx):
        x, y = batch
        out, l1_feats, l2_feats, l3_feats = self(x)
        logits = torch.log_softmax(out, dim=1)
        ce_loss = F.nll_loss(logits, y)
        loss = perceptual_loss(l1_feats, l2_feats, l3_feats) + ce_loss
        return loss
```

```
model = MNISTRepresentator.load_from_checkpoint(PATH)
x = mnist_image()
feature_maps = model(x)
```

Or maybe we have a model that we use to do generation

```
class LitMNISTDreamer(LightningModule):

    def forward(self, z):
        imgs = self.decoder(z)
        return imgs

    def training_step(self, batch, batch_idx):
        x, y = batch
        representation = self.encoder(x)
        imgs = self(representation)

        loss = perceptual_loss(imgs, x)
        return loss
```

```
model = LitMNISTDreamer.load_from_checkpoint(PATH)
z = sample_noise()
generated_imgs = model(z)
```

How you split up what goes in *forward* vs *training_step* depends on how you want to use this model for prediction.

---

## 2.10 Extensibility

Although lightning makes everything super simple, it doesn't sacrifice any flexibility or control. Lightning offers multiple ways of managing the training state.

### 2.10.1 Training overrides

Any part of the training, validation and testing loop can be modified. For instance, if you wanted to do your own backward pass, you would override the default implementation

```
def backward(self, use_amp, loss, optimizer):
    if use_amp:
        with amp.scale_loss(loss, optimizer) as scaled_loss:
            scaled_loss.backward()
    else:
        loss.backward()
```

With your own

```
class LitMNIST(LightningModule):

    def backward(self, use_amp, loss, optimizer):
        # do a custom way of backward
        loss.backward(retain_graph=True)
```

Or if you wanted to initialize ddp in a different way than the default one

```
def configure_ddp(self, model, device_ids):
    # Lightning DDP simply routes to test_step, val_step, etc...
    model = LightningDistributedDataParallel(
        model,
        device_ids=device_ids,
        find_unused_parameters=True
    )
    return model
```

you could do your own:

```
class LitMNIST(LightningModule):

    def configure_ddp(self, model, device_ids):

        model = Horovod(model)
        # model = Ray(model)
        return model
```

Every single part of training is configurable this way. For a full list look at *LightningModule*.

## 2.11 Callbacks

Another way to add arbitrary functionality is to add a custom callback for hooks that you might care about

```
from pytorch_lightning.callbacks import Callback

class MyPrintingCallback(Callback):

    def on_init_start(self, trainer):
        print('Starting to init trainer!')

    def on_init_end(self, trainer):
        print('Trainer is init now')

    def on_train_end(self, trainer, pl_module):
        print('do something when training ends')
```

And pass the callbacks into the trainer

```
trainer = Trainer(callbacks=[MyPrintingCallback()])
```

**Note:** See full list of 12+ hooks in the *Callbacks*.

---

## 2.12 Child Modules

Research projects tend to test different approaches to the same dataset. This is very easy to do in Lightning with inheritance.

For example, imagine we now want to train an Autoencoder to use as a feature extractor for MNIST images. Recall that *LitMNIST* already defines all the dataloading etc. . . The only things that change in the *Autoencoder* model are the init, forward, training, validation and test step.

```
class Encoder(torch.nn.Module):
    pass

class Decoder(torch.nn.Module):
    pass

class AutoEncoder(LitMNIST):

    def __init__(self):
        super().__init__()
        self.encoder = Encoder()
        self.decoder = Decoder()

    def forward(self, x):
        generated = self.decoder(x)

    def training_step(self, batch, batch_idx):
        x, _ = batch
```

(continues on next page)

(continued from previous page)

```
        representation = self.encoder(x)
        x_hat = self(representation)

        loss = MSE(x, x_hat)
        return loss

    def validation_step(self, batch, batch_idx):
        return self._shared_eval(batch, batch_idx, 'val')

    def test_step(self, batch, batch_idx):
        return self._shared_eval(batch, batch_idx, 'test')

    def _shared_eval(self, batch, batch_idx, prefix):
        x, y = batch
        representation = self.encoder(x)
        x_hat = self(representation)

        loss = F.nll_loss(logits, y)
        return {f'{prefix}_loss': loss}
```

and we can train this using the same trainer

```
autoencoder = AutoEncoder()
trainer = Trainer()
trainer.fit(autoencoder)
```

And remember that the forward method is to define the practical use of a LightningModule. In this case, we want to use the *AutoEncoder* to extract image representations

```
some_images = torch.Tensor(32, 1, 28, 28)
representations = autoencoder(some_images)
```

---

## 2.13 Transfer Learning

### 2.13.1 Using Pretrained Models

Sometimes we want to use a LightningModule as a pretrained model. This is fine because a LightningModule is just a *torch.nn.Module*!

**Note:** Remember that a LightningModule is EXACTLY a torch.nn.Module but with more capabilities.

Let's use the *AutoEncoder* as a feature extractor in a separate model.

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

class AutoEncoder(LightningModule):
    def __init__(self):
        self.encoder = Encoder()
        self.decoder = Decoder()
```

(continues on next page)

(continued from previous page)

```
class CIFAR10Classifier(LightningModule):
    def __init__(self):
        # init the pretrained LightningModule
        self.feature_extractor = AutoEncoder.load_from_checkpoint(PATH)
        self.feature_extractor.freeze()

        # the autoencoder outputs a 100-dim representation and CIFAR-10 has 10 classes
        self.classifier = nn.Linear(100, 10)

    def forward(self, x):
        representations = self.feature_extractor(x)
        x = self.classifier(representations)
        ...
```

We used our pretrained Autoencoder (a LightningModule) for transfer learning!

### 2.13.2 Example: Imagenet (computer Vision)

```
import torchvision.models as models

class ImagenetTransferLearning(LightningModule):
    def __init__(self):
        # init a pretrained resnet
        num_target_classes = 10
        self.feature_extractor = models.resnet50(
                                    pretrained=True,
                                    num_classes=num_target_classes)
        self.feature_extractor.eval()

        # use the pretrained model to classify cifar-10 (10 image classes)
        self.classifier = nn.Linear(2048, num_target_classes)

    def forward(self, x):
        representations = self.feature_extractor(x)
        x = self.classifier(representations)
        ...
```

Finetune

```
model = ImagenetTransferLearning()
trainer = Trainer()
trainer.fit(model)
```

And use it to predict your data of interest

```
model = ImagenetTransferLearning.load_from_checkpoint(PATH)
model.freeze()

x = some_images_from_cifar10()
predictions = model(x)
```

We used a pretrained model on imagenet, finetuned on CIFAR-10 to predict on CIFAR-10. In the non-academic world we would finetune on a tiny dataset you have and predict on your dataset.

### 2.13.3 Example: BERT (NLP)

Lightning is completely agnostic to what's used for transfer learning so long as it is a *torch.nn.Module* subclass.

Here's a model that uses Huggingface transformers.

```
class BertMNLIFinetuner(LightningModule):

    def __init__(self):
        super().__init__()

        self.bert = BertModel.from_pretrained('bert-base-cased', output_
↪attentions=True)
        self.W = nn.Linear(bert.config.hidden_size, 3)
        self.num_classes = 3


    def forward(self, input_ids, attention_mask, token_type_ids):

        h, _, attn = self.bert(input_ids=input_ids,
                         attention_mask=attention_mask,
                         token_type_ids=token_type_ids)

        h_cls = h[:, 0]
        logits = self.W(h_cls)
        return logits, attn
```

CHAPTER

THREE

# FAST PERFORMANCE

Here are some best practices to increase your performance.

---

## 3.1 Dataloaders

When building your Dataloader set *num_workers* > 0 and *pin_memory=True* (only for GPUs).

```
Dataloader(dataset, num_workers=8, pin_memory=True)
```

### 3.1.1 num_workers

The question of how many *num_workers* is tricky. Here's a summary of some references, [1], and our suggestions.

1. num_workers=0 means ONLY the main process will load batches (that can be a bottleneck).
2. num_workers=1 means ONLY one worker (just not the main process) will load data but it will still be slow.
3. The num_workers depends on the batch size and your machine.
4. A general place to start is to set *num_workers* equal to the number of CPUs on that machine.

**Warning:** Increasing num_workers will ALSO increase your CPU memory consumption.

The best thing to do is to increase the nun_workers slowly and stop once you see no more improvement in your training speed.

### 3.1.2 Spawn

When using *distributed_backend=ddp_spawn* (the ddp default) or TPU training, the way multiple GPUs/TPU cores are used is by calling *.spawn()* under the hood. The problem is that PyTorch has issues with *num_workers* > 0 when using .spawn(). For this reason we recommend you use *distributed_backend=ddp* so you can increase the *num_workers*, however your script has to be callable like so:

```
python my_program.py --gpus X
```

---

## 3.2 .item(), .numpy(), .cpu()

Don't call .item() anywhere on your code. Use *.detach()* instead to remove the connected graph calls. Lightning takes a great deal of care to be optimized for this.

---

## 3.3 empty_cache()

Don't call this unnecessarily! Every time you call this ALL your GPUs have to wait to sync.

---

## 3.4 construct tensors directly on device

LightningModules know what device they are on! construct tensors on the device directly to avoid CPU->Device transfer.

```
# bad
t = tensor.rand(2, 2).cuda()

# good (self is lightningModule)
t = tensor.rand(2,2, device=self.device)
```

---

## 3.5 Use DDP not DP

DP performs three GPU transfers for EVERY batch:

1. Copy model to device.
2. Copy data to device.
3. Copy outputs of each device back to master.

Whereas DDP only performs 1 transfer to sync gradients. Because of this, DDP is MUCH faster than DP.

---

## 3.6 16-bit precision

Use 16-bit to decrease the memory (and thus increase your batch size). On certain GPUs (V100s, 2080tis), 16-bit calculations are also faster. However, know that 16-bit and multi-processing (any DDP) can have issues. Here are some common problems.

1. **CUDA error: an illegal memory access was encountered.** The solution is likely setting a specific CUDA, CUDNN, PyTorch version combination.
2. *CUDA error: device-side assert triggered.* This is a general catch-all error. To see the actual error run your script like so:

```
# won't see what the error is
python main.py

# will see what the error is
CUDA_LAUNCH_BLOCKING=1 python main.py
```

We also recommend using 16-bit native found in PyTorch 1.6. Just install this version and Lightning will automatically use it.

CHAPTER

FOUR

# CALLBACKS

Lightning has a callback system to execute arbitrary code. Callbacks should capture NON-ESSENTIAL logic that is NOT required for your *LightningModule* to run.

An overall Lightning system should have:

1. Trainer for all engineering
2. LightningModule for all research code.
3. Callbacks for non-essential code.

Example:

```
class MyPrintingCallback(Callback):

    def on_init_start(self, trainer):
        print('Starting to init trainer!')

    def on_init_end(self, trainer):
        print('trainer is init now')

    def on_train_end(self, trainer, pl_module):
        print('do something when training ends')

trainer = Trainer(callbacks=[MyPrintingCallback()])
```

```
Starting to init trainer!
trainer is init now
```

We successfully extended functionality without polluting our super clean *LightningModule* research code.

---

## 4.1 Callback Base

Abstract base class used to build new callbacks.

**class** pytorch_lightning.callbacks.base.**Callback**

Bases: abc.ABC

Abstract base class used to build new callbacks.

**on_batch_end**(*trainer*, *pl_module*)

Called when the training batch ends.

**on_batch_start** (*trainer*, *pl_module*)
Called when the training batch begins.

**on_epoch_end** (*trainer*, *pl_module*)
Called when the epoch ends.

**on_epoch_start** (*trainer*, *pl_module*)
Called when the epoch begins.

**on_fit_end** (*trainer*)
Called when fit ends

**on_fit_start** (*trainer*)
Called when fit begins

**on_init_end** (*trainer*)
Called when the trainer initialization ends, model has not yet been set.

**on_init_start** (*trainer*)
Called when the trainer initialization begins, model has not yet been set.

**on_keyboard_interrupt** (*trainer*, *pl_module*)
Called when the training is interrupted by KeyboardInterrupt.

**on_sanity_check_end** (*trainer*, *pl_module*)
Called when the validation sanity check ends.

**on_sanity_check_start** (*trainer*, *pl_module*)
Called when the validation sanity check starts.

**on_test_batch_end** (*trainer*, *pl_module*)
Called when the test batch ends.

**on_test_batch_start** (*trainer*, *pl_module*)
Called when the test batch begins.

**on_test_end** (*trainer*, *pl_module*)
Called when the test ends.

**on_test_start** (*trainer*, *pl_module*)
Called when the test begins.

**on_train_end** (*trainer*, *pl_module*)
Called when the train ends.

**on_train_start** (*trainer*, *pl_module*)
Called when the train begins.

**on_validation_batch_end** (*trainer*, *pl_module*)
Called when the validation batch ends.

**on_validation_batch_start** (*trainer*, *pl_module*)
Called when the validation batch begins.

**on_validation_end** (*trainer*, *pl_module*)
Called when the validation loop ends.

**on_validation_start** (*trainer*, *pl_module*)
Called when the validation loop begins.

**setup** (*trainer*, *stage*)
Called when fit or test begins

**teardown** (*trainer*, *stage*)
Called when fit or test ends

## 4.2 Early Stopping

Monitor a validation metric and stop training when it stops improving.

**class** pytorch_lightning.callbacks.early_stopping.**EarlyStopping** (*monitor='val_loss'*, *min_delta=0.0*, *patience=3*, *verbose=False*, *mode='auto'*, *strict=True*)

Bases: *pytorch_lightning.callbacks.base.Callback*

**Parameters**

- **monitor** (str) – quantity to be monitored. Default: 'val_loss'.
- **min_delta** (float) – minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than *min_delta*, will count as no improvement. Default: 0.
- **patience** (int) – number of validation epochs with no improvement after which training will be stopped. Default: 0.
- **verbose** (bool) – verbosity mode. Default: False.
- **mode** (str) – one of {auto, min, max}. In *min* mode, training will stop when the quantity monitored has stopped decreasing; in *max* mode it will stop when the quantity monitored has stopped increasing; in *auto* mode, the direction is automatically inferred from the name of the monitored quantity. Default: 'auto'.
- **strict** (bool) – whether to crash the training if *monitor* is not found in the validation metrics. Default: True.

Example:

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.callbacks import EarlyStopping
>>> early_stopping = EarlyStopping('val_loss')
>>> trainer = Trainer(early_stop_callback=early_stopping)
```

**_validate_condition_metric** (*logs*)
Checks that the condition metric for early stopping is good :param _sphinx_paramlinks_pytorch_lightning.callbacks.early_stopping.EarlyStopping._validate_condition_metric.logs: :return:

**on_train_end** (*trainer*, *pl_module*)
Called when the train ends.

**on_train_start** (*trainer*, *pl_module*)
Called when the train begins.

**on_validation_end** (*trainer*, *pl_module*)
Called when the validation loop ends.

## 4.3 Gradient Accumulator

Change gradient accumulation factor according to scheduling.

**class** pytorch_lightning.callbacks.gradient_accumulation_scheduler.**GradientAccumulationSched**

Bases: *pytorch_lightning.callbacks.base.Callback*

Change gradient accumulation factor according to scheduling.

**Parameters** **scheduling** (dict) – scheduling in format {epoch: accumulation_factor}

> **Warning:** Epochs indexing starts from "1" until v0.6.x, but will start from "0" in v0.8.0.

Example:

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.callbacks import GradientAccumulationScheduler

# at epoch 5 start accumulating every 2 batches
>>> accumulator = GradientAccumulationScheduler(scheduling={5: 2})
>>> trainer = Trainer(callbacks=[accumulator])

# alternatively, pass the scheduling dict directly to the Trainer
>>> trainer = Trainer(accumulate_grad_batches={5: 2})
```

**on_epoch_start** (*trainer*, *pl_module*)

Called when the epoch begins.

---

## 4.4 Learning Rate Logger

Log learning rate for lr schedulers during training

**class** pytorch_lightning.callbacks.lr_logger.**LearningRateLogger**

Bases: *pytorch_lightning.callbacks.base.Callback*

Automatically logs learning rate for learning rate schedulers during training.

Example:

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.callbacks import LearningRateLogger
>>> lr_logger = LearningRateLogger()
>>> trainer = Trainer(callbacks=[lr_logger])
```

Logging names are automatically determined based on optimizer class name. In case of multiple optimizers of same type, they will be named *Adam*, *Adam-1* etc. If a optimizer has multiple parameter groups they will be named *Adam/pg1*, *Adam/pg2* etc. To control naming, pass in a *name* keyword in the construction of the learning rate schdulers

Example:

```
def configure_optimizer(self):
    optimizer = torch.optim.Adam(...)
    lr_scheduler = {'scheduler': torch.optim.lr_schedulers.LambdaLR(optimizer, ...
↪)
                    'name': 'my_logging_name'}
    return [optimizer], [lr_scheduler]
```

**_extract_lr**(*trainer*, *interval*)
Extracts learning rates for lr schedulers and saves information into dict structure.

**on_batch_start**(*trainer*, *pl_module*)
Called when the training batch begins.

**on_epoch_start**(*trainer*, *pl_module*)
Called when the epoch begins.

**on_train_start**(*trainer*, *pl_module*)
Called before training, determines unique names for all lr schedulers in the case of multiple of the same type or in the case of multiple parameter groups

## 4.5 Model Checkpointing

Automatically save model checkpoints during training.

**class** pytorch_lightning.callbacks.model_checkpoint.**ModelCheckpoint**(*filepath=None*, *monitor='val_loss'*, *verbose=False*, *save_last=False*, *save_top_k=1*, *save_weights_only=False*, *mode='auto'*, *period=1*, *prefix=''*)

Bases: *pytorch_lightning.callbacks.base.Callback*

Save the model after every epoch if it improves.

After training finishes, use `best_model_path` to retrieve the path to the best checkpoint file and `best_model_score` to retrieve its score.

**Parameters**

- **filepath** (`Optional`[`str`]) – path to save the model file. Can contain named formatting options to be auto-filled.

  Example:

```
# custom path
# saves a file like: my/path/epoch_0.ckpt
>>> checkpoint_callback = ModelCheckpoint('my/path/')

# save any arbitrary metrics like `val_loss`, etc. in name
# saves a file like: my/path/epoch=2-val_loss=0.2_other_metric=0.3.
↪ckpt
```

(continues on next page)

(continued from previous page)

```
>>> checkpoint_callback = ModelCheckpoint(
...     filepath='my/path/{epoch}-{val_loss:.2f}-{other_metric:.2f}
↪'
... )
```

Can also be set to *None*, then it will be set to default location during trainer construction.

- **monitor** (str) – quantity to monitor.
- **verbose** (bool) – verbosity mode. Default: `False`.
- **save_last** (bool) – always saves the model at the end of the epoch. Default: `False`.
- **save_top_k** (int) – if *save_top_k == k*, the best k models according to the quantity monitored will be saved. if `save_top_k == 0`, no models are saved. if `save_top_k == -1`, all models are saved. Please note that the monitors are checked every *period* epochs. if `save_top_k >= 2` and the callback is called multiple times inside an epoch, the name of the saved file will be appended with a version count starting with *v0*.
- **mode** (str) – one of {auto, min, max}. If `save_top_k != 0`, the decision to overwrite the current save file is made based on either the maximization or the minimization of the monitored quantity. For *val_acc*, this should be *max*, for *val_loss* this should be *min*, etc. In *auto* mode, the direction is automatically inferred from the name of the monitored quantity.
- **save_weights_only** (bool) – if `True`, then only the model's weights will be saved (`model.save_weights(filepath)`), else the full model is saved (`model.save(filepath)`).
- **period** (int) – Interval (number of epochs) between checkpoints.

Example:

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.callbacks import ModelCheckpoint

# saves checkpoints to 'my/path/' whenever 'val_loss' has a new min
>>> checkpoint_callback = ModelCheckpoint(filepath='my/path/')
>>> trainer = Trainer(checkpoint_callback=checkpoint_callback)

# save epoch and val_loss in name
# saves a file like: my/path/sample-mnist_epoch=02_val_loss=0.32.ckpt
>>> checkpoint_callback = ModelCheckpoint(
...     filepath='my/path/sample-mnist_{epoch:02d}-{val_loss:.2f}'
... )

# retrieve the best checkpoint after training
checkpoint_callback = ModelCheckpoint(filepath='my/path/')
trainer = Trainer(checkpoint_callback=checkpoint_callback)
model = ...
trainer.fit(model)
checkpoint_callback.best_model_path
```

**format_checkpoint_name**(*epoch*, *metrics*, *ver=None*)

Generate a filename according to the defined template.

Example:

```
>>> tmpdir = os.path.dirname(__file__)
>>> ckpt = ModelCheckpoint(os.path.join(tmpdir, '{epoch}'))
```

(continues on next page)

(continued from previous page)

```
>>> os.path.basename(ckpt.format_checkpoint_name(0, {}))
'epoch=0.ckpt'
>>> ckpt = ModelCheckpoint(os.path.join(tmpdir, '{epoch:03d}'))
>>> os.path.basename(ckpt.format_checkpoint_name(5, {}))
'epoch=005.ckpt'
>>> ckpt = ModelCheckpoint(os.path.join(tmpdir, '{epoch}-{val_loss:.2f}'))
>>> os.path.basename(ckpt.format_checkpoint_name(2, dict(val_loss=0.123456)))
'epoch=2-val_loss=0.12.ckpt'
>>> ckpt = ModelCheckpoint(os.path.join(tmpdir, '{missing:d}'))
>>> os.path.basename(ckpt.format_checkpoint_name(0, {}))
'missing=0.ckpt'
```

**on_validation_end**(*trainer*, *pl_module*)
Called when the validation loop ends.

## 4.6 Progress Bars

Use or override one of the progress bar callbacks.

**class** pytorch_lightning.callbacks.progress.**ProgressBar**(*refresh_rate=1*, *process_position=0*)

Bases: *pytorch_lightning.callbacks.progress.ProgressBarBase*

This is the default progress bar used by Lightning. It prints to *stdout* using the `tqdm` package and shows up to four different bars:

- **sanity check progress:** the progress during the sanity check run
- **main progress:** shows training + validation progress combined. It also accounts for multiple validation runs during training when *val_check_interval* is used.
- **validation progress:** only visible during validation; shows total progress over all validation datasets.
- **test progress:** only active when testing; shows total progress over all test datasets.

For infinite datasets, the progress bar never ends.

If you want to customize the default `tqdm` progress bars used by Lightning, you can override specific methods of the callback class and pass your custom implementation to the *Trainer*:

Example:

```
class LitProgressBar(ProgressBar):

    def init_validation_tqdm(self):
        bar = super().init_validation_tqdm()
        bar.set_description('running validation ...')
        return bar

bar = LitProgressBar()
trainer = Trainer(callbacks=[bar])
```

**Parameters**

- **refresh_rate** (int) – Determines at which rate (in number of batches) the progress bars get updated. Set it to 0 to disable the display. By default, the *Trainer* uses this implementation of the progress bar and sets the refresh rate to the value provided to the *progress_bar_refresh_rate* argument in the *Trainer*.
- **process_position** (int) – Set this to a value greater than 0 to offset the progress bars by this many lines. This is useful when you have progress bars defined elsewhere and want to show all of them together. This corresponds to *process_position* in the *Trainer*.

**disable**()
You should provide a way to disable the progress bar. The *Trainer* will call this to disable the output on processes that have a rank different from 0, e.g., in multi-node training.

**Return type** None

**enable**()
You should provide a way to enable the progress bar. The *Trainer* will call this in e.g. pre-training routines like the *learning rate finder* to temporarily enable and disable the main progress bar.

**Return type** None

**init_sanity_tqdm**()
Override this to customize the tqdm bar for the validation sanity run.

**Return type** tqdm

**init_test_tqdm**()
Override this to customize the tqdm bar for testing.

**Return type** tqdm

**init_train_tqdm**()
Override this to customize the tqdm bar for training.

**Return type** tqdm

**init_validation_tqdm**()
Override this to customize the tqdm bar for validation.

**Return type** tqdm

**on_batch_end**(*trainer*, *pl_module*)
Called when the training batch ends.

**on_epoch_start**(*trainer*, *pl_module*)
Called when the epoch begins.

**on_sanity_check_end**(*trainer*, *pl_module*)
Called when the validation sanity check ends.

**on_sanity_check_start**(*trainer*, *pl_module*)
Called when the validation sanity check starts.

**on_test_batch_end**(*trainer*, *pl_module*)
Called when the test batch ends.

**on_test_end**(*trainer*, *pl_module*)
Called when the test ends.

**on_test_start**(*trainer*, *pl_module*)
Called when the test begins.

**on_train_end**(*trainer*, *pl_module*)
Called when the train ends.

**on_train_start**(*trainer*, *pl_module*)
Called when the train begins.

**on_validation_batch_end**(*trainer*, *pl_module*)
Called when the validation batch ends.

**on_validation_end**(*trainer*, *pl_module*)
Called when the validation loop ends.

**on_validation_start**(*trainer*, *pl_module*)
Called when the validation loop begins.

**class** pytorch_lightning.callbacks.progress.**ProgressBarBase**
Bases: *pytorch_lightning.callbacks.base.Callback*

The base class for progress bars in Lightning. It is a *Callback* that keeps track of the batch progress in the *Trainer*. You should implement your highly custom progress bars with this as the base class.

Example:

```
class LitProgressBar(ProgressBarBase):

    def __init__(self):
        super().__init__()  # don't forget this :)
        self.enable = True

    def disable(self):
        self.enable = False

    def on_batch_end(self, trainer, pl_module):
        super().on_batch_end(trainer, pl_module)  # don't forget this :)
        percent = (self.train_batch_idx / self.total_train_batches) * 100
        sys.stdout.flush()
        sys.stdout.write(f'{percent:.01f} percent complete \r')

bar = LitProgressBar()
trainer = Trainer(callbacks=[bar])
```

**disable**()
You should provide a way to disable the progress bar. The *Trainer* will call this to disable the output on processes that have a rank different from 0, e.g., in multi-node training.

**enable**()
You should provide a way to enable the progress bar. The *Trainer* will call this in e.g. pre-training routines like the *learning rate finder* to temporarily enable and disable the main progress bar.

**on_batch_end**(*trainer*, *pl_module*)
Called when the training batch ends.

**on_epoch_start**(*trainer*, *pl_module*)
Called when the epoch begins.

**on_init_end**(*trainer*)
Called when the trainer initialization ends, model has not yet been set.

**on_test_batch_end**(*trainer*, *pl_module*)
Called when the test batch ends.

**on_test_start**(*trainer*, *pl_module*)
Called when the test begins.

**on_train_start**(*trainer*, *pl_module*)
Called when the train begins.

**on_validation_batch_end**(*trainer*, *pl_module*)
Called when the validation batch ends.

**on_validation_start**(*trainer*, *pl_module*)
Called when the validation loop begins.

**property test_batch_idx**
The current batch index being processed during testing. Use this to update your progress bar.

**Return type** `int`

**property total_test_batches**
The total number of training batches during testing, which may change from epoch to epoch. Use this to set the total number of iterations in the progress bar. Can return `inf` if the test dataloader is of infinite size.

**Return type** `int`

**property total_train_batches**
The total number of training batches during training, which may change from epoch to epoch. Use this to set the total number of iterations in the progress bar. Can return `inf` if the training dataloader is of infinite size.

**Return type** `int`

**property total_val_batches**
The total number of training batches during validation, which may change from epoch to epoch. Use this to set the total number of iterations in the progress bar. Can return `inf` if the validation dataloader is of infinite size.

**Return type** `int`

**property train_batch_idx**
The current batch index being processed during training. Use this to update your progress bar.

**Return type** `int`

**property val_batch_idx**
The current batch index being processed during validation. Use this to update your progress bar.

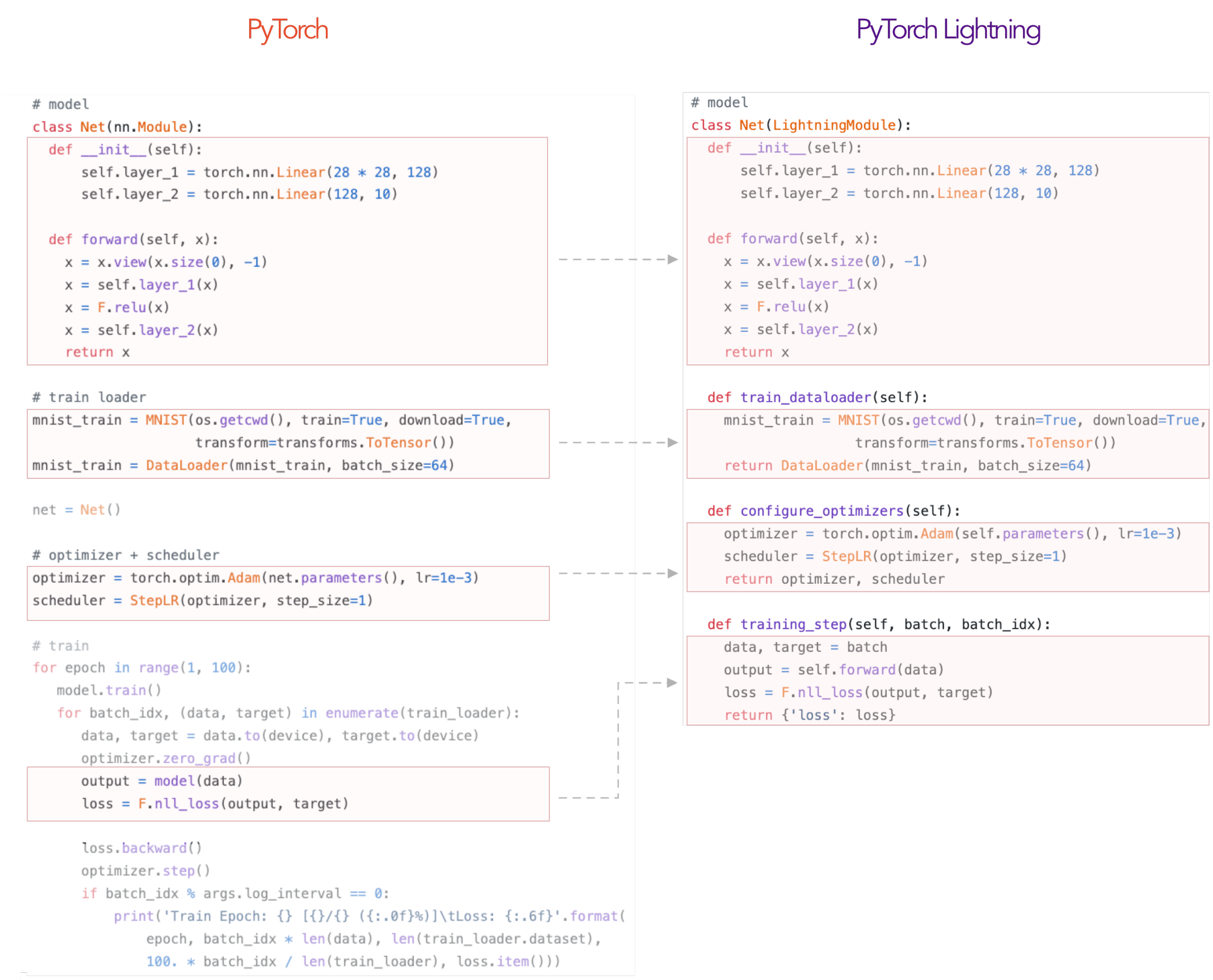
**Return type** `int`

pytorch_lightning.callbacks.progress.**convert_inf**(*x*)
The tqdm doesn't support inf values. We have to convert it to None.

# CHAPTER FIVE

# LIGHTNINGMODULE

A *LightningModule* organizes your PyTorch code into the following sections:




Notice a few things.

1. It's the SAME code.
2. The PyTorch code IS NOT abstracted - just organized.
3. All the other code that's not in the *LightningModule* has been automated for you by the trainer.

```
net = Net()
trainer = Trainer()
trainer.fit(net)
```

4. There are no .cuda() or .to() calls... Lightning does these for you.

```
# don't do in lightning
x = torch.Tensor(2, 3)
x = x.cuda()
x = x.to(device)

# do this instead
x = x  # leave it alone!

# or to init a new tensor
new_x = torch.Tensor(2, 3)
new_x = new_x.type_as(x.type())
```

5. There are no samplers for distributed, Lightning also does this for you.

```
# Don't do in Lightning...
data = MNIST(...)
sampler = DistributedSampler(data)
DataLoader(data, sampler=sampler)

# do this instead
data = MNIST(...)
DataLoader(data)
```

6. A `LightningModule` is a `torch.nn.Module` but with added functionality. Use it as such!

```
net = Net.load_from_checkpoint(PATH)
net.freeze()
out = net(x)
```

Thus, to use Lightning, you just need to organize your code which takes about 30 minutes, (and let's be real, you probably should do anyhow).

---

## 5.1 Minimal Example

Here are the only required methods.

```
>>> import pytorch_lightning as pl
>>> class LitModel(pl.LightningModule):
...
...     def __init__(self):
...         super().__init__()
...         self.l1 = torch.nn.Linear(28 * 28, 10)
...
...     def forward(self, x):
...         return torch.relu(self.l1(x.view(x.size(0), -1)))
...
...     def training_step(self, batch, batch_idx):
```

(continues on next page)

(continued from previous page)

```
...         x, y = batch
...         y_hat = self(x)
...         return {'loss': F.cross_entropy(y_hat, y)}
...
...     def train_dataloader(self):
...         return DataLoader(MNIST(os.getcwd(), train=True, download=True,
...                                 transform=transforms.ToTensor()), batch_size=32)
...
...     def configure_optimizers(self):
...         return torch.optim.Adam(self.parameters(), lr=0.02)
```

Which you can train by doing:

```
trainer = pl.Trainer()
model = LitModel()

trainer.fit(model)
```

---

## 5.2 Training loop structure

The general pattern is that each loop (training, validation, test loop) has 3 methods:

- `___step`
- `___step_end`
- `___epoch_end`

To show how Lightning calls these, let's use the validation loop as an example:

```
val_outs = []
for val_batch in val_data:
    # do something with each batch
    out = validation_step(val_batch)
    val_outs.append(out)

# do something with the outputs for all batches
# like calculate validation set accuracy or loss
validation_epoch_end(val_outs)
```

If we use dp or ddp2 mode, we can also define the `XXX_step_end` method to operate on all parts of the batch:

```
val_outs = []
for val_batch in val_data:
    batches = split_batch(val_batch)
    dp_outs = []
    for sub_batch in batches:
        dp_out = validation_step(sub_batch)
        dp_outs.append(dp_out)

    out = validation_step_end(dp_outs)
    val_outs.append(out)

# do something with the outputs for all batches
```

(continues on next page)

(continued from previous page)

```
# like calculate validation set accuracy or loss
validation_epoch_end(val_outs)
```

### 5.2.1 Add validation loop

Thus, if we wanted to add a validation loop you would add this to your `LightningModule`:

```
>>> class LitModel(pl.LightningModule):
...     def validation_step(self, batch, batch_idx):
...         x, y = batch
...         y_hat = self(x)
...         return {'val_loss': F.cross_entropy(y_hat, y)}
...
...     def validation_epoch_end(self, outputs):
...         val_loss_mean = torch.stack([x['val_loss'] for x in outputs]).mean()
...         return {'val_loss': val_loss_mean}
...
...     def val_dataloader(self):
...         # can also return a list of val dataloaders
...         return DataLoader(...)
```

### 5.2.2 Add test loop

```
>>> class LitModel(pl.LightningModule):
...     def test_step(self, batch, batch_idx):
...         x, y = batch
...         y_hat = self(x)
...         return {'test_loss': F.cross_entropy(y_hat, y)}
...
...     def test_epoch_end(self, outputs):
...         test_loss_mean = torch.stack([x['test_loss'] for x in outputs]).mean()
...         return {'test_loss': test_loss_mean}
...
...     def test_dataloader(self):
...         # can also return a list of test dataloaders
...         return DataLoader(...)
```

However, the test loop won't ever be called automatically to make sure you don't run your test data by accident. Instead you have to explicitly call:

```
# call after training
trainer = Trainer()
trainer.fit(model)
trainer.test()

# or call with pretrained model
model = MyLightningModule.load_from_checkpoint(PATH)
trainer = Trainer()
trainer.test(model)
```

---

## 5.3 Training_step_end method

When using *LightningDataParallel* or *LightningDistributedDataParallel*, the *training_step()* will be operating on a portion of the batch. This is normally ok but in special cases like calculating NCE loss using negative samples, we might want to perform a softmax across all samples in the batch.

For these types of situations, each loop has an additional `__step_end` method which allows you to operate on the pieces of the batch:

```
training_outs = []
for train_batch in train_data:
    # dp, ddp2 splits the batch
    sub_batches = split_batches_for_dp(batch)

    # run training_step on each piece of the batch
    batch_parts_outputs = [training_step(sub_batch) for sub_batch in sub_batches]

    # do softmax with all pieces
    out = training_step_end(batch_parts_outputs)
    training_outs.append(out)

# do something with the outputs for all batches
# like calculate validation set accuracy or loss
training_epoch_end(val_outs)
```

---

## 5.4 Remove cuda calls

In a *LightningModule*, all calls to `.cuda()` and `.to(device)` should be removed. Lightning will do these automatically. This will allow your code to work on CPUs, TPUs and GPUs.

When you init a new tensor in your code, just use `type_as()`:

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

    # put the z on the appropriate gpu or tpu core
    z = sample_noise()
    z = z.type_as(x)
```

---

## 5.5 Data preparation

Data preparation in PyTorch follows 5 steps:

1. Download
2. Clean and (maybe) save to disk
3. Load inside `Dataset`
4. Apply transforms (rotate, tokenize, etc…)
5. Wrap inside a `DataLoader`

When working in distributed settings, steps 1 and 2 have to be done from a single GPU, otherwise you will overwrite these files from every GPU. The `LightningModule` has the `prepare_data` method to allow for this:

```
>>> class LitModel(pl.LightningModule):
...     def prepare_data(self):
...         # download
...         MNIST(os.getcwd(), train=True, download=True, transform=transforms.
↪ToTensor())
...         MNIST(os.getcwd(), train=False, download=True, transform=transforms.
↪ToTensor())
...
...     def setup(self, stage):
...         mnist_train = MNIST(os.getcwd(), train=True, download=False,␣
↪transform=transforms.ToTensor())
...         mnist_test = MNIST(os.getcwd(), train=False, download=False,␣
↪transform=transforms.ToTensor())
...         # train/val split
...         mnist_train, mnist_val = random_split(mnist_train, [55000, 5000])
...
...         # assign to use in dataloaders
...         self.train_dataset = mnist_train
...         self.val_dataset = mnist_val
...         self.test_dataset = mnist_test
...
...     def train_dataloader(self):
...         return DataLoader(self.train_dataset, batch_size=64)
...
...     def val_dataloader(self):
...         return DataLoader(self.mnist_val, batch_size=64)
...
...     def test_dataloader(self):
...         return DataLoader(self.mnist_test, batch_size=64)
```

**Note:** `prepare_data()` is called once.

**Note:** Do anything with data that needs to happen ONLY once here, like download, tokenize, etc…

## 5.6 Lifecycle

The methods in the `LightningModule` are called in this order:

1. `__init__()`
2. `prepare_data()`
3. `configure_optimizers()`
4. `train_dataloader()`

If you define a validation loop then

5. `val_dataloader()`

And if you define a test loop:

6. `test_dataloader()`

**Note:** `test_dataloader()` is only called with `.test()`

In every epoch, the loop methods are called in this frequency:

1. `validation_step()` called every batch
2. `validation_epoch_end()` called every epoch

## 5.7 Live demo

Check out this COLAB for a live demo.

## 5.8 LightningModule Class

**class** `pytorch_lightning.core.`**`LightningModule`**`(*args, **kwargs)`

Bases: `abc.ABC`, `pytorch_lightning.utilities.device_dtype_mixin.DeviceDtypeModuleMixin`, `pytorch_lightning.core.grads.GradInformation`, `pytorch_lightning.core.saving.ModelIO`, `pytorch_lightning.core.hooks.ModelHooks`, `torch.nn.Module`

**`_LightningModule__get_hparams_assignment_variable`**`()`

looks at the code of the class to figure out what the user named self.hparams this only happens when the user explicitly sets self.hparams

**classmethod `_auto_collect_arguments`**`(frame=None)`

Collect all module arguments in the current constructor and all child constructors. The child constructors are all the `__init__` methods that reach the current class through (chained) `super().__init__()` calls.

**Parameters** **frame** – instance frame

**Returns** arguments dictionary of the first instance parents_arguments: arguments dictionary of the parent's instances

**Return type** self_arguments

**`_init_slurm_connection`**`()`

Sets up environment variables necessary for pytorch distributed communications based on slurm environment.

**Return type** `None`

**`configure_apex`**`(amp, model, optimizers, amp_level)`

Override to init AMP your own way. Must return a model and list of optimizers.

**Parameters**

- **amp** (`object`) – pointer to amp library object.
- **model** (`LightningModule`) – pointer to current `LightningModule`.
- **optimizers** (`List[Optimizer]`) – list of optimizers passed in `configure_optimizers()`.
- **amp_level** (`str`) – AMP mode chosen ('O1', 'O2', etc...)

**Return type** Tuple[*LightningModule*, List[Optimizer]]

**Returns** Apex wrapped model and optimizers

### Examples

```
# Default implementation used by Trainer.
def configure_apex(self, amp, model, optimizers, amp_level):
    model, optimizers = amp.initialize(
        model, optimizers, opt_level=amp_level,
    )

    return model, optimizers
```

**configure_ddp**(*model*, *device_ids*)

Override to init DDP in your own way or with your own wrapper. The only requirements are that:

1. On a validation batch the call goes to `model.validation_step`.
2. On a training batch the call goes to `model.training_step`.
3. On a testing batch, the call goes to `model.test_step`.+

**Parameters**

- **model** (*LightningModule*) – the *LightningModule* currently being optimized.
- **device_ids** (List[int]) – the list of GPU ids.

**Return type** DistributedDataParallel

**Returns** DDP wrapped model

### Examples

```
# default implementation used in Trainer
def configure_ddp(self, model, device_ids):
    # Lightning DDP simply routes to test_step, val_step, etc...
    model = LightningDistributedDataParallel(
        model,
        device_ids=device_ids,
        find_unused_parameters=True
    )
    return model
```

**configure_optimizers**()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

**Return type** Union[Optimizer, Sequence[Optimizer], Dict, Sequence[Dict], Tuple[List, List], None]

**Returns**

Any of these 6 options.

- Single optimizer.
- List or Tuple - List of optimizers.

- Two lists - The first list has multiple optimizers, the second a list of LR schedulers (or lr_dict).
- Dictionary, with an 'optimizer' key, and (optionally) a 'lr_scheduler' key which value is a single LR scheduler or lr_dict.
- Tuple of dictionaries as described, with an optional 'frequency' key.
- None - Fit will run without any optimizer.

**Note:** The 'frequency' value is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1: In the former case, all optimizers will operate on the given batch in each optimization step. In the latter, only one optimizer will operate on the given batch at every step.

The lr_dict is a dictionary which contains scheduler and its associated configuration. It has five keys. The default configuration is shown below.

```
{
    'scheduler': lr_scheduler, # The LR schduler
    'interval': 'epoch', # The unit of the scheduler's step size
    'frequency': 1, # The frequency of the scheduler
    'reduce_on_plateau': False, # For ReduceLROnPlateau scheduler
    'monitor': 'val_loss' # Metric to monitor
}
```

If user only provides LR schedulers, then their configuration will set to default as shown above.

### Examples

```
# most cases
def configure_optimizers(self):
    opt = Adam(self.parameters(), lr=1e-3)
    return opt

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    generator_opt = Adam(self.model_gen.parameters(), lr=0.01)
    disriminator_opt = Adam(self.model_disc.parameters(), lr=0.02)
    return generator_opt, disriminator_opt

# example with learning rate schedulers
def configure_optimizers(self):
    generator_opt = Adam(self.model_gen.parameters(), lr=0.01)
    disriminator_opt = Adam(self.model_disc.parameters(), lr=0.02)
    discriminator_sched = CosineAnnealing(discriminator_opt, T_max=10)
    return [generator_opt, disriminator_opt], [discriminator_sched]

# example with step-based learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_disc.parameters(), lr=0.02)
    gen_sched = {'scheduler': ExponentialLR(gen_opt, 0.99),
                 'interval': 'step'}  # called after each training step
```

(continues on next page)

(continued from previous page)

```
    dis_sched = CosineAnnealing(discriminator_opt, T_max=10) # called every␣
↪epoch
    return [gen_opt, dis_opt], [gen_sched, dis_sched]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`,␣
↪Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_disc.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

**Note:** Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer and learning rate scheduler as needed.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers for you.
- If you use multiple optimizers, *training_step()* will have an additional `optimizer_idx` parameter.
- If you use LBFGS Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the *optimizer_step()* hook.
- If you only want to call a learning rate scheduler every `x` step or epoch, or want to monitor a custom metric, you can specify these in a lr_dict:

```
{
    'scheduler': lr_scheduler,
    'interval': 'step',  # or 'epoch'
    'monitor': 'val_f1',
    'frequency': x,
}
```

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

Same as `torch.nn.Module.forward()`, however in Lightning you want this to define the operations you want to use for prediction (i.e.: on a server or as a feature extractor).

Normally you'd call `self()` from your *training_step()* method. This makes it easy to write a complex system for training with the outputs you'd want in a prediction setting.

You may also find the *auto_move_data()* decorator useful when using the module outside Lightning in a production setting.

**Parameters**

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

**Returns** Predicted output

### Examples

```
# example if we were using this model as a feature extractor
def forward(self, x):
    feature_maps = self.convnet(x)
    return feature_maps

def training_step(self, batch, batch_idx):
    x, y = batch
    feature_maps = self(x)
    logits = self.classifier(feature_maps)

    # ...
    return loss

# splitting it this way allows model to be used a feature extractor
model = MyModelAbove()

inputs = server.get_request()
results = model(inputs)
server.write_results(results)

# -------------
# This is in stark contrast to torch.nn.Module where normally you would have␣
↪this:
def forward(self, batch):
    x, y = batch
    feature_maps = self.convnet(x)
    logits = self.classifier(feature_maps)
    return logits
```

**freeze**()

Freeze all params for inference.

### Example

```
model = MyLightningModule(...)
model.freeze()
```

**Return type** None

**get_progress_bar_dict**()

Additional items to be displayed in the progress bar.

**Return type** Dict[str, Union[int, str]]

**Returns** Dictionary with the items to be displayed in the progress bar.

**get_tqdm_dict**()

Additional items to be displayed in the progress bar.

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

**Returns** Dictionary with the items to be displayed in the progress bar.

> **Warning:** Deprecated since v0.7.3. Use `get_progress_bar_dict()` instead.

**init_ddp_connection**(*global_rank, world_size, is_slurm_managing_tasks=True*)

Override to define your custom way of setting up a distributed environment.

Lightning's implementation uses env:// init by default and sets the first node as root for SLURM managed cluster.

**Parameters**

- **global_rank** (`int`) – The global process idx.
- **world_size** (`int`) – Number of GPUs being use across all nodes. (num_nodes * num_gpus).
- **is_slurm_managing_tasks** (`bool`) – is cluster managed by SLURM.

**Return type** `None`

**on_load_checkpoint**(*checkpoint*)

Called by Lightning to restore your model. If you saved something with `on_save_checkpoint()` this is your chance to restore this.

**Parameters** **checkpoint** (`Dict[str, Any]`) – Loaded checkpoint

### Example

```
def on_load_checkpoint(self, checkpoint):
    # 99% of the time you don't need to implement this method
    self.something_cool_i_want_to_save = checkpoint['something_cool_i_want_to_
↪save']
```

**Note:** Lightning auto-restores global step, epoch, and train state including amp scaling. There is no need for you to restore anything regarding training.

**Return type** `None`

**on_save_checkpoint**(*checkpoint*)

Called by Lightning when saving a checkpoint to give you a chance to store anything else you might want to save.

**Parameters** **checkpoint** (`Dict[str, Any]`) – Checkpoint to be saved

### Example

```
def on_save_checkpoint(self, checkpoint):
    # 99% of use cases you don't need to implement this method
    checkpoint['something_cool_i_want_to_save'] = my_cool_pickable_object
```

**Note:** Lightning saves all aspects of training (epoch, global step, etc...) including amp scaling. There is no need for you to store anything about training.

**Return type** `None`

**optimizer_step**(*epoch*, *batch_idx*, *optimizer*, *optimizer_idx*, *second_order_closure=None*)

Override this method to adjust the default way the `Trainer` calls each optimizer. By default, Lightning calls `step()` and `zero_grad()` as shown in the example once per optimizer.

**Parameters**

- **epoch** (`int`) – Current epoch
- **batch_idx** (`int`) – Index of current batch
- **optimizer** (`Optimizer`) – A PyTorch optimizer
- **optimizer_idx** (`int`) – If you used multiple optimizers this indexes into that list.
- **second_order_closure** (`Optional[Callable]`) – closure for second order methods

### Examples

```
# DEFAULT
def optimizer_step(self, current_epoch, batch_idx, optimizer, optimizer_idx,
                        second_order_closure=None):
    optimizer.step()
    optimizer.zero_grad()

# Alternating schedule for optimizer steps (i.e.: GANs)
def optimizer_step(self, current_epoch, batch_idx, optimizer, optimizer_idx,
                        second_order_closure=None):
    # update generator opt every 2 steps
    if optimizer_idx == 0:
        if batch_idx % 2 == 0 :
            optimizer.step()
            optimizer.zero_grad()

    # update discriminator opt every 4 steps
    if optimizer_idx == 1:
        if batch_idx % 4 == 0 :
            optimizer.step()
            optimizer.zero_grad()

    # ...
    # add as many optimizers as you want
```

Here's another example showing how to use this for more advanced things such as learning rate warm-up:

```python
# learning rate warm-up
def optimizer_step(self, current_epoch, batch_idx, optimizer,
                    optimizer_idx, second_order_closure=None):
    # warm up lr
    if self.trainer.global_step < 500:
        lr_scale = min(1., float(self.trainer.global_step + 1) / 500.)
        for pg in optimizer.param_groups:
            pg['lr'] = lr_scale * self.learning_rate

    # update params
    optimizer.step()
    optimizer.zero_grad()
```

**Note:** If you also override the *on_before_zero_grad()* model hook don't forget to add the call to it before `optimizer.zero_grad()` yourself.

**Return type** `None`

**prepare_data**()

Use this to download and prepare data.

**Warning:** DO NOT set state to the model (use *setup* instead) since this is NOT called on every GPU in DDP/TPU

Example:

```python
def prepare_data(self):
    # good
    download_data()
    tokenize()
    etc()

    # bad
    self.split = data_split
    self.some_state = some_other_state()
```

In DDP prepare_data can be called in two ways (using Trainer(prepare_data_per_node)):

1. Once per node. This is the default and is only called on LOCAL_RANK=0.
2. Once in total. Only called on GLOBAL_RANK=0.

Example:

```python
# DEFAULT
# called once per node on LOCAL_RANK=0 of that node
Trainer(prepare_data_per_node=True)

# call on GLOBAL_RANK=0 (great for shared file systems)
Trainer(prepare_data_per_node=False)
```

This is called before requesting the dataloaders:

```
model.prepare_data()
    if ddp/tpu: init()
model.setup(step)
model.train_dataloader()
model.val_dataloader()
model.test_dataloader()
```

**Return type** None

**print**(**args*, ***kwargs*)

Prints only from process 0. Use this in any distributed mode to log only once.

**Parameters**

- ***args** – The thing to print. Will be passed to Python's built-in print function.
- ****kwargs** – Will be passed to Python's built-in print function.

### Example

```
def forward(self, x):
    self.print(x, 'in forward')
```

**Return type** None

**save_hyperparameters**(**args*, *frame=None*)

Save all model arguments.

**Parameters** **args** – single object of *dict*, *NameSpace* or *OmegaConf* or string names or argumenst from class *__init__*

```
>>> from collections import OrderedDict
>>> class ManuallyArgsModel(LightningModule):
...     def __init__(self, arg1, arg2, arg3):
...         super().__init__()
...         # manually assine arguments
...         self.save_hyperparameters('arg1', 'arg3')
...     def forward(self, *args, **kwargs):
...         ...
>>> model = ManuallyArgsModel(1, 'abc', 3.14)
>>> model.hparams
"arg1": 1
"arg3": 3.14
```

```
>>> class AutomaticArgsModel(LightningModule):
...     def __init__(self, arg1, arg2, arg3):
...         super().__init__()
...         # equivalent automatic
...         self.save_hyperparameters()
...     def forward(self, *args, **kwargs):
...         ...
>>> model = AutomaticArgsModel(1, 'abc', 3.14)
>>> model.hparams
"arg1": 1
"arg2": abc
"arg3": 3.14
```

```
>>> class SingleArgModel(LightningModule):
...     def __init__(self, params):
...         super().__init__()
...         # manually assign single argument
...         self.save_hyperparameters(params)
...     def forward(self, *args, **kwargs):
...         ...
>>> model = SingleArgModel(Namespace(p1=1, p2='abc', p3=3.14))
>>> model.hparams
"p1": 1
"p2": abc
"p3": 3.14
```

**Return type** None

**tbptt_split_batch**(*batch*, *split_size*)

When using truncated backpropagation through time, each batch must be split along the time dimension. Lightning handles this by default, but for custom behavior override this function.

**Parameters**

- **batch** (Tensor) – Current batch
- **split_size** (int) – The size of the split

**Return type** list

**Returns** List of batch splits. Each split will be passed to *training_step()* to enable truncated back propagation through time. The default implementation splits root level Tensors and Sequences at dim=1 (i.e. time dim). It assumes that each time dim is the same length.

### Examples

```
def tbptt_split_batch(self, batch, split_size):
  splits = []
  for t in range(0, time_dims[0], split_size):
      batch_split = []
      for i, x in enumerate(batch):
          if isinstance(x, torch.Tensor):
              split_x = x[:, t:t + split_size]
          elif isinstance(x, collections.Sequence):
              split_x = [None] * len(x)
              for batch_idx in range(len(x)):
                  split_x[batch_idx] = x[batch_idx][t:t + split_size]

          batch_split.append(split_x)

      splits.append(batch_split)

  return splits
```

**Note:** Called in the training loop after *on_batch_start()* if *truncated_bptt_steps* > 0. Each returned batch split is passed separately to *training_step()*.

**test_dataloader**()

Implement one or multiple PyTorch DataLoaders for testing.

The dataloader you return will not be called every epoch unless you set *reload_dataloaders_every_epoch* to True.

It's recommended that all data downloads and preparation happen in *prepare_data()*.

- *fit()*
- ...
- *prepare_data()*
- *train_dataloader()*
- *val_dataloader()*
- *test_dataloader()*

---

**Note:** Lightning adds the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

---

**Return type** Union[DataLoader, List[DataLoader]]

**Returns** Single or multiple PyTorch DataLoaders.

### Example

```
def test_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=False, transform=transform,
                    download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=False
    )

    return loader
```

---

**Note:** If you don't need a test dataset and a *test_step()*, you don't need to implement this method.

---

**test_end**(*outputs*)

**Warning:** Deprecated in v0.7.0. Use *test_epoch_end()* instead. Will be removed in 1.0.0.

**test_epoch_end**(*outputs*)

Called at the end of a test epoch with the output of all test steps.

```
# the pseudocode for these calls
test_outs = []
for test_batch in test_data:
    out = test_step(test_batch)
```

(continues on next page)

(continued from previous page)

```
    test_outs.append(out)
test_epoch_end(test_outs)
```

**Parameters** **outputs** (Union[List[Dict[str, Tensor]], List[List[Dict[str, Tensor]]]]) – List of outputs you defined in *test_step_end()*, or if there are multiple dataloaders, a list containing a list of outputs for each dataloader

**Returns**

Dict has the following optional keys:

- progress_bar -> Dict for progress bar display. Must have only tensors.
- log -> Dict of metrics to add to logger. Must have only tensors (no images, etc).

**Return type** Dict or OrderedDict

**Note:** If you didn't define a *test_step()*, this won't be called.

- The outputs here are strictly for logging or progress bar.
- If you don't need to display anything, don't return anything.
- If you want to manually set current step, specify it with the 'step' key in the 'log' Dict

### Examples

With a single dataloader:

```
def test_epoch_end(self, outputs):
    test_acc_mean = 0
    for output in outputs:
        test_acc_mean += output['test_acc']

    test_acc_mean /= len(outputs)
    tqdm_dict = {'test_acc': test_acc_mean.item()}

    # show test_loss and test_acc in progress bar but only log test_loss
    results = {
        'progress_bar': tqdm_dict,
        'log': {'test_acc': test_acc_mean.item()}
    }
    return results
```

With multiple dataloaders, *outputs* will be a list of lists. The outer list contains one entry per dataloader, while the inner list contains the individual outputs of each test step for that dataloader.

```
def test_epoch_end(self, outputs):
    test_acc_mean = 0
    i = 0
    for dataloader_outputs in outputs:
        for output in dataloader_outputs:
            test_acc_mean += output['test_acc']
            i += 1
```

(continues on next page)

(continued from previous page)

```
    test_acc_mean /= i
    tqdm_dict = {'test_acc': test_acc_mean.item()}

    # show test_loss and test_acc in progress bar but only log test_loss
    results = {
        'progress_bar': tqdm_dict,
        'log': {'test_acc': test_acc_mean.item(), 'step': self.current_epoch}
    }
    return results
```

**test_step**(**args, **kwargs*)

Operates on a single batch of data from the test set. In this step you'd normally generate examples or calculate anything of interest such as accuracy.

```
# the pseudocode for these calls
test_outs = []
for test_batch in test_data:
    out = test_step(test_batch)
    test_outs.append(out)
test_epoch_end(test_outs)
```

**Parameters**

- **batch** (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (*int*) – The index of this batch.
- **dataloader_idx** (*int*) – The index of the dataloader that produced this batch (only if multiple test datasets used).

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

**Returns** Dict or OrderedDict - passed to the *test_epoch_end()* method. If you defined *test_step_end()* it will go to that first.

```
# if you have one test dataloader:
def test_step(self, batch, batch_idx)

# if you have multiple test dataloaders:
def test_step(self, batch, batch_idx, dataloader_idx)
```

## Examples

```
# CASE 1: A single test dataset
def test_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
```

(continues on next page)

(continued from previous page)

```
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # all optional...
    # return whatever you need for the collation function test_epoch_end
    output = OrderedDict({
        'val_loss': loss_val,
        'val_acc': torch.tensor(val_acc), # everything must be a tensor
    })

    # return an optional dict
    return output
```

If you pass in multiple validation datasets, `test_step()` will have an additional argument.

```
# CASE 2: multiple test datasets
def test_step(self, batch, batch_idx, dataset_idx):
    # dataset_idx tells you which dataset this is.
```

**Note:** If you don't need to validate you don't need to implement this method.

**Note:** When the `test_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of the test epoch, the model goes back to training mode and gradients are enabled.

**test_step_end**(**args*, ***kwargs*)

Use this when testing with dp or ddp2 because `test_step()` will operate on only part of the batch. However, this is still optional and only needed for things like softmax or NCE loss.

**Note:** If you later switch to ddp or some other mode, this will still be called so that you don't have to change your code.

```
# pseudocode
sub_batches = split_batches_for_dp(batch)
batch_parts_outputs = [test_step(sub_batch) for sub_batch in sub_batches]
test_step_end(batch_parts_outputs)
```

**Parameters** **batch_parts_outputs** – What you return in `test_step()` for each batch part.

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

**Returns** Dict or OrderedDict - passed to the `test_epoch_end()`.

### Examples

```
# WITHOUT test_step_end
# if used in DP or DDP2, this batch is 1/num_gpus large
def test_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self(x)
    loss = self.softmax(out)
    loss = nce_loss(loss)
    return {'loss': loss}

# --------------
# with test_step_end to do softmax over the full batch
def test_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self(x)
    return {'out': out}

def test_step_end(self, outputs):
    # this out is now the full size of the batch
    out = outputs['out']

    # this softmax now uses the full batch size
    loss = nce_loss(loss)
    return {'loss': loss}
```

**See also:**

See the *Multi-GPU training* guide for more details.

**tng_dataloader**()

> **Warning:** Deprecated in v0.5.0. Use *train_dataloader()* instead. Will be removed in 1.0.0.

**train_dataloader**()

Implement a PyTorch DataLoader for training.

**Return type** `DataLoader`

**Returns** Single PyTorch `DataLoader`.

The dataloader you return will not be called every epoch unless you set *reload_dataloaders_every_epoch* to `True`.

It's recommended that all data downloads and preparation happen in *prepare_data()*.

- *fit()*
- ...
- *prepare_data()*
- *train_dataloader()*

**Note:** Lightning adds the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

### Example

```
def train_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=True, transform=transform,
                    download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=True
    )
    return loader
```

**training_end**(**args*, ***kwargs*)

**Warning:** Deprecated in v0.7.0. Use *training_step_end()* instead.

**training_epoch_end**(*outputs*)
Called at the end of the training epoch with the outputs of all training steps.

```
# the pseudocode for these calls
train_outs = []
for train_batch in train_data:
    out = training_step(train_batch)
    train_outs.append(out)
training_epoch_end(train_outs)
```

**Parameters outputs** (Union[List[Dict[str, Tensor]], List[List[Dict[str, Tensor]]]]) – List of outputs you defined in *training_step()*, or if there are multiple dataloaders, a list containing a list of outputs for each dataloader.

**Return type** Dict[str, Dict[str, Tensor]]

**Returns**

Dict or OrderedDict. May contain the following optional keys:

- log (metrics to be added to the logger; only tensors)
- progress_bar (dict for progress bar display)
- any metric used in a callback (e.g. early stopping).

**Note:** If this method is not overridden, this won't be called.

- The outputs here are strictly for logging or progress bar.
- If you don't need to display anything, don't return anything.

- If you want to manually set current step, you can specify the 'step' key in the 'log' dict.

### Examples

With a single dataloader:

```
def training_epoch_end(self, outputs):
    train_acc_mean = 0
    for output in outputs:
        train_acc_mean += output['train_acc']

    train_acc_mean /= len(outputs)

    # log training accuracy at the end of an epoch
    results = {
        'log': {'train_acc': train_acc_mean.item()},
        'progress_bar': {'train_acc': train_acc_mean},
    }
    return results
```

With multiple dataloaders, `outputs` will be a list of lists. The outer list contains one entry per dataloader, while the inner list contains the individual outputs of each training step for that dataloader.

```
def training_epoch_end(self, outputs):
    train_acc_mean = 0
    i = 0
    for dataloader_outputs in outputs:
        for output in dataloader_outputs:
            train_acc_mean += output['train_acc']
            i += 1

    train_acc_mean /= i

    # log training accuracy at the end of an epoch
    results = {
        'log': {'train_acc': train_acc_mean.item(), 'step': self.current_
↪epoch}
        'progress_bar': {'train_acc': train_acc_mean},
    }
    return results
```

**training_step**(**args*, ***kwargs*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

**Parameters**

- **batch** (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (*int*) – Integer displaying index of this batch
- **optimizer_idx** (*int*) – When using multiple optimizers, this argument will also be present.
- **hiddens** (`Tensor`) – Passed in if *truncated_bptt_steps* > 0.

**Return type** `Union`[`int`, `Dict`[`str`, `Union`[`Tensor`, `Dict`[`str`, `Tensor`]]]]

**Returns**

Dict with loss key and optional log or progress bar keys. When implementing *training_step()*, return whatever you need in that step:

- loss -> tensor scalar **REQUIRED**
- progress_bar -> Dict for progress bar display. Must have only tensors
- log -> Dict of metrics to add to logger. Must have only tensors (no images, etc)

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

### Examples

```
def training_step(self, batch, batch_idx):
    x, y, z = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, x)

    logger_logs = {'training_loss': loss} # optional (MUST ALL BE TENSORS)

    # if using TestTubeLogger or TensorBoardLogger you can nest scalars
    logger_logs = {'losses': logger_logs} # optional (MUST ALL BE TENSORS)

    output = {
        'loss': loss, # required
        'progress_bar': {'training_loss': loss}, # optional (MUST ALL BE␣
↪TENSORS)
        'log': logger_logs
    }

    # return a dict
    return output
```

If you define multiple optimizers, this step will be called with an additional `optimizer_idx` parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
    if optimizer_idx == 1:
        # do training_step with decoder
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    ...
    out, hiddens = self.lstm(data, hiddens)
    ...

    return {
```

(continues on next page)

(continued from previous page)

```
        "loss": ...,
        "hiddens": hiddens  # remember to detach() this
    }
```

### Notes

The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

**training_step_end**(**args*, ***kwargs*)

Use this when training with dp or ddp2 because `training_step()` will operate on only part of the batch. However, this is still optional and only needed for things like softmax or NCE loss.

**Note:** If you later switch to ddp or some other mode, this will still be called so that you don't have to change your code

```
# pseudocode
sub_batches = split_batches_for_dp(batch)
batch_parts_outputs = [training_step(sub_batch) for sub_batch in sub_batches]
training_step_end(batch_parts_outputs)
```

**Parameters** **batch_parts_outputs** – What you return in *training_step* for each batch part.

**Return type** `Dict`[`str`, `Union`[`Tensor`, `Dict`[`str`, `Tensor`]]]

**Returns**

Dict with loss key and optional log or progress bar keys.

- loss -> tensor scalar **REQUIRED**
- progress_bar -> Dict for progress bar display. Must have only tensors
- log -> Dict of metrics to add to logger. Must have only tensors (no images, etc)

### Examples

```
# WITHOUT training_step_end
# if used in DP or DDP2, this batch is 1/num_gpus large
def training_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self(x)
    loss = self.softmax(out)
    loss = nce_loss(loss)
    return {'loss': loss}

# --------------
# with training_step_end to do softmax over the full batch
def training_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch
```

(continues on next page)

(continued from previous page)

```
        out = self(x)
        return {'out': out}

    def training_step_end(self, outputs):
        # this out is now the full size of the batch
        out = outputs['out']

        # this softmax now uses the full batch size
        loss = nce_loss(loss)
        return {'loss': loss}
```

**See also:**

See the *Multi-GPU training* guide for more details.

**unfreeze**()

Unfreeze all parameters for training.

```
model = MyLightningModule(...)
model.unfreeze()
```

**Return type** None

**val_dataloader**()

Implement one or multiple PyTorch DataLoaders for validation.

The dataloader you return will not be called every epoch unless you set *reload_dataloaders_every_epoch* to True.

It's recommended that all data downloads and preparation happen in *prepare_data()*.

- *fit()*
- ...
- *prepare_data()*
- *train_dataloader()*
- *val_dataloader()*
- *test_dataloader()*

**Note:** Lightning adds the correct sampler for distributed and arbitrary hardware There is no need to set it yourself.

**Return type** Union[DataLoader, List[DataLoader]]

**Returns** Single or multiple PyTorch DataLoaders.

### Examples

```
def val_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=False,
                    transform=transform, download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=False
    )

    return loader

# can also return multiple dataloaders
def val_dataloader(self):
    return [loader_a, loader_b, ..., loader_n]
```

**Note:** If you don't need a validation dataset and a `validation_step()`, you don't need to implement this method.

**Note:** In the case where you return multiple validation dataloaders, the `validation_step()` will have an argument `dataset_idx` which matches the order here.

**validation_end**(*outputs*)

**Warning:** Deprecated in v0.7.0. Use `validation_epoch_end()` instead. Will be removed in 1.0.0.

**validation_epoch_end**(*outputs*)

Called at the end of the validation epoch with the outputs of all validation steps.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

**Parameters** **outputs** (`Union`[`List`[`Dict`[`str`, `Tensor`]], `List`[`List`[`Dict`[`str`, `Tensor`]]]]) – List of outputs you defined in `validation_step()`, or if there are multiple dataloaders, a list containing a list of outputs for each dataloader.

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

**Returns**

Dict or OrderedDict. May have the following optional keys:

- progress_bar (dict for progress bar display; only tensors)
- log (dict of metrics to add to logger; only tensors).

**Note:** If you didn't define a `validation_step()`, this won't be called.

- The outputs here are strictly for logging or progress bar.
- If you don't need to display anything, don't return anything.
- If you want to manually set current step, you can specify the 'step' key in the 'log' dict.

### Examples

With a single dataloader:

```
def validation_epoch_end(self, outputs):
    val_acc_mean = 0
    for output in outputs:
        val_acc_mean += output['val_acc']

    val_acc_mean /= len(outputs)
    tqdm_dict = {'val_acc': val_acc_mean.item()}

    # show val_acc in progress bar but only log val_loss
    results = {
        'progress_bar': tqdm_dict,
        'log': {'val_acc': val_acc_mean.item()}
    }
    return results
```

With multiple dataloaders, *outputs* will be a list of lists. The outer list contains one entry per dataloader, while the inner list contains the individual outputs of each validation step for that dataloader.

```
def validation_epoch_end(self, outputs):
    val_acc_mean = 0
    i = 0
    for dataloader_outputs in outputs:
        for output in dataloader_outputs:
            val_acc_mean += output['val_acc']
            i += 1

    val_acc_mean /= i
    tqdm_dict = {'val_acc': val_acc_mean.item()}

    # show val_loss and val_acc in progress bar but only log val_loss
    results = {
        'progress_bar': tqdm_dict,
        'log': {'val_acc': val_acc_mean.item(), 'step': self.current_epoch}
    }
    return results
```

**validation_step**(**args*, ***kwargs*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
```

(continues on next page)

(continued from previous page)

```
    out = validation_step(train_batch)
    val_outs.append(out)
    validation_epoch_end(val_outs)
```

**Parameters**

- **batch** (Tensor | (Tensor, ...) | [Tensor, ...]) – The output of your DataLoader. A tensor, tuple or list.
- **batch_idx** (*int*) – The index of this batch
- **dataloader_idx** (*int*) – The index of the dataloader that produced this batch (only if multiple val datasets used)

**Return type** Dict[str, Tensor]

**Returns** Dict or OrderedDict - passed to *validation_epoch_end()*. If you defined *validation_step_end()* it will go to that first.

```
# pseudocode of order
out = validation_step()
if defined('validation_step_end'):
    out = validation_step_end(out)
out = validation_epoch_end(out)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx)

# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx)
```

### Examples

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # all optional...
    # return whatever you need for the collation function validation_epoch_end
    output = OrderedDict({
        'val_loss': loss_val,
```

(continues on next page)

(continued from previous page)

```
        'val_acc': torch.tensor(val_acc), # everything must be a tensor
    })

    # return an optional dict
    return output
```

If you pass in multiple val datasets, validation_step will have an additional argument.

```
# CASE 2: multiple validation datasets
def validation_step(self, batch, batch_idx, dataset_idx):
    # dataset_idx tells you which dataset this is.
```

**Note:** If you don't need to validate you don't need to implement this method.

**Note:** When the *validation_step()* is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

**validation_step_end**(**args*, ***kwargs*)

Use this when validating with dp or ddp2 because *validation_step()* will operate on only part of the batch. However, this is still optional and only needed for things like softmax or NCE loss.

**Note:** If you later switch to ddp or some other mode, this will still be called so that you don't have to change your code.

```
# pseudocode
sub_batches = split_batches_for_dp(batch)
batch_parts_outputs = [validation_step(sub_batch) for sub_batch in sub_
→batches]
validation_step_end(batch_parts_outputs)
```

**Parameters** **batch_parts_outputs** – What you return in *validation_step()* for each batch part.

**Return type** Dict[str, Tensor]

**Returns** Dict or OrderedDict - passed to the *validation_epoch_end()* method.

### Examples

```
# WITHOUT validation_step_end
# if used in DP or DDP2, this batch is 1/num_gpus large
def validation_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self(x)
    loss = self.softmax(out)
    loss = nce_loss(loss)
```

(continues on next page)

(continued from previous page)

```
    return {'loss': loss}

# ---------------
# with validation_step_end to do softmax over the full batch
def validation_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self(x)
    return {'out': out}

def validation_epoch_end(self, outputs):
    # this out is now the full size of the batch
    out = outputs['out']

    # this softmax now uses the full batch size
    loss = nce_loss(loss)
    return {'loss': loss}
```

**See also:**

See the *Multi-GPU training* guide for more details.

**_device = None**
device reference

**_dtype = None**
Current dtype

**current_epoch = None**
The current epoch

**global_step = None**
Total training batches seen across all epochs

**logger = None**
Pointer to the logger object

**property on_gpu**
True if your model is currently running on GPUs. Useful to set flags around the LightningModule for different CPU vs GPU behavior.

**trainer = None**
Pointer to the trainer object

**use_amp = None**
True if using amp

**use_ddp = None**
True if using ddp

**use_ddp2 = None**
True if using ddp2

**use_dp = None**
True if using dp

pytorch_lightning.core.**data_loader**(*fn*)
Decorator to make any fx with this use the lazy property.

> **Warning:** This decorator deprecated in v0.7.0 and it will be removed v0.9.0.

CHAPTER

SIX

# LOGGERS

Lightning supports the most popular logging frameworks (TensorBoard, Comet, Weights and Biases, etc…). To use a logger, simply pass it into the *Trainer*. Lightning uses TensorBoard by default.

```
from pytorch_lightning import Trainer
from pytorch_lightning import loggers
tb_logger = loggers.TensorBoardLogger('logs/')
trainer = Trainer(logger=tb_logger)
```

Choose from any of the others such as MLflow, Comet, Neptune, WandB, …

```
comet_logger = loggers.CometLogger(save_dir='logs/')
trainer = Trainer(logger=comet_logger)
```

To use multiple loggers, simply pass in a `list` or `tuple` of loggers …

```
tb_logger = loggers.TensorBoardLogger('logs/')
comet_logger = loggers.CometLogger(save_dir='logs/')
trainer = Trainer(logger=[tb_logger, comet_logger])
```

**Note:** All loggers log by default to `os.getcwd()`. To change the path without creating a logger set `Trainer(default_root_dir='/your/path/to/save/checkpoints')`

## 6.1 Custom Logger

You can implement your own logger by writing a class that inherits from `LightningLoggerBase`. Use the `rank_zero_only()` decorator to make sure that only the first process in DDP training logs data.

```
from pytorch_lightning.utilities import rank_zero_only
from pytorch_lightning.loggers import LightningLoggerBase
class MyLogger(LightningLoggerBase):

    @rank_zero_only
    def log_hyperparams(self, params):
        # params is an argparse.Namespace
        # your code to record hyperparameters goes here
        pass

    @rank_zero_only
    def log_metrics(self, metrics, step):
        # metrics is a dictionary of metric names and values
```

(continues on next page)

(continued from previous page)

```
        # your code to record metrics goes here
        pass

    def save(self):
        # Optional. Any code necessary to save logger data goes here
        pass

    @rank_zero_only
    def finalize(self, status):
        # Optional. Any code that needs to be run after training
        # finishes goes here
        pass
```

If you write a logger that may be useful to others, please send a pull request to add it to Lighting!

---

## 6.2 Using loggers

Call the logger anywhere except `__init__` in your `LightningModule` by doing:

```
from pytorch_lightning import LightningModule
class LitModel(LightningModule):
    def training_step(self, batch, batch_idx):
        # example
        self.logger.experiment.whatever_method_summary_writer_supports(...)

        # example if logger is a tensorboard logger
        self.logger.experiment.add_image('images', grid, 0)
        self.logger.experiment.add_graph(model, images)

    def any_lightning_module_function_or_hook(self):
        self.logger.experiment.add_histogram(...)
```

Read more in the Experiment Logging use case.

---

## 6.3 Supported Loggers

The following are loggers we support

### 6.3.1 Comet

**class** `pytorch_lightning.loggers.comet.`**`CometLogger`**(*api_key=None*, *save_dir=None*, *workspace=None*, *project_name=None*, *rest_api_key=None*, *experiment_name=None*, *experiment_key=None*, ***kwargs*)

Bases: `pytorch_lightning.loggers.base.LightningLoggerBase`

Log using Comet.ml. Install it with pip:

```
pip install comet-ml
```

Comet requires either an API Key (online mode) or a local directory path (offline mode).

**ONLINE MODE**

### Example

```
>>> import os
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.loggers import CometLogger
>>> # arguments made to CometLogger are passed on to the comet_ml.Experiment class
>>> comet_logger = CometLogger(
...     api_key=os.environ.get('COMET_API_KEY'),
...     workspace=os.environ.get('COMET_WORKSPACE'),  # Optional
...     save_dir='.',  # Optional
...     project_name='default_project',  # Optional
...     rest_api_key=os.environ.get('COMET_REST_API_KEY'),  # Optional
...     experiment_name='default'  # Optional
... )
>>> trainer = Trainer(logger=comet_logger)
```

**OFFLINE MODE**

### Example

```
>>> from pytorch_lightning.loggers import CometLogger
>>> # arguments made to CometLogger are passed on to the comet_ml.Experiment class
>>> comet_logger = CometLogger(
...     save_dir='.',
...     workspace=os.environ.get('COMET_WORKSPACE'),  # Optional
...     project_name='default_project',  # Optional
...     rest_api_key=os.environ.get('COMET_REST_API_KEY'),  # Optional
...     experiment_name='default'  # Optional
... )
>>> trainer = Trainer(logger=comet_logger)
```

**Parameters**

- **api_key** (`Optional[str]`) – Required in online mode. API key, found on Comet.ml
- **save_dir** (`Optional[str]`) – Required in offline mode. The path for the directory to save local comet logs
- **workspace** (`Optional[str]`) – Optional. Name of workspace for this user
- **project_name** (`Optional[str]`) – Optional. Send your experiment to a specific project. Otherwise will be sent to Uncategorized Experiments. If the project name does not already exist, Comet.ml will create a new project.
- **rest_api_key** (`Optional[str]`) – Optional. Rest API key found in Comet.ml settings. This is used to determine version number
- **experiment_name** (`Optional[str]`) – Optional. String representing the name for this particular experiment on Comet.ml.

- **experiment_key** (Optional[str]) – Optional. If set, restores from existing experiment.

**finalize**(*status*)
When calling `self.experiment.end()`, that experiment won't log any more data to Comet. That's why, if you need to log any more data, you need to create an ExistingCometExperiment. For example, to log data when testing your model after training, because when training is finalized *CometLogger.finalize()* is called.

This happens automatically in the *experiment()* property, when `self._experiment` is set to `None`, i.e. `self.reset_experiment()`.

**Return type** None

**log_hyperparams**(*params*)
Record hyperparameters.

**Parameters params** (Union[Dict[str, Any], Namespace]) – Namespace containing the hyperparameters

**Return type** None

**log_metrics**(*metrics*, *step=None*)
Records metrics. This method logs metrics as as soon as it received them. If you want to aggregate metrics for one specific *step*, use the *agg_and_log_metrics()* method.

**Parameters**

- **metrics** (Dict[str, Union[Tensor, float]]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded

**Return type** None

**property experiment**
Actual Comet object. To use Comet features in your *LightningModule* do the following.

Example:

```
self.logger.experiment.some_comet_function()
```

**Return type** BaseExperiment

**property name**
Return the experiment name.

**Return type** str

**property version**
Return the experiment version.

**Return type** str

### 6.3.2 MLFlow

**class** pytorch_lightning.loggers.mlflow.**MLFlowLogger**(*experiment_name='default'*, *tracking_uri=None*, *tags=None*, *save_dir=None*)

Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Log using MLflow. Install it with pip:

```
pip install mlflow
```

#### Example

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.loggers import MLFlowLogger
>>> mlf_logger = MLFlowLogger(
...     experiment_name="default",
...     tracking_uri="file:./ml-runs"
... )
>>> trainer = Trainer(logger=mlf_logger)
```

Use the logger anywhere in you *LightningModule* as follows:

```
>>> from pytorch_lightning import LightningModule
>>> class LitModel(LightningModule):
...     def training_step(self, batch, batch_idx):
...         # example
...         self.logger.experiment.whatever_ml_flow_supports(...)
...
...     def any_lightning_module_function_or_hook(self):
...         self.logger.experiment.whatever_ml_flow_supports(...)
```

**Parameters**

- **experiment_name** (str) – The name of the experiment
- **tracking_uri** (Optional[str]) – Address of local or remote tracking server. If not provided, defaults to the service set by mlflow.tracking.set_tracking_uri.
- **tags** (Optional[Dict[str, Any]]) – A dictionary tags for the experiment.

**finalize**(*status='FINISHED'*)

Do any processing that is necessary to finalize an experiment.

**Parameters** **status** (str) – Status that the experiment finished with (e.g. success, failed, aborted)

**Return type** None

**log_hyperparams**(*params*)

Record hyperparameters.

**Parameters** **params** (Union[Dict[str, Any], Namespace]) – Namespace containing the hyperparameters

**Return type** None

**log_metrics**(*metrics*, *step=None*)

Records metrics. This method logs metrics as as soon as it received them. If you want to aggregate metrics for one specific *step*, use the *agg_and_log_metrics()* method.

**Parameters**

- **metrics** (Dict[str, float]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded

**Return type** None

**property experiment**

Actual MLflow object. To use mlflow features in your *LightningModule* do the following.

Example:

```
self.logger.experiment.some_mlflow_function()
```

**Return type** MlflowClient

**property name**

Return the experiment name.

**Return type** str

**property version**

Return the experiment version.

**Return type** str

### 6.3.3 Neptune

**class** pytorch_lightning.loggers.neptune.**NeptuneLogger**(*api_key=None*, *project_name=None*, *close_after_fit=True*, *offline_mode=False*, *experiment_name=None*, *upload_source_files=None*, *params=None*, *properties=None*, *tags=None*, ***kwargs*)

Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Log using Neptune. Install it with pip:

```
pip install neptune-client
```

The Neptune logger can be used in the online mode or offline (silent) mode. To log experiment data in online mode, *NeptuneLogger* requires an API key. In offline mode, the logger does not connect to Neptune.

**ONLINE MODE**

### Example

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.loggers import NeptuneLogger
>>> # arguments made to NeptuneLogger are passed on to the neptune.experiments.
↪Experiment class
>>> # We are using an api_key for the anonymous user "neptuner" but you can use␣
↪your own.
>>> neptune_logger = NeptuneLogger(
...     api_key='ANONYMOUS',
...     project_name='shared/pytorch-lightning-integration',
...     experiment_name='default',  # Optional,
...     params={'max_epochs': 10},  # Optional,
...     tags=['pytorch-lightning', 'mlp']  # Optional,
... )
>>> trainer = Trainer(max_epochs=10, logger=neptune_logger)
```

**OFFLINE MODE**

### Example

```
>>> from pytorch_lightning.loggers import NeptuneLogger
>>> # arguments made to NeptuneLogger are passed on to the neptune.experiments.
↪Experiment class
>>> neptune_logger = NeptuneLogger(
...     offline_mode=True,
...     project_name='USER_NAME/PROJECT_NAME',
...     experiment_name='default',  # Optional,
...     params={'max_epochs': 10},  # Optional,
...     tags=['pytorch-lightning', 'mlp']  # Optional,
... )
>>> trainer = Trainer(max_epochs=10, logger=neptune_logger)
```

Use the logger anywhere in you *LightningModule* as follows:

```
>>> from pytorch_lightning import LightningModule
>>> class LitModel(LightningModule):
...     def training_step(self, batch, batch_idx):
...         # log metrics
...         self.logger.experiment.log_metric('acc_train', ...)
...         # log images
...         self.logger.experiment.log_image('worse_predictions', ...)
...         # log model checkpoint
...         self.logger.experiment.log_artifact('model_checkpoint.pt', ...)
...         self.logger.experiment.whatever_neptune_supports(...)
...
...     def any_lightning_module_function_or_hook(self):
...         self.logger.experiment.log_metric('acc_train', ...)
...         self.logger.experiment.log_image('worse_predictions', ...)
...         self.logger.experiment.log_artifact('model_checkpoint.pt', ...)
...         self.logger.experiment.whatever_neptune_supports(...)
```

If you want to log objects after the training is finished use `close_after_fit=False`:

```
neptune_logger = NeptuneLogger(
    ...
```

(continues on next page)

(continued from previous page)

```
    close_after_fit=False,
    ...
)
trainer = Trainer(logger=neptune_logger)
trainer.fit()

# Log test metrics
trainer.test(model)

# Log additional metrics
from sklearn.metrics import accuracy_score

accuracy = accuracy_score(y_true, y_pred)
neptune_logger.experiment.log_metric('test_accuracy', accuracy)

# Log charts
from scikitplot.metrics import plot_confusion_matrix
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(16, 12))
plot_confusion_matrix(y_true, y_pred, ax=ax)
neptune_logger.experiment.log_image('confusion_matrix', fig)

# Save checkpoints folder
neptune_logger.experiment.log_artifact('my/checkpoints')

# When you are done, stop the experiment
neptune_logger.experiment.stop()
```

**See also:**

- An Example experiment showing the UI of Neptune.
- Tutorial on how to use Pytorch Lightning with Neptune.

**Parameters**

- **api_key** (`Optional[str]`) – Required in online mode. Neptune API token, found on https://neptune.ai. Read how to get your API key. It is recommended to keep it in the *NEPTUNE_API_TOKEN* environment variable and then you can leave `api_key=None`.
- **project_name** (`Optional[str]`) – Required in online mode. Qualified name of a project in a form of "namespace/project_name" for example "tom/minst-classification". If `None`, the value of *NEPTUNE_PROJECT* environment variable will be taken. You need to create the project in https://neptune.ai first.
- **offline_mode** (`bool`) – Optional default `False`. If `True` no logs will be sent to Neptune. Usually used for debug purposes.
- **close_after_fit** (`Optional[bool]`) – Optional default `True`. If `False` the experiment will not be closed after training and additional metrics, images or artifacts can be logged. Also, remember to close the experiment explicitly by running `neptune_logger.experiment.stop()`.
- **experiment_name** (`Optional[str]`) – Optional. Editable name of the experiment. Name is displayed in the experiment's Details (Metadata section) and in experiments view as a column.

- **upload_source_files** (Optional[List[str]]) – Optional. List of source files to be uploaded. Must be list of str or single str. Uploaded sources are displayed in the experiment's Source code tab. If `None` is passed, the Python file from which the experiment was created will be uploaded. Pass an empty list (`[]`) to upload no files. Unix style pathname pattern expansion is supported. For example, you can pass `'\*.py'` to upload all python source files from the current directory. For recursion lookup use `'\**/\*.py'` (for Python 3.5 and later). For more information see `glob` library.
- **params** (Optional[Dict[str, Any]]) – Optional. Parameters of the experiment. After experiment creation params are read-only. Parameters are displayed in the experiment's Parameters section and each key-value pair can be viewed in the experiments view as a column.
- **properties** (Optional[Dict[str, Any]]) – Optional. Default is `{}`. Properties of the experiment. They are editable after the experiment is created. Properties are displayed in the experiment's Details section and each key-value pair can be viewed in the experiments view as a column.
- **tags** (Optional[List[str]]) – Optional. Default is `[]`. Must be list of str. Tags of the experiment. They are editable after the experiment is created (see: `append_tag()` and `remove_tag()`). Tags are displayed in the experiment's Details section and can be viewed in the experiments view as a column.

**append_tags**(*tags*)
Appends tags to the neptune experiment.

**Parameters** **tags** (Union[str, Iterable[str]]) – Tags to add to the current experiment. If str is passed, a single tag is added. If multiple - comma separated - str are passed, all of them are added as tags. If list of str is passed, all elements of the list are added as tags.

**Return type** `None`

**finalize**(*status*)
Do any processing that is necessary to finalize an experiment.

**Parameters** **status** (str) – Status that the experiment finished with (e.g. success, failed, aborted)

**Return type** `None`

**log_artifact**(*artifact*, *destination=None*)
Save an artifact (file) in Neptune experiment storage.

**Parameters**

- **artifact** (str) – A path to the file in local filesystem.
- **destination** (Optional[str]) – Optional. Default is `None`. A destination path. If `None` is passed, an artifact file name will be used.

**Return type** `None`

**log_hyperparams**(*params*)
Record hyperparameters.

**Parameters** **params** (Union[Dict[str, Any], Namespace]) – Namespace containing the hyperparameters

**Return type** `None`

**log_image**(*log_name*, *image*, *step=None*)
Log image data in Neptune experiment

**Parameters**

- **log_name** (str) – The name of log, i.e. bboxes, visualisations, sample_images.
- **image** (Union[str, Image, Any]) – The value of the log (data-point). Can be one of the following types: PIL image, *matplotlib.figure.Figure*, path to image file (str)
- **step** (Optional[int]) – Step number at which the metrics should be recorded, must be strictly increasing

**Return type** None

**log_metric**(*metric_name*, *metric_value*, *step=None*)

Log metrics (numeric values) in Neptune experiments.

**Parameters**

- **metric_name** (str) – The name of log, i.e. mse, loss, accuracy.
- **metric_value** (Union[Tensor, float, str]) – The value of the log (data-point).
- **step** (Optional[int]) – Step number at which the metrics should be recorded, must be strictly increasing

**Return type** None

**log_metrics**(*metrics*, *step=None*)

Log metrics (numeric values) in Neptune experiments.

**Parameters**

- **metrics** (Dict[str, Union[Tensor, float]]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded, must be strictly increasing

**Return type** None

**log_text**(*log_name*, *text*, *step=None*)

Log text data in Neptune experiments.

**Parameters**

- **log_name** (str) – The name of log, i.e. mse, my_text_data, timing_info.
- **text** (str) – The value of the log (data-point).
- **step** (Optional[int]) – Step number at which the metrics should be recorded, must be strictly increasing

**Return type** None

**set_property**(*key*, *value*)

Set key-value pair as Neptune experiment property.

**Parameters**

- **key** (str) – Property key.
- **value** (Any) – New value of a property.

**Return type** None

**property experiment**

Actual Neptune object. To use neptune features in your *LightningModule* do the following.

Example:

```
self.logger.experiment.some_neptune_function()
```

**Return type** `Experiment`

**property name**
Return the experiment name.

**Return type** `str`

**property version**
Return the experiment version.

**Return type** `str`

### 6.3.4 Tensorboard

**class** `pytorch_lightning.loggers.tensorboard.`**TensorBoardLogger**(*save_dir*, *name='default'*, *version=None*, ***kwargs*)

Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Log to local file system in TensorBoard format. Implemented using `SummaryWriter`. Logs are saved to `os.path.join(save_dir, name, version)`. This is the default logger in Lightning, it comes preinstalled.

#### Example

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.loggers import TensorBoardLogger
>>> logger = TensorBoardLogger("tb_logs", name="my_model")
>>> trainer = Trainer(logger=logger)
```

**Parameters**

- **save_dir** (`str`) – Save directory
- **name** (`Optional[str]`) – Experiment name. Defaults to `'default'`. If it is the empty string then no per-experiment subdirectory is used.
- **version** (`Union[int, str, None]`) – Experiment version. If version is not specified the logger inspects the save directory for existing versions, then automatically assigns the next available version. If it is a string then it is used as the run-specific subdirectory name, otherwise `'version_${version}'` is used.
- ****kwargs** – Other arguments are passed directly to the `SummaryWriter` constructor.

**finalize**(*status*)
Do any processing that is necessary to finalize an experiment.

**Parameters status** (`str`) – Status that the experiment finished with (e.g. success, failed, aborted)

**Return type** `None`

**log_hyperparams**(*params*, *metrics=None*)
Record hyperparameters.

**Parameters** **params** (Union[Dict[str, Any], Namespace]) – Namespace containing the hyperparameters

**Return type** None

**log_metrics** (*metrics*, *step=None*)

Records metrics. This method logs metrics as as soon as it received them. If you want to aggregate metrics for one specific *step*, use the *agg_and_log_metrics()* method.

**Parameters**

- **metrics** (Dict[str, float]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded

**Return type** None

**save** ()

Save log data.

**Return type** None

**property experiment**

Actual tensorboard object. To use TensorBoard features in your *LightningModule* do the following.

Example:

```
self.logger.experiment.some_tensorboard_function()
```

**Return type** SummaryWriter

**property log_dir**

The directory for this run's tensorboard checkpoint. By default, it is named `'version_${self.version}'` but it can be overridden by passing a string value for the constructor's version parameter instead of `None` or an int.

**Return type** str

**property name**

Return the experiment name.

**Return type** str

**property root_dir**

Parent directory for all tensorboard checkpoint subdirectories. If the experiment name parameter is `None` or the empty string, no experiment subdirectory is used and the checkpoint will be saved in "save_dir/version_dir"

**Return type** str

**property version**

Return the experiment version.

**Return type** int

### 6.3.5 Test-tube

**class** pytorch_lightning.loggers.test_tube.**TestTubeLogger**(*save_dir*, *name='default'*, *description=None*, *debug=False*, *version=None*, *create_git_tag=False*)

Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Log to local file system in TensorBoard format but using a nicer folder structure (see full docs). Install it with pip:

```
pip install test_tube
```

#### Example

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.loggers import TestTubeLogger
>>> logger = TestTubeLogger("tt_logs", name="my_exp_name")
>>> trainer = Trainer(logger=logger)
```

Use the logger anywhere in your *LightningModule* as follows:

```
>>> from pytorch_lightning import LightningModule
>>> class LitModel(LightningModule):
...     def training_step(self, batch, batch_idx):
...         # example
...         self.logger.experiment.whatever_method_summary_writer_supports(...)
...
...     def any_lightning_module_function_or_hook(self):
...         self.logger.experiment.add_histogram(...)
```

**Parameters**

- **save_dir** (str) – Save directory
- **name** (str) – Experiment name. Defaults to 'default'.
- **description** (Optional[str]) – A short snippet about this experiment
- **debug** (bool) – If True, it doesn't log anything.
- **version** (Optional[int]) – Experiment version. If version is not specified the logger inspects the save directory for existing versions, then automatically assigns the next available version.
- **create_git_tag** (bool) – If True creates a git tag to save the code used in this experiment.

**close**()

Do any cleanup that is necessary to close an experiment.

**Return type** None

**finalize**(*status*)

Do any processing that is necessary to finalize an experiment.

**Parameters** **status** (str) – Status that the experiment finished with (e.g. success, failed, aborted)

**Return type** None

**log_hyperparams**(*params*)
Record hyperparameters.

**Parameters params** (Union[Dict[str, Any], Namespace]) – Namespace containing the hyperparameters

**Return type** None

**log_metrics**(*metrics*, *step=None*)
Records metrics. This method logs metrics as as soon as it received them. If you want to aggregate metrics for one specific *step*, use the *agg_and_log_metrics()* method.

**Parameters**

- **metrics** (Dict[str, float]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded

**Return type** None

**save**()
Save log data.

**Return type** None

**property experiment**
Actual TestTube object. To use TestTube features in your *LightningModule* do the following.

Example:

```
self.logger.experiment.some_test_tube_function()
```

**Return type** Experiment

**property name**
Return the experiment name.

**Return type** str

**property version**
Return the experiment version.

**Return type** int

### 6.3.6 Trains

**class** pytorch_lightning.loggers.trains.**TrainsLogger**(*project_name=None*, *task_name=None*, *task_type='training'*, *reuse_last_task_id=True*, *output_uri=None*, *auto_connect_arg_parser=True*, *auto_connect_frameworks=True*, *auto_resource_monitoring=True*)
Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Log using allegro.ai TRAINS. Install it with pip:

```
pip install trains
```

### Example

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.loggers import TrainsLogger
>>> trains_logger = TrainsLogger(
...     project_name='pytorch lightning',
...     task_name='default',
...     output_uri='.',
... )
TRAINS Task: ...
TRAINS results page: ...
>>> trainer = Trainer(logger=trains_logger)
```

Use the logger anywhere in your *LightningModule* as follows:

```
>>> from pytorch_lightning import LightningModule
>>> class LitModel(LightningModule):
...     def training_step(self, batch, batch_idx):
...         # example
...         self.logger.experiment.whatever_trains_supports(...)
...
...     def any_lightning_module_function_or_hook(self):
...         self.logger.experiment.whatever_trains_supports(...)
```

**Parameters**

- **project_name** (Optional[str]) – The name of the experiment's project. Defaults to None.
- **task_name** (Optional[str]) – The name of the experiment. Defaults to None.
- **task_type** (str) – The name of the experiment. Defaults to 'training'.
- **reuse_last_task_id** (bool) – Start with the previously used task id. Defaults to True.
- **output_uri** (Optional[str]) – Default location for output models. Defaults to None.
- **auto_connect_arg_parser** (bool) – Automatically grab the ArgumentParser and connect it with the task. Defaults to True.
- **auto_connect_frameworks** (bool) – If True, automatically patch to trains backend. Defaults to True.
- **auto_resource_monitoring** (bool) – If True, machine vitals will be sent along side the task scalars. Defaults to True.

### Examples

```
>>> logger = TrainsLogger("pytorch lightning", "default", output_uri=".")
TRAINS Task: ...
TRAINS results page: ...
>>> logger.log_metrics({"val_loss": 1.23}, step=0)
>>> logger.log_text("sample test")
sample test
>>> import numpy as np
>>> logger.log_artifact("confusion matrix", np.ones((2, 3)))
>>> logger.log_image("passed", "Image 1", np.random.randint(0, 255, (200, 150, 3),
↪ dtype=np.uint8))
```

**classmethod bypass_mode**()
Returns the bypass mode state.

**Note:** *GITHUB_ACTIONS* env will automatically set bypass_mode to `True` unless overridden specifically with `TrainsLogger.set_bypass_mode(False)`.

**Return type** `bool`

**Returns** If True, all outside communication is skipped.

**finalize**(*status=None*)
Do any processing that is necessary to finalize an experiment.

**Parameters** **status** (`Optional`[`str`]) – Status that the experiment finished with (e.g. success, failed, aborted)

**Return type** `None`

**log_artifact**(*name*, *artifact*, *metadata=None*, *delete_after_upload=False*)
Save an artifact (file/object) in TRAINS experiment storage.

**Parameters**

- **name** (`str`) – Artifact name. Notice! it will override the previous artifact if the name already exists.
- **artifact** (`Union`[`str`, `Path`, `Dict`[`str`, `Any`], `ndarray`, `Image`]) – Artifact object to upload. Currently supports:
  - string / `pathlib.Path` are treated as path to artifact file to upload If a wildcard or a folder is passed, a zip file containing the local files will be created and uploaded.
  - dict will be stored as .json file and uploaded
  - `pandas.DataFrame` will be stored as .csv.gz (compressed CSV file) and uploaded
  - `numpy.ndarray` will be stored as .npz and uploaded
  - `PIL.Image.Image` will be stored to .png file and uploaded
- **metadata** (`Optional`[`Dict`[`str`, `Any`]]) – Simple key/value dictionary to store on the artifact. Defaults to `None`.
- **delete_after_upload** (`bool`) – If `True`, the local artifact will be deleted (only applies if `artifact` is a local file). Defaults to `False`.

**Return type** `None`

**log_hyperparams** (*params*)
Log hyperparameters (numeric values) in TRAINS experiments.

**Parameters** **params** (Union[Dict[str, Any], Namespace]) – The hyperparameters that passed through the model.

**Return type** None

**log_image** (*title*, *series*, *image*, *step=None*)
Log Debug image in TRAINS experiment

**Parameters**

- **title** (str) – The title of the debug image, i.e. "failed", "passed".
- **series** (str) – The series name of the debug image, i.e. "Image 0", "Image 1".
- **image** (Union[str, ndarray, Image, Tensor]) – Debug image to log. If numpy.ndarray or torch.Tensor, the image is assumed to be the following:
  - shape: CHW
  - color space: RGB
  - value range: [0., 1.] (float) or [0, 255] (uint8)
- **step** (Optional[int]) – Step number at which the metrics should be recorded. Defaults to None.

**Return type** None

**log_metric** (*title*, *series*, *value*, *step=None*)
Log metrics (numeric values) in TRAINS experiments. This method will be called by the users.

**Parameters**

- **title** (str) – The title of the graph to log, e.g. loss, accuracy.
- **series** (str) – The series name in the graph, e.g. classification, localization.
- **value** (float) – The value to log.
- **step** (Optional[int]) – Step number at which the metrics should be recorded. Defaults to None.

**Return type** None

**log_metrics** (*metrics*, *step=None*)
Log metrics (numeric values) in TRAINS experiments. This method will be called by Trainer.

**Parameters**

- **metrics** (Dict[str, float]) – The dictionary of the metrics. If the key contains "/", it will be split by the delimiter, then the elements will be logged as "title" and "series" respectively.
- **step** (Optional[int]) – Step number at which the metrics should be recorded. Defaults to None.

**Return type** None

**log_text** (*text*)
Log console text data in TRAINS experiment.

**Parameters** **text** (str) – The value of the log (data-point).

**Return type** None

**classmethod set_bypass_mode**(*bypass*)
Will bypass all outside communication, and will drop all logs. Should only be used in "standalone mode", when there is no access to the *trains-server*.

**Parameters** **bypass** (`bool`) – If `True`, all outside communication is skipped.

**Return type** `None`

**classmethod set_credentials**(*api_host=None*, *web_host=None*, *files_host=None*, *key=None*, *secret=None*)
Set new default TRAINS-server host and credentials. These configurations could be overridden by either OS environment variables or trains.conf configuration file.

**Note:** Credentials need to be set *prior* to Logger initialization.

**Parameters**

- **api_host** (`Optional[str]`) – Trains API server url, example: `host='http://localhost:8008'`
- **web_host** (`Optional[str]`) – Trains WEB server url, example: `host='http://localhost:8080'`
- **files_host** (`Optional[str]`) – Trains Files server url, example: `host='http://localhost:8081'`
- **key** (`Optional[str]`) – user key/secret pair, example: `key='thisisakey123'`
- **secret** (`Optional[str]`) – user key/secret pair, example: `secret='thisisseceret123'`

**Return type** `None`

**property experiment**
Actual TRAINS object. To use TRAINS features in your *LightningModule* do the following.

Example:

```
self.logger.experiment.some_trains_function()
```

**Return type** `Task`

**property id**
ID is a uuid (string) representing this specific experiment in the entire system.

**Return type** `Optional[str]`

**property name**
Name is a human readable non-unique name (str) of the experiment.

**Return type** `Optional[str]`

**property version**
Return the experiment version.

**Return type** `Optional[str]`

CHAPTER

SEVEN

# METRICS

This is a general package for PyTorch Metrics. These can also be used with regular non-lightning PyTorch code. Metrics are used to monitor model performance.

In this package we provide two major pieces of functionality.

1. A Metric class you can use to implement metrics with built-in distributed (ddp) support which are device agnostic.
2. A collection of ready to use pupular metrics. There are two types of metrics: Class metrics and Functional metrics.
3. A interface to call sklearns metrics

Example:

```
from pytorch_lightning.metrics.functional import accuracy

pred = torch.tensor([0, 1, 2, 3])
target = torch.tensor([0, 1, 2, 2])

# calculates accuracy across all GPUs and all Nodes used in training
accuracy(pred, target)
```

**Warning:** The metrics package is still in development! If we're missing a metric or you find a mistake, please send a PR! to a few metrics. Please feel free to create an issue/PR if you have a proposed metric or have found a bug.

## 7.1 Implement a metric

You can implement metrics as either a PyTorch metric or a Numpy metric (It is recommend to use PyTorch metrics when possible, since Numpy metrics slow down training).

Use `TensorMetric` to implement native PyTorch metrics. This class handles automated DDP syncing and converts all inputs and outputs to tensors.

Use `NumpyMetric` to implement numpy metrics. This class handles automated DDP syncing and converts all inputs and outputs to tensors.

**Warning:** Numpy metrics might slow down your training substantially, since every metric computation requires a GPU sync to convert tensors to numpy.

### 7.1.1 TensorMetric

Here's an example showing how to implement a TensorMetric

```
class RMSE(TensorMetric):
    def forward(self, x, y):
        return torch.sqrt(torch.mean(torch.pow(x-y, 2.0)))
```

**class** pytorch_lightning.metrics.metric.**TensorMetric**(*name*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.Metric*

Base class for metric implementation operating directly on tensors. All inputs and outputs will be casted to tensors if necessary. Already handles DDP sync and input/output conversions.

**Parameters**

- **name** (str) – the metric's name
- **reduce_group** (Optional[Any]) – the process group for DDP reduces (only needed for DDP training). Defaults to all processes (world)
- **reduce_op** (Optional[Any]) – the operation to perform during reduction within DDP (only needed for DDP training). Defaults to sum.

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

Call self as a function.

**Return type** Tensor

### 7.1.2 NumpyMetric

Here's an example showing how to implement a NumpyMetric

```
class RMSE(NumpyMetric):
    def forward(self, x, y):
        return np.sqrt(np.mean(np.power(x-y, 2.0)))
```

**class** pytorch_lightning.metrics.metric.**NumpyMetric**(*name*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.Metric*

Base class for metric implementation operating on numpy arrays. All inputs will be casted to numpy if necessary and all outputs will be casted to tensors if necessary. Already handles DDP sync and input/output conversions.

**Parameters**

- **name** (str) – the metric's name
- **reduce_group** (Optional[Any]) – the process group for DDP reduces (only needed for DDP training). Defaults to all processes (world)

- **reduce_op** (Optional[Any]) – the operation to perform during reduction within DDP (only needed for DDP training). Defaults to sum.

**__call__**(**args*, ***kwargs*)
Call self as a function.

**Return type** Tensor

## 7.2 Class Metrics

Class metrics can be instantiated as part of a module definition (even with just plain PyTorch).

```
from pytorch_lightning.metrics import Accuracy

# Plain PyTorch
class MyModule(Module):
    def __init__(self):
        super().__init__()
        self.metric = Accuracy()

    def forward(self, x, y):
        y_hat = ...
        acc = self.metric(y_hat, y)

# PyTorch Lightning
class MyModule(LightningModule):
    def __init__(self):
        super().__init__()
        self.metric = Accuracy()

    def training_step(self, batch, batch_idx):
        x, y = batch
        y_hat = ...
        acc = self.metric(y_hat, y)
```

These metrics even work when using distributed training:

```
model = MyModule()
trainer = Trainer(gpus=8, num_nodes=2)

# any metric automatically reduces across GPUs (even the ones you implement using␣
↪Lightning)
trainer.fit(model)
```

### 7.2.1 Accuracy

**class** pytorch_lightning.metrics.classification.**Accuracy**(*num_classes=None*, *reduction='elementwise_mean'*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the accuracy classification score

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = Accuracy()
>>> metric(pred, target)
tensor(0.7500)
```

**Parameters**

- **num_classes** (Optional[int]) – number of classes
- **reduction** (str) – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward**(*pred*, *target*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the classification score.

## 7.2.2 AveragePrecision

**class** pytorch_lightning.metrics.classification.**AveragePrecision**(*pos_label=1*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the average precision score

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = AveragePrecision()
>>> metric(pred, target)
tensor(0.3333)
```

**Parameters**

- **pos_label** (int) – positive label indicator

- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward** (*pred*, *target*, *sample_weight=None*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – groundtruth labels
- **sample_weight** (Optional[Sequence]) – the weights per sample

**Returns** classification score

**Return type** torch.Tensor

### 7.2.3 AUROC

**class** pytorch_lightning.metrics.classification.**AUROC** (*pos_label=1*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the area under curve (AUC) of the receiver operator characteristic (ROC)

#### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = AUROC()
>>> metric(pred, target)
tensor(0.3333)
```

**Parameters**

- **pos_label** (int) – positive label indicator
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward** (*pred*, *target*, *sample_weight=None*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – groundtruth labels
- **sample_weight** (Optional[Sequence]) – the weights per sample

**Returns** classification score

**Return type** torch.Tensor

### 7.2.4 ConfusionMatrix

**class** pytorch_lightning.metrics.classification.**ConfusionMatrix**(*normalize=False*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the confusion matrix C where each entry C_{i,j} is the number of observations in group i that were predicted in group j.

#### Example

```
>>> pred = torch.tensor([0, 1, 2, 2])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = ConfusionMatrix()
>>> metric(pred, target)
tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 2.]])
```

**Parameters**

- **normalize** (bool) – whether to compute a normalized confusion matrix
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward**(*pred*, *target*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the confusion matrix.

### 7.2.5 DiceCoefficient

**class** pytorch_lightning.metrics.classification.**DiceCoefficient**(*include_background=False*, *nan_score=0.0*, *no_fg_score=0.0*, *reduction='elementwise_mean'*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the dice coefficient

### Example

```
>>> pred = torch.tensor([[0.85, 0.05, 0.05, 0.05],
...                      [0.05, 0.85, 0.05, 0.05],
...                      [0.05, 0.05, 0.85, 0.05],
...                      [0.05, 0.05, 0.05, 0.85]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> metric = DiceCoefficient()
>>> metric(pred, target)
tensor(0.3333)
```

**Parameters**

- **include_background** (bool) – whether to also compute dice for the background
- **nan_score** (float) – score to return, if a NaN occurs during computation (denom zero)
- **no_fg_score** (float) – score to return, if no foreground pixel was found in target
- **reduction** (str) – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward**(*pred*, *target*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted probability for each label
- **target** (Tensor) – groundtruth labels

**Returns** the calculated dice coefficient

**Return type** torch.Tensor

## 7.2.6 F1

**class** pytorch_lightning.metrics.classification.**F1**(*num_classes=None*, *reduction='elementwise_mean'*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the F1 score, which is the harmonic mean of the precision and recall. It ranges between 1 and 0, where 1 is perfect and the worst value is 0.

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = F1()
>>> metric(pred, target)
tensor(0.6667)
```

**Parameters**

- **num_classes** (Optional[int]) – number of classes
- **reduction** (str) – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward**(*pred*, *target*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – groundtruth labels

**Returns** classification score

**Return type** torch.Tensor

## 7.2.7 FBeta

**class** pytorch_lightning.metrics.classification.**FBeta**(*beta*, *num_classes=None*, *reduction='elementwise_mean'*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

**Computes the FBeta Score, which is the weighted harmonic mean of precision and recall.** It ranges between 1 and 0, where 1 is perfect and the worst value is 0.

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = FBeta(0.25)
>>> metric(pred, target)
tensor(0.7361)
```

**Parameters**

- **beta** (float) – determines the weight of recall in the combined score.
- **num_classes** (Optional[int]) – number of classes

- **reduction** (str) – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for DDP reduction

**forward** (*pred*, *target*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – groundtruth labels

**Returns** classification score

**Return type** torch.Tensor

### 7.2.8 PrecisionRecall

**class** pytorch_lightning.metrics.classification.**PrecisionRecall** (*pos_label=1*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorCollectionMetric*

Computes the precision recall curve

#### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = PrecisionRecall()
>>> prec, recall, thr = metric(pred, target)
>>> prec
tensor([0.3333, 0.0000, 0.0000, 1.0000])
>>> recall
tensor([1., 0., 0., 0.])
>>> thr
tensor([1., 2., 3.])
```

**Parameters**

- **pos_label** (int) – positive label indicator
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward** (*pred*, *target*, *sample_weight=None*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels

- **target** (Tensor) – groundtruth labels
- **sample_weight** (Optional[Sequence]) – the weights per sample

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

**Returns**

- precision values
- recall values
- threshold values

### 7.2.9 Precision

**class** pytorch_lightning.metrics.classification.**Precision**(*num_classes=None*, *reduction='elementwise_mean'*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the precision score

#### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = Precision(num_classes=4)
>>> metric(pred, target)
tensor(0.7500)
```

**Parameters**

- **num_classes** (Optional[int]) – number of classes
- **reduction** (str) – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward**(*pred*, *target*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the classification score.

### 7.2.10 Recall

**class** pytorch_lightning.metrics.classification.**Recall**(*num_classes=None*, *reduction='elementwise_mean'*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the recall score

#### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = Recall()
>>> metric(pred, target)
tensor(0.6250)
```

**Parameters**

- **num_classes** (Optional[int]) – number of classes
- **reduction** (str) – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward**(*pred*, *target*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the classification score.

### 7.2.11 ROC

**class** pytorch_lightning.metrics.classification.**ROC**(*pos_label=1*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorCollectionMetric*

Computes the Receiver Operator Characteristic (ROC)

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = ROC()
>>> fps, tps, thresholds = metric(pred, target)
>>> fps
tensor([0.0000, 0.3333, 0.6667, 0.6667, 1.0000])
>>> tps
tensor([0., 0., 0., 1., 1.])
>>> thresholds
tensor([4., 3., 2., 1., 0.])
```

**Parameters**

- **pos_label** (int) – positive label indicator
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward**(*pred*, *target*, *sample_weight=None*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – groundtruth labels
- **sample_weight** (Optional[Sequence]) – the weights per sample

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

**Returns**

- false positive rate
- true positive rate
- thresholds

## 7.2.12 MAE

**class** pytorch_lightning.metrics.regression.**MAE**(*reduction='elementwise_mean'*)

Bases: *pytorch_lightning.metrics.metric.Metric*

Computes the root mean absolute loss or L1-loss.

**Parameters** **reduction** (str) – a method for reducing mse over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements

### Example

```
>>> pred = torch.tensor([0., 1, 2, 3])
>>> target = torch.tensor([0., 1, 2, 2])
>>> metric = MAE()
>>> metric(pred, target)
tensor(0.2500)
```

**forward**(*pred*, *target*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the mae loss.

## 7.2.13 MSE

**class** pytorch_lightning.metrics.regression.**MSE**(*reduction='elementwise_mean'*)
Bases: *pytorch_lightning.metrics.metric.Metric*

Computes the mean squared loss.

**Parameters** **reduction** (str) – a method for reducing mse over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements

### Example

```
>>> pred = torch.tensor([0., 1, 2, 3])
>>> target = torch.tensor([0., 1, 2, 2])
>>> metric = MSE()
>>> metric(pred, target)
tensor(0.2500)
```

**forward**(*pred*, *target*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the mse loss.

### 7.2.14 MulticlassROC

**class** pytorch_lightning.metrics.classification.**MulticlassROC**(*num_classes=None*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorCollectionMetric*

Computes the multiclass ROC

#### Example

```
>>> pred = torch.tensor([[0.85, 0.05, 0.05, 0.05],
...                      [0.05, 0.85, 0.05, 0.05],
...                      [0.05, 0.05, 0.85, 0.05],
...                      [0.05, 0.05, 0.05, 0.85]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> metric = MulticlassROC()
>>> classes_roc = metric(pred, target)
>>> metric(pred, target)
((tensor([0., 0., 1.]), tensor([0., 1., 1.]), tensor([1.8500, 0.8500, 0.0500])),
 (tensor([0., 0., 1.]), tensor([0., 1., 1.]), tensor([1.8500, 0.8500, 0.0500])),
 (tensor([0.0000, 0.3333, 1.0000]), tensor([0., 0., 1.]), tensor([1.8500, 0.8500,␣
↪0.0500])),
 (tensor([0.0000, 0.3333, 1.0000]), tensor([0., 0., 1.]), tensor([1.8500, 0.8500,␣
↪0.0500])))
```

**Parameters**

- **num_classes** (Optional[int]) – number of classes
- **reduction** – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward**(*pred*, *target*, *sample_weight=None*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted probability for each label
- **target** (Tensor) – groundtruth labels
- **sample_weight** (Optional[Sequence]) – Weights for each sample defining the sample's impact on the score

**Returns** A tuple consisting of one tuple per class, holding false positive rate, true positive rate and thresholds

**Return type** tuple

### 7.2.15 MulticlassPrecisionRecall

**class** pytorch_lightning.metrics.classification.**MulticlassPrecisionRecall**(*num_classes=None*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorCollectionMetric*

Computes the multiclass PR Curve

#### Example

```
>>> pred = torch.tensor([[0.85, 0.05, 0.05, 0.05],
...                      [0.05, 0.85, 0.05, 0.05],
...                      [0.05, 0.05, 0.85, 0.05],
...                      [0.05, 0.05, 0.05, 0.85]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> metric = MulticlassPrecisionRecall()
>>> metric(pred, target)
((tensor([1., 1.]), tensor([1., 0.]), tensor([0.8500])),
 (tensor([1., 1.]), tensor([1., 0.]), tensor([0.8500])),
 (tensor([0.2500, 0.0000, 1.0000]), tensor([1., 0., 0.]), tensor([0.0500, 0.
↪8500])),
 (tensor([0.2500, 0.0000, 1.0000]), tensor([1., 0., 0.]), tensor([0.0500, 0.
↪8500])))
```

**Parameters**

- **num_classes** (Optional[int]) – number of classes
- **reduction** – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward**(*pred*, *target*, *sample_weight=None*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted probability for each label
- **target** (Tensor) – groundtruth labels
- **sample_weight** (Optional[Sequence]) – Weights for each sample defining the sample's impact on the score

**Returns** A tuple consisting of one tuple per class, holding precision, recall and thresholds

**Return type** tuple

### 7.2.16 IoU

**class** pytorch_lightning.metrics.classification.**IoU**(*remove_bg=False*, *reduction='elementwise_mean'*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the intersection over union.

#### Example

```
>>> pred = torch.tensor([[0, 0, 0, 0, 0, 0, 0, 0],
...                      [0, 0, 1, 1, 1, 0, 0, 0],
...                      [0, 0, 0, 0, 0, 0, 0, 0]])
>>> target = torch.tensor([[0, 0, 0, 0, 0, 0, 0, 0],
...                        [0, 0, 0, 1, 1, 1, 0, 0],
...                        [0, 0, 0, 0, 0, 0, 0, 0]])
>>> metric = IoU()
>>> metric(pred, target)
tensor(0.7045)
```

**Parameters**

- **remove_bg** (bool) – Flag to state whether a background class has been included within input parameters. If true, will remove background class. If false, return IoU over all classes. Assumes that background is '0' class in input tensor
- **reduction** (str) – a method for reducing IoU over labels (default: takes the mean) Available reduction methods:
  - elementwise_mean: takes the mean
  - none: pass array
  - sum: add elements

**forward**(*y_pred*, *y_true*, *sample_weight=None*)

Actual metric calculation.

### 7.2.17 RMSE

**class** pytorch_lightning.metrics.regression.**RMSE**(*reduction='elementwise_mean'*)

Bases: *pytorch_lightning.metrics.metric.Metric*

Computes the root mean squared loss.

**Parameters** **reduction** (str) – a method for reducing mse over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements

### Example

```
>>> pred = torch.tensor([0., 1, 2, 3])
>>> target = torch.tensor([0., 1, 2, 2])
>>> metric = RMSE()
>>> metric(pred, target)
tensor(0.5000)
```

**forward** (*pred*, *target*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the rmse loss.

## 7.2.18 RMSLE

**class** pytorch_lightning.metrics.regression.**RMSLE** (*reduction='elementwise_mean'*)
Bases: *pytorch_lightning.metrics.metric.Metric*

Computes the root mean squared log loss.

**Parameters** **reduction** (str) – a method for reducing mse over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements

### Example

```
>>> pred = torch.tensor([0., 1, 2, 3])
>>> target = torch.tensor([0., 1, 2, 2])
>>> metric = RMSLE()
>>> metric(pred, target)
tensor(0.0207)
```

**forward** (*pred*, *target*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the rmsle loss.

## 7.3 Functional Metrics

Functional metrics can be called anywhere (even used with just plain PyTorch).

```
from pytorch_lightning.metrics.functional import accuracy

pred = torch.tensor([0, 1, 2, 3])
target = torch.tensor([0, 1, 2, 2])

# calculates accuracy across all GPUs and all Nodes used in training
accuracy(pred, target)
```

These metrics even work when using distributed training:

```
class MyModule(...):
    def forward(self, x, y):
        return accuracy(x, y)

model = MyModule()
trainer = Trainer(gpus=8, num_nodes=2)

# any metric automatically reduces across GPUs (even the ones you implement using␣
↪Lightning)
trainer.fit(model)
```

### 7.3.1 accuracy (F)

`pytorch_lightning.metrics.functional.`**`accuracy`**(*pred*, *target*, *num_classes=None*, *reduction='elementwise_mean'*)

Computes the accuracy classification score

**Parameters**

- **`pred`** (`Tensor`) – predicted labels
- **`target`** (`Tensor`) – ground truth labels
- **`num_classes`** (`Optional[int]`) – number of classes
- **`reduction`** – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods:
  - elementwise_mean: takes the mean
  - none: pass array
  - sum: add elements

**Return type** `Tensor`

**Returns** A Tensor with the classification score.

#### Example

```
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> accuracy(x, y)
tensor(0.7500)
```

### 7.3.2 auc (F)

`pytorch_lightning.metrics.functional.`**`auc`**(*x*, *y*, *reorder=True*)

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

**Parameters**

- **x** (`Tensor`) – x-coordinates
- **y** (`Tensor`) – y-coordinates
- **reorder** (`bool`) – reorder coordinates, so they are increasing.

**Return type** `Tensor`

**Returns** Tensor containing AUC score (float)

#### Example

```
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> auc(x, y)
tensor(4.)
```

### 7.3.3 auroc (F)

`pytorch_lightning.metrics.functional.`**`auroc`**(*pred*, *target*, *sample_weight=None*, *pos_label=1.0*)

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

**Parameters**

- **pred** (`Tensor`) – estimated probabilities
- **target** (`Tensor`) – ground-truth labels
- **sample_weight** (`Optional[Sequence]`) – sample weights
- **pos_label** (`int`) – the label for the positive class (default: 1.)

### Example

```
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> auroc(x, y)
tensor(0.3333)
```

**Return type** Tensor

## 7.3.4 average_precision (F)

`pytorch_lightning.metrics.functional.average_precision`(*pred*, *target*, *sample_weight=None*, *pos_label=1.0*)

**Parameters**

- **pred** (Tensor) – estimated probabilities
- **target** (Tensor) – ground-truth labels
- **sample_weight** (Optional[Sequence]) – sample weights
- **pos_label** (int) – the label for the positive class (default: 1.)

### Example

```
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> average_precision(x, y)
tensor(0.3333)
```

**Return type** Tensor

## 7.3.5 confusion_matrix (F)

`pytorch_lightning.metrics.functional.confusion_matrix`(*pred*, *target*, *normalize=False*)

Computes the confusion matrix C where each entry C_{i,j} is the number of observations in group i that were predicted in group j.

**Parameters**

- **pred** (Tensor) – estimated targets
- **target** (Tensor) – ground truth labels
- **normalize** (bool) – normalizes confusion matrix

**Return type** Tensor

**Returns** Tensor, confusion matrix C [num_classes, num_classes ]

#### Example

```
>>> x = torch.tensor([1, 2, 3])
>>> y = torch.tensor([0, 2, 3])
>>> confusion_matrix(x, y)
tensor([[0., 1., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.]])
```

### 7.3.6 dice_score (F)

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

**Parameters**

- **pred** (Tensor) – estimated probabilities
- **target** (Tensor) – ground-truth labels
- **bg** (bool) – whether to also compute dice for the background
- **nan_score** (float) – score to return, if a NaN occurs during computation (denom zero)
- **no_fg_score** (float) – score to return, if no foreground pixel was found in target
- **reduction** (str) – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods:
  - elementwise_mean: takes the mean
  - none: pass array
  - sum: add elements

#### Example

```
>>> pred = torch.tensor([[0.85, 0.05, 0.05, 0.05],
...                      [0.05, 0.85, 0.05, 0.05],
...                      [0.05, 0.05, 0.85, 0.05],
...                      [0.05, 0.05, 0.05, 0.85]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> average_precision(pred, target)
tensor(0.2500)
```

**Return type** Tensor

### 7.3.7 f1_score (F)

pytorch_lightning.metrics.functional.**f1_score**(*pred*, *target*, *num_classes=None*, *reduction='elementwise_mean'*)

Computes the F1-score (a.k.a F-measure), which is the harmonic mean of the precision and recall. It ranges between 1 and 0, where 1 is perfect and the worst value is 0.

**Parameters**

- **pred** (Tensor) – estimated probabilities
- **target** (Tensor) – ground-truth labels
- **num_classes** (Optional[int]) – number of classes
- **reduction** – method for reducing F1-score (default: takes the mean) Available reduction methods:
  - elementwise_mean: takes the mean
  - none: pass array
  - sum: add elements.

**Return type** Tensor

**Returns** Tensor containing F1-score

#### Example

```
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> f1_score(x, y)
tensor(0.6667)
```

### 7.3.8 fbeta_score (F)

pytorch_lightning.metrics.functional.**fbeta_score**(*pred*, *target*, *beta*, *num_classes=None*, *reduction='elementwise_mean'*)

Computes the F-beta score which is a weighted harmonic mean of precision and recall. It ranges between 1 and 0, where 1 is perfect and the worst value is 0.

**Parameters**

- **pred** (Tensor) – estimated probabilities
- **target** (Tensor) – ground-truth labels
- **beta** (float) – weights recall when combining the score. beta < 1: more weight to precision. beta > 1 more weight to recall beta = 0: only precision beta -> inf: only recall
- **num_classes** (Optional[int]) – number of classes
- **reduction** (str) – method for reducing F-score (default: takes the mean) Available reduction methods:
  - elementwise_mean: takes the mean
  - none: pass array
  - sum: add elements.

**Return type** Tensor

**Returns** Tensor with the value of F-score. It is a value between 0-1.

### Example

```
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> fbeta_score(x, y, 0.2)
tensor(0.7407)
```

### 7.3.9 multiclass_precision_recall_curve (F)

pytorch_lightning.metrics.functional.**multiclass_precision_recall_curve**(*pred, target, sample_weight=None, num_classes=None*)

Computes precision-recall pairs for different thresholds given a multiclass scores.

**Parameters**

- **pred** (Tensor) – estimated probabilities
- **target** (Tensor) – ground-truth labels
- **sample_weight** (Optional[Sequence]) – sample weight
- **num_classes** (Optional[int]) – number of classes

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

**Returns** number of classes, precision, recall, thresholds

### Example

```
>>> pred = torch.tensor([[0.85, 0.05, 0.05, 0.05],
...                      [0.05, 0.85, 0.05, 0.05],
...                      [0.05, 0.05, 0.85, 0.05],
...                      [0.05, 0.05, 0.05, 0.85]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> nb_classes, precision, recall, thresholds = multiclass_precision_recall_
↪curve(pred, target)
>>> nb_classes
(tensor([1., 1.]), tensor([1., 0.]), tensor([0.8500]))
>>> precision
(tensor([1., 1.]), tensor([1., 0.]), tensor([0.8500]))
>>> recall
(tensor([0.2500, 0.0000, 1.0000]), tensor([1., 0., 0.]), tensor([0.0500, 0.8500]))
>>> thresholds
(tensor([0.2500, 0.0000, 1.0000]), tensor([1., 0., 0.]), tensor([0.0500, 0.8500]))
```

### 7.3.10 multiclass_roc (F)

pytorch_lightning.metrics.functional.**multiclass_roc**(*pred*, *target*, *sample_weight=None*, *num_classes=None*)

Computes the Receiver Operating Characteristic (ROC) for multiclass predictors.

**Parameters**

- **pred** (Tensor) – estimated probabilities
- **target** (Tensor) – ground-truth labels
- **sample_weight** (Optional[Sequence]) – sample weights
- **num_classes** (Optional[int]) – number of classes (default: None, computes automatically from data)

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

**Returns** returns roc for each class. Number of classes, false-positive rate (fpr), true-positive rate (tpr), thresholds

#### Example

```
>>> pred = torch.tensor([[0.85, 0.05, 0.05, 0.05],
...                      [0.05, 0.85, 0.05, 0.05],
...                      [0.05, 0.05, 0.85, 0.05],
...                      [0.05, 0.05, 0.05, 0.85]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> multiclass_roc(pred, target)
((tensor([0., 0., 1.]), tensor([0., 1., 1.]), tensor([1.8500, 0.8500, 0.0500])),
 (tensor([0., 0., 1.]), tensor([0., 1., 1.]), tensor([1.8500, 0.8500, 0.0500])),
 (tensor([0.0000, 0.3333, 1.0000]), tensor([0., 0., 1.]), tensor([1.8500, 0.8500,␣
↪0.0500])),
 (tensor([0.0000, 0.3333, 1.0000]), tensor([0., 0., 1.]), tensor([1.8500, 0.8500,␣
↪0.0500])))
```

### 7.3.11 precision (F)

pytorch_lightning.metrics.functional.**precision**(*pred*, *target*, *num_classes=None*, *reduction='elementwise_mean'*)

Computes precision score.

**Parameters**

- **pred** (Tensor) – estimated probabilities
- **target** (Tensor) – ground-truth labels
- **num_classes** (Optional[int]) – number of classes
- **reduction** (str) – method for reducing precision values (default: takes the mean) Available reduction methods:
  - elementwise_mean: takes the mean
  - none: pass array
  - sum: add elements

**Return type** Tensor

**Returns** Tensor with precision.

#### Example

```
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> precision(x, y)
tensor(0.7500)
```

### 7.3.12 precision_recall (F)

pytorch_lightning.metrics.functional.**precision_recall**(*pred*, *target*, *num_classes=None*, *reduction='elementwise_mean'*)

Computes precision and recall for different thresholds

**Parameters**

- **pred** (Tensor) – estimated probabilities
- **target** (Tensor) – ground-truth labels
- **num_classes** (Optional[int]) – number of classes
- **reduction** (str) – method for reducing precision-recall values (default: takes the mean) Available reduction methods:
  - elementwise_mean: takes the mean
  - none: pass array
  - sum: add elements

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

**Returns** Tensor with precision and recall

#### Example

```
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> precision_recall(x, y)
(tensor(0.7500), tensor(0.6250))
```

### 7.3.13 precision_recall_curve (F)

pytorch_lightning.metrics.functional.**precision_recall_curve**(*pred*, *target*, *sample_weight=None*, *pos_label=1.0*)

Computes precision-recall pairs for different thresholds.

**Parameters**

- **pred** (Tensor) – estimated probabilities

- **target** (Tensor) – ground-truth labels
- **sample_weight** (Optional[Sequence]) – sample weights
- **pos_label** (int) – the label for the positive class (default: 1.)

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

**Returns** precision, recall, thresholds

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> precision, recall, thresholds = precision_recall_curve(pred, target)
>>> precision
tensor([0.3333, 0.0000, 0.0000, 1.0000])
>>> recall
tensor([1., 0., 0., 0.])
>>> thresholds
tensor([1, 2, 3])
```

## 7.3.14 recall (F)

pytorch_lightning.metrics.functional.**recall**(*pred*, *target*, *num_classes=None*, *reduction='elementwise_mean'*)

Computes recall score.

**Parameters**

- **pred** (Tensor) – estimated probabilities
- **target** (Tensor) – ground-truth labels
- **num_classes** (Optional[int]) – number of classes
- **reduction** (str) – method for reducing recall values (default: takes the mean) Available reduction methods:
  - elementwise_mean: takes the mean
  - none: pass array
  - sum: add elements

**Return type** Tensor

**Returns** Tensor with recall.

### Example

```
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> recall(x, y)
tensor(0.6250)
```

### 7.3.15 roc (F)

`pytorch_lightning.metrics.functional.roc(pred, target, sample_weight=None, pos_label=1.0)`

Computes the Receiver Operating Characteristic (ROC). It assumes classifier is binary.

**Parameters**

- **pred** (Tensor) – estimated probabilities
- **target** (Tensor) – ground-truth labels
- **sample_weight** (Optional[Sequence]) – sample weights
- **pos_label** (int) – the label for the positive class (default: 1)

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

**Returns** false-positive rate (fpr), true-positive rate (tpr), thresholds

#### Example

```
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> fpr, tpr, thresholds = roc(x, y)
>>> fpr
tensor([0.0000, 0.3333, 0.6667, 0.6667, 1.0000])
>>> tpr
tensor([0., 0., 0., 1., 1.])
>>> thresholds
tensor([4, 3, 2, 1, 0])
```

### 7.3.16 stat_scores (F)

`pytorch_lightning.metrics.functional.stat_scores(pred, target, class_index, argmax_dim=1)`

Calculates the number of true positive, false positive, true negative and false negative for a specific class

**Parameters**

- **pred** (Tensor) – prediction tensor
- **target** (Tensor) – target tensor
- **class_index** (int) – class to calculate over
- **argmax_dim** (int) – if pred is a tensor of probabilities, this indicates the axis the argmax transformation will be applied over

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

**Returns** True Positive, False Positive, True Negative, False Negative

### Example

```
>>> x = torch.tensor([1, 2, 3])
>>> y = torch.tensor([0, 2, 3])
>>> tp, fp, tn, fn, sup = stat_scores(x, y, class_index=1)
>>> tp, fp, tn, fn, sup
(tensor(0), tensor(1), tensor(2), tensor(0), tensor(0))
```

## 7.3.17 iou (F)

`pytorch_lightning.metrics.functional.`**`iou`**(*pred*, *target*, *num_classes=None*, *remove_bg=False*, *reduction='elementwise_mean'*)

Intersection over union, or Jaccard index calculation.

**Parameters**

- **pred** (`Tensor`) – Tensor containing predictions
- **target** (`Tensor`) – Tensor containing targets
- **num_classes** (`Optional[int]`) – Optionally specify the number of classes
- **remove_bg** (`bool`) – Flag to state whether a background class has been included within input parameters. If true, will remove background class. If false, return IoU over all classes. Assumes that background is '0' class in input tensor
- **reduction** (`str`) – a method for reducing IoU over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements

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

**Return type** IoU score

### Example

```
>>> target = torch.randint(0, 1, (10, 25, 25))
>>> pred = torch.tensor(target)
>>> pred[2:5, 7:13, 9:15] = 1 - pred[2:5, 7:13, 9:15]
>>> iou(pred, target)
tensor(0.4914)
```

## 7.3.18 stat_scores_multiple_classes (F)

`pytorch_lightning.metrics.functional.`**`stat_scores_multiple_classes`**(*pred*, *target*, *num_classes=None*, *argmax_dim=1*)

Calls the stat_scores function iteratively for all classes, thus calculating the number of true postive, false postive, true negative and false negative for each class

**Parameters**

- **pred** (`Tensor`) – prediction tensor
- **target** (`Tensor`) – target tensor

- **class_index** – class to calculate over
- **argmax_dim** (int) – if pred is a tensor of probabilities, this indicates the axis the argmax transformation will be applied over

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

**Returns** True Positive, False Positive, True Negative, False Negative

### Example

```
>>> x = torch.tensor([1, 2, 3])
>>> y = torch.tensor([0, 2, 3])
>>> tps, fps, tns, fns, sups = stat_scores_multiple_classes(x, y)
>>> tps
tensor([0., 0., 1., 1.])
>>> fps
tensor([0., 1., 0., 0.])
>>> tns
tensor([2., 2., 2., 2.])
>>> fns
tensor([1., 0., 0., 0.])
>>> sups
tensor([1., 0., 1., 1.])
```

## 7.4 Metric pre-processing

### 7.4.1 to_categorical (F)

pytorch_lightning.metrics.functional.**to_categorical**(*tensor*, *argmax_dim=1*)

Converts a tensor of probabilities to a dense label tensor

**Parameters**

- **tensor** (Tensor) – probabilities to get the categorical label [N, d1, d2, …]
- **argmax_dim** (int) – dimension to apply (default: 1)

**Return type** Tensor

**Returns** A tensor with categorical labels [N, d2, …]

### Example

```
>>> x = torch.tensor([[0.2, 0.5], [0.9, 0.1]])
>>> to_categorical(x)
tensor([1, 0])
```

### 7.4.2 to_onehot (F)

`pytorch_lightning.metrics.functional.`**`to_onehot`**(*tensor*, *n_classes=None*)

Converts a dense label tensor to one-hot format

**Parameters**

- **tensor** (`Tensor`) – dense label tensor, with shape [N, d1, d2, ...]
- **n_classes** (`Optional[int]`) – number of classes C

**Output:** A sparse label tensor with shape [N, C, d1, d2, ...]

#### Example

```
>>> x = torch.tensor([1, 2, 3])
>>> to_onehot(x)
tensor([[0, 1, 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 1]])
```

**Return type** `Tensor`

## 7.5 Sklearn interface

Lightning supports sklearns metrics module as a backend for calculating metrics. Sklearns metrics are well tested and robust, but requires conversion between pytorch and numpy thus may slow down your computations.

To use the sklearn backend of metrics simply import as

```
import pytorch_lightning.metrics.sklearns import plm
metric = plm.Accuracy(normalize=True)
val = metric(pred, target)
```

Each converted sklearn metric comes has the same interface as its originally counterpart (e.g. accuracy takes the additional *normalize* keyword). Like the native Lightning metrics these converted sklearn metrics also come with built-in distributed (ddp) support.

### 7.5.1 SklearnMetric (sk)

`pytorch_lightning.metrics.sklearns.`**`SklearnMetric`**(*metric_name*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*, ***kwargs*)

Bridge between PyTorch Lightning and scikit-learn metrics

> **Warning:** Every metric call will cause a GPU synchronization, which may slow down your code

**Note:** The order of targets and predictions may be different from the order typically used in PyTorch

### 7.5.2 Accuracy (sk)

`pytorch_lightning.metrics.sklearns.`**`Accuracy`**(*normalize=True*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*)

Calculates the Accuracy Score

**Warning:** Every metric call will cause a GPU synchronization, which may slow down your code

#### Example

```
>>> y_pred = torch.tensor([0, 1, 2, 3])
>>> y_true = torch.tensor([0, 1, 2, 2])
>>> metric = Accuracy()
>>> metric(y_pred, y_true)
tensor([0.7500])
```

### 7.5.3 AUC (sk)

`pytorch_lightning.metrics.sklearns.`**`AUC`**(*reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*)

Calculates the Area Under the Curve using the trapoezoidal rule

**Warning:** Every metric call will cause a GPU synchronization, which may slow down your code

#### Example

```
>>> y_pred = torch.tensor([0, 1, 2, 3])
>>> y_true = torch.tensor([0, 1, 2, 2])
>>> metric = AUC()
>>> metric(y_pred, y_true)
tensor([4.])
```

### 7.5.4 AveragePrecision (sk)

`pytorch_lightning.metrics.sklearns.`**`AveragePrecision`**(*average='macro'*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*)

Calculates the average precision (AP) score.

### 7.5.5 ConfusionMatrix (sk)

`pytorch_lightning.metrics.sklearns.`**`ConfusionMatrix`**(*labels=None, reduce_group=torch.distributed.group.WORLD, reduce_op=torch.distributed.ReduceOp.SUM*)

Compute confusion matrix to evaluate the accuracy of a classification By definition a confusion matrix $C$ is such that $C_{i,j}$ is equal to the number of observations known to be in group $i$ but predicted to be in group $j$.

#### Example

```
>>> y_pred = torch.tensor([0, 1, 2, 1])
>>> y_true = torch.tensor([0, 1, 2, 2])
>>> metric = ConfusionMatrix()
>>> metric(y_pred, y_true)
tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 1., 1.]])
```

### 7.5.6 F1 (sk)

`pytorch_lightning.metrics.sklearns.`**`F1`**(*labels=None, pos_label=1, average='macro', reduce_group=torch.distributed.group.WORLD, reduce_op=torch.distributed.ReduceOp.SUM*)

Compute the F1 score, also known as balanced F-score or F-measure The F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0. The relative contribution of precision and recall to the F1 score are equal. The formula for the F1 score is:

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

In the multi-class and multi-label case, this is the weighted average of the F1 score of each class.

#### Example

```
>>> y_pred = torch.tensor([0, 1, 2, 3])
>>> y_true = torch.tensor([0, 1, 2, 2])
>>> metric = F1()
>>> metric(y_pred, y_true)
tensor([0.6667])
```

**References**

- [1] Wikipedia entry for the F1-score

### 7.5.7 FBeta (sk)

`pytorch_lightning.metrics.sklearns.FBeta`(*beta*, *labels=None*, *pos_label=1*, *average='macro'*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*)

Compute the F-beta score. The *beta* parameter determines the weight of precision in the combined score. `beta < 1` lends more weight to precision, while `beta > 1` favors recall (`beta -> 0` considers only precision, `beta -> inf` only recall).

#### Example

```
>>> y_pred = torch.tensor([0, 1, 2, 3])
>>> y_true = torch.tensor([0, 1, 2, 2])
>>> metric = FBeta(beta=0.25)
>>> metric(y_pred, y_true)
tensor([0.7361])
```

#### References

- [1] R. Baeza-Yates and B. Ribeiro-Neto (2011). Modern Information Retrieval. Addison Wesley, pp. 327-328.
- [2] Wikipedia entry for the F1-score

### 7.5.8 Precision (sk)

`pytorch_lightning.metrics.sklearns.Precision`(*labels=None*, *pos_label=1*, *average='macro'*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*)

Compute the precision The precision is the ratio `tp / (tp + fp)` where `tp` is the number of true positives and `fp` the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative. The best value is 1 and the worst value is 0.

#### Example

```
>>> y_pred = torch.tensor([0, 1, 2, 3])
>>> y_true = torch.tensor([0, 1, 2, 2])
>>> metric = Precision()
>>> metric(y_pred, y_true)
tensor([0.7500])
```

### 7.5.9 Recall (sk)

`pytorch_lightning.metrics.sklearns.`**`Recall`**(*labels=None*, *pos_label=1*, *average='macro'*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*)

Compute the recall The recall is the ratio `tp / (tp + fn)` where `tp` is the number of true positives and `fn` the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples. The best value is 1 and the worst value is 0.

#### Example

```
>>> y_pred = torch.tensor([0, 1, 2, 3])
>>> y_true = torch.tensor([0, 1, 2, 2])
>>> metric = Recall()
>>> metric(y_pred, y_true)
tensor([0.6250])
```

### 7.5.10 PrecisionRecallCurve (sk)

`pytorch_lightning.metrics.sklearns.`**`PrecisionRecallCurve`**(*pos_label=1*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*)

Compute precision-recall pairs for different probability thresholds

**Note:** This implementation is restricted to the binary classification task.

The precision is the ratio `tp / (tp + fp)` where `tp` is the number of true positives and `fp` the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative. The recall is the ratio `tp / (tp + fn)` where `tp` is the number of true positives and `fn` the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples. The last precision and recall values are 1. and 0. respectively and do not have a corresponding threshold. This ensures that the graph starts on the x axis.

### 7.5.11 ROC (sk)

`pytorch_lightning.metrics.sklearns.`**`ROC`**(*pos_label=1*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*)

Compute Receiver operating characteristic (ROC)

**Note:** this implementation is restricted to the binary classification task.

### Example

```
>>> y_pred = torch.tensor([0, 1, 2, 3])
>>> y_true = torch.tensor([0, 1, 2, 2])
>>> metric = ROC()
>>> fps, tps = metric(y_pred, y_true)
>>> fps
tensor([0.0000, 0.3333, 0.6667, 0.6667, 1.0000])
>>> tps
tensor([0., 0., 0., 1., 1.])
```

### References

- [1] Wikipedia entry for the Receiver operating characteristic

## 7.5.12 AUROC (sk)

`pytorch_lightning.metrics.sklearns.`**AUROC**(*average='macro'*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*)

Compute Area Under the Curve (AUC) from prediction scores

**Note:** this implementation is restricted to the binary classification task or multilabel classification task in label indicator format.

CHAPTER

EIGHT

# MODEL HOOKS

There are cases when you might want to do something different at different parts of the training/validation loop. To enable a hook, simply override the method in your LightningModule and the trainer will call it at the correct time.

**Contributing** If there's a hook you'd like to add, simply:

1. Fork PyTorchLightning.
2. Add the hook to *pytorch_lightning.core.hooks.ModelHooks*.
3. Add it in the correct place in *pytorch_lightning.trainer* where it should be called.

---

## 8.1 Hooks lifecycle

### 8.1.1 Training set-up

- *init_ddp_connection()*
- *init_optimizers()*
- *configure_apex()*
- *configure_ddp()*
- *train_dataloader()*
- *test_dataloader()*
- *val_dataloader()*
- *summarize()*
- *restore_weights()*

---

### 8.1.2 Training loop

- *on_epoch_start()*
- *on_batch_start()*
- *tbptt_split_batch()*
- *training_step()*
- *training_step_end()* (optional)
- *on_before_zero_grad()*
- *backward()*
- *on_after_backward()*
- `optimizer.step()`
- *on_batch_end()*
- *training_epoch_end()*
- *on_epoch_end()*

---

### 8.1.3 Validation loop

- `model.zero_grad()`
- `model.eval()`
- `torch.set_grad_enabled(False)`
- *validation_step()*
- *validation_step_end()*
- *validation_epoch_end()*
- `model.train()`
- `torch.set_grad_enabled(True)`
- *on_post_performance_check()*

---

### 8.1.4 Test loop

- `model.zero_grad()`
- `model.eval()`
- `torch.set_grad_enabled(False)`
- *test_step()*
- *test_step_end()*
- *test_epoch_end()*
- `model.train()`

- `torch.set_grad_enabled(True)`
- *`on_post_performance_check()`*

## 8.2 General hooks

**class** `pytorch_lightning.core.hooks.`**`ModelHooks`**(**args*, ***kwargs*)

Bases: `torch.nn.Module`

**`backward`**(*trainer*, *loss*, *optimizer*, *optimizer_idx*)

Override backward with your own implementation if you need to.

**Parameters**

- **`trainer`** – Pointer to the trainer
- **`loss`** (`Tensor`) – Loss is already scaled by accumulated grads
- **`optimizer`** (`Optimizer`) – Current optimizer being used
- **`optimizer_idx`** (`int`) – Index of the current optimizer being used

Called to perform backward step. Feel free to override as needed.

The loss passed in has already been scaled for accumulated gradients if requested.

Example:

```
def backward(self, use_amp, loss, optimizer):
    if use_amp:
        with amp.scale_loss(loss, optimizer) as scaled_loss:
            scaled_loss.backward()
    else:
        loss.backward()
```

**Return type** `None`

**`on_after_backward`**()

Called in the training loop after loss.backward() and before optimizers do anything. This is the ideal place to inspect or log gradient information.

Example:

```
def on_after_backward(self):
    # example to inspect gradient information in tensorboard
    if self.trainer.global_step % 25 == 0:  # don't make the tf file huge
        params = self.state_dict()
        for k, v in params.items():
            grads = v
            name = k
            self.logger.experiment.add_histogram(tag=name, values=grads,
                                                 global_step=self.trainer.
↪global_step)
```

**Return type** `None`

**`on_batch_end`**()

Called in the training loop after the batch.

**Return type** None

**on_batch_start**(*batch*)
Called in the training loop before anything happens for that batch.

If you return -1 here, you will skip training for the rest of the current epoch.

**Parameters** **batch** (Any) – The batched data as it is returned by the training DataLoader.

**Return type** None

**on_before_zero_grad**(*optimizer*)
Called after optimizer.step() and before optimizer.zero_grad().

Called in the training loop after taking an optimizer step and before zeroing grads. Good place to inspect weight information with weights updated.

This is where it is called:

```
for optimizer in optimizers:
    optimizer.step()
    model.on_before_zero_grad(optimizer) # < ---- called here
    optimizer.zero_grad
```

**Parameters** **optimizer** (Optimizer) – The optimizer for which grads should be zeroed.

**Return type** None

**on_epoch_end**()
Called in the training loop at the very end of the epoch.

**Return type** None

**on_epoch_start**()
Called in the training loop at the very beginning of the epoch.

**Return type** None

**on_fit_end**()
Called at the very end of fit. If on DDP it is called on every process

**on_fit_start**()
Called at the very beginning of fit. If on DDP it is called on every process

**on_post_performance_check**()
Called at the very end of the validation loop.

**Return type** None

**on_pre_performance_check**()
Called at the very beginning of the validation loop.

**Return type** None

**on_sanity_check_start**()
Called before starting evaluation.

**Warning:** Deprecated. Will be removed in v0.9.0.

**on_train_end**()
Called at the end of training before logger experiment is closed.

**Return type** None

**on_train_start**()
Called at the beginning of training before sanity check.

**Return type** `None`

**setup**(*stage*)
Called at the beginning of fit and test. This is a good hook when you need to build models dynamically or adjust something about them. This hook is called on every process when using DDP.

**Parameters** **stage** (`str`) – either 'fit' or 'test'

Example:

```
class LitModel(...):
    def __init__(self):
        self.l1 = None

    def prepare_data(self):
        download_data()
        tokenize()

        # don't do this
        self.something = else

    def setup(step):
        data = Load_data(...)
        self.l1 = nn.Linear(28, data.num_classes)
```

**teardown**(*stage*)
Called at the end of fit and test.

**Parameters** **stage** (`str`) – either 'fit' or 'test'

**transfer_batch_to_device**(*batch, device*)
Override this hook if your `DataLoader` returns tensors wrapped in a custom data structure.

The data types listed below (and any arbitrary nesting of them) are supported out of the box:

- `torch.Tensor`
- `list`
- `dict`
- `tuple`
- `torchtext.data.Batch` (COMING SOON)

For anything else, you need to define how the data is moved to the target device (CPU, GPU, TPU, . . . ).

Example:

```
def transfer_batch_to_device(self, batch, device)
    if isinstance(batch, CustomBatch):
        # move all tensors in your custom data structure to the device
        batch.samples = batch.samples.to(device)
        batch.targets = batch.targets.to(device)
    else:
        batch = super().transfer_batch_to_device(data, device)
    return batch
```

**Parameters**

- **batch** (Any) – A batch of data that needs to be transferred to a new device.
- **device** (device) – The target device as defined in PyTorch.

**Return type** Any

**Returns** A reference to the data on the new device.

**Note:** This hook should only transfer the data and not modify it, nor should it move the data to any other device than the one passed in as argument (unless you know what you are doing). The *Trainer* already takes care of splitting the batch and determines the target devices.

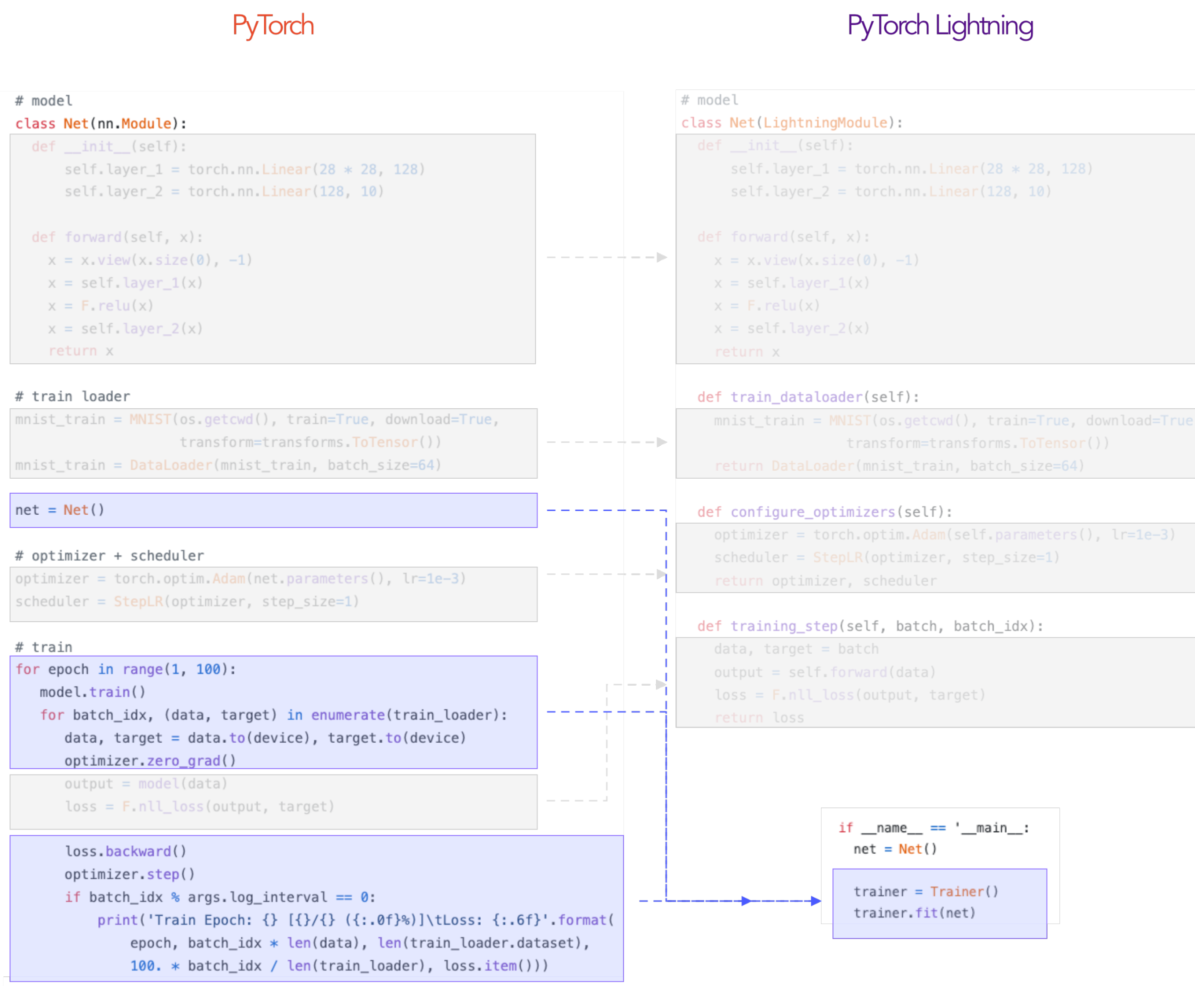
**See also:**

- *move_data_to_device()*
- *apply_to_collection()*

# CHAPTER NINE

# TRAINER

Once you've organized your PyTorch code into a LightningModule, the Trainer automates everything else.




This abstraction achieves the following:

1. You maintain control over all aspects via PyTorch code without an added abstraction.
2. The trainer uses best practices embedded by contributors and users from top AI labs such as Facebook AI Research, NYU, MIT, Stanford, etc. . .
3. The trainer allows overriding any key part that you don't want automated.

## 9.1 Basic use

This is the basic use of the trainer:

```
model = MyLightningModule()

trainer = Trainer()
trainer.fit(model)
```

## 9.2 Best Practices

For cluster computing, it's recommended you structure your main.py file this way

```
from argparse import ArgumentParser

def main(hparams):
    model = LightningModule()
    trainer = Trainer(gpus=hparams.gpus)
    trainer.fit(model)

if __name__ == '__main__':
    parser = ArgumentParser()
    parser.add_argument('--gpus', default=None)
    args = parser.parse_args()

    main(args)
```

So you can run it like so:

```
python main.py --gpus 2
```

**Note:** If you want to stop a training run early, you can press "Ctrl + C" on your keyboard. The trainer will catch the *KeyboardInterrupt* and attempt a graceful shutdown, including running callbacks such as *on_train_end*. The trainer object will also set an attribute *interrupted* to *True* in such cases. If you have a callback which shuts down compute resources, for example, you can conditionally run the shutdown logic for only uninterrupted runs.

## 9.3 Testing

Once you're done training, feel free to run the test set! (Only right before publishing your paper or pushing to production)

```
trainer.test()
```

---

## 9.4 Deployment / prediction

You just trained a LightningModule which is also just a torch.nn.Module. Use it to do whatever!

```
# load model
pretrained_model = LightningModule.load_from_checkpoint(PATH)
pretrained_model.freeze()

# use it for finetuning
def forward(self, x):
    features = pretrained_model(x)
    classes = classifier(features)

# or for prediction
out = pretrained_model(x)
api_write({'response': out}
```

You may wish to run the model on a variety of devices. Instead of moving the data manually to the correct device, decorate the forward method (or any other method you use for inference) with `auto_move_data()` and Lightning will take care of the rest.

---

## 9.5 Reproducibility

To ensure full reproducibility from run to run you need to set seeds for pseudo-random generators, and set `deterministic`` flag in `Trainer`.

Example:

```
from pytorch_lightning import Trainer, seed_everything

seed_everything(42)
# sets seeds for numpy, torch, python.random and PYTHONHASHSEED.
model = Model()
trainer = Trainer(deterministic=True)
```

---

## 9.6 Trainer flags

### 9.6.1 accumulate_grad_batches

Accumulates grads every k batches or as set up in the dict.

```
# default used by the Trainer (no accumulation)
trainer = Trainer(accumulate_grad_batches=1)
```

Example:

```
# accumulate every 4 batches (effective batch size is batch*4)
trainer = Trainer(accumulate_grad_batches=4)

# no accumulation for epochs 1-4. accumulate 3 for epochs 5-10. accumulate 20 after␣
↪that
trainer = Trainer(accumulate_grad_batches={5: 3, 10: 20})
```

### 9.6.2 amp_level

The optimization level to use (O1, O2, etc. . . ) for 16-bit GPU precision (using NVIDIA apex under the hood).

Check NVIDIA apex docs for level

Example:

```
# default used by the Trainer
trainer = Trainer(amp_level='O1')
```

### 9.6.3 auto_scale_batch_size

Automatically tries to find the largest batch size that fits into memory, before any training.

```
# default used by the Trainer (no scaling of batch size)
trainer = Trainer(auto_scale_batch_size=None)

# run batch size scaling, result overrides hparams.batch_size
trainer = Trainer(auto_scale_batch_size='binsearch')
```

### 9.6.4 auto_lr_find

Runs a learning rate finder algorithm (see this paper) before any training, to find optimal initial learning rate.

```
# default used by the Trainer (no learning rate finder)
trainer = Trainer(auto_lr_find=False)
```

Example:

```
# run learning rate finder, results override hparams.learning_rate
trainer = Trainer(auto_lr_find=True)

# run learning rate finder, results override hparams.my_lr_arg
trainer = Trainer(auto_lr_find='my_lr_arg')
```

**Note:** See the *learning rate finder guide*

### 9.6.5 benchmark

If true enables cudnn.benchmark. This flag is likely to increase the speed of your system if your input sizes don't change. However, if it does, then it will likely make your system slower.

The speedup comes from allowing the cudnn auto-tuner to find the best algorithm for the hardware [see discussion here].

Example:

```
# default used by the Trainer
trainer = Trainer(benchmark=False)
```

### 9.6.6 deterministic

If true enables cudnn.deterministic. Might make your system slower, but ensures reproducibility. Also sets `$HOROVOD_FUSION_THRESHOLD=0`.

For more info check [pytorch docs].

Example:

```
# default used by the Trainer
trainer = Trainer(deterministic=False)
```

### 9.6.7 callbacks

Add a list of user defined callbacks. These callbacks DO NOT replace the explicit callbacks (loggers, EarlyStopping or ModelCheckpoint).

**Note:** Only user defined callbacks (ie: Not EarlyStopping or ModelCheckpoint)

```
# a list of callbacks
callbacks = [PrintCallback()]
trainer = Trainer(callbacks=callbacks)
```

Example:

```
from pytorch_lightning.callbacks import Callback

class PrintCallback(Callback):
    def on_train_start(self, trainer, pl_module):
        print("Training is started!")
    def on_train_end(self, trainer, pl_module):
        print("Training is done.")
```

### 9.6.8 check_val_every_n_epoch

Check val every n train epochs.

Example:

```
# default used by the Trainer
trainer = Trainer(check_val_every_n_epoch=1)

# run val loop every 10 training epochs
trainer = Trainer(check_val_every_n_epoch=10)
```

### 9.6.9 checkpoint_callback

Callback for checkpointing.

```
from pytorch_lightning.callbacks import ModelCheckpoint
trainer = Trainer(checkpoint_callback=ModelCheckpoint())
```

Example:

```
from pytorch_lightning.callbacks import ModelCheckpoint

# default used by the Trainer
checkpoint_callback = ModelCheckpoint(
    filepath=os.getcwd(),
    save_top_k=True,
    verbose=True,
    monitor='val_loss',
    mode='min',
    prefix=''
)
```

### 9.6.10 default_root_dir

Default path for logs and weights when no logger or *pytorch_lightning.callbacks.ModelCheckpoint* callback passed. On certain clusters you might want to separate where logs and checkpoints are stored. If you don't then use this method for convenience.

Example:

```
# default used by the Trainer
trainer = Trainer(default_root_path=os.getcwd())
```

### 9.6.11 distributed_backend

The distributed backend to use.

- (`dp`) is DataParallel (split batch among GPUs of same machine)
- (`ddp`) is DistributedDataParallel (each gpu on each node trains, and syncs grads)
- (`ddp_cpu`) is DistributedDataParallel on CPU (same as *ddp*, but does not use GPUs. Useful for multi-node CPU training or single-node debugging. Note that this will **not** give a speedup on a single node, since Torch already makes effient use of multiple CPUs on a single machine.)

- **(`ddp2`) dp on node, ddp across nodes. Useful for things like increasing** the number of negative samples

```
# default used by the Trainer
trainer = Trainer(distributed_backend=None)
```

Example:

```
# dp = DataParallel
trainer = Trainer(gpus=2, distributed_backend='dp')

# ddp = DistributedDataParallel
trainer = Trainer(gpus=2, num_nodes=2, distributed_backend='ddp')

# ddp2 = DistributedDataParallel + dp
trainer = Trainer(gpus=2, num_nodes=2, distributed_backend='ddp2')
```

**Note:** this option does not apply to TPU. TPUs use `ddp` by default (over each core)

**See also:**

- *Multi-GPU training guide*
- *Multi-node (SLURM) guide*

### 9.6.12 early_stop_callback

Callback for early stopping. early_stop_callback (*pytorch_lightning.callbacks.EarlyStopping*)

- **True: A default callback monitoring 'val_loss' is created.** Will raise an error if 'val_loss' is not found.
- False: Early stopping will be disabled.
- None: The default callback monitoring 'val_loss' is created.
- Default: None.

```
from pytorch_lightning.callbacks import EarlyStopping

# default used by the Trainer
early_stop = EarlyStopping(
    monitor='val_loss',
    patience=3,
    strict=False,
    verbose=False,
    mode='min'
)
trainer = Trainer(early_stop_callback=early_stop)
```

**Note:** If 'val_loss' is not found will work as if early stopping is disabled.

### 9.6.13 fast_dev_run

Runs 1 batch of train, test and val to find any bugs (ie: a sort of unit test).

Under the hood the pseudocode looks like this:

```
# loading
__init__()
prepare_data

# test training step
training_batch = next(train_dataloader)
training_step(training_batch)

# test val step
val_batch = next(val_dataloader)
out = validation_step(val_batch)
validation_epoch_end([out])
```

```
# default used by the Trainer
trainer = Trainer(fast_dev_run=False)

# runs 1 train, val, test  batch and program ends
trainer = Trainer(fast_dev_run=True)
```

### 9.6.14 gpus

- Number of GPUs to train on
- or Which GPUs to train on
- can handle strings

```
# default used by the Trainer (ie: train on CPU)
trainer = Trainer(gpus=None)
```

Example:

```
# int: train on 2 gpus
trainer = Trainer(gpus=2)

# list: train on GPUs 1, 4 (by bus ordering)
trainer = Trainer(gpus=[1, 4])
trainer = Trainer(gpus='1, 4') # equivalent

# -1: train on all gpus
trainer = Trainer(gpus=-1)
trainer = Trainer(gpus='-1') # equivalent

# combine with num_nodes to train on multiple GPUs across nodes
# uses 8 gpus in total
trainer = Trainer(gpus=2, num_nodes=4)
```

**See also:**

- *Multi-GPU training guide*

### 9.6.15 gradient_clip_val

Gradient clipping value

- 0 means don't clip.

```
# default used by the Trainer
trainer = Trainer(gradient_clip_val=0.0)
```

### 9.6.16 limit_test_batches

How much of test dataset to check.

```
# default used by the Trainer
trainer = Trainer(limit_test_batches=1.0)

# run through only 25% of the test set each epoch
trainer = Trainer(limit_test_batches=0.25)

# run for only 10 batches
trainer = Trainer(limit_test_batches=10)
```

### 9.6.17 limit_val_batches

How much of validation dataset to check. Useful when debugging or testing something that happens at the end of an epoch.

```
# default used by the Trainer
trainer = Trainer(limit_val_batches=1.0)

# run through only 25% of the validation set each epoch
trainer = Trainer(limit_val_batches=0.25)

# run for only 10 batches
trainer = Trainer(limit_val_batches=10)
```

### 9.6.18 log_gpu_memory

Options:

- None
- 'min_max'
- 'all'

```
# default used by the Trainer
trainer = Trainer(log_gpu_memory=None)

# log all the GPUs (on master node only)
trainer = Trainer(log_gpu_memory='all')

# log only the min and max memory on the master node
trainer = Trainer(log_gpu_memory='min_max')
```

**Note:** Might slow performance because it uses the output of nvidia-smi.

### 9.6.19 log_save_interval

Writes logs to disk this often.

```
# default used by the Trainer
trainer = Trainer(log_save_interval=100)
```

### 9.6.20 logger

*Logger* (or iterable collection of loggers) for experiment tracking.

```
from pytorch_lightning.loggers import TensorBoardLogger

# default logger used by trainer
logger = TensorBoardLogger(
    save_dir=os.getcwd(),
    version=1,
    name='lightning_logs'
)
Trainer(logger=logger)
```

### 9.6.21 max_epochs

Stop training once this number of epochs is reached

```
# default used by the Trainer
trainer = Trainer(max_epochs=1000)
```

### 9.6.22 min_epochs

Force training for at least these many epochs

```
# default used by the Trainer
trainer = Trainer(min_epochs=1)
```

### 9.6.23 max_steps

Stop training after this number of steps Training will stop if max_steps or max_epochs have reached (earliest).

```
# Default (disabled)
trainer = Trainer(max_steps=None)

# Stop after 100 steps
trainer = Trainer(max_steps=100)
```

### 9.6.24 min_steps

Force training for at least these number of steps. Trainer will train model for at least min_steps or min_epochs (latest).

```
# Default (disabled)
trainer = Trainer(min_steps=None)

# Run at least for 100 steps (disable min_epochs)
trainer = Trainer(min_steps=100, min_epochs=0)
```

### 9.6.25 num_nodes

Number of GPU nodes for distributed training.

```
# default used by the Trainer
trainer = Trainer(num_nodes=1)

# to train on 8 nodes
trainer = Trainer(num_nodes=8)
```

### 9.6.26 num_processes

Number of processes to train with. Automatically set to the number of GPUs when using `distrbuted_backend="ddp"`. Set to a number greater than 1 when using `distributed_backend="ddp_cpu"` to mimic distributed training on a machine without GPUs. This is useful for debugging, but **will not** provide any speedup, since single-process Torch already makes effient use of multiple CPUs.

```
# Simulate DDP for debugging on your GPU-less laptop
trainer = Trainer(distributed_backend="ddp_cpu", num_processes=2)
```

### 9.6.27 num_sanity_val_steps

Sanity check runs n batches of val before starting the training routine. This catches any bugs in your validation without having to wait for the first validation check. The Trainer uses 5 steps by default. Turn it off or modify it here.

```
# default used by the Trainer
trainer = Trainer(num_sanity_val_steps=5)

# turn it off
trainer = Trainer(num_sanity_val_steps=0)
```

### 9.6.28 num_tpu_cores

> **Warning:** Deprecated since version 0.7.6.
>
> Use *tpu_cores* instead. Will remove 0.9.0.

Example:

```
python -m torch_xla.distributed.xla_dist
--tpu=$TPU_POD_NAME
--conda-env=torch-xla-nightly
--env=XLA_USE_BF16=1
-- python your_trainer_file.py
```

### 9.6.29 prepare_data_per_node

If True will call *prepare_data()* on LOCAL_RANK=0 for every node. If False will only call from NODE_RANK=0, LOCAL_RANK=0

```
# default
Trainer(prepare_data_per_node=True)

# use only NODE_RANK=0, LOCAL_RANK=0
Trainer(prepare_data_per_node=False)
```

### 9.6.30 tpu_cores

- How many TPU cores to train on (1 or 8).
- Which TPU core to train on [1-8]

A single TPU v2 or v3 has 8 cores. A TPU pod has up to 2048 cores. A slice of a POD means you get as many cores as you request.

Your effective batch size is batch_size * total tpu cores.

**Note:** No need to add a DistributedDataSampler, Lightning automatically does it for you.

This parameter can be either 1 or 8.

```
# your_trainer_file.py

# default used by the Trainer (ie: train on CPU)
trainer = Trainer(tpu_cores=None)

# int: train on a single core
trainer = Trainer(tpu_cores=1)

# list: train on a single selected core
trainer = Trainer(tpu_cores=[2])

# int: train on all cores few cores
trainer = Trainer(tpu_cores=8)

# for 8+ cores must submit via xla script with
# a max of 8 cores specified. The XLA script
# will duplicate script onto each TPU in the POD
trainer = Trainer(tpu_cores=8)
```

To train on more than 8 cores (ie: a POD), submit this script using the xla_dist script.

Example:

```
python -m torch_xla.distributed.xla_dist
--tpu=$TPU_POD_NAME
--conda-env=torch-xla-nightly
--env=XLA_USE_BF16=1
-- python your_trainer_file.py
```

### 9.6.31 overfit_pct

> **Warning:** Deprecated since version 0.8.0..
>
> Use *overfit_batches*. Will remove 1.0.0.

### 9.6.32 overfit_batches

Uses this much data of the training set. If will use the same training set for validation and testing. If the training Dataloaders(shuffle=True), Lightning will automatically disable it.

Useful for quickly debugging or trying to overfit on purpose.

```
# default used by the Trainer
trainer = Trainer(overfit_batches=0.0)

# use only 1% of the train set (and use the train set for val and test)
trainer = Trainer(overfit_batches=0.01)

# overfit on 10 of the same batches
trainer = Trainer(overfit_batches=10)
```

### 9.6.33 precision

Full precision (32), half precision (16). Can be used on CPU, GPU or TPUs.

If used on TPU will use torch.bfloat16 but tensor printing will still show torch.float32.

```
# default used by the Trainer
trainer = Trainer(precision=32)

# 16-bit precision
trainer = Trainer(precision=16)
```

Example:

```
# one day
trainer = Trainer(precision=8|4|2)
```

### 9.6.34 print_nan_grads

**Warning:** Deprecated since version 0.7.2..

Has no effect. When detected, NaN grads will be printed automatically. Will remove 0.9.0.

### 9.6.35 process_position

Orders the progress bar. Useful when running multiple trainers on the same node.

```
# default used by the Trainer
trainer = Trainer(process_position=0)
```

**Note:** This argument is ignored if a custom callback is passed to `callbacks`.

### 9.6.36 profiler

To profile individual steps during training and assist in identifying bottlenecks.

See the *profiler documentation*. for more details.

```
from pytorch_lightning.profiler import SimpleProfiler, AdvancedProfiler

# default used by the Trainer
trainer = Trainer(profiler=None)

# to profile standard training events
trainer = Trainer(profiler=True)

# equivalent to profiler=True
trainer = Trainer(profiler=SimpleProfiler())

# advanced profiler for function-level stats
trainer = Trainer(profiler=AdvancedProfiler())
```

### 9.6.37 progress_bar_refresh_rate

How often to refresh progress bar (in steps). In notebooks, faster refresh rates (lower number) is known to crash them because of their screen refresh rates, so raise it to 50 or more.

```
# default used by the Trainer
trainer = Trainer(progress_bar_refresh_rate=1)

# disable progress bar
trainer = Trainer(progress_bar_refresh_rate=0)
```

**Note:** This argument is ignored if a custom callback is passed to `callbacks`.

### 9.6.38 reload_dataloaders_every_epoch

Set to True to reload dataloaders every epoch.

```
# if False (default)
train_loader = model.train_dataloader()
for epoch in epochs:
    for batch in train_loader:
        ...

# if True
for epoch in epochs:
    train_loader = model.train_dataloader()
    for batch in train_loader:
```

### 9.6.39 replace_sampler_ddp

Enables auto adding of distributed sampler.

```
# default used by the Trainer
trainer = Trainer(replace_sampler_ddp=True)
```

By setting to False, you have to add your own distributed sampler:

```
# default used by the Trainer
sampler = torch.utils.data.distributed.DistributedSampler(dataset, shuffle=True)
dataloader = DataLoader(dataset, batch_size=32, sampler=sampler)
```

### 9.6.40 resume_from_checkpoint

To resume training from a specific checkpoint pass in the path here.

```
# default used by the Trainer
trainer = Trainer(resume_from_checkpoint=None)

# resume from a specific checkpoint
trainer = Trainer(resume_from_checkpoint='some/path/to/my_checkpoint.ckpt')
```

### 9.6.41 row_log_interval

How often to add logging rows (does not write to disk)

```
# default used by the Trainer
trainer = Trainer(row_log_interval=50)
```

use_amp:

> **Warning:** Deprecated since version 0.7.0.
>
> Use *precision* instead. Will remove 0.9.0.

### 9.6.42 show_progress_bar

> **Warning:** Deprecated since version 0.7.2.
>
> Set *progress_bar_refresh_rate* to 0 instead. Will remove 0.9.0.

### 9.6.43 val_percent_check

> **Warning:** deprecated in v0.8.0 please use *limit_val_batches*. Will remove in 0.10.0

### 9.6.44 test_percent_check

> **Warning:** deprecated in v0.8.0 please use *limit_test_batches*. Will remove in 0.10.0

### 9.6.45 train_percent_check

> **Warning:** deprecated in v0.8.0 please use *limit_train_batches*. Will remove in 0.10.0

### 9.6.46 track_grad_norm

- no tracking (-1)
- Otherwise tracks that norm (2 for 2-norm)

```
# default used by the Trainer
trainer = Trainer(track_grad_norm=-1)

# track the 2-norm
trainer = Trainer(track_grad_norm=2)
```

### 9.6.47 limit_train_batches

How much of training dataset to check. Useful when debugging or testing something that happens at the end of an epoch.

```
# default used by the Trainer
trainer = Trainer(limit_train_batches=1.0)
```

Example:

```
# default used by the Trainer
trainer = Trainer(limit_train_batches=1.0)

# run through only 25% of the training set each epoch
trainer = Trainer(limit_train_batches=0.25)
```

(continues on next page)

(continued from previous page)

```
# run through only 10 batches of the training set each epoch
trainer = Trainer(limit_train_batches=10)
```

### 9.6.48 truncated_bptt_steps

Truncated back prop breaks performs backprop every k steps of a much longer sequence.

If this is enabled, your batches will automatically get truncated and the trainer will apply Truncated Backprop to it.

(Williams et al. "An efficient gradient-based algorithm for on-line training of recurrent network trajectories.")

```
# default used by the Trainer (ie: disabled)
trainer = Trainer(truncated_bptt_steps=None)

# backprop every 5 steps in a batch
trainer = Trainer(truncated_bptt_steps=5)
```

**Note:** Make sure your batches have a sequence dimension.

Lightning takes care to split your batch along the time-dimension.

```
# we use the second as the time dimension
# (batch, time, ...)
sub_batch = batch[0, 0:t, ...]
```

Using this feature requires updating your LightningModule's *pytorch_lightning.core.LightningModule.training_step()* to include a *hiddens* arg with the hidden

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hiddens from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)

    return {
        "loss": ...,
        "hiddens": hiddens  # remember to detach() this
    }
```

To modify how the batch is split, override *pytorch_lightning.core.LightningModule.tbptt_split_batch()*:

```
class LitMNIST(LightningModule):
    def tbptt_split_batch(self, batch, split_size):
        # do your own splitting on the batch
        return splits
```

### 9.6.49 val_check_interval

How often within one training epoch to check the validation set. Can specify as float or int.

- use (float) to check within a training epoch
- use (int) to check every n steps (batches)

```
# default used by the Trainer
trainer = Trainer(val_check_interval=1.0)

# check validation set 4 times during a training epoch
trainer = Trainer(val_check_interval=0.25)

# check validation set every 1000 training batches
# use this when using iterableDataset and your dataset has no length
# (ie: production cases with streaming data)
trainer = Trainer(val_check_interval=1000)
```

### 9.6.50 weights_save_path

Directory of where to save weights if specified.

```
# default used by the Trainer
trainer = Trainer(weights_save_path=os.getcwd())

# save to your custom path
trainer = Trainer(weights_save_path='my/path')
```

Example:

```
# if checkpoint callback used, then overrides the weights path
# **NOTE: this saves weights to some/path NOT my/path
checkpoint = ModelCheckpoint(filepath='some/path')
trainer = Trainer(
    checkpoint_callback=checkpoint,
    weights_save_path='my/path'
)
```

### 9.6.51 weights_summary

Prints a summary of the weights when training begins. Options: 'full', 'top', None.

```
# default used by the Trainer (ie: print summary of top level modules)
trainer = Trainer(weights_summary='top')

# print full summary of all modules and submodules
trainer = Trainer(weights_summary='full')

# don't print a summary
trainer = Trainer(weights_summary=None)
```

## 9.7 Trainer class

**class** pytorch_lightning.trainer.**Trainer**(*logger=True, checkpoint_callback=True, early_stop_callback=False, callbacks=None, default_root_dir=None, gradient_clip_val=0, process_position=0, num_nodes=1, num_processes=1, gpus=None, auto_select_gpus=False, tpu_cores=None, log_gpu_memory=None, progress_bar_refresh_rate=1, overfit_batches=0.0, track_grad_norm=-1, check_val_every_n_epoch=1, fast_dev_run=False, accumulate_grad_batches=1, max_epochs=1000, min_epochs=1, max_steps=None, min_steps=None, limit_train_batches=1.0, limit_val_batches=1.0, limit_test_batches=1.0, val_check_interval=1.0, log_save_interval=100, row_log_interval=50, distributed_backend=None, precision=32, print_nan_grads=False, weights_summary='top', weights_save_path=None, num_sanity_val_steps=2, truncated_bptt_steps=None, resume_from_checkpoint=None, profiler=None, benchmark=False, deterministic=False, reload_dataloaders_every_epoch=False, auto_lr_find=False, replace_sampler_ddp=True, terminate_on_nan=False, auto_scale_batch_size=False, prepare_data_per_node=True, amp_level='O1', num_tpu_cores=None, use_amp=None, show_progress_bar=None, val_percent_check=None, test_percent_check=None, train_percent_check=None, overfit_pct=None*)

Bases: `pytorch_lightning.trainer.training_io.TrainerIOMixin`, `pytorch_lightning.trainer.optimizers.TrainerOptimizersMixin`, `pytorch_lightning.trainer.auto_mix_precision.TrainerAMPMixin`, `pytorch_lightning.trainer.distrib_parts.TrainerDPMixin`, `pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin`, `pytorch_lightning.trainer.logging.TrainerLoggingMixin`, `pytorch_lightning.trainer.model_hooks.TrainerModelHooksMixin`, `pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixin`, `pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin`, `pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin`, `pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin`, `pytorch_lightning.trainer.callback_config.TrainerCallbackConfigMixin`, `pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin`, `pytorch_lightning.trainer.lr_finder.TrainerLRFinderMixin`, `pytorch_lightning.trainer.deprecated_api.TrainerDeprecatedAPITillVer0_9`, `pytorch_lightning.trainer.deprecated_api.TrainerDeprecatedAPITillVer0_10`

Customize every aspect of training via flags

**Parameters**

- **logger** (Union[*LightningLoggerBase*, Iterable[*LightningLoggerBase*], bool]) – Logger (or iterable collection of loggers) for experiment tracking.
- **checkpoint_callback** (Union[*ModelCheckpoint*, bool]) – Callback for checkpointing.
- **early_stop_callback** (*pytorch_lightning.callbacks.EarlyStopping*) –
- **callbacks** (Optional[List[*Callback*]]) – Add a list of callbacks.
- **default_root_dir** (Optional[str]) – Default path for logs and weights when no logger/ckpt_callback passed
- **gradient_clip_val** (float) – 0 means don't clip.
- **gradient_clip** –

> **Warning:** Deprecated since version 0.7.0.
>
> Use *gradient_clip_val* instead. Will remove 0.9.0.

- **process_position** (int) – orders the progress bar when running multiple models on same machine.
- **num_nodes** (int) – number of GPU nodes for distributed training.
- **nb_gpu_nodes** –

> **Warning:** Deprecated since version 0.7.0.
>
> Use *num_nodes* instead. Will remove 0.9.0.

- **gpus** (Union[int, str, List[int], None]) – Which GPUs to train on.
- **auto_select_gpus** (bool) – If enabled and *gpus* is an integer, pick available gpus automatically. This is especially useful when GPUs are configured to be in "exclusive mode", such that only one process at a time can access them.
- **tpu_cores** (Union[List[int], int, None]) – How many TPU cores to train on (1 or 8) / Single TPU to train on [1]
- **num_tpu_cores** (Optional[int]) – How many TPU cores to train on (1 or 8) .. warning:: .. deprecated:: 0.7.6. Will remove 0.9.0.
- **log_gpu_memory** (Optional[str]) – None, 'min_max', 'all'. Might slow performance
- **show_progress_bar** –

> **Warning:** Deprecated since version 0.7.2.
>
> Set *progress_bar_refresh_rate* to positive integer to enable. Will remove 0.9.0.

- **progress_bar_refresh_rate** (int) – How often to refresh progress bar (in steps). Value `0` disables progress bar. Ignored when a custom callback is passed to *callbacks*.

- **overfit_batches** (Union[int, float]) – Overfit a percent of training data (float) or a set number of batches (int).
- **overfit_pct** (Optional[float]) –

  > **Warning:** Deprecated since version 0.8.0.
  >
  > Use *overfit_batches* instead. Will remove 0.10.0.

- **track_grad_norm** (Union[int, float, str]) – -1 no tracking. Otherwise tracks that p-norm. May be set to 'inf' infinity-norm.
- **check_val_every_n_epoch** (int) – Check val every n train epochs.
- **fast_dev_run** (bool) – runs 1 batch of train, test and val to find any bugs (ie: a sort of unit test).
- **accumulate_grad_batches** (Union[int, Dict[int, int], List[list]]) – Accumulates grads every k batches or as set up in the dict.
- **max_epochs** (int) – Stop training once this number of epochs is reached.
- **max_nb_epochs** –

  > **Warning:** Deprecated since version 0.7.0.
  >
  > Use *max_epochs* instead. Will remove 0.9.0.

- **min_epochs** (int) – Force training for at least these many epochs
- **min_nb_epochs** –

  > **Warning:** Deprecated since version 0.7.0.
  >
  > Use *min_epochs* instead. Will remove 0.9.0.

- **max_steps** (Optional[int]) – Stop training after this number of steps. Disabled by default (None).
- **min_steps** (Optional[int]) – Force training for at least these number of steps. Disabled by default (None).
- **limit_train_batches** (Union[int, float]) – How much of training dataset to check.
- **limit_val_batches** (Union[int, float]) – How much of validation dataset to check (floats = percent, int = num_batches)
- **limit_test_batches** (Union[int, float]) – How much of test dataset to check (floats = percent, int = num_batches)
- **train_percent_check** (Optional[float]) –

  > **Warning:** Deprecated since version 0.8.0.
  >
  > Use *limit_train_batches* instead. Will remove v0.10.0.

- **val_percent_check** (Optional[float]) –

  > **Warning:** Deprecated since version 0.8.0.
  >
  > Use *limit_val_batches* instead. Will remove v0.10.0.

- **test_percent_check** (Optional[float]) –

  > **Warning:** Deprecated since version 0.8.0.
  >
  > Use *limit_test_batches* instead. Will remove v0.10.0.

- **val_check_interval** (Union[int, float]) – How often within one training epoch to check the validation set
- **log_save_interval** (int) – Writes logs to disk this often
- **row_log_interval** (int) – How often to add logging rows (does not write to disk)
- **add_row_log_interval** –

  > **Warning:** Deprecated since version 0.7.0.
  >
  > Use *row_log_interval* instead. Will remove 0.9.0.

- **distributed_backend** (Optional[str]) – The distributed backend to use (dp, ddp, ddp2, ddp_spawn)
- **use_amp** –

  > **Warning:** Deprecated since version 0.7.0.
  >
  > Use *precision* instead. Will remove 0.9.0.

- **precision** (int) – Full precision (32), half precision (16).
- **print_nan_grads** (bool) –

  > **Warning:** Deprecated since version 0.7.2.
  >
  > Has no effect. When detected, NaN grads will be printed automatically. Will remove 0.9.0.

- **weights_summary** (Optional[str]) – Prints a summary of the weights when training begins.
- **weights_save_path** (Optional[str]) – Where to save weights if specified. Will override default_root_dir for checkpoints only. Use this if for whatever reason you need the checkpoints stored in a different place than the logs written in *default_root_dir*.
- **amp_level** (str) – The optimization level to use (O1, O2, etc…).
- **num_sanity_val_steps** (int) – Sanity check runs n batches of val before starting the training routine.

- **truncated_bptt_steps** (Optional[int]) – Truncated back prop breaks performs backprop every k steps of
- **resume_from_checkpoint** (Optional[str]) – To resume training from a specific checkpoint pass in the path here. This can be a URL.
- **profiler** (Union[*BaseProfiler*, bool, None]) – To profile individual steps during training and assist in
- **reload_dataloaders_every_epoch** (bool) – Set to True to reload dataloaders every epoch
- **auto_lr_find** (Union[bool, str]) – If set to True, will *initially* run a learning rate finder, trying to optimize initial learning for faster convergence. Sets learning rate in self.lr or self.learning_rate in the LightningModule. To use a different key, set a string instead of True with the key name.
- **replace_sampler_ddp** (bool) – Explicitly enables or disables sampler replacement. If not specified this will toggled automatically ddp is used
- **benchmark** (bool) – If true enables cudnn.benchmark.
- **deterministic** (bool) – If true enables cudnn.deterministic
- **terminate_on_nan** (bool) – If set to True, will terminate training (by raising a *ValueError*) at the end of each training batch, if any of the parameters or the loss are NaN or +/-inf.
- **auto_scale_batch_size** (Union[str, bool]) – If set to True, will *initially* run a batch size finder trying to find the largest batch size that fits into memory. The result will be stored in self.batch_size in the LightningModule. Additionally, can be set to either *power* that estimates the batch size through a power search or *binsearch* that estimates the batch size through a binary search.
- **prepare_data_per_node** (bool) – If True, each LOCAL_RANK=0 will call prepare data. Otherwise only NODE_RANK=0, LOCAL_RANK=0 will prepare data

**classmethod add_argparse_args**(*parent_parser*)

Extends existing argparse by default *Trainer* attributes.

**Parameters** **parent_parser** (ArgumentParser) – The custom cli arguments parser, which will be extended by the Trainer default arguments.

Only arguments of the allowed types (str, float, int, bool) will extend the *parent_parser*.

### Examples

```
>>> import argparse
>>> import pprint
>>> parser = argparse.ArgumentParser()
>>> parser = Trainer.add_argparse_args(parser)
>>> args = parser.parse_args([])
>>> pprint.pprint(vars(args))
{...
 'check_val_every_n_epoch': 1,
 'checkpoint_callback': True,
 'default_root_dir': None,
 'deterministic': False,
 'distributed_backend': None,
 'early_stop_callback': False,
```

(continues on next page)

(continued from previous page)

```
...
'logger': True,
'max_epochs': 1000,
'max_steps': None,
'min_epochs': 1,
'min_steps': None,
...
'profiler': None,
'progress_bar_refresh_rate': 1,
...}
```

**Return type** `ArgumentParser`

**check_model_configuration** (*model*)
Checks that the model is configured correctly before training or testing is started.

**Parameters** **model** (*LightningModule*) – The model to check the configuration.

**fit** (*model*, *train_dataloader=None*, *val_dataloaders=None*)
Runs the full optimization routine.

**Parameters**

- **model** (*LightningModule*) – Model to fit.
- **train_dataloader** (`Optional`[`DataLoader`]) – A Pytorch DataLoader with training samples. If the model has a predefined train_dataloader method this will be skipped.
- **val_dataloaders** (`Union`[`DataLoader`, `List`[`DataLoader`], `None`]) – Either a single Pytorch Dataloader or a list of them, specifying validation samples. If the model has a predefined val_dataloaders method this will be skipped

Example:

```
# Option 1,
# Define the train_dataloader() and val_dataloader() fxs
# in the lightningModule
# RECOMMENDED FOR MOST RESEARCH AND APPLICATIONS TO MAINTAIN READABILITY
trainer = Trainer()
model = LightningModule()
trainer.fit(model)

# Option 2
# in production cases we might want to pass different datasets to the same␣
↪model
# Recommended for PRODUCTION SYSTEMS
train, val = DataLoader(...), DataLoader(...)
trainer = Trainer()
model = LightningModule()
trainer.fit(model, train_dataloader=train, val_dataloaders=val)

# Option 1 & 2 can be mixed, for example the training set can be
# defined as part of the model, and validation can then be feed to .fit()
```

**classmethod from_argparse_args** (*args*, ***kwargs*)
Create an instance from CLI arguments.

**Parameters**

- **args** (Union[Namespace, ArgumentParser]) – The parser or namespace to take arguments from. Only known arguments will be parsed and passed to the *Trainer*.
- ****kwargs** – Additional keyword arguments that may override ones in the parser or namespace. These must be valid Trainer arguments.

### Example

```
>>> parser = ArgumentParser(add_help=False)
>>> parser = Trainer.add_argparse_args(parser)
>>> parser.add_argument('--my_custom_arg', default='something')
>>> args = Trainer.parse_argparser(parser.parse_args(""))
>>> trainer = Trainer.from_argparse_args(args, logger=False)
```

**Return type** *Trainer*

**classmethod get_deprecated_arg_names**()

Returns a list with deprecated Trainer arguments.

**Return type** List

**classmethod get_init_arguments_and_types**()

Scans the Trainer signature and returns argument names, types and default values.

**Returns** (argument name, set with argument types, argument default value).

**Return type** List with tuples of 3 values

### Examples

```
>>> args = Trainer.get_init_arguments_and_types()
>>> import pprint
>>> pprint.pprint(sorted(args))
[('accumulate_grad_batches',
  (<class 'int'>, typing.Dict[int, int], typing.List[list]),
  1),
 ...
 ('callbacks',
  (typing.List[pytorch_lightning.callbacks.base.Callback],
   <class 'NoneType'>),
   None),
 ('check_val_every_n_epoch', (<class 'int'>,), 1),
 ...
 ('max_epochs', (<class 'int'>,), 1000),
 ...
 ('precision', (<class 'int'>,), 32),
 ('prepare_data_per_node', (<class 'bool'>,), True),
 ('print_nan_grads', (<class 'bool'>,), False),
 ('process_position', (<class 'int'>,), 0),
 ('profiler',
  (<class 'pytorch_lightning.profiler.profilers.BaseProfiler'>,
   <class 'bool'>,
   <class 'NoneType'>),
  None),
 ...
```

**static parse_argparser**(*arg_parser*)
Parse CLI arguments, required for custom bool types.

**Return type** `Namespace`

**run_pretrain_routine**(*model*)
Sanity check a few things before starting actual training.

**Parameters model** (*LightningModule*) – The model to run sanity test on.

**test**(*model=None*, *test_dataloaders=None*, *ckpt_path='best'*)
Separates from fit to make sure you never run on your test set until you want to.

**Parameters**

- **model** (`Optional`[*LightningModule*]) – The model to test.
- **test_dataloaders** (`Union`[`DataLoader`, `List`[`DataLoader`], `None`]) – Either a single Pytorch Dataloader or a list of them, specifying validation samples.
- **ckpt_path** (`Optional`[`str`]) – Either `best` or path to the checkpoint you wish to test. If `None`, use the weights from the last epoch to test. Default to `best`.

Example:

```
# Option 1
# run test with the best checkpoint from ``ModelCheckpoint`` after fitting.
test = DataLoader(...)
trainer = Trainer()
model = LightningModule()

trainer.fit(model)
trainer.test(test_dataloaders=test)

# Option 2
# run test with the specified checkpoint after fitting
test = DataLoader(...)
trainer = Trainer()
model = LightningModule()

trainer.fit(model)
trainer.test(test_dataloaders=test, ckpt_path='path/to/checkpoint.ckpt')

# Option 3
# run test with the weights from the end of training after fitting
test = DataLoader(...)
trainer = Trainer()
model = LightningModule()

trainer.fit(model)
trainer.test(test_dataloaders=test, ckpt_path=None)

# Option 4
# run test from a loaded model. ``ckpt_path`` is ignored in this case.
test = DataLoader(...)
model = LightningModule.load_from_checkpoint('path/to/checkpoint.ckpt')
trainer = Trainer()
trainer.test(model, test_dataloaders=test)
```

**property is_global_zero**
this is just empty shell for code implemented in other class.

**Type** Warning

**property num_gpus**
this is just empty shell for code implemented in other class.

**Type** Warning

**Return type** `int`

**property progress_bar_dict**
Read-only for progress bar metrics.

**Return type** `dict`

**property slurm_job_id**
this is just empty shell for code implemented in other class.

**Type** Warning

**Return type** `Optional[int]`

`pytorch_lightning.trainer.`**`seed_everything`**`(`*seed=None*`)`
Function that sets seed for pseudo-random number generators in: pytorch, numpy, python.random and sets PYTHONHASHSEED environment variable.

**Return type** `int`

CHAPTER

TEN

# 16-BIT TRAINING

Lightning offers 16-bit training for CPUs, GPUs and TPUs.

---

## 10.1 GPU 16-bit

16 bit precision can cut your memory footprint by half. If using volta architecture GPUs it can give a dramatic training speed-up as well.

**Note:** PyTorch 1.6+ is recommended for 16-bit

### 10.1.1 Native torch

When using PyTorch 1.6+ Lightning uses the native amp implementation to support 16-bit.

```
# turn on 16-bit
trainer = Trainer(precision=16)
```

### 10.1.2 Apex 16-bit

If you are using an earlier version of PyTorch Lightning uses Apex to support 16-bit.

Follow these instructions to install Apex. To use 16-bit precision, do two things:

1. Install Apex
2. Set the "precision" trainer flag.

```
$ git clone https://github.com/NVIDIA/apex
$ cd apex

# ------------------------
# OPTIONAL: on your cluster you might need to load cuda 10 or 9
# depending on how you installed PyTorch

# see available modules
module avail

```

(continues on next page)

(continued from previous page)

```
# load correct cuda before install
module load cuda-10.0
# ------------------------

# make sure you've loaded a cuda version > 4.0 and < 7.0
module load gcc-6.1.0

$ pip install -v --no-cache-dir --global-option="--cpp_ext" --global-option="--cuda_
↪ext" ./
```

**Warning:** NVIDIA Apex and DDP have instability problems. We recommend native 16-bit in PyTorch 1.6+

### 10.1.3 Enable 16-bit

```
# turn on 16-bit
trainer = Trainer(amp_level='O2', precision=16)
```

If you need to configure the apex init for your particular use case or want to use a different way of doing 16-bit training, override `pytorch_lightning.core.LightningModule.configure_apex()`.

---

## 10.2 TPU 16-bit

16-bit on TPus is much simpler. To use 16-bit with TPUs set precision to 16 when using the tpu flag

```
# DEFAULT
trainer = Trainer(tpu_cores=8, precision=32)

# turn on 16-bit
trainer = Trainer(tpu_cores=8, precision=16)
```

CHAPTER

ELEVEN

# COMPUTING CLUSTER (SLURM)

Lightning automates the details behind training on a SLURM-powered cluster.

---

## 11.1 Multi-node training

To train a model using multiple nodes, do the following:

1. Design your *LightningModule*.
2. Enable ddp in the trainer

```
# train on 32 GPUs across 4 nodes
trainer = Trainer(gpus=8, num_nodes=4, distributed_backend='ddp')
```

3. It's a good idea to structure your training script like this:

```
# train.py
def main(hparams):
    model = LightningTemplateModel(hparams)

    trainer = pl.Trainer(
        gpus=8,
        num_nodes=4,
        distributed_backend='ddp'
    )

    trainer.fit(model)


if __name__ == '__main__':
    root_dir = os.path.dirname(os.path.realpath(__file__))
    parent_parser = ArgumentParser(add_help=False)
    hyperparams = parser.parse_args()

    # TRAIN
    main(hyperparams)
```

4. Create the appropriate SLURM job:

```
# (submit.sh)
#!/bin/bash -l

```

(continues on next page)

(continued from previous page)

```
# SLURM SUBMIT SCRIPT
#SBATCH --nodes=4
#SBATCH --gres=gpu:8
#SBATCH --ntasks-per-node=8
#SBATCH --mem=0
#SBATCH --time=0-02:00:00

# activate conda env
source activate $1

# debugging flags (optional)
export NCCL_DEBUG=INFO
export PYTHONFAULTHANDLER=1

# on your cluster you might need these:
# set the network interface
# export NCCL_SOCKET_IFNAME=^docker0,lo

# might need the latest cuda
# module load NCCL/2.4.7-1-cuda.10.0

# run script from above
srun python3 train.py
```

5. If you want auto-resubmit (read below), add this line to the submit.sh script

```
#SBATCH --signal=SIGUSR1@90
```

6. Submit the SLURM job

```
sbatch submit.sh
```

**Note:** When running in DDP mode, any errors in your code will show up as an NCCL issue. Set the *NCCL_DEBUG=INFO* flag to see the ACTUAL error.

Normally now you would need to add a `DistributedSampler` to your dataset, however Lightning automates this for you. But if you still need to set a sampler set the Trainer flag `replace_sampler_ddp` to `False`.

Here's an example of how to add your own sampler (again, not needed with Lightning).

```
# in your LightningModule
def train_dataloader(self):
    dataset = MyDataset()
    dist_sampler = torch.utils.data.distributed.DistributedSampler(dataset)
    dataloader = Dataloader(dataset, sampler=dist_sampler)
    return dataloader

# in your training script
trainer = Trainer(replace_sampler_ddp=False)
```

## 11.2 Wall time auto-resubmit

When you use Lightning in a SLURM cluster, it automatically detects when it is about to run into the wall time and does the following:

1. Saves a temporary checkpoint.
2. Requeues the job.
3. When the job starts, it loads the temporary checkpoint.

To get this behavior make sure to add the correct signal to your SLURM script

```
# 90 seconds before training ends
SBATCH --signal=SIGUSR1@90
```

---

## 11.3 Building SLURM scripts

Instead of manually building SLURM scripts, you can use the SlurmCluster object to do this for you. The SlurmCluster can also run a grid search if you pass in a HyperOptArgumentParser.

Here is an example where you run a grid search of 9 combinations of hyperparameters. See also the multi-node examples here.

```
# grid search 3 values of learning rate and 3 values of number of layers for your net
# this generates 9 experiments (lr=1e-3, layers=16), (lr=1e-3, layers=32),
# (lr=1e-3, layers=64), ... (lr=1e-1, layers=64)
parser = HyperOptArgumentParser(strategy='grid_search', add_help=False)
parser.opt_list('--learning_rate', default=0.001, type=float,
                options=[1e-3, 1e-2, 1e-1], tunable=True)
parser.opt_list('--layers', default=1, type=float, options=[16, 32, 64], tunable=True)
hyperparams = parser.parse_args()

# Slurm cluster submits 9 jobs, each with a set of hyperparams
cluster = SlurmCluster(
    hyperparam_optimizer=hyperparams,
    log_path='/some/path/to/save',
)

# OPTIONAL FLAGS WHICH MAY BE CLUSTER DEPENDENT
# which interface your nodes use for communication
cluster.add_command('export NCCL_SOCKET_IFNAME=^docker0,lo')

# see output of the NCCL connection process
# NCCL is how the nodes talk to each other
cluster.add_command('export NCCL_DEBUG=INFO')

# setting a master port here is a good idea.
cluster.add_command('export MASTER_PORT=%r' % PORT)

# ************** DON'T FORGET THIS ***************
# MUST load the latest NCCL version
cluster.load_modules(['NCCL/2.4.7-1-cuda.10.0'])
```

(continues on next page)

(continued from previous page)

```
# configure cluster
cluster.per_experiment_nb_nodes = 12
cluster.per_experiment_nb_gpus = 8

cluster.add_slurm_cmd(cmd='ntasks-per-node', value=8, comment='1 task per gpu')

# submit a script with 9 combinations of hyper params
# (lr=1e-3, layers=16), (lr=1e-3, layers=32), (lr=1e-3, layers=64), ... (lr=1e-1,␣
↪layers=64)
cluster.optimize_parallel_cluster_gpu(
    main,
    nb_trials=9, # how many permutations of the grid search to run
    job_name='name_for_squeue'
)
```

The other option is that you generate scripts on your own via a bash command or use another library.

---

## 11.4 Self-balancing architecture (COMING SOON)

Here Lightning distributes parts of your module across available GPUs to optimize for speed and memory.

CHAPTER

TWELVE

# CHILD MODULES

Research projects tend to test different approaches to the same dataset. This is very easy to do in Lightning with inheritance.

For example, imagine we now want to train an Autoencoder to use as a feature extractor for MNIST images. Recall that *LitMNIST* already defines all the dataloading etc... The only things that change in the *Autoencoder* model are the init, forward, training, validation and test step.

```
class Encoder(torch.nn.Module):
    pass

class Decoder(torch.nn.Module):
    pass

class AutoEncoder(LitMNIST):

    def __init__(self):
        super().__init__()
        self.encoder = Encoder()
        self.decoder = Decoder()

    def forward(self, x):
        generated = self.decoder(x)

    def training_step(self, batch, batch_idx):
        x, _ = batch

        representation = self.encoder(x)
        x_hat = self(representation)

        loss = MSE(x, x_hat)
        return loss

    def validation_step(self, batch, batch_idx):
        return self._shared_eval(batch, batch_idx, 'val')

    def test_step(self, batch, batch_idx):
        return self._shared_eval(batch, batch_idx, 'test')

    def _shared_eval(self, batch, batch_idx, prefix):
        x, y = batch
        representation = self.encoder(x)
        x_hat = self(representation)

        loss = F.nll_loss(logits, y)
```

(continues on next page)

(continued from previous page)

```
        return {f'{prefix}_loss': loss}
```

and we can train this using the same trainer

```
autoencoder = AutoEncoder()
trainer = Trainer()
trainer.fit(autoencoder)
```

And remember that the forward method is to define the practical use of a LightningModule. In this case, we want to use the *AutoEncoder* to extract image representations

```
some_images = torch.Tensor(32, 1, 28, 28)
representations = autoencoder(some_images)
```

CHAPTER

THIRTEEN

# DEBUGGING

The following are flags that make debugging much easier.

---

## 13.1 fast_dev_run

This flag runs a "unit test" by running 1 training batch and 1 validation batch. The point is to detect any bugs in the training/validation loop without having to wait for a full epoch to crash.

(See: *fast_dev_run* argument of *Trainer*)

```
trainer = Trainer(fast_dev_run=True)
```

---

## 13.2 Inspect gradient norms

Logs (to a logger), the norm of each weight matrix.

(See: *track_grad_norm* argument of *Trainer*)

```
# the 2-norm
trainer = Trainer(track_grad_norm=2)
```

---

## 13.3 Log GPU usage

Logs (to a logger) the GPU usage for each GPU on the master machine.

(See: *log_gpu_memory* argument of *Trainer*)

```
trainer = Trainer(log_gpu_memory=True)
```

---

## 13.4 Make model overfit on subset of data

A good debugging technique is to take a tiny portion of your data (say 2 samples per class), and try to get your model to overfit. If it can't, it's a sign it won't work with large datasets.

(See: *overfit_batches* argument of *Trainer*)

```
# use only 1% of training data (and use the same training Dataloader (with shuffle␣
↪off) in val and test)
trainer = Trainer(overfit_batches=0.01)

# or overfit a number of batches
trainer = Trainer(overfit_batches=0.01)
```

With this flag, the train, val, and test sets will all be the same train set. We will also replace the sampler in the training set to turn off shuffle for you.

---

## 13.5 Print a summary of your LightningModule

Whenever the `.fit()` function gets called, the Trainer will print the weights summary for the LightningModule. By default it only prints the top-level modules. If you want to show all submodules in your network, use the *'full'* option:

```
trainer = Trainer(weights_summary='full')
```

You can also display the intermediate input- and output sizes of all your layers by setting the `example_input_array` attribute in your LightningModule. It will print a table like this

```
  | Name  | Type        | Params | In sizes  | Out sizes
--------------------------------------------------------------
0 | net   | Sequential  | 132 K  | [10, 256] | [10, 512]
1 | net.0 | Linear      | 131 K  | [10, 256] | [10, 512]
2 | net.1 | BatchNorm1d | 1 K    | [10, 512] | [10, 512]
```

when you call `.fit()` on the Trainer. This can help you find bugs in the composition of your layers.

**See Also:**

- *weights_summary* Trainer argument
- *ModelSummary*

---

## 13.6 Shorten epochs

Sometimes it's helpful to only use a percentage of your training, val or test data (or a set number of batches). For example, you can use 20% of the training set and 1% of the validation set.

On larger datasets like Imagenet, this can help you debug or test a few things faster than waiting for a full epoch.

```
# use only 10% of training data and 1% of val data
trainer = Trainer(limit_train_batches=0.1, limit_val_batches=0.01)

# use 10 batches of train and 5 batches of val
trainer = Trainer(limit_train_batches=10, limit_val_batches=5)
```

---

## 13.7 Set the number of validation sanity steps

Lightning runs a few steps of validation in the beginning of training. This avoids crashing in the validation loop sometime deep into a lengthy training loop.

(See: `num_sanity_val_steps` argument of `Trainer`)

```
# DEFAULT
trainer = Trainer(num_sanity_val_steps=5)
```

CHAPTER

FOURTEEN

# EXPERIMENT LOGGING

## 14.1 Comet.ml

Comet.ml is a third-party logger. To use `CometLogger` as your logger do the following. First, install the package:

```
pip install comet-ml
```

Then configure the logger and pass it to the `Trainer`:

```
import os
from pytorch_lightning.loggers import CometLogger
comet_logger = CometLogger(
    api_key=os.environ.get('COMET_API_KEY'),
    workspace=os.environ.get('COMET_WORKSPACE'),  # Optional
    save_dir='.',  # Optional
    project_name='default_project',  # Optional
    rest_api_key=os.environ.get('COMET_REST_API_KEY'),  # Optional
    experiment_name='default'  # Optional
)
trainer = Trainer(logger=comet_logger)
```

The `CometLogger` is available anywhere except `__init__` in your `LightningModule`.

```
class MyModule(LightningModule):
    def any_lightning_module_function_or_hook(self):
        some_img = fake_image()
        self.logger.experiment.add_image('generated_images', some_img, 0)
```

**See also:**

`CometLogger` docs.

## 14.2 MLflow

MLflow is a third-party logger. To use `MLFlowLogger` as your logger do the following. First, install the package:

```
pip install mlflow
```

Then configure the logger and pass it to the `Trainer`:

```
from pytorch_lightning.loggers import MLFlowLogger
mlf_logger = MLFlowLogger(
    experiment_name="default",
    tracking_uri="file:./ml-runs"
)
trainer = Trainer(logger=mlf_logger)
```

**See also:**

`MLFlowLogger` docs.

---

## 14.3 Neptune.ai

Neptune.ai is a third-party logger. To use `NeptuneLogger` as your logger do the following. First, install the package:

```
pip install neptune-client
```

Then configure the logger and pass it to the `Trainer`:

```
from pytorch_lightning.loggers import NeptuneLogger
neptune_logger = NeptuneLogger(
    api_key='ANONYMOUS',  # replace with your own
    project_name='shared/pytorch-lightning-integration',
    experiment_name='default',  # Optional,
    params={'max_epochs': 10},  # Optional,
    tags=['pytorch-lightning', 'mlp'],  # Optional,
)
trainer = Trainer(logger=neptune_logger)
```

The `NeptuneLogger` is available anywhere except `__init__` in your `LightningModule`.

```
class MyModule(LightningModule):
    def any_lightning_module_function_or_hook(self):
        some_img = fake_image()
        self.logger.experiment.add_image('generated_images', some_img, 0)
```

**See also:**

`NeptuneLogger` docs.

---

## 14.4 allegro.ai TRAINS

allegro.ai is a third-party logger. To use *TrainsLogger* as your logger do the following. First, install the package:

```
pip install trains
```

Then configure the logger and pass it to the *Trainer*:

```
from pytorch_lightning.loggers import TrainsLogger
trains_logger = TrainsLogger(
    project_name='examples',
    task_name='pytorch lightning test',
)
trainer = Trainer(logger=trains_logger)
```

The *TrainsLogger* is available anywhere in your *LightningModule*.

```
class MyModule(LightningModule):
    def __init__(self):
        some_img = fake_image()
        self.logger.experiment.log_image('debug', 'generated_image_0', some_img, 0)
```

**See also:**

*TrainsLogger* docs.

---

## 14.5 Tensorboard

To use TensorBoard as your logger do the following.

```
from pytorch_lightning.loggers import TensorBoardLogger
logger = TensorBoardLogger('tb_logs', name='my_model')
trainer = Trainer(logger=logger)
```

The *TensorBoardLogger* is available anywhere except `__init__` in your *LightningModule*.

```
class MyModule(LightningModule):
    def any_lightning_module_function_or_hook(self):
        some_img = fake_image()
        self.logger.experiment.add_image('generated_images', some_img, 0)
```

**See also:**

*TensorBoardLogger* docs.

---

## 14.6 Test Tube

Test Tube is a TensorBoard logger but with nicer file structure. To use *TestTubeLogger* as your logger do the following. First, install the package:

```
pip install test_tube
```

Then configure the logger and pass it to the *Trainer*:

```
from pytorch_lightning.loggers import TestTubeLogger
logger = TestTubeLogger('tb_logs', name='my_model')
trainer = Trainer(logger=logger)
```

The *TestTubeLogger* is available anywhere except `__init__` in your *LightningModule*.

```
class MyModule(LightningModule):
    def any_lightning_module_function_or_hook(self):
        some_img = fake_image()
        self.logger.experiment.add_image('generated_images', some_img, 0)
```

**See also:**

*TestTubeLogger* docs.

---

## 14.7 Weights and Biases

Weights and Biases is a third-party logger. To use *WandbLogger* as your logger do the following. First, install the package:

```
pip install wandb
```

Then configure the logger and pass it to the *Trainer*:

```
from pytorch_lightning.loggers import WandbLogger
wandb_logger = WandbLogger()
trainer = Trainer(logger=wandb_logger)
```

The *WandbLogger* is available anywhere except `__init__` in your *LightningModule*.

```
class MyModule(LightningModule):
    def any_lightning_module_function_or_hook(self):
        some_img = fake_image()
        self.logger.experiment.log({
            "generated_images": [wandb.Image(some_img, caption="...")]
        })
```

**See also:**

*WandbLogger* docs.

---

## 14.8 Multiple Loggers

Lightning supports the use of multiple loggers, just pass a list to the *Trainer*.

```
from pytorch_lightning.loggers import TensorBoardLogger, TestTubeLogger
logger1 = TensorBoardLogger('tb_logs', name='my_model')
logger2 = TestTubeLogger('tb_logs', name='my_model')
trainer = Trainer(logger=[logger1, logger2])
```

The loggers are available as a list anywhere except `__init__` in your *LightningModule*.

```
class MyModule(LightningModule):
    def any_lightning_module_function_or_hook(self):
        some_img = fake_image()
        # Option 1
        self.logger.experiment[0].add_image('generated_images', some_img, 0)
        # Option 2
        self.logger[0].experiment.add_image('generated_images', some_img, 0)
```

CHAPTER

FIFTEEN

# EXPERIMENT REPORTING

Lightning supports many different experiment loggers. These loggers allow you to monitor losses, images, text, etc... as training progresses. They usually provide a GUI to visualize and can sometimes even snapshot hyperparameters used in each experiment.

---

## 15.1 Control logging frequency

It may slow training down to log every single batch. Trainer has an option to log every k batches instead.

```
k = 10
trainer = Trainer(row_log_interval=k)
```

---

## 15.2 Control log writing frequency

Writing to a logger can be expensive. In Lightning you can set the interval at which you want to log using this trainer flag.

**See also:**

*Trainer*

```
k = 100
trainer = Trainer(log_save_interval=k)
```

---

## 15.3 Log metrics

To plot metrics into whatever logger you passed in (tensorboard, comet, neptune, TRAINS, etc...)

1. training_epoch_end, validation_epoch_end, test_epoch_end will all log anything in the "log" key of the return dict.

```
def training_epoch_end(self, outputs):
    loss = some_loss()
    ...

    logs = {'train_loss': loss}
    results = {'log': logs}
    return results

def validation_epoch_end(self, outputs):
    loss = some_loss()
    ...

    logs = {'val_loss': loss}
    results = {'log': logs}
    return results

def test_epoch_end(self, outputs):
    loss = some_loss()
    ...

    logs = {'test_loss': loss}
    results = {'log': logs}
    return results
```

2. In addition, you can also use any arbitrary functionality from a particular logger from within your LightningModule. For instance, here we log images using tensorboard.

```
def training_step(self, batch, batch_idx):
    self.generated_imgs = self.decoder.generate()

    sample_imgs = self.generated_imgs[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('generated_images', grid, 0)

    ...
    return results
```

---

## 15.4 Modify progress bar

Each return dict from the training_end, validation_end, testing_end and training_step also has a key called "progress_bar".

Here we show the validation loss in the progress bar

```
def validation_epoch_end(self, outputs):
    loss = some_loss()
    ...

    logs = {'val_loss': loss}
    results = {'progress_bar': logs}
    return results
```

---

## 15.5 Snapshot hyperparameters

When training a model, it's useful to know what hyperparams went into that model. When Lightning creates a checkpoint, it stores a key "hparams" with the hyperparams.

```
lightning_checkpoint = torch.load(filepath, map_location=lambda storage, loc: storage)
hyperparams = lightning_checkpoint['hparams']
```

Some loggers also allow logging the hyperparams used in the experiment. For instance, when using the TestTubeLogger or the TensorBoardLogger, all hyperparams will show in the hparams tab.

---

## 15.6 Snapshot code

Loggers also allow you to snapshot a copy of the code used in this experiment. For example, TestTubeLogger does this with a flag:

```
from pytorch_lightning.loggers import TestTubeLogger
logger = TestTubeLogger('.', create_git_tag=True)
```

CHAPTER

SIXTEEN

# EARLY STOPPING

## 16.1 Stopping an epoch early

You can stop an epoch early by overriding `on_batch_start()` to return *-1* when some condition is met.

If you do this repeatedly, for every epoch you had originally requested, then this will stop your entire run.

---

## 16.2 Default Epoch End Callback Behavior

By default early stopping will be enabled if *'val_loss'* is found in `validation_epoch_end()`'s return dict. Otherwise training will proceed with early stopping disabled.

---

## 16.3 Enable Early Stopping using the EarlyStopping Callback

The `EarlyStopping` callback can be used to monitor a validation metric and stop the training when no improvement is observed.

There are two ways to enable the EarlyStopping callback:

- Set *early_stop_callback=True*. The callback will look for 'val_loss' in the dict returned by `validation_epoch_end()` and raise an error if *val_loss* is not present.

```
trainer = Trainer(early_stop_callback=True)
```

- Create the callback object and pass it to the trainer. This allows for further customization.

```
early_stop_callback = EarlyStopping(
   monitor='val_accuracy',
   min_delta=0.00,
   patience=3,
   verbose=False,
   mode='max'
)
trainer = Trainer(early_stop_callback=early_stop_callback)
```

In case you need early stopping in a different part of training, subclass EarlyStopping and change where it is called:

```
class MyEarlyStopping(EarlyStopping):

    def on_validation_end(self, trainer, pl_module):
        # override this to disable early stopping at the end of val loop
        pass

    def on_train_end(self, trainer, pl_module):
        # instead, do it at the end of training loop
        self._run_early_stopping_check(trainer, pl_module)
```

**Note:** The EarlyStopping callback runs at the end of every validation epoch, which, under the default configuration, happen after every training epoch. However, the frequency of validation can be modified by setting various parameters on the *Trainer*, for example *check_val_every_n_epoch* and *val_check_interval*. It must be noted that the *patience* parameter counts the number of validation epochs with no improvement, and not the number of training epochs. Therefore, with parameters *check_val_every_n_epoch=10* and *patience=3*, the trainer will perform at least 40 training epochs before being stopped.

**See also:**

- *Trainer*
- *EarlyStopping*

## 16.4 Disable Early Stopping with callbacks on epoch end

To disable early stopping pass `False` to the *early_stop_callback*. Note that `None` will not disable early stopping but will lead to the default behaviour.

**See also:**

- *Trainer*
- *EarlyStopping*

CHAPTER

SEVENTEEN

# FAST TRAINING

There are multiple options to speed up different parts of the training by choosing to train on a subset of data. This could be done for speed or debugging purposes.

---

## 17.1 Check validation every n epochs

If you have a small dataset you might want to check validation every n epochs

```
# DEFAULT
trainer = Trainer(check_val_every_n_epoch=1)
```

---

## 17.2 Force training for min or max epochs

It can be useful to force training for a minimum number of epochs or limit to a max number.

**See also:**

*Trainer*

```
# DEFAULT
trainer = Trainer(min_epochs=1, max_epochs=1000)
```

---

## 17.3 Set validation check frequency within 1 training epoch

For large datasets it's often desirable to check validation multiple times within a training loop. Pass in a float to check that often within 1 training epoch. Pass in an int k to check every k training batches. Must use an int if using an IterableDataset.

```
# DEFAULT
trainer = Trainer(val_check_interval=0.95)

# check every .25 of an epoch
trainer = Trainer(val_check_interval=0.25)
```

(continues on next page)

(continued from previous page)

```
# check every 100 train batches (ie: for IterableDatasets or fixed frequency)
trainer = Trainer(val_check_interval=100)
```

## 17.4 Use data subset for training, validation and test

If you don't want to check 100% of the training/validation/test set (for debugging or if it's huge), set these flags.

```
# DEFAULT
trainer = Trainer(
    limit_train_batches=1.0,
    limit_val_batches=1.0,
    limit_test_batches=1.0
)

# check 10%, 20%, 30% only, respectively for training, validation and test set
trainer = Trainer(
    limit_train_batches=0.1,
    limit_val_batches=0.2,
    limit_test_batches=0.3
)
```

**Note:** `limit_train_batches`, `limit_val_batches` and `limit_test_batches` will be overwritten by `overfit_batches` if `overfit_batches` > 0. `limit_val_batches` will be ignored if `fast_dev_run=True`.

**Note:** If you set `limit_val_batches=0`, validation will be disabled.

CHAPTER

EIGHTEEN

# HYPERPARAMETERS

Lightning has utilities to interact seamlessly with the command line ArgumentParser and plays well with the hyperparameter optimization framework of your choice.

---

## 18.1 ArgumentParser

Lightning is designed to augment a lot of the functionality of the built-in Python ArgumentParser

```
from argparse import ArgumentParser
parser = ArgumentParser()
parser.add_argument('--layer_1_dim', type=int, default=128)
args = parser.parse_args()
```

This allows you to call your program like so:

```
python trainer.py --layer_1_dim 64
```

---

## 18.2 Argparser Best Practices

It is best practice to layer your arguments in three sections.

1. Trainer args (gpus, num_nodes, etc. . . )
2. Model specific arguments (layer_dim, num_layers, learning_rate, etc. . . )
3. Program arguments (data_path, cluster_email, etc. . . )

We can do this as follows. First, in your LightningModule, define the arguments specific to that module. Remember that data splits or data paths may also be specific to a module (ie: if your project has a model that trains on Imagenet and another on CIFAR-10).

```
class LitModel(LightningModule):

    @staticmethod
    def add_model_specific_args(parent_parser):
        parser = ArgumentParser(parents=[parent_parser], add_help=False)
        parser.add_argument('--encoder_layers', type=int, default=12)
        parser.add_argument('--data_path', type=str, default='/some/path')
        return parser
```

Now in your main trainer file, add the Trainer args, the program args, and add the model args

```
# ----------------
# trainer_main.py
# ----------------
from argparse import ArgumentParser
parser = ArgumentParser()

# add PROGRAM level args
parser.add_argument('--conda_env', type=str, default='some_name')
parser.add_argument('--notification_email', type=str, default='will@email.com')

# add model specific args
parser = LitModel.add_model_specific_args(parser)

# add all the available trainer options to argparse
# ie: now --gpus --num_nodes ... --fast_dev_run all work in the cli
parser = Trainer.add_argparse_args(parser)

args = parser.parse_args()
```

Now you can call run your program like so

```
python trainer_main.py --gpus 2 --num_nodes 2 --conda_env 'my_env' --encoder_layers 12
```

Finally, make sure to start the training like so:

```
# init the trainer like this
trainer = Trainer.from_argparse_args(args, early_stopping_callback=...)

# NOT like this
trainer = Trainer(gpus=hparams.gpus, ...)

# init the model with Namespace directly
model = LitModel(args)

# or init the model with all the key-value pairs
dict_args = vars(args)
model = LitModel(**dict_args)
```

---

## 18.3 LightningModule hyperparameters

Often times we train many versions of a model. You might share that model or come back to it a few months later at which point it is very useful to know how that model was trained (ie: what learning_rate, neural network, etc…).

Lightning has a few ways of saving that information for you in checkpoints and yaml files. The goal here is to improve readability and reproducibility

1. The first way is to ask lightning to save the values anything in the __init__ for you to the checkpoint. This also makes those values available via *self.hparams*.

```
class LitMNIST(LightningModule):

    def __init__(self, layer_1_dim=128, learning_rate=1e-2, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        super().__init__()
        # call this to save (layer_1_dim=128, learning_rate=1e-4) to the checkpoint
        self.save_hyperparameters()

        # equivalent
        self.save_hyperparameters(['layer_1_dim', 'learning_rate'])

        # this now works
        self.hparams.layer_1_dim
```

2. Sometimes your init might have objects or other parameters you might not want to save. In that case, choose only a few

```
class LitMNIST(LightningModule):

    def __init__(self, loss_fx, generator_network, layer_1_dim=128 **kwargs):
        super().__init__()
        self.layer_1_dim = layer_1_dim
        self.loss_fx = loss_fx

        # call this to save (layer_1_dim=128) to the checkpoint
        self.save_hyperparameters(['layer_1_dim'])

# to load specify the other args
model = LitMNIST.load_from_checkpoint(PATH, loss_fx=torch.nn.SomeOtherLoss, generator_
↪network=MyGenerator())
```

3. Assign to *self.hparams*. Anything assigned to *self.hparams* will also be saved automatically

```
# using a argparse.Namespace
class LitMNIST(LightningModule):

    def __init__(self, hparams, *args, **kwargs):
        super().__init__()
        self.hparams = hparams

        self.layer_1 = torch.nn.Linear(28 * 28, self.hparams.layer_1_dim)
        self.layer_2 = torch.nn.Linear(self.hparams.layer_1_dim, self.hparams.layer_2_
↪dim)
        self.layer_3 = torch.nn.Linear(self.hparams.layer_2_dim, 10)

    def train_dataloader(self):
        return DataLoader(mnist_train, batch_size=self.hparams.batch_size)
```

4. You can also save full objects such as *dict* or *Namespace* to the checkpoint.

```
# using a argparse.Namespace
class LitMNIST(LightningModule):

    def __init__(self, conf, *args, **kwargs):
        super().__init__()
        self.hparams = conf

        # equivalent
        self.save_hyperparameters(conf)
```

(continues on next page)

(continued from previous page)

```
        self.layer_1 = torch.nn.Linear(28 * 28, self.hparams.layer_1_dim)
        self.layer_2 = torch.nn.Linear(self.hparams.layer_1_dim, self.hparams.layer_2_
↪dim)
        self.layer_3 = torch.nn.Linear(self.hparams.layer_2_dim, 10)

conf = OmegaConf.create(...)
model = LitMNIST(conf)

# this works
model.hparams.anything
```

## 18.4 Trainer args

To recap, add ALL possible trainer flags to the argparser and init the Trainer this way

```
parser = ArgumentParser()
parser = Trainer.add_argparse_args(parser)
hparams = parser.parse_args()

trainer = Trainer.from_argparse_args(hparams)

# or if you need to pass in callbacks
trainer = Trainer.from_argparse_args(hparams, checkpoint_callback=..., callbacks=[...
↪])
```

## 18.5 Multiple Lightning Modules

We often have multiple Lightning Modules where each one has different arguments. Instead of polluting the main.py file, the LightningModule lets you define arguments for each one.

```
class LitMNIST(LightningModule):

    def __init__(self, layer_1_dim, **kwargs):
        super().__init__()
        self.layer_1 = torch.nn.Linear(28 * 28, layer_1_dim)

    @staticmethod
    def add_model_specific_args(parent_parser):
        parser = ArgumentParser(parents=[parent_parser], add_help=False)
        parser.add_argument('--layer_1_dim', type=int, default=128)
        return parser
```

```
class GoodGAN(LightningModule):

    def __init__(self, encoder_layers, **kwargs):
        super().__init__()
        self.encoder = Encoder(layers=encoder_layers)
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def add_model_specific_args(parent_parser):
        parser = ArgumentParser(parents=[parent_parser], add_help=False)
        parser.add_argument('--encoder_layers', type=int, default=12)
        return parser
```

Now we can allow each model to inject the arguments it needs in the `main.py`

```
def main(args):
    dict_args = vars(args)

    # pick model
    if args.model_name == 'gan':
        model = GoodGAN(**dict_args)
    elif args.model_name == 'mnist':
        model = LitMNIST(**dict_args)

    trainer = Trainer.from_argparse_args(args)
    trainer.fit(model)

if __name__ == '__main__':
    parser = ArgumentParser()
    parser = Trainer.add_argparse_args(parser)

    # figure out which model to use
    parser.add_argument('--model_name', type=str, default='gan', help='gan or mnist')

    # THIS LINE IS KEY TO PULL THE MODEL NAME
    temp_args, _ = parser.parse_known_args()

    # let the model add what it wants
    if temp_args.model_name == 'gan':
        parser = GoodGAN.add_model_specific_args(parser)
    elif temp_args.model_name == 'mnist':
        parser = LitMNIST.add_model_specific_args(parser)

    args = parser.parse_args()

    # train
    main(args)
```

and now we can train MNIST or the GAN using the command line interface!

```
$ python main.py --model_name gan --encoder_layers 24
$ python main.py --model_name mnist --layer_1_dim 128
```

## 18.6 Hyperparameter Optimization

Lightning is fully compatible with the hyperparameter optimization libraries! Here are some useful ones:

- Hydra
- Optuna

CHAPTER

# NINETEEN

# LEARNING RATE FINDER

For training deep neural networks, selecting a good learning rate is essential for both better performance and faster convergence. Even optimizers such as *Adam* that are self-adjusting the learning rate can benefit from more optimal choices.

To reduce the amount of guesswork concerning choosing a good initial learning rate, a *learning rate finder* can be used. As described in this paper a learning rate finder does a small run where the learning rate is increased after each processed batch and the corresponding loss is logged. The result of this is a *lr* vs. *loss* plot that can be used as guidance for choosing a optimal initial lr.

> **Warning:** For the moment, this feature only works with models having a single optimizer. LR support for DDP is not implemented yet, it is comming soon.

---

## 19.1 Using Lightning's built-in LR finder

In the most basic use case, this feature can be enabled during trainer construction with `Trainer(auto_lr_find=True)`. When `.fit(model)` is called, the LR finder will automatically run before any training is done. The `lr` that is found and used will be written to the console and logged together with all other hyperparameters of the model.

```
# default: no automatic learning rate finder
trainer = Trainer(auto_lr_find=False)
```

This flag sets your learning rate which can be accessed via `self.lr` or `self.learning_rate`.

```
class LitModel(LightningModule):

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

    def configure_optimizers(self):
        return Adam(self.parameters(), lr=(self.lr or self.learning_rate))

# finds learning rate automatically
# sets hparams.lr or hparams.learning_rate to that learning rate
trainer = Trainer(auto_lr_find=True)
```

To use an arbitrary value set it as auto_lr_find

```
# to set to your own hparams.my_value
trainer = Trainer(auto_lr_find='my_value')
```

Under the hood, when you call fit it runs the learning rate finder before actually calling fit.

```
# when you call .fit() this happens
# 1. find learning rate
# 2. actually run fit
trainer.fit(model)
```

If you want to inspect the results of the learning rate finder before doing any actual training or just play around with the parameters of the algorithm, this can be done by invoking the `lr_find` method of the trainer. A typical example of this would look like

```
model = MyModelClass(hparams)
trainer = Trainer()

# Run learning rate finder
lr_finder = trainer.lr_find(model)

# Results can be found in
lr_finder.results

# Plot with
fig = lr_finder.plot(suggest=True)
fig.show()

# Pick point based on plot, or get suggestion
new_lr = lr_finder.suggestion()

# update hparams of the model
model.hparams.lr = new_lr

# Fit model
trainer.fit(model)
```

The figure produced by `lr_finder.plot()` should look something like the figure below. It is recommended to not pick the learning rate that achives the lowest loss, but instead something in the middle of the sharpest downward slope (red point). This is the point returned py `lr_finder.suggestion()`.

The parameters of the algorithm can be seen below.

**class** `pytorch_lightning.trainer.lr_finder.`**`TrainerLRFinderMixin`**

Bases: `abc.ABC`

**`_run_lr_finder_internally`**(*model*)

Call lr finder internally during Trainer.fit()

**abstract fit**(**args*)

Warning: this is just empty shell for code implemented in other class.

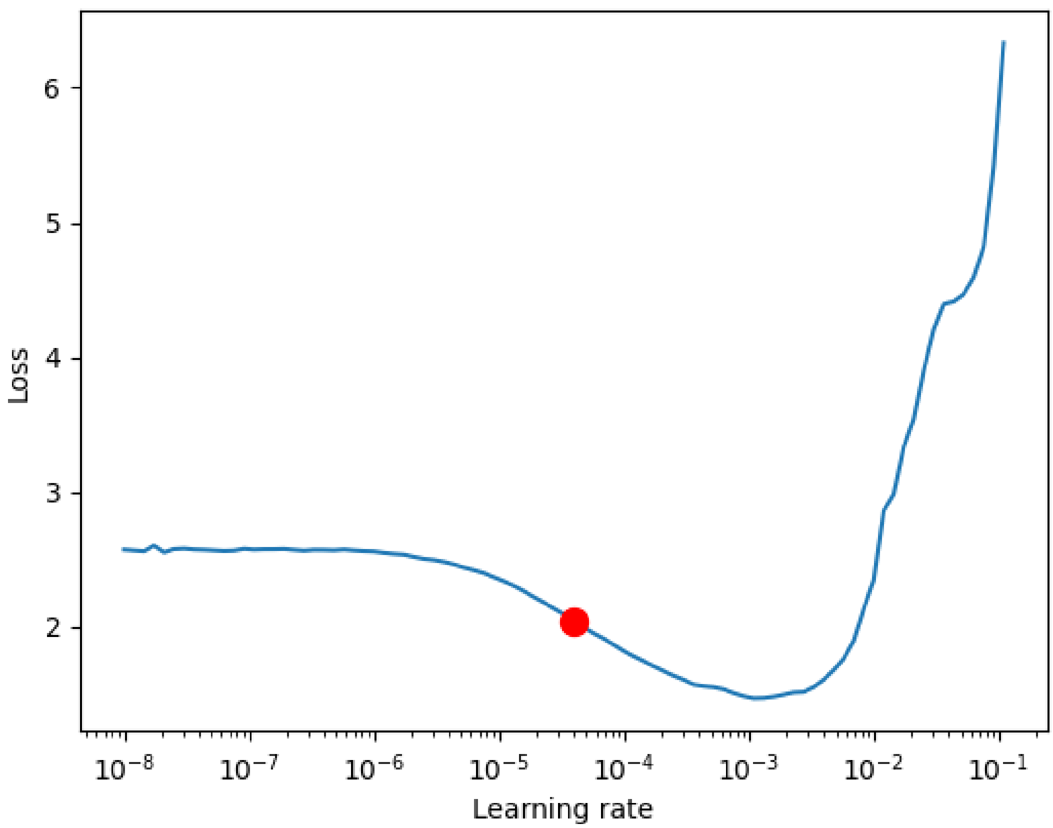
**abstract init_optimizers**(**args*)

Warning: this is just empty shell for code implemented in other class.

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

**`lr_find`**(*model*, *train_dataloader=None*, *val_dataloaders=None*, *min_lr=1e-08*, *max_lr=1*, *num_training=100*, *mode='exponential'*, *early_stop_threshold=4.0*, *num_accumulation_steps=None*)

lr_find enables the user to do a range test of good initial learning rates, to reduce the amount of guesswork


6
5
4
3
2
Loss
$10^{-8}$
$10^{-7}$
$10^{-6}$
$10^{-5}$
$10^{-4}$
$10^{-3}$
$10^{-2}$
$10^{-1}$
Learning rate

in picking a good starting learning rate.

**Parameters**

- **model** (*LightningModule*) – Model to do range testing for
- **train_dataloader** (Optional[DataLoader]) – A PyTorch DataLoader with training samples. If the model has a predefined train_dataloader method this will be skipped.
- **min_lr** (float) – minimum learning rate to investigate
- **max_lr** (float) – maximum learning rate to investigate
- **num_training** (int) – number of learning rates to test
- **mode** (str) – search strategy, either 'linear' or 'exponential'. If set to 'linear' the learning rate will be searched by linearly increasing after each batch. If set to 'exponential', will increase learning rate exponentially.
- **early_stop_threshold** (float) – threshold for stopping the search. If the loss at any point is larger than early_stop_threshold*best_loss then the search is stopped. To disable, set to None.
- **num_accumulation_steps** – deprepecated, number of batches to calculate loss over. Set trainer argument `accumulate_grad_batches` instead.

Example:

```
# Setup model and trainer
model = MyModelClass(hparams)
trainer = pl.Trainer()

# Run lr finder
lr_finder = trainer.lr_find(model, ...)

# Inspect results
fig = lr_finder.plot(); fig.show()
suggested_lr = lr_finder.suggestion()

# Overwrite lr and create new model
hparams.lr = suggested_lr
model = MyModelClass(hparams)

# Ready to train with new learning rate
trainer.fit(model)
```

**abstract restore**(**args*)
Warning: this is just empty shell for code implemented in other class.

**abstract save_checkpoint**(**args*)
Warning: this is just empty shell for code implemented in other class.

CHAPTER

TWENTY

# MULTI-GPU TRAINING

Lightning supports multiple ways of doing distributed training.

---

## 20.1 Preparing your code

To train on CPU/GPU/TPU without changing your code, we need to build a few good habits :)

### 20.1.1 Delete .cuda() or .to() calls

Delete any calls to .cuda() or .to(device).

```
# before lightning
def forward(self, x):
    x = x.cuda(0)
    layer_1.cuda(0)
    x_hat = layer_1(x)

# after lightning
def forward(self, x):
    x_hat = layer_1(x)
```

### 20.1.2 Init tensors using type_as

When you need to create a new tensor, use *type_as*. This will make your code scale to any arbitrary number of GPUs or TPUs with Lightning.

```
# before lightning
def forward(self, x):
    z = torch.Tensor(2, 3)
    z = z.cuda(0)

# with lightning
def forward(self, x):
    z = torch.Tensor(2, 3)
    z = z.type_as(x, device=self.device)
```

The LightningModule knows what device it is on. You can access the reference via *self.device*.

### 20.1.3 Remove samplers

In PyTorch, you must use *torch.nn.DistributedSampler* for multi-node or TPU training in PyTorch. The sampler makes sure each GPU sees the appropriate part of your data.

```
# without lightning
def train_dataloader(self):
    dataset = MNIST(...)
    sampler = None

    if self.on_tpu:
        sampler = DistributedSampler(dataset)

    return DataLoader(dataset, sampler=sampler)
```

Lightning adds the correct samplers when needed, so no need to explicitly add samplers.

```
# with lightning
def train_dataloader(self):
    dataset = MNIST(...)
    return DataLoader(dataset)
```

**Note:** You can disable this behavior with *Trainer(replace_sampler_ddp=False)*

**Note:** For iterable datasets, we don't do this automatically.

### 20.1.4 Make models pickleable

It's very likely your code is already pickleable, in that case no change in necessary. However, if you run a distributed model and get the following error:

```
self._launch(process_obj)
File "/net/software/local/python/3.6.5/lib/python3.6/multiprocessing/popen_spawn_
↪posix.py", line 47,
in _launch reduction.dump(process_obj, fp)
File "/net/software/local/python/3.6.5/lib/python3.6/multiprocessing/reduction.py",␣
↪line 60, in dump
ForkingPickler(file, protocol).dump(obj)
_pickle.PicklingError: Can't pickle <function <lambda> at 0x2b599e088ae8>:
attribute lookup <lambda> on __main__ failed
```

This means something in your model definition, transforms, optimizer, dataloader or callbacks cannot be pickled, and the following code will fail:

```
import pickle
pickle.dump(some_object)
```

This is a limitation of using multiple processes for distributed training within PyTorch. To fix this issue, find your piece of code that cannot be pickled. The end of the stacktrace is usually helpful. ie: in the stacktrace example here, there seems to be a lambda function somewhere in the code which cannot be pickled.

```
self._launch(process_obj)
File "/net/software/local/python/3.6.5/lib/python3.6/multiprocessing/popen_spawn_
↪posix.py", line 47,
in _launch reduction.dump(process_obj, fp)
File "/net/software/local/python/3.6.5/lib/python3.6/multiprocessing/reduction.py",␣
↪line 60, in dump
ForkingPickler(file, protocol).dump(obj)
_pickle.PicklingError: Can't pickle [THIS IS THE THING TO FIND AND DELETE]:
attribute lookup <lambda> on __main__ failed
```

---

## 20.2 Select GPU devices

You can select the GPU devices using ranges, a list of indices or a string containing a comma separated list of GPU ids:

```
# DEFAULT (int) specifies how many GPUs to use
Trainer(gpus=k)

# Above is equivalent to
Trainer(gpus=list(range(k)))

# Specify which GPUs to use (don't use when running on cluster)
Trainer(gpus=[0, 1])

# Equivalent using a string
Trainer(gpus='0, 1')

# To use all available GPUs put -1 or '-1'
# equivalent to list(range(torch.cuda.available_devices()))
Trainer(gpus=-1)
```

The table below lists examples of possible input formats and how they are interpreted by Lightning. Note in particular the difference between *gpus=0*, *gpus=[0]* and *gpus="0"*.

| *gpus* | Type | Parsed | Meaning |
|---|---|---|---|
| None | NoneType | None | CPU |
| 0 | int | None | CPU |
| 3 | int | [0, 1, 2] | first 3 GPUs |
| -1 | int | [0, 1, 2, . . . ] | all available GPUs |
| [0] | list | [0] | GPU 0 |
| [1, 3] | list | [1, 3] | GPUs 1 and 3 |
| "0" | str | [0] | GPU 0 |
| "3" | str | [3] | GPU 3 |
| "1, 3" | str | [1, 3] | GPUs 1 and 3 |
| "-1" | str | [0, 1, 2, . . . ] | all available GPUs |

### 20.2.1 Remove CUDA flags

CUDA flags make certain GPUs visible to your script. Lightning sets these for you automatically, there's NO NEED to do this yourself.

```
# lightning will set according to what you give the trainer
os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
os.environ["CUDA_VISIBLE_DEVICES"] = "0"
```

However, when using a cluster, Lightning will NOT set these flags (and you should not either). SLURM will set these for you. For more details see the *SLURM cluster guide*.

---

## 20.3 Distributed modes

Lightning allows multiple ways of training

- Data Parallel (*distributed_backend='dp'*) (multiple-gpus, 1 machine)
- DistributedDataParallel (*distributed_backend='ddp'*) (multiple-gpus across many machines (python script based)).
- DistributedDataParallel (*distributed_backend='ddp_spawn'*) (multiple-gpus across many machines (spawn based)).
- DistributedDataParallel 2 (*distributed_backend='ddp2'*) (DP in a machine, DDP across machines).
- Horovod (*distributed_backend='horovod'*) (multi-machine, multi-gpu, configured at runtime)
- TPUs (*tpu_cores=8|x*) (tpu or TPU pod)

**Note:** If you request multiple GPUs or nodes without setting a mode, DDP will be automatically used.

For a deeper understanding of what Lightning is doing, feel free to read this guide.

### 20.3.1 Data Parallel

DataParallel (DP) splits a batch across k GPUs. That is, if you have a batch of 32 and use DP with 2 gpus, each GPU will process 16 samples, after which the root node will aggregate the results.

**Warning:** DP use is discouraged by PyTorch and Lightning. Use DDP which is more stable and at least 3x faster

```
# train on 2 GPUs (using DP mode)
trainer = Trainer(gpus=2, distributed_backend='dp')
```

### 20.3.2 Distributed Data Parallel

DistributedDataParallel (DDP) works as follows:

1. Each GPU across each node gets its own process.
2. Each GPU gets visibility into a subset of the overall dataset. It will only ever see that subset.
3. Each process inits the model.

**Note:** Make sure to set the random seed so that each model initializes with the same weights.

4. Each process performs a full forward and backward pass in parallel.
5. The gradients are synced and averaged across all processes.
6. Each process updates its optimizer.

```
# train on 8 GPUs (same machine (ie: node))
trainer = Trainer(gpus=8, distributed_backend='ddp')

# train on 32 GPUs (4 nodes)
trainer = Trainer(gpus=8, distributed_backend='ddp', num_nodes=4)
```

This Lightning implementation of DDP calls your script under the hood multiple times with the correct environment variables. If your code does not support this (ie: jupyter notebook, colab, or a nested script without a root package), use *dp* or *ddp_spawn*.

```
# example for 3 GPUs DDP
MASTER_ADDR=localhost MASTER_PORT=random() WORLD_SIZE=3 NODE_RANK=0 LOCAL_RANK=0␣
↪python my_file.py --gpus 3 --etc
MASTER_ADDR=localhost MASTER_PORT=random() WORLD_SIZE=3 NODE_RANK=1 LOCAL_RANK=0␣
↪python my_file.py --gpus 3 --etc
MASTER_ADDR=localhost MASTER_PORT=random() WORLD_SIZE=3 NODE_RANK=2 LOCAL_RANK=0␣
↪python my_file.py --gpus 3 --etc
```

We use DDP this way because *ddp_spawn* has a few limitations (due to Python and PyTorch):

1. Since *.spawn()* trains the model in subprocesses, the model on the main process does not get updated.
2. Dataloader(num_workers=N), where N is large, bottlenecks training with DDP… ie: it will be VERY slow or won't work at all. This is a PyTorch limitation.
3. Forces everything to be picklable.

However, if you don't mind these limitations, you can use *ddp_spawn*.

### 20.3.3 Distributed Data Parallel 2

In certain cases, it's advantageous to use all batches on the same machine instead of a subset. For instance, you might want to compute a NCE loss where it pays to have more negative samples.

In this case, we can use DDP2 which behaves like DP in a machine and DDP across nodes. DDP2 does the following:

1. Copies a subset of the data to each node.
2. Inits a model on each node.
3. Runs a forward and backward pass using DP.

4. Syncs gradients across nodes.
5. Applies the optimizer updates.

```
# train on 32 GPUs (4 nodes)
trainer = Trainer(gpus=8, distributed_backend='ddp2', num_nodes=4)
```

### 20.3.4 Distributed Data Parallel Spawn

*ddp_spawn* is exactly like *ddp* except that it uses .spawn to start the training processes.

> **Warning:** It is STRONGLY recommended to use *DDP* for speed and performance.

```
mp.spawn(self.ddp_train, nprocs=self.num_processes, args=(model, ))
```

If your script does not support being called from the command line (ie: it is nested without a root project module) you can use the following method:

```
# train on 8 GPUs (same machine (ie: node))
trainer = Trainer(gpus=8, distributed_backend='ddp')
```

We STRONGLY discourage this use because it has limitations (due to Python and PyTorch):

1. The model you pass in will not update. Please save a checkpoint and restore from there.
2. Set Dataloader(num_workers=0) or it will bottleneck training.

*ddp* is MUCH faster than *ddp_spawn*. We recommend you

1. Install a top-level module for your project using setup.py

```
# setup.py
#!/usr/bin/env python

from setuptools import setup, find_packages

setup(name='src',
      version='0.0.1',
      description='Describe Your Cool Project',
      author='',
      author_email='',
      url='https://github.com/YourSeed',  # REPLACE WITH YOUR OWN GITHUB PROJECT LINK
      install_requires=[
            'pytorch-lightning'
      ],
      packages=find_packages()
      )
```

2. Setup your project like so:

```
/project
    /src
        some_file.py
        /or_a_folder
    setup.py
```

3. Install as a root-level package

```
cd /project
pip install -e .
```

You can then call your scripts anywhere

```
cd /project/src
python some_file.py --distributed_backend 'ddp' --gpus 8
```

### 20.3.5 Horovod

Horovod allows the same training script to be used for single-GPU, multi-GPU, and multi-node training.

Like Distributed Data Parallel, every process in Horovod operates on a single GPU with a fixed subset of the data. Gradients are averaged across all GPUs in parallel during the backward pass, then synchronously applied before beginning the next step.

The number of worker processes is configured by a driver application (*horovodrun* or *mpirun*). In the training script, Horovod will detect the number of workers from the environment, and automatically scale the learning rate to compensate for the increased total batch size.

Horovod can be configured in the training script to run with any number of GPUs / processes as follows:

```
# train Horovod on GPU (number of GPUs / machines provided on command-line)
trainer = Trainer(distributed_backend='horovod', gpus=1)

# train Horovod on CPU (number of processes / machines provided on command-line)
trainer = Trainer(distributed_backend='horovod')
```

When starting the training job, the driver application will then be used to specify the total number of worker processes:

```
# run training with 4 GPUs on a single machine
horovodrun -np 4 python train.py

# run training with 8 GPUs on two machines (4 GPUs each)
horovodrun -np 8 -H hostname1:4,hostname2:4 python train.py
```

See the official Horovod documentation for details on installation and performance tuning.

### 20.3.6 DP/DDP2 caveats

In DP and DDP2 each GPU within a machine sees a portion of a batch. DP and ddp2 roughly do the following:

```
def distributed_forward(batch, model):
    batch = torch.Tensor(32, 8)
    gpu_0_batch = batch[:8]
    gpu_1_batch = batch[8:16]
    gpu_2_batch = batch[16:24]
    gpu_3_batch = batch[24:]

    y_0 = model_copy_gpu_0(gpu_0_batch)
    y_1 = model_copy_gpu_1(gpu_1_batch)
    y_2 = model_copy_gpu_2(gpu_2_batch)
    y_3 = model_copy_gpu_3(gpu_3_batch)
```

(continues on next page)

(continued from previous page)

```
    return [y_0, y_1, y_2, y_3]
```

So, when Lightning calls any of the *training_step*, *validation_step*, *test_step* you will only be operating on one of those pieces.

```
# the batch here is a portion of the FULL batch
def training_step(self, batch, batch_idx):
    y_0 = batch
```

For most metrics, this doesn't really matter. However, if you want to add something to your computational graph (like softmax) using all batch parts you can use the *training_step_end* step.

```
def training_step_end(self, outputs):
    # only use when  on dp
    outputs = torch.cat(outputs, dim=1)
    softmax = softmax(outputs, dim=1)
    out = softmax.mean()
    return out
```

In pseudocode, the full sequence is:

```
# get data
batch = next(dataloader)

# copy model and data to each gpu
batch_splits = split_batch(batch, num_gpus)
models = copy_model_to_gpus(model)

# in parallel, operate on each batch chunk
all_results = []
for gpu_num in gpus:
    batch_split = batch_splits[gpu_num]
    gpu_model = models[gpu_num]
    out = gpu_model(batch_split)
    all_results.append(out)

# use the full batch for something like softmax
full out = model.training_step_end(all_results)
```

To illustrate why this is needed, let's look at DataParallel

```
def training_step(self, batch, batch_idx):
    x, y = batch
    y_hat = self(batch)

    # on dp or ddp2 if we did softmax now it would be wrong
    # because batch is actually a piece of the full batch
    return y_hat

def training_step_end(self, batch_parts_outputs):
    # batch_parts_outputs has outputs of each part of the batch

    # do softmax here
    outputs = torch.cat(outputs, dim=1)
    softmax = softmax(outputs, dim=1)
```

(continues on next page)

(continued from previous page)

```
    out = softmax.mean()

    return out
```

If *training_step_end* is defined it will be called regardless of TPU, DP, DDP, etc… which means it will behave the same regardless of the backend.

Validation and test step have the same option when using DP.

```
def validation_step_end(self, batch_parts_outputs):
    ...

def test_step_end(self, batch_parts_outputs):
    ...
```

### 20.3.7 Distributed and 16-bit precision

Due to an issue with Apex and DataParallel (PyTorch and NVIDIA issue), Lightning does not allow 16-bit and DP training. We tried to get this to work, but it's an issue on their end.

Below are the possible configurations we support.

| 1 GPU | 1+ GPUs | DP | DDP | 16-bit | command |
|---|---|---|---|---|---|
| Y | | | | | *Trainer(gpus=1)* |
| Y | | | | Y | *Trainer(gpus=1, use_amp=True)* |
| | Y | Y | | | *Trainer(gpus=k, distributed_backend='dp')* |
| | Y | | Y | | *Trainer(gpus=k, distributed_backend='ddp')* |
| | Y | | Y | Y | *Trainer(gpus=k, distributed_backend='ddp', use_amp=True)* |

### 20.3.8 Implement Your Own Distributed (DDP) training

If you need your own way to init PyTorch DDP you can override `pytorch_lightning.core.LightningModule.()`.

If you also need to use your own DDP implementation, override: `pytorch_lightning.core.LightningModule.configure_ddp()`.

## 20.4 Batch size

When using distributed training make sure to modify your learning rate according to your effective batch size.

Let's say you have a batch size of 7 in your dataloader.

```
class LitModel(LightningModule):

    def train_dataloader(self):
        return Dataset(..., batch_size=7)
```

In (DDP, Horovod) your effective batch size will be 7 * gpus * num_nodes.

```
# effective batch size = 7 * 8
Trainer(gpus=8, distributed_backend='ddp|horovod')

# effective batch size = 7 * 8 * 10
Trainer(gpus=8, num_nodes=10, distributed_backend='ddp|horovod')
```

In DDP2, your effective batch size will be 7 * num_nodes. The reason is that the full batch is visible to all GPUs on the node when using DDP2.

```
# effective batch size = 7
Trainer(gpus=8, distributed_backend='ddp2')

# effective batch size = 7 * 10
Trainer(gpus=8, num_nodes=10, distributed_backend='ddp2')
```

**Note:** Huge batch sizes are actually really bad for convergence. Check out: Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

## 20.5 PytorchElastic

Lightning supports the use of PytorchElastic to enable fault-tolerent and elastic distributed job scheduling. To use it, specify the 'ddp' or 'ddp2' backend and the number of gpus you want to use in the trainer.

```
Trainer(gpus=8, distributed_backend='ddp')
```

Following the PytorchElastic Quickstart documentation, you then need to start a single-node etcd server on one of the hosts:

```
etcd --enable-v2
     --listen-client-urls http://0.0.0.0:2379,http://127.0.0.1:4001
     --advertise-client-urls PUBLIC_HOSTNAME:2379
```

And then launch the elastic job with:

```
python -m torchelastic.distributed.launch
        --nnodes=MIN_SIZE:MAX_SIZE
        --nproc_per_node=TRAINERS_PER_NODE
        --rdzv_id=JOB_ID
        --rdzv_backend=etcd
        --rdzv_endpoint=ETCD_HOST:ETCD_PORT
        YOUR_LIGHTNING_TRAINING_SCRIPT.py (--arg1 ... train script args...)
```

See the official PytorchElastic documentation for details on installation and more use cases.

## 20.6 Jupyter Notebooks

Unfortunately any *ddp_* is not supported in jupyter notebooks. Please use *dp* for multiple GPUs. This is a known Jupyter issue. If you feel like taking a stab at adding this support, feel free to submit a PR!

---

## 20.7 Pickle Errors

Multi-GPU training sometimes requires your model to be pickled. If you run into an issue with pickling try the following to figure out the issue

```
import pickle

model = YourModel()
pickle.dumps(model)
```

However, if you use *ddp* the pickling requirement is not there and you should be fine. If you use *ddp_spawn* the pickling requirement remains. This is a limitation of Python.

CHAPTER

# TWENTYONE

# MULTIPLE DATASETS

Lightning supports multiple dataloaders in a few ways.

1. Create a dataloader that iterates both datasets under the hood.
2. In the validation and test loop you also have the option to return multiple dataloaders which lightning will call sequentially.

---

## 21.1 Multiple training dataloaders

For training, the best way to use multiple-dataloaders is to create a Dataloader class which wraps both your dataloaders. (This of course also works for testing and validation dataloaders).

(reference)

```
class ConcatDataset(torch.utils.data.Dataset):
    def __init__(self, *datasets):
        self.datasets = datasets

    def __getitem__(self, i):
        return tuple(d[i] for d in self.datasets)

    def __len__(self):
        return min(len(d) for d in self.datasets)

class LitModel(LightningModule):

    def train_dataloader(self):
        concat_dataset = ConcatDataset(
            datasets.ImageFolder(traindir_A),
            datasets.ImageFolder(traindir_B)
        )

        loader = torch.utils.data.DataLoader(
            concat_dataset,
            batch_size=args.batch_size,
            shuffle=True,
            num_workers=args.workers,
            pin_memory=True
        )
        return loader
```

(continues on next page)

(continued from previous page)

```
    def val_dataloader(self):
        # SAME
        ...

    def test_dataloader(self):
        # SAME
        ...
```

## 21.2 Test/Val dataloaders

For validation, test dataloaders lightning also gives you the additional option of passing in multiple dataloaders back from each call.

See the following for more details:

- *val_dataloader()*
- *test_dataloader()*

```
def val_dataloader(self):
    loader_1 = Dataloader()
    loader_2 = Dataloader()
    return [loader_1, loader_2]
```

CHAPTER

TWENTYTWO

# SAVING AND LOADING WEIGHTS

Lightning can automate saving and loading checkpoints.

## 22.1 Checkpoint saving

A Lightning checkpoint has everything needed to restore a training session including:

- 16-bit scaling factor (apex)
- Current epoch
- Global step
- Model state_dict
- State of all optimizers
- State of all learningRate schedulers
- State of all callbacks
- The hyperparameters used for that model if passed in as hparams (Argparse.Namespace)

### 22.1.1 Automatic saving

Checkpointing is enabled by default to the current working directory. To change the checkpoint path pass in:

```
trainer = Trainer(default_root_dir='/your/path/to/save/checkpoints')
```

To modify the behavior of checkpointing pass in your own callback.

```
from pytorch_lightning.callbacks import ModelCheckpoint

# DEFAULTS used by the Trainer
checkpoint_callback = ModelCheckpoint(
    filepath=os.getcwd(),
    save_top_k=True,
    verbose=True,
    monitor='val_loss',
    mode='min',
    prefix=''
)

trainer = Trainer(checkpoint_callback=checkpoint_callback)
```

Or disable it by passing

```
trainer = Trainer(checkpoint_callback=False)
```

The Lightning checkpoint also saves the arguments passed into the LightningModule init under the *module_arguments* key in the checkpoint.

```
class MyLightningModule(LightningModule):

   def __init__(self, learning_rate, *args, **kwargs):
        super().__init__()

# all init args were saved to the checkpoint
checkpoint = torch.load(CKPT_PATH)
print(checkpoint['module_arguments'])
# {'learning_rate': the_value}
```

### 22.1.2 Manual saving

You can manually save checkpoints and restore your model from the checkpointed state.

```
model = MyLightningModule(hparams)
trainer.fit(model)
trainer.save_checkpoint("example.ckpt")
new_model = MyModel.load_from_checkpoint(checkpoint_path="example.ckpt")
```

## 22.2 Checkpoint Loading

To load a model along with its weights, biases and *module_arguments* use following method.

```
model = MyLightingModule.load_from_checkpoint(PATH)

print(model.learning_rate)
# prints the learning_rate you used in this checkpoint

model.eval()
y_hat = model(x)
```

But if you don't want to use the values saved in the checkpoint, pass in your own here

```
class LitModel(LightningModule):

    def __init__(self, in_dim, out_dim):
        super().__init__()
        self.in_dim = in_dim
        self.out_dim = out_dim
        self.l1 = nn.Linear(self.in_dim, self.out_dim)
```

you can restore the model like this

```
# if you train and save the model like this it will use these values when loading
# the weights. But you can overwrite this
LitModel(in_dim=32, out_dim=10)
```

(continues on next page)

(continued from previous page)

```
# uses in_dim=32, out_dim=10
model = LitModel.load_from_checkpoint(PATH)

# uses in_dim=128, out_dim=10
model = LitModel.load_from_checkpoint(PATH, in_dim=128, out_dim=10)
```

## 22.3 Restoring Training State

If you don't just want to load weights, but instead restore the full training, do the following:

```
model = LitModel()
trainer = Trainer(resume_from_checkpoint='some/path/to/my_checkpoint.ckpt')

# automatically restores model, epoch, step, LR schedulers, apex, etc...
trainer.fit(model)
```

CHAPTER

TWENTYTHREE

# OPTIMIZATION

## 23.1 Learning rate scheduling

Every optimizer you use can be paired with any LearningRateScheduler.

```python
# no LR scheduler
def configure_optimizers(self):
    return Adam(...)

# Adam + LR scheduler
def configure_optimizers(self):
    optimizer = Adam(...)
    scheduler = ReduceLROnPlateau(optimizer, ...)
    return [optimizer], [scheduler]

# Two optimizers each with a scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return [optimizer1, optimizer2], [scheduler1, scheduler2]

# Same as above with additional params passed to the first scheduler
def configure_optimizers(self):
    optimizers = [Adam(...), SGD(...)]
    schedulers = [
        {
            'scheduler': ReduceLROnPlateau(optimizers[0], ...),
            'monitor': 'val_recall', # Default: val_loss
            'interval': 'epoch',
            'frequency': 1
        },
        LambdaLR(optimizers[1], ...)
    ]
    return optimizers, schedulers
```

## 23.2 Use multiple optimizers (like GANs)

To use multiple optimizers return > 1 optimizers from *pytorch_lightning.core.LightningModule.configure_optimizers()*

```
# one optimizer
def configure_optimizers(self):
   return Adam(...)

# two optimizers, no schedulers
def configure_optimizers(self):
   return Adam(...), SGD(...)

# Two optimizers, one scheduler for adam only
def configure_optimizers(self):
   return [Adam(...), SGD(...)], [ReduceLROnPlateau()]
```

Lightning will call each optimizer sequentially:

```
for epoch in epochs:
   for batch in data:
      for opt in optimizers:
         train_step(opt)
         opt.step()

   for scheduler in scheduler:
      scheduler.step()
```

---

## 23.3 Step optimizers at arbitrary intervals

To do more interesting things with your optimizers such as learning rate warm-up or odd scheduling, override the `optimizer_step()` function.

For example, here step optimizer A every 2 batches and optimizer B every 4 batches

```
def optimizer_step(self, current_epoch, batch_nb, optimizer, optimizer_i, second_
↪order_closure=None):
    optimizer.step()
    optimizer.zero_grad()

# Alternating schedule for optimizer steps (ie: GANs)
def optimizer_step(self, current_epoch, batch_nb, optimizer, optimizer_i, second_
↪order_closure=None):
    # update generator opt every 2 steps
    if optimizer_i == 0:
        if batch_nb % 2 == 0 :
            optimizer.step()
            optimizer.zero_grad()

    # update discriminator opt every 4 steps
    if optimizer_i == 1:
        if batch_nb % 4 == 0 :
            optimizer.step()
```

(continues on next page)

(continued from previous page)

```
            optimizer.zero_grad()

    # ...
    # add as many optimizers as you want
```

Here we add a learning-rate warm up

```
# learning rate warm-up
def optimizer_step(self, current_epoch, batch_nb, optimizer, optimizer_i, second_
↪order_closure=None):
    # warm up lr
    if self.trainer.global_step < 500:
        lr_scale = min(1., float(self.trainer.global_step + 1) / 500.)
        for pg in optimizer.param_groups:
            pg['lr'] = lr_scale * self.hparams.learning_rate

    # update params
    optimizer.step()
    optimizer.zero_grad()
```

CHAPTER

# TWENTYFOUR

# PERFORMANCE AND BOTTLENECK PROFILER

Profiling your training run can help you understand if there are any bottlenecks in your code.

## 24.1 Built-in checks

PyTorch Lightning supports profiling standard actions in the training loop out of the box, including:

- on_epoch_start
- on_epoch_end
- on_batch_start
- tbptt_split_batch
- model_forward
- model_backward
- on_after_backward
- optimizer_step
- on_batch_end
- training_step_end
- on_training_end

## 24.2 Enable simple profiling

If you only wish to profile the standard actions, you can set *profiler=True* when constructing your *Trainer* object.

```
trainer = Trainer(..., profiler=True)
```

The profiler's results will be printed at the completion of a training *fit()*.

```
Profiler Report

Action                  |  Mean duration (s)    |  Total time (s)
-----------------------------------------------------------------
on_epoch_start          |  5.993e-06            |  5.993e-06
get_train_batch         |  0.0087412            |  16.398
on_batch_start          |  5.0865e-06           |  0.0095372
```

(continues on next page)

(continued from previous page)

```
model_forward          |  0.0017818          |  3.3408
model_backward         |  0.0018283          |  3.4282
on_after_backward      |  4.2862e-06         |  0.0080366
optimizer_step         |  0.0011072          |  2.0759
on_batch_end           |  4.5202e-06         |  0.0084753
on_epoch_end           |  3.919e-06          |  3.919e-06
on_train_end           |  5.449e-06          |  5.449e-06
```

## 24.3 Advanced Profiling

If you want more information on the functions called during each event, you can use the *AdvancedProfiler*. This option uses Python's cProfiler to provide a report of time spent on *each* function called within your code.

```
profiler = AdvancedProfiler()
trainer = Trainer(..., profiler=profiler)
```

The profiler's results will be printed at the completion of a training *fit()*. This profiler report can be quite long, so you can also specify an *output_filename* to save the report instead of logging it to the output in your terminal. The output below shows the profiling for the action *get_train_batch*.

```
Profiler Report

Profile stats for: get_train_batch
        4869394 function calls (4863767 primitive calls) in 18.893 seconds
Ordered by: cumulative time
List reduced from 76 to 10 due to restriction <10>
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
3752/1876    0.011    0.000   18.887    0.010 {built-in method builtins.next}
    1876     0.008    0.000   18.877    0.010 dataloader.py:344(__next__)
    1876     0.074    0.000   18.869    0.010 dataloader.py:383(_next_data)
    1875     0.012    0.000   18.721    0.010 fetch.py:42(fetch)
    1875     0.084    0.000   18.290    0.010 fetch.py:44(<listcomp>)
    60000    1.759    0.000   18.206    0.000 mnist.py:80(__getitem__)
    60000    0.267    0.000   13.022    0.000 transforms.py:68(__call__)
    60000    0.182    0.000    7.020    0.000 transforms.py:93(__call__)
    60000    1.651    0.000    6.839    0.000 functional.py:42(to_tensor)
    60000    0.260    0.000    5.734    0.000 transforms.py:167(__call__)
```

You can also reference this profiler in your LightningModule to profile specific actions of interest. If you don't want to always have the profiler turned on, you can optionally pass a *PassThroughProfiler* which will allow you to skip profiling without having to make any code changes. Each profiler has a method *profile()* which returns a context handler. Simply pass in the name of your action that you want to track and the profiler will record performance for code executed within this context.

```python
from pytorch_lightning.profiler import Profiler, PassThroughProfiler

class MyModel(LightningModule):
    def __init__(self, profiler=None):
        self.profiler = profiler or PassThroughProfiler()

    def custom_processing_step(self, data):
        with profiler.profile('my_custom_action'):
            # custom processing step
```

(continues on next page)

(continued from previous page)

```
        return data

profiler = Profiler()
model = MyModel(profiler)
trainer = Trainer(profiler=profiler, max_epochs=1)
```

**class** pytorch_lightning.profiler.**BaseProfiler**(*output_streams=None*)
Bases: abc.ABC

If you wish to write a custom profiler, you should inhereit from this class.

**Params:** stream_out: callable

**describe**()
Logs a profile report after the conclusion of the training run.

**Return type** None

**profile**(*action_name*)
Yields a context manager to encapsulate the scope of a profiled action.

Example:

```
with self.profile('load training data'):
    # load training data code
```

The profiler will start once you've entered the context and will automatically stop once you exit the code block.

**Return type** None

**abstract start**(*action_name*)
Defines how to start recording an action.

**Return type** None

**abstract stop**(*action_name*)
Defines how to record the duration once an action is complete.

**Return type** None

**abstract summary**()
Create profiler summary in text format.

**Return type** str

**class** pytorch_lightning.profiler.**SimpleProfiler**(*output_filename=None*)
Bases: *pytorch_lightning.profiler.profilers.BaseProfiler*

This profiler simply records the duration of actions (in seconds) and reports the mean duration of each action and the total time spent over the entire training run.

**Params:**

**output_filename (str): optionally save profile results to file instead of printing** to std out when training is finished.

**describe**()
Logs a profile report after the conclusion of the training run.

**start**(*action_name*)
Defines how to start recording an action.

**Return type** None

**stop** (*action_name*)
Defines how to record the duration once an action is complete.

**Return type** None

**summary** ()
Create profiler summary in text format.

**Return type** str

**class** pytorch_lightning.profiler.**AdvancedProfiler** (*output_filename=None, line_count_restriction=1.0*)
Bases: *pytorch_lightning.profiler.profilers.BaseProfiler*

This profiler uses Python's cProfiler to record more detailed information about time spent in each function call recorded during a given action. The output is quite verbose and you should only use this if you want very detailed reports.

**Parameters**

- **output_filename** (Optional[str]) – optionally save profile results to file instead of printing to std out when training is finished.
- **line_count_restriction** (float) – this can be used to limit the number of functions reported for each action. either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines)

**describe** ()
Logs a profile report after the conclusion of the training run.

**start** (*action_name*)
Defines how to start recording an action.

**Return type** None

**stop** (*action_name*)
Defines how to record the duration once an action is complete.

**Return type** None

**summary** ()
Create profiler summary in text format.

**Return type** str

**class** pytorch_lightning.profiler.**PassThroughProfiler**
Bases: *pytorch_lightning.profiler.profilers.BaseProfiler*

This class should be used when you don't want the (small) overhead of profiling. The Trainer uses this class by default.

Params: stream_out: callable

**start** (*action_name*)
Defines how to start recording an action.

**Return type** None

**stop** (*action_name*)
Defines how to record the duration once an action is complete.

**Return type** None

**summary**()

Create profiler summary in text format.

**Return type** str

CHAPTER

# TWENTYFIVE

# SINGLE GPU TRAINING

Make sure you are running on a machine that has at least one GPU. Lightning handles all the NVIDIA flags for you, there's no need to set them yourself.

```
# train on 1 GPU (using dp mode)
trainer = Trainer(gpus=1)
```

# CHAPTER TWENTYSIX

# SEQUENTIAL DATA

Lightning has built in support for dealing with sequential data.

## 26.1 Packed sequences as inputs

When using PackedSequence, do 2 things:

1. return either a padded tensor in dataset or a list of variable length tensors in the dataloader collate_fn (example above shows the list implementation).
2. Pack the sequence in forward or training and validation steps depending on use case.

```
# For use in dataloader
def collate_fn(batch):
    x = [item[0] for item in batch]
    y = [item[1] for item in batch]
    return x, y

# In module
def training_step(self, batch, batch_nb):
    x = rnn.pack_sequence(batch[0], enforce_sorted=False)
    y = rnn.pack_sequence(batch[1], enforce_sorted=False)
```

## 26.2 Truncated Backpropagation Through Time

There are times when multiple backwards passes are needed for each batch. For example, it may save memory to use Truncated Backpropagation Through Time when training RNNs.

Lightning can handle TBTT automatically via this flag.

```
# DEFAULT (single backwards pass per batch)
trainer = Trainer(truncated_bptt_steps=None)

# (split batch into sequences of size 2)
trainer = Trainer(truncated_bptt_steps=2)
```

**Note:** If you need to modify how the batch is split, override *pytorch_lightning.core.LightningModule.tbptt_split_batch()*.

**Note:** Using this feature requires updating your LightningModule's *pytorch_lightning.core.LightningModule.training_step()* to include a *hiddens* arg.

## 26.3 Iterable Datasets

Lightning supports using IterableDatasets as well as map-style Datasets. IterableDatasets provide a more natural option when using sequential data.

**Note:** When using an IterableDataset you must set the val_check_interval to 1.0 (the default) or to an int (specifying the number of training batches to run before validation) when initializing the Trainer. This is due to the fact that the IterableDataset does not have a __len__ and Lightning requires this to calculate the validation interval when val_check_interval is less than one.

```
# IterableDataset
class CustomDataset(IterableDataset):

    def __init__(self, data):
        self.data_source

    def __iter__(self):
        return iter(self.data_source)

# Setup DataLoader
def train_dataloader(self):
    seq_data = ['A', 'long', 'time', 'ago', 'in', 'a', 'galaxy', 'far', 'far', 'away']
    iterable_dataset = CustomDataset(seq_data)

    dataloader = DataLoader(dataset=iterable_dataset, batch_size=5)
    return dataloader
```

```
# Set val_check_interval
trainer = Trainer(val_check_interval=100)
```

CHAPTER

# TWENTYSEVEN

# TRAINING TRICKS

Lightning implements various tricks to help during training

---

## 27.1 Accumulate gradients

Accumulated gradients runs K small batches of size N before doing a backwards pass. The effect is a large effective batch size of size KxN.

**See also:**

*Trainer*

```
# DEFAULT (ie: no accumulated grads)
trainer = Trainer(accumulate_grad_batches=1)
```

---

## 27.2 Gradient Clipping

Gradient clipping may be enabled to avoid exploding gradients. Specifically, this will clip the gradient norm computed over all model parameters together.

**See also:**

*Trainer*

```
# DEFAULT (ie: don't clip)
trainer = Trainer(gradient_clip_val=0)

# clip gradients with norm above 0.5
trainer = Trainer(gradient_clip_val=0.5)
```

---

## 27.3 Auto scaling of batch size

Auto scaling of batch size may be enabled to find the largest batch size that fits into memory. Larger batch size often yields better estimates of gradients, but may also result in longer training time. Inspired by https://github.com/BlackHC/toma.

**See also:**

*Trainer*

```
# DEFAULT (ie: don't scale batch size automatically)
trainer = Trainer(auto_scale_batch_size=None)

# Autoscale batch size
trainer = Trainer(auto_scale_batch_size=None|'power'|'binsearch')
```

Currently, this feature supports two modes *'power'* scaling and *'binsearch'* scaling. In *'power'* scaling, starting from a batch size of 1 keeps doubling the batch size until an out-of-memory (OOM) error is encountered. Setting the argument to *'binsearch'* continues to finetune the batch size by performing a binary search.

**Note:** This feature expects that a *batch_size* field in the *hparams* of your model, i.e., *model.hparams.batch_size* should exist and will be overridden by the results of this algorithm. Additionally, your *train_dataloader()* method should depend on this field for this feature to work i.e.

```
def train_dataloader(self):
    return DataLoader(train_dataset, batch_size=self.batch_size)
```

**Warning:** Due to these constraints, this features does *NOT* work when passing dataloaders directly to *.fit()*.

The scaling algorithm has a number of parameters that the user can control by invoking the trainer method *.scale_batch_size* themself (see description below).

```
# Use default in trainer construction
trainer = Trainer()

# Invoke method
new_batch_size = trainer.scale_batch_size(model, ...)

# Override old batch size
model.hparams.batch_size = new_batch_size

# Fit as normal
trainer.fit(model)
```

**The algorithm in short works by:**

1. Dumping the current state of the model and trainer
2. **Iteratively until convergence or maximum number of tries *max_trials* (default 25) has been reached:**

   - Call *fit()* method of trainer. This evaluates *steps_per_trial* (default 3) number of training steps. Each training step can trigger an OOM error if the tensors (training batch, weights, gradients ect.) allocated during the steps have a too large memory footprint.

- If an OOM error is encountered, decrease batch size else increase it. How much the batch size is increased/decreased is determined by the choosen stratrgy.

3. The found batch size is saved to *model.hparams.batch_size*

4. Restore the initial state of model and trainer

**class** pytorch_lightning.trainer.training_tricks.**TrainerTrainingTricksMixin**

Bases: abc.ABC

**abstract fit**(**args*)

Warning: this is just empty shell for code implemented in other class.

**abstract get_model**()

Warning: this is just empty shell for code implemented in other class.

**Return type** *LightningModule*

**abstract restore**(**args*)

Warning: this is just empty shell for code implemented in other class.

**abstract save_checkpoint**(**args*)

Warning: this is just empty shell for code implemented in other class.

**scale_batch_size**(*model, mode='power', steps_per_trial=3, init_val=2, max_trials=25, batch_arg_name='batch_size'*)

Will iteratively try to find the largest batch size for a given model that does not give an out of memory (OOM) error.

**Parameters**

- **model** (*LightningModule*) – Model to fit.
- **mode** (str) – string setting the search mode. Either *power* or *binsearch*. If mode is *power* we keep multiplying the batch size by 2, until we get an OOM error. If mode is 'binsearch', we will initially also keep multiplying by 2 and after encountering an OOM error do a binary search between the last successful batch size and the batch size that failed.
- **steps_per_trial** (int) – number of steps to run with a given batch size. Idealy 1 should be enough to test if a OOM error occurs, however in practise a few are needed
- **init_val** (int) – initial batch size to start the search with
- **max_trials** (int) – max number of increase in batch size done before algorithm is terminated

**Warning:** Batch size finder is not supported for DDP yet, it is coming soon.

CHAPTER

# TWENTYEIGHT

# TRANSFER LEARNING

## 28.1 Using Pretrained Models

Sometimes we want to use a LightningModule as a pretrained model. This is fine because a LightningModule is just a *torch.nn.Module*!

**Note:** Remember that a LightningModule is EXACTLY a torch.nn.Module but with more capabilities.

Let's use the *AutoEncoder* as a feature extractor in a separate model.

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

class AutoEncoder(LightningModule):
    def __init__(self):
        self.encoder = Encoder()
        self.decoder = Decoder()

class CIFAR10Classifier(LightningModule):
    def __init__(self):
        # init the pretrained LightningModule
        self.feature_extractor = AutoEncoder.load_from_checkpoint(PATH)
        self.feature_extractor.freeze()

        # the autoencoder outputs a 100-dim representation and CIFAR-10 has 10 classes
        self.classifier = nn.Linear(100, 10)

    def forward(self, x):
        representations = self.feature_extractor(x)
        x = self.classifier(representations)
        ...
```

We used our pretrained Autoencoder (a LightningModule) for transfer learning!

## 28.2 Example: Imagenet (computer Vision)

```
import torchvision.models as models

class ImagenetTransferLearning(LightningModule):
    def __init__(self):
        # init a pretrained resnet
        num_target_classes = 10
        self.feature_extractor = models.resnet50(
                                    pretrained=True,
                                    num_classes=num_target_classes)
        self.feature_extractor.eval()

        # use the pretrained model to classify cifar-10 (10 image classes)
        self.classifier = nn.Linear(2048, num_target_classes)

    def forward(self, x):
        representations = self.feature_extractor(x)
        x = self.classifier(representations)
        ...
```

Finetune

```
model = ImagenetTransferLearning()
trainer = Trainer()
trainer.fit(model)
```

And use it to predict your data of interest

```
model = ImagenetTransferLearning.load_from_checkpoint(PATH)
model.freeze()

x = some_images_from_cifar10()
predictions = model(x)
```

We used a pretrained model on imagenet, finetuned on CIFAR-10 to predict on CIFAR-10. In the non-academic world we would finetune on a tiny dataset you have and predict on your dataset.

## 28.3 Example: BERT (NLP)

Lightning is completely agnostic to what's used for transfer learning so long as it is a *torch.nn.Module* subclass.

Here's a model that uses Huggingface transformers.

```
class BertMNLIFinetuner(LightningModule):

    def __init__(self):
        super().__init__()

        self.bert = BertModel.from_pretrained('bert-base-cased', output_
↪attentions=True)
        self.W = nn.Linear(bert.config.hidden_size, 3)
        self.num_classes = 3
```

(continues on next page)

(continued from previous page)

```
    def forward(self, input_ids, attention_mask, token_type_ids):

        h, _, attn = self.bert(input_ids=input_ids,
                               attention_mask=attention_mask,
                               token_type_ids=token_type_ids)

        h_cls = h[:, 0]
        logits = self.W(h_cls)
        return logits, attn
```

CHAPTER

# TWENTYNINE

# TPU SUPPORT

Lightning supports running on TPUs. At this moment, TPUs are available on Google Cloud (GCP), Google Colab and Kaggle Environments. For more information on TPUs watch this video.

---

## 29.1 Live demo

Check out this Google Colab to see how to train MNIST on TPUs.

---

## 29.2 TPU Terminology

A TPU is a Tensor processing unit. Each TPU has 8 cores where each core is optimized for 128x128 matrix multiplies. In general, a single TPU is about as fast as 5 V100 GPUs!

A TPU pod hosts many TPUs on it. Currently, TPU pod v2 has 2048 cores! You can request a full pod from Google cloud or a "slice" which gives you some subset of those 2048 cores.

---

## 29.3 How to access TPUs

To access TPUs there are two main ways.

1. Using google colab.
2. Using Google Cloud (GCP).
3. Using Kaggle.

---

## 29.4 Colab TPUs

Colab is like a jupyter notebook with a free GPU or TPU hosted on GCP.

To get a TPU on colab, follow these steps:

1. Go to https://colab.research.google.com/.
2. Click "new notebook" (bottom right of pop-up).
3. Click runtime > change runtime settings. Select Python 3, and hardware accelerator "TPU". This will give you a TPU with 8 cores.
4. Next, insert this code into the first cell and execute. This will install the xla library that interfaces between PyTorch and the TPU.

```
!curl https://raw.githubusercontent.com/pytorch/xla/master/contrib/scripts/env-
↪setup.py -o pytorch-xla-env-setup.py
!python pytorch-xla-env-setup.py --version nightly --apt-packages libomp5␣
↪libopenblas-dev
```

5. Once the above is done, install PyTorch Lightning (v 0.7.0+).

```
!pip install pytorch-lightning
```

6. Then set up your LightningModule as normal.

---

## 29.5 DistributedSamplers

Lightning automatically inserts the correct samplers - no need to do this yourself!

Usually, with TPUs (and DDP), you would need to define a DistributedSampler to move the right chunk of data to the appropriate TPU. As mentioned, this is not needed in Lightning

**Note:** Don't add distributedSamplers. Lightning does this automatically

If for some reason you still need to, this is how to construct the sampler for TPU use

```
import torch_xla.core.xla_model as xm

def train_dataloader(self):
    dataset = MNIST(
        os.getcwd(),
        train=True,
        download=True,
        transform=transforms.ToTensor()
    )

    # required for TPU support
    sampler = None
    if use_tpu:
        sampler = torch.utils.data.distributed.DistributedSampler(
            dataset,
```

(continues on next page)

(continued from previous page)

```
            num_replicas=xm.xrt_world_size(),
            rank=xm.get_ordinal(),
            shuffle=True
        )

    loader = DataLoader(
        dataset,
        sampler=sampler,
        batch_size=32
    )

    return loader
```

Configure the number of TPU cores in the trainer. You can only choose 1 or 8. To use a full TPU pod skip to the TPU pod section.

```
import pytorch_lightning as pl

my_model = MyLightningModule()
trainer = pl.Trainer(tpu_cores=8)
trainer.fit(my_model)
```

That's it! Your model will train on all 8 TPU cores.

---

## 29.6 Single TPU core training

Lightning supports training on a single TPU core. Just pass the TPU core ID [1-8] in a list.

```
trainer = pl.Trainer(tpu_cores=[1])
```

---

## 29.7 Distributed Backend with TPU

The `` `distributed_backend` `` option used for GPUs does not apply to TPUs. TPUs work in DDP mode by default (distributing over each core)

---

## 29.8 TPU Pod

To train on more than 8 cores, your code actually doesn't change! All you need to do is submit the following command:

```
$ python -m torch_xla.distributed.xla_dist
--tpu=$TPU_POD_NAME
--conda-env=torch-xla-nightly
-- python /usr/share/torch-xla-0.5/pytorch/xla/test/test_train_imagenet.py --fake_data
```

---

## 29.9 16 bit precision

Lightning also supports training in 16-bit precision with TPUs. By default, TPU training will use 32-bit precision. To enable 16-bit, also set the 16-bit flag.

```
import pytorch_lightning as pl

my_model = MyLightningModule()
trainer = pl.Trainer(tpu_cores=8, precision=16)
trainer.fit(my_model)
```

Under the hood the xla library will use the bfloat16 type.

---

## 29.10 About XLA

XLA is the library that interfaces PyTorch with the TPUs. For more information check out XLA.

Guide for troubleshooting XLA

CHAPTER

THIRTY

# TEST SET

Lightning forces the user to run the test set separately to make sure it isn't evaluated by mistake.

---

## 30.1 Test after fit

To run the test set after training completes, use this method.

```
# run full training
trainer.fit(model)

# (1) load the best checkpoint automatically (lightning tracks this for you)
trainer.test()

# (2) don't load a checkpoint, instead use the model with the latest weights
trainer.test(ckpt_path=None)

# (3) test using a specific checkpoint
trainer.test(ckpt_path='/path/to/my_checkpoint.ckpt')

# (4) test with an explicit model (will use this model and not load a checkpoint)
trainer.test(model)
```

---

## 30.2 Test multiple models

You can run the test set on multiple models using the same trainer instance.

```
model1 = LitModel()
model2 = GANModel()

trainer = Trainer()
trainer.test(model1)
trainer.test(model2)
```

---

## 30.3 Test pre-trained model

To run the test set on a pre-trained model, use this method.

```
model = MyLightningModule.load_from_checkpoint(
    checkpoint_path='/path/to/pytorch_checkpoint.ckpt',
    hparams_file='/path/to/test_tube/experiment/version/hparams.yaml',
    map_location=None
)

# init trainer with whatever options
trainer = Trainer(...)

# test (pass in the model)
trainer.test(model)
```

In this case, the options you pass to trainer will be used when running the test set (ie: 16-bit, dp, ddp, etc…)

---

## 30.4 Test with additional data loaders

You can still run inference on a test set even if the *test_dataloader* method hasn't been defined within your `LightningModule` instance. This would be the case when your test data is not available at the time your model was declared.

```
# setup your data loader
test = DataLoader(...)

# test (pass in the loader)
trainer.test(test_dataloaders=test)
```

You can either pass in a single dataloader or a list of them. This optional named parameter can be used in conjunction with any of the above use cases.

CHAPTER

# THIRTYONE

# CONTRIBUTOR COVENANT CODE OF CONDUCT

## 31.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

## 31.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

## 31.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

## 31.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

## 31.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at waf2107@columbia.edu. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

## 31.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq

CHAPTER

THIRTYTWO

# CONTRIBUTING

Welcome to the PyTorch Lightning community! We're building the most advanced research platform on the planet to implement the latest, best practices that the amazing PyTorch team rolls out!

## 32.1 Main Core Value: One less thing to remember

Simplify the API as much as possible from the user perspective. Any additions or improvements should minimize the things the user needs to remember.

For example: One benefit of the validation_step is that the user doesn't have to remember to set the model to .eval(). This helps users avoid all sorts of subtle errors.

## 32.2 Lightning Design Principles

We encourage all sorts of contributions you're interested in adding! When coding for lightning, please follow these principles.

### 32.2.1 No PyTorch Interference

We don't want to add any abstractions on top of pure PyTorch. This gives researchers all the control they need without having to learn yet another framework.

### 32.2.2 Simple Internal Code

It's useful for users to look at the code and understand very quickly what's happening. Many users won't be engineers. Thus we need to value clear, simple code over condensed ninja moves. While that's super cool, this isn't the project for that :)

### 32.2.3 Force User Decisions To Best Practices

There are 1,000 ways to do something. However, eventually one popular solution becomes standard practice, and everyone follows. We try to find the best way to solve a particular problem, and then force our users to use it for readability and simplicity. A good example is accumulated gradients. There are many different ways to implement it, we just pick one and force users to use it. A bad forced decision would be to make users use a specific library to do something.

When something becomes a best practice, we add it to the framework. This is usually something like bits of code in utils or in the model file that everyone keeps adding over and over again across projects. When this happens, bring that code inside the trainer and add a flag for it.

### 32.2.4 Simple External API

What makes sense to you may not make sense to others. When creating an issue with an API change suggestion, please validate that it makes sense for others. Treat code changes the way you treat a startup: validate that it's a needed feature, then add if it makes sense for many people.

### 32.2.5 Backward-compatible API

We all hate updating our deep learning packages because we don't want to refactor a bunch of stuff. In Lightning, we make sure every change we make which could break an API is backwards compatible with good deprecation warnings.

**You shouldn't be afraid to upgrade Lightning :)**

### 32.2.6 Gain User Trust

As a researcher you can't have any part of your code going wrong. So, make thorough tests to ensure that every implementation of a new trick or subtle change is correct.

### 32.2.7 Interoperability

Have a favorite feature from other libraries like fast.ai or transformers? Those should just work with lightning as well. Grab your favorite model or learning rate scheduler from your favorite library and run it in Lightning.

---

## 32.3 Contribution Types

We are currently looking for help implementing new features or adding bug fixes.

A lot of good work has already been done in project mechanics (requirements/base.txt, setup.py, pep8, badges, ci, etc…) so we're in a good state there thanks to all the early contributors (even pre-beta release)!

### 32.3.1 Bug Fixes:

1. Submit a github issue - try to describe what happened so others can reproduce it too (config, code samples, expected vs. actual behaviour). Note, that the sample code shall be minimal and if needed with publicly available data.
2. Try to fix it or recommend a solution… We highly recommend to use test driven approach
    - convert your minimal code example to a unit/integration test with assert on expected results
    - start with debugging the issue… you can run just this particular test in your IDE and draft a fix
    - verify that your test case fails on the master branch and only passes with the fix applied
3. Submit a PR!

***Note****, even if you do not find the solution, sending a PR with a test covering the issue is a valid contribution and we can help you or finish it with you :]*

### 32.3.2 New Features:

1. Submit a github issue - describe what is the motivation of such feature (adding the use case or an example is helpful).
2. Let's discuss to determine the feature scope.
3. Submit a PR! (with updated docs and tests).

---

## 32.4 Guidelines

### 32.4.1 Original code

All added or edited code shall be the own original work of the particular contributor. If you use some third-party implementation, all such blocks/functions/modules shall be properly referred and if possible also agreed by code's author. For example - `This code is inpired from http://...`. In case you adding new dependencies, make sure that they are compatible with the actual PyTorch Lightning license (ie. dependencies should be *at least* as permissive as the PyTorch Lightning license).

### 32.4.2 Coding Style

1. Use f-strings for output formation (except logging when we stay with lazy `logging.info("Hello %s!`, name);
2. Black code formatter is used using `pre-commit` hook.

### 32.4.3 Documentation

We are using Sphinx with Napoleon extension. Moreover we set Google style to follow with type convention.

- Napoleon formatting with Google style
- ReStructured Text (reST)
- Paragraph-level markup

See following short example of a sample function taking one position string and optional

```python
from typing import Optional

def my_func(param_a: int, param_b: Optional[float] = None) -> str:
    """Sample function.

    Args:
        param_a: first parameter
        param_b: second parameter

    Return:
        sum of both numbers

    Example:
        Sample doctest example...
        >>> my_func(1, 2)
        3

    .. note:: If you want to add something.
    """
    p = param_b if param_b else 0
    return str(param_a + p)
```

When updating the docs make sure to build them first locally and visually inspect the html files (in the browser) for formatting errors. In certain cases, a missing blank line or a wrong indent can lead to a broken layout. Run these commands

```
pip install -r requirements/docs.txt
cd docs
make html
```

and open `docs/build/html/index.html` in your browser.

Notes:

- You need to have LaTeX installed for rendering math equations. You can for example install TeXLive by doing one of the following:
  - on Ubuntu (Linux) run `apt-get install texlive` or otherwise follow the instructions on the TeXLive website
  - use the RTD docker image
- with PL used class meta you need to use python 3.7 or higher

When you send a PR the continuous integration will run tests and build the docs. You can access a preview of the html pages in the *Artifacts* tab in CircleCI when you click on the task named *ci/circleci: Build-Docs* at the bottom of the PR page.

### 32.4.4 Testing

Testing your work locally will help you speed up the process since it allows you to focus on particular (failing) test-cases. To setup a local development environment, install both local and test dependencies:

```
python -m pip install -r requirements/devel.txt
python -m pip install -r requirements/examples.txt
python -m pip pre-commit install
```

You can run the full test-case in your terminal via this bash script:

```
bash .run_local_tests.sh
```

Note: if your computer does not have multi-GPU nor TPU these tests are skipped.

For convenience, you can also use your own CircleCI building which will be triggered with each commit. This is useful if you do not test against all required dependency versions. To do so, login to CircleCI and enable your forked project in the dashboard. It will just work after that.

### 32.4.5 Pull Request

We welcome any useful contribution! For your convenience here's a recommended workflow:

1. Think about what you want to do - fix a bug, repair docs, etc.
2. Start your work locally (usually until you need our CI testing)
    - create a branch and prepare your changes
    - hint: do not work with your master directly, it may become complicated when you need to rebase
    - hint: give your PR a good name! it will be useful later when you may work on multiple tasks/PRs
3. Create a "Draft PR" which is clearly marked, to let us know you don't need feedback yet.
4. When you feel ready for integrating your work, mark your PR "Ready for review".
5. Use tags in PR name for following cases:
    - **[blocked by #]** if you work is depending on others changes
    - **[wip]** when you start to re-edit your work, mark it so no one will accidentally merge it in meantime

### 32.4.6 Question & Answer

1. **How can I help/contribute?**

   All help is very welcome - reporting bugs, solving issues and preparing bug fixes. To solve some issues you can start with label good first issue or chose something close to your domain with label help wanted. Before you start to implement anything check that the issue description that it is clear and self-assign the task to you (if it is not possible, just comment that you take it and we assign it to you…).

2. **Is there a recommendation for branch names?**

   We do not rely on the name convention so far you are working with your own fork. Anyway it would be nice to follow this convention `<type>/<issue-id>_<short-name>` where the types are: `bugfix`, `feature`, `docs`, `tests`, …

3. **How to rebase my PR?**

   We recommend creating a PR in separate branch other than `master`, especially if you plan submitting several changes and do not want to wait until the fist one is resolved (we can work on them in parallel). Update your master with upstream (assuming you have already set upstream)

   ```
   git fetch --all --prune
   git checkout master
   git merge upstream/master
   ```

   checkout your feature branch

   ```
   git checkout my-PR-branch
   git rebase master
   # follow git instructions to resolve conflists
   git push -f
   ```

CHAPTER

THIRTYTHREE

# HOW TO BECOME A CORE CONTRIBUTOR

Thanks for your interest in joining the Lightning team! We're a rapidly growing project which is poised to become the go-to framework for DL researchers! We're currently recruiting for a team of 5 core maintainers.

As a core maintainer you will have a strong say in the direction of the project. Big changes will require a majority of maintainers to agree.

## 33.1 Code of conduct

First and foremost, you'll be evaluated against these core values. Any code we commit or feature we add needs to align with those core values.

## 33.2 The bar for joining the team

Lightning is being used to solve really hard problems at the top AI labs in the world. As such, the bar for adding team members is extremely high. Candidates must have solid engineering skills, have a good eye for user experience, and must be a power user of Lightning and PyTorch.

With that said, the Lightning team will be diverse and a reflection of an inclusive AI community. You don't have to be an engineer to conntribute! Scientists with great usability intuition and PyTorch ninja skills are welcomed!

## 33.3 Responsibilities:

The responsibilities mainly revolve around 3 things.

### 33.3.1 Github issues

- Here we want to help users have an amazing experience. These range from questions from new people getting into DL to questions from researchers about doing something esoteric with Lightning Often, these issues require some sort of bug fix, document clarification or new functionality to be scoped out.
- To become a core member you must resolve at least 10 Github issues which align with the API design goals for Lightning. By the end of these 10 issues I should feel comfortable in the way you answer user questions Pleasant/helpful tone.
- Can abstract from that issue or bug into functionality that might solve other related issues or makes the platform more flexible.

- Don't make users feel like they don't know what they're doing. We're here to help and to make everyone's experience delightful.

### 33.3.2 Pull requests

- Here we need to ensure the code that enters Lightning is high quality. For each PR we need to:
- Make sure code coverage does not decrease
- Documents are updated
- Code is elegant and simple
- Code is NOT overly engineered or hard to read
- Ask yourself, could a non-engineer understand what's happening here?
- Make sure new tests are written
- Is this NECESSARY for Lightning? There are some PRs which are just purely about adding engineering complexity which have no place in Lightning. Guidance
- Some other PRs are for people who are wanting to get involved and add something unnecessary. We do want their help though! So don't approve the PR, but direct them to a Github issue that they might be interested in helping with instead!
- To be considered for core contributor, please review 10 PRs and help the authors land it on master. Once you've finished the review, ping me for a sanity check. At the end of 10 PRs if your PR reviews are inline with expectations described above, then you can merge PRs on your own going forward, otherwise we'll do a few more until we're both comfortable :)

### 33.3.3 Project directions

There are some big decisions which the project must make. For these I expect core contributors to have something meaningful to add if it's their area of expertise.

### 33.3.4 Diversity

Lightning should reflect the broader community it serves. As such we should have scientists/researchers from different fields contributing!

The first 5 core contributors will fit this profile. Thus if you overlap strongly with experiences and expertise as someone else on the team, you might have to wait until the next set of contributors are added.

### 33.3.5 Summary: Requirements to apply

- Solve 10 Github issues. The goal is to be inline with expectations for solving issues by the last one so you can do them on your own. If not, I might ask you to solve a few more specific ones.
- Do 10 PR reviews. The goal is to be inline with expectations for solving issues by the last one so you can do them on your own. If not, I might ask you to solve a few more specific ones.

If you want to be considered, ping me on gitter and start tracking your progress here.

CHAPTER

# THIRTYFOUR

## WHAT DOES THIS PR DO?

Fixes # (issue)

CHAPTER

THIRTYFIVE

# BEFORE SUBMITTING

- [ ] Was this discussed/approved via a Github issue? (no need for typos and docs improvements)
- [ ] Did you read the contributor guideline, Pull Request section?
- [ ] Did you make sure your PR does only one thing, instead of bundling different changes together? Otherwise, we ask you create a separate PR for every change.
- [ ] Did you make sure to update the documentation with your changes?
- [ ] Did you write any new necessary tests?
- [ ] Did you verify new and existing tests pass locally with your changes?
- [ ] If you made a notable change (that affects users), did you update the CHANGELOG?

## 35.1 PR review

Anyone in the community is free to review the PR once the tests have passed.If we didn't discuss your PR in Github issues there's a high chance it will not be merged.

## 35.2 Did you have fun?

Make sure you had fun coding

CHAPTER

THIRTYSIX

# PYTORCH LIGHTNING GOVERNANCE | PERSONS OF INTEREST

## 36.1 Leads

- William Falcon (williamFalcon) (Lightning founder)
- Jirka Borovec (Borda)
- Ethan Harris (ethanwharris) (Torchbearer founder)
- Matthew Painter (MattPainter01) (Torchbearer founder)
- Justus Schock (justusschock) (Former Core Member PyTorch Ignite)

## 36.2 Core Maintainers

- Nic Eggert (neggert)
- Jeff Ling (jeffling)
- Jeremy Jordan (jeremyjordan)
- Tullie Murrell (tullie)
- Adrian Wälchli (awaelchli)
- Nicki Skafte (skaftenicki)

CHAPTER

# THIRTYSEVEN

# INDICES AND TABLES

- genindex
- modindex
- search

## 37.1 pytorch_lightning.core package

A *LightningModule* organizes your PyTorch code into the following sections:

Notice a few things.

1. It's the SAME code.
2. The PyTorch code IS NOT abstracted - just organized.
3. All the other code that's not in the *LightningModule* has been automated for you by the trainer.

```
net = Net()
trainer = Trainer()
trainer.fit(net)
```

4. There are no .cuda() or .to() calls… Lightning does these for you.

```
# don't do in lightning
x = torch.Tensor(2, 3)
x = x.cuda()
x = x.to(device)

# do this instead
x = x  # leave it alone!

# or to init a new tensor
new_x = torch.Tensor(2, 3)
new_x = new_x.type_as(x.type())
```

5. There are no samplers for distributed, Lightning also does this for you.

```
# Don't do in Lightning...
data = MNIST(...)
sampler = DistributedSampler(data)
DataLoader(data, sampler=sampler)

# do this instead
```

(continues on next page)

PyTorch

```python
# model
class Net(nn.Module):
  def __init__(self):
      self.layer_1 = torch.nn.Linear(28 * 28, 128)
      self.layer_2 = torch.nn.Linear(128, 10)

  def forward(self, x):
    x = x.view(x.size(0), -1)
    x = self.layer_1(x)
    x = F.relu(x)
    x = self.layer_2(x)
    return x

# train loader
mnist_train = MNIST(os.getcwd(), train=True, download=True,
                    transform=transforms.ToTensor())
mnist_train = DataLoader(mnist_train, batch_size=64)

net = Net()

# optimizer + scheduler
optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)
scheduler = StepLR(optimizer, step_size=1)

# train
for epoch in range(1, 100):
   model.train()
   for batch_idx, (data, target) in enumerate(train_loader):
      data, target = data.to(device), target.to(device)
      optimizer.zero_grad()
      output = model(data)
      loss = F.nll_loss(output, target)

      loss.backward()
      optimizer.step()
      if batch_idx % args.log_interval == 0:
          print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
              epoch, batch_idx * len(data), len(train_loader.dataset),
              100. * batch_idx / len(train_loader), loss.item()))
```

PyTorch Lightning

```python
# model
class Net(LightningModule):
  def __init__(self):
      self.layer_1 = torch.nn.Linear(28 * 28, 128)
      self.layer_2 = torch.nn.Linear(128, 10)

  def forward(self, x):
    x = x.view(x.size(0), -1)
    x = self.layer_1(x)
    x = F.relu(x)
    x = self.layer_2(x)
    return x

  def train_dataloader(self):
    mnist_train = MNIST(os.getcwd(), train=True, download=True,
                        transform=transforms.ToTensor())
    return DataLoader(mnist_train, batch_size=64)

  def configure_optimizers(self):
    optimizer = torch.optim.Adam(self.parameters(), lr=1e-3)
    scheduler = StepLR(optimizer, step_size=1)
    return optimizer, scheduler

  def training_step(self, batch, batch_idx):
    data, target = batch
    output = self.forward(data)
    loss = F.nll_loss(output, target)
    return {'loss': loss}
```

(continued from previous page)

```
data = MNIST(...)
DataLoader(data)
```

6. A *LightningModule* is a `torch.nn.Module` but with added functionality. Use it as such!

```
net = Net.load_from_checkpoint(PATH)
net.freeze()
out = net(x)
```

Thus, to use Lightning, you just need to organize your code which takes about 30 minutes, (and let's be real, you probably should do anyhow).

---

### 37.1.1 Minimal Example

Here are the only required methods.

```
>>> import pytorch_lightning as pl
>>> class LitModel(pl.LightningModule):
...
...     def __init__(self):
...         super().__init__()
...         self.l1 = torch.nn.Linear(28 * 28, 10)
...
...     def forward(self, x):
...         return torch.relu(self.l1(x.view(x.size(0), -1)))
...
...     def training_step(self, batch, batch_idx):
...         x, y = batch
...         y_hat = self(x)
...         return {'loss': F.cross_entropy(y_hat, y)}
...
...     def train_dataloader(self):
...         return DataLoader(MNIST(os.getcwd(), train=True, download=True,
...                           transform=transforms.ToTensor()), batch_size=32)
...
...     def configure_optimizers(self):
...         return torch.optim.Adam(self.parameters(), lr=0.02)
```

Which you can train by doing:

```
trainer = pl.Trainer()
model = LitModel()

trainer.fit(model)
```

---

### 37.1.2 Training loop structure

The general pattern is that each loop (training, validation, test loop) has 3 methods:

- `___step`
- `___step_end`
- `___epoch_end`

To show how Lightning calls these, let's use the validation loop as an example:

```
val_outs = []
for val_batch in val_data:
    # do something with each batch
    out = validation_step(val_batch)
    val_outs.append(out)

# do something with the outputs for all batches
# like calculate validation set accuracy or loss
validation_epoch_end(val_outs)
```

If we use dp or ddp2 mode, we can also define the `XXX_step_end` method to operate on all parts of the batch:

```
val_outs = []
for val_batch in val_data:
    batches = split_batch(val_batch)
    dp_outs = []
    for sub_batch in batches:
        dp_out = validation_step(sub_batch)
        dp_outs.append(dp_out)

    out = validation_step_end(dp_outs)
    val_outs.append(out)

# do something with the outputs for all batches
# like calculate validation set accuracy or loss
validation_epoch_end(val_outs)
```

#### Add validation loop

Thus, if we wanted to add a validation loop you would add this to your *LightningModule*:

```
>>> class LitModel(pl.LightningModule):
...     def validation_step(self, batch, batch_idx):
...         x, y = batch
...         y_hat = self(x)
...         return {'val_loss': F.cross_entropy(y_hat, y)}
...
...     def validation_epoch_end(self, outputs):
...         val_loss_mean = torch.stack([x['val_loss'] for x in outputs]).mean()
...         return {'val_loss': val_loss_mean}
...
...     def val_dataloader(self):
...         # can also return a list of val dataloaders
...         return DataLoader(...)
```

### Add test loop

```
>>> class LitModel(pl.LightningModule):
...     def test_step(self, batch, batch_idx):
...         x, y = batch
...         y_hat = self(x)
...         return {'test_loss': F.cross_entropy(y_hat, y)}
...
...     def test_epoch_end(self, outputs):
...         test_loss_mean = torch.stack([x['test_loss'] for x in outputs]).mean()
...         return {'test_loss': test_loss_mean}
...
...     def test_dataloader(self):
...         # can also return a list of test dataloaders
...         return DataLoader(...)
```

However, the test loop won't ever be called automatically to make sure you don't run your test data by accident. Instead you have to explicitly call:

```
# call after training
trainer = Trainer()
trainer.fit(model)
trainer.test()

# or call with pretrained model
model = MyLightningModule.load_from_checkpoint(PATH)
trainer = Trainer()
trainer.test(model)
```

---

## 37.1.3 Training_step_end method

When using *LightningDataParallel* or *LightningDistributedDataParallel*, the *training_step()* will be operating on a portion of the batch. This is normally ok but in special cases like calculating NCE loss using negative samples, we might want to perform a softmax across all samples in the batch.

For these types of situations, each loop has an additional `__step_end` method which allows you to operate on the pieces of the batch:

```
training_outs = []
for train_batch in train_data:
    # dp, ddp2 splits the batch
    sub_batches = split_batches_for_dp(batch)

    # run training_step on each piece of the batch
    batch_parts_outputs = [training_step(sub_batch) for sub_batch in sub_batches]

    # do softmax with all pieces
    out = training_step_end(batch_parts_outputs)
    training_outs.append(out)

# do something with the outputs for all batches
# like calculate validation set accuracy or loss
training_epoch_end(val_outs)
```

---

### 37.1.4 Remove cuda calls

In a *LightningModule*, all calls to `.cuda()` and `.to(device)` should be removed. Lightning will do these automatically. This will allow your code to work on CPUs, TPUs and GPUs.

When you init a new tensor in your code, just use `type_as()`:

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

    # put the z on the appropriate gpu or tpu core
    z = sample_noise()
    z = z.type_as(x)
```

---

### 37.1.5 Data preparation

Data preparation in PyTorch follows 5 steps:

1. Download
2. Clean and (maybe) save to disk
3. Load inside `Dataset`
4. Apply transforms (rotate, tokenize, etc…)
5. Wrap inside a `DataLoader`

When working in distributed settings, steps 1 and 2 have to be done from a single GPU, otherwise you will overwrite these files from every GPU. The *LightningModule* has the *prepare_data* method to allow for this:

```
>>> class LitModel(pl.LightningModule):
...     def prepare_data(self):
...         # download
...         MNIST(os.getcwd(), train=True, download=True, transform=transforms.
↪ToTensor())
...         MNIST(os.getcwd(), train=False, download=True, transform=transforms.
↪ToTensor())
...
...     def setup(self, stage):
...         mnist_train = MNIST(os.getcwd(), train=True, download=False,␣
↪transform=transforms.ToTensor())
...         mnist_test = MNIST(os.getcwd(), train=False, download=False,␣
↪transform=transforms.ToTensor())
...         # train/val split
...         mnist_train, mnist_val = random_split(mnist_train, [55000, 5000])
...
...         # assign to use in dataloaders
...         self.train_dataset = mnist_train
...         self.val_dataset = mnist_val
...         self.test_dataset = mnist_test
...
...     def train_dataloader(self):
...         return DataLoader(self.train_dataset, batch_size=64)
...
...     def val_dataloader(self):
...         return DataLoader(self.mnist_val, batch_size=64)
```

(continues on next page)

(continued from previous page)

```
...
...     def test_dataloader(self):
...         return DataLoader(self.mnist_test, batch_size=64)
```

**Note:** *prepare_data()* is called once.

**Note:** Do anything with data that needs to happen ONLY once here, like download, tokenize, etc…

### 37.1.6 Lifecycle

The methods in the *LightningModule* are called in this order:

1. `__init__()`
2. *prepare_data()*
3. *configure_optimizers()*
4. *train_dataloader()*

If you define a validation loop then

5. *val_dataloader()*

And if you define a test loop:

6. *test_dataloader()*

**Note:** *test_dataloader()* is only called with `.test()`

In every epoch, the loop methods are called in this frequency:

1. *validation_step()* called every batch
2. *validation_epoch_end()* called every epoch

### 37.1.7 Live demo

Check out this COLAB for a live demo.

### 37.1.8 LightningModule Class

**class** pytorch_lightning.core.**LightningModule**(**args*, ***kwargs*)

Bases: `abc.ABC`, *pytorch_lightning.utilities.device_dtype_mixin.DeviceDtypeModuleMixin*, *pytorch_lightning.core.grads.GradInformation*, *pytorch_lightning.core.saving.ModelIO*, *pytorch_lightning.core.hooks.ModelHooks*, `torch.nn.Module`

**_LightningModule__get_hparams_assignment_variable**()

looks at the code of the class to figure out what the user named self.hparams this only happens when the user explicitly sets self.hparams

**classmethod _auto_collect_arguments**(*frame=None*)
Collect all module arguments in the current constructor and all child constructors. The child constructors are all the `__init__` methods that reach the current class through (chained) `super().__init__()` calls.

**Parameters** **frame** – instance frame

**Returns** arguments dictionary of the first instance parents_arguments: arguments dictionary of the parent's instances

**Return type** self_arguments

**_init_slurm_connection**()
Sets up environment variables necessary for pytorch distributed communications based on slurm environment.

**Return type** `None`

**_set_hparams**(*hp*)

**Return type** `None`

**configure_apex**(*amp, model, optimizers, amp_level*)
Override to init AMP your own way. Must return a model and list of optimizers.

**Parameters**

- **amp** (`object`) – pointer to amp library object.
- **model** (*LightningModule*) – pointer to current *LightningModule*.
- **optimizers** (`List[Optimizer]`) – list of optimizers passed in *configure_optimizers()*.
- **amp_level** (`str`) – AMP mode chosen ('O1', 'O2', etc…)

**Return type** `Tuple[LightningModule, List[Optimizer]]`

**Returns** Apex wrapped model and optimizers

### Examples

```
# Default implementation used by Trainer.
def configure_apex(self, amp, model, optimizers, amp_level):
    model, optimizers = amp.initialize(
        model, optimizers, opt_level=amp_level,
    )

    return model, optimizers
```

**configure_ddp**(*model, device_ids*)
Override to init DDP in your own way or with your own wrapper. The only requirements are that:

1. On a validation batch the call goes to `model.validation_step`.
2. On a training batch the call goes to `model.training_step`.
3. On a testing batch, the call goes to `model.test_step`.+

**Parameters**

- **model** (*LightningModule*) – the *LightningModule* currently being optimized.

- **device_ids** (List[int]) – the list of GPU ids.

**Return type** DistributedDataParallel

**Returns** DDP wrapped model

### Examples

```
# default implementation used in Trainer
def configure_ddp(self, model, device_ids):
    # Lightning DDP simply routes to test_step, val_step, etc...
    model = LightningDistributedDataParallel(
        model,
        device_ids=device_ids,
        find_unused_parameters=True
    )
    return model
```

**configure_optimizers**()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

**Return type** Union[Optimizer, Sequence[Optimizer], Dict, Sequence[Dict], Tuple[List, List], None]

**Returns**

Any of these 6 options.

- Single optimizer.
- List or Tuple - List of optimizers.
- Two lists - The first list has multiple optimizers, the second a list of LR schedulers (or lr_dict).
- Dictionary, with an 'optimizer' key, and (optionally) a 'lr_scheduler' key which value is a single LR scheduler or lr_dict.
- Tuple of dictionaries as described, with an optional 'frequency' key.
- None - Fit will run without any optimizer.

---

**Note:** The 'frequency' value is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1: In the former case, all optimizers will operate on the given batch in each optimization step. In the latter, only one optimizer will operate on the given batch at every step.

The lr_dict is a dictionary which contains scheduler and its associated configuration. It has five keys. The default configuration is shown below.

```
{
    'scheduler': lr_scheduler, # The LR schduler
    'interval': 'epoch', # The unit of the scheduler's step size
    'frequency': 1, # The frequency of the scheduler
    'reduce_on_plateau': False, # For ReduceLROnPlateau scheduler
    'monitor': 'val_loss' # Metric to monitor
}
```

If user only provides LR schedulers, then their configuration will set to default as shown above.

### Examples

```
# most cases
def configure_optimizers(self):
    opt = Adam(self.parameters(), lr=1e-3)
    return opt

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    generator_opt = Adam(self.model_gen.parameters(), lr=0.01)
    disriminator_opt = Adam(self.model_disc.parameters(), lr=0.02)
    return generator_opt, disriminator_opt

# example with learning rate schedulers
def configure_optimizers(self):
    generator_opt = Adam(self.model_gen.parameters(), lr=0.01)
    disriminator_opt = Adam(self.model_disc.parameters(), lr=0.02)
    discriminator_sched = CosineAnnealing(discriminator_opt, T_max=10)
    return [generator_opt, disriminator_opt], [discriminator_sched]

# example with step-based learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_disc.parameters(), lr=0.02)
    gen_sched = {'scheduler': ExponentialLR(gen_opt, 0.99),
                 'interval': 'step'}  # called after each training step
    dis_sched = CosineAnnealing(discriminator_opt, T_max=10) # called every␣
↪epoch
    return [gen_opt, dis_opt], [gen_sched, dis_sched]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`,␣
↪Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_disc.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

**Note:** Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer and learning rate scheduler as needed.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers for you.
- If you use multiple optimizers, *`training_step()`* will have an additional `optimizer_idx` parameter.

- If you use LBFGS Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the *`optimizer_step()`* hook.
- If you only want to call a learning rate scheduler every `x` step or epoch, or want to monitor a custom metric, you can specify these in a lr_dict:

```
{
    'scheduler': lr_scheduler,
    'interval': 'step',  # or 'epoch'
    'monitor': 'val_f1',
    'frequency': x,
}
```

---

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

Same as `torch.nn.Module.forward()`, however in Lightning you want this to define the operations you want to use for prediction (i.e.: on a server or as a feature extractor).

Normally you'd call `self()` from your *`training_step()`* method. This makes it easy to write a complex system for training with the outputs you'd want in a prediction setting.

You may also find the *`auto_move_data()`* decorator useful when using the module outside Lightning in a production setting.

**Parameters**

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

**Returns** Predicted output

### Examples

```
# example if we were using this model as a feature extractor
def forward(self, x):
    feature_maps = self.convnet(x)
    return feature_maps

def training_step(self, batch, batch_idx):
    x, y = batch
    feature_maps = self(x)
    logits = self.classifier(feature_maps)

    # ...
    return loss

# splitting it this way allows model to be used a feature extractor
model = MyModelAbove()

inputs = server.get_request()
results = model(inputs)
server.write_results(results)
```

(continues on next page)

(continued from previous page)

```
# -------------
# This is in stark contrast to torch.nn.Module where normally you would have␣
↪this:
def forward(self, batch):
    x, y = batch
    feature_maps = self.convnet(x)
    logits = self.classifier(feature_maps)
    return logits
```

**freeze**()

Freeze all params for inference.

### Example

```
model = MyLightningModule(...)
model.freeze()
```

**Return type** None

**get_progress_bar_dict**()

Additional items to be displayed in the progress bar.

**Return type** Dict[str, Union[int, str]]

**Returns** Dictionary with the items to be displayed in the progress bar.

**get_tqdm_dict**()

Additional items to be displayed in the progress bar.

**Return type** Dict[str, Union[int, str]]

**Returns** Dictionary with the items to be displayed in the progress bar.

**Warning:** Deprecated since v0.7.3. Use *get_progress_bar_dict()* instead.

**init_ddp_connection**(*global_rank*, *world_size*, *is_slurm_managing_tasks=True*)

Override to define your custom way of setting up a distributed environment.

Lightning's implementation uses env:// init by default and sets the first node as root for SLURM managed cluster.

**Parameters**

- **global_rank** (int) – The global process idx.
- **world_size** (int) – Number of GPUs being use across all nodes. (num_nodes * num_gpus).
- **is_slurm_managing_tasks** (bool) – is cluster managed by SLURM.

**Return type** None

**on_load_checkpoint**(*checkpoint*)

Called by Lightning to restore your model. If you saved something with *on_save_checkpoint()* this is your chance to restore this.

**Parameters** **checkpoint** (Dict[str, Any]) – Loaded checkpoint

### Example

```
def on_load_checkpoint(self, checkpoint):
    # 99% of the time you don't need to implement this method
    self.something_cool_i_want_to_save = checkpoint['something_cool_i_want_to_
↪save']
```

**Note:** Lightning auto-restores global step, epoch, and train state including amp scaling. There is no need for you to restore anything regarding training.

**Return type** `None`

**on_save_checkpoint**(*checkpoint*)
Called by Lightning when saving a checkpoint to give you a chance to store anything else you might want to save.

**Parameters** **checkpoint** (`Dict`[`str`, `Any`]) – Checkpoint to be saved

### Example

```
def on_save_checkpoint(self, checkpoint):
    # 99% of use cases you don't need to implement this method
    checkpoint['something_cool_i_want_to_save'] = my_cool_pickable_object
```

**Note:** Lightning saves all aspects of training (epoch, global step, etc...) including amp scaling. There is no need for you to store anything about training.

**Return type** `None`

**optimizer_step**(*epoch*, *batch_idx*, *optimizer*, *optimizer_idx*, *second_order_closure=None*)
Override this method to adjust the default way the *Trainer* calls each optimizer. By default, Lightning calls `step()` and `zero_grad()` as shown in the example once per optimizer.

**Parameters**

- **epoch** (`int`) – Current epoch
- **batch_idx** (`int`) – Index of current batch
- **optimizer** (`Optimizer`) – A PyTorch optimizer
- **optimizer_idx** (`int`) – If you used multiple optimizers this indexes into that list.
- **second_order_closure** (`Optional`[`Callable`]) – closure for second order methods

### Examples

```
# DEFAULT
def optimizer_step(self, current_epoch, batch_idx, optimizer, optimizer_idx,
                   second_order_closure=None):
    optimizer.step()
    optimizer.zero_grad()

# Alternating schedule for optimizer steps (i.e.: GANs)
def optimizer_step(self, current_epoch, batch_idx, optimizer, optimizer_idx,
                   second_order_closure=None):
    # update generator opt every 2 steps
    if optimizer_idx == 0:
        if batch_idx % 2 == 0 :
            optimizer.step()
            optimizer.zero_grad()

    # update discriminator opt every 4 steps
    if optimizer_idx == 1:
        if batch_idx % 4 == 0 :
            optimizer.step()
            optimizer.zero_grad()

    # ...
    # add as many optimizers as you want
```

Here's another example showing how to use this for more advanced things such as learning rate warm-up:

```
# learning rate warm-up
def optimizer_step(self, current_epoch, batch_idx, optimizer,
                    optimizer_idx, second_order_closure=None):
    # warm up lr
    if self.trainer.global_step < 500:
        lr_scale = min(1., float(self.trainer.global_step + 1) / 500.)
        for pg in optimizer.param_groups:
            pg['lr'] = lr_scale * self.learning_rate

    # update params
    optimizer.step()
    optimizer.zero_grad()
```

**Note:** If you also override the *on_before_zero_grad()* model hook don't forget to add the call to it before `optimizer.zero_grad()` yourself.

**Return type** `None`

**prepare_data**()

Use this to download and prepare data.

**Warning:** DO NOT set state to the model (use *setup* instead) since this is NOT called on every GPU in DDP/TPU

Example:

```python
def prepare_data(self):
    # good
    download_data()
    tokenize()
    etc()

    # bad
    self.split = data_split
    self.some_state = some_other_state()
```

In DDP prepare_data can be called in two ways (using Trainer(prepare_data_per_node)):

1. Once per node. This is the default and is only called on LOCAL_RANK=0.
2. Once in total. Only called on GLOBAL_RANK=0.

Example:

```python
# DEFAULT
# called once per node on LOCAL_RANK=0 of that node
Trainer(prepare_data_per_node=True)

# call on GLOBAL_RANK=0 (great for shared file systems)
Trainer(prepare_data_per_node=False)
```

This is called before requesting the dataloaders:

```python
model.prepare_data()
    if ddp/tpu: init()
model.setup(step)
model.train_dataloader()
model.val_dataloader()
model.test_dataloader()
```

**Return type** `None`

**print**(**args*, ***kwargs*)

Prints only from process 0. Use this in any distributed mode to log only once.

**Parameters**

- ***args** – The thing to print. Will be passed to Python's built-in print function.
- ****kwargs** – Will be passed to Python's built-in print function.

### Example

```python
def forward(self, x):
    self.print(x, 'in forward')
```

**Return type** `None`

**save_hyperparameters**(**args*, *frame=None*)

Save all model arguments.

**Parameters** **args** – single object of *dict*, *NameSpace* or *OmegaConf* or string names or argumenst from class *__init__*

```
>>> from collections import OrderedDict
>>> class ManuallyArgsModel(LightningModule):
...     def __init__(self, arg1, arg2, arg3):
...         super().__init__()
...         # manually assine arguments
...         self.save_hyperparameters('arg1', 'arg3')
...     def forward(self, *args, **kwargs):
...         ...
>>> model = ManuallyArgsModel(1, 'abc', 3.14)
>>> model.hparams
"arg1": 1
"arg3": 3.14
```

```
>>> class AutomaticArgsModel(LightningModule):
...     def __init__(self, arg1, arg2, arg3):
...         super().__init__()
...         # equivalent automatic
...         self.save_hyperparameters()
...     def forward(self, *args, **kwargs):
...         ...
>>> model = AutomaticArgsModel(1, 'abc', 3.14)
>>> model.hparams
"arg1": 1
"arg2": abc
"arg3": 3.14
```

```
>>> class SingleArgModel(LightningModule):
...     def __init__(self, params):
...         super().__init__()
...         # manually assign single argument
...         self.save_hyperparameters(params)
...     def forward(self, *args, **kwargs):
...         ...
>>> model = SingleArgModel(Namespace(p1=1, p2='abc', p3=3.14))
>>> model.hparams
"p1": 1
"p2": abc
"p3": 3.14
```

**Return type** `None`

**summarize**(*mode='top'*)

**Return type** `ModelSummary`

**tbptt_split_batch**(*batch, split_size*)

When using truncated backpropagation through time, each batch must be split along the time dimension. Lightning handles this by default, but for custom behavior override this function.

**Parameters**

- **batch** (`Tensor`) – Current batch
- **split_size** (`int`) – The size of the split

**Return type** `list`

**Returns** List of batch splits. Each split will be passed to `training_step()` to enable truncated back propagation through time. The default implementation splits root level Tensors and Sequences at dim=1 (i.e. time dim). It assumes that each time dim is the same length.

### Examples

```
def tbptt_split_batch(self, batch, split_size):
  splits = []
  for t in range(0, time_dims[0], split_size):
      batch_split = []
      for i, x in enumerate(batch):
          if isinstance(x, torch.Tensor):
              split_x = x[:, t:t + split_size]
          elif isinstance(x, collections.Sequence):
              split_x = [None] * len(x)
              for batch_idx in range(len(x)):
                  split_x[batch_idx] = x[batch_idx][t:t + split_size]

          batch_split.append(split_x)

      splits.append(batch_split)

  return splits
```

**Note:** Called in the training loop after *on_batch_start()* if *truncated_bptt_steps* > 0. Each returned batch split is passed separately to *training_step()*.

**test_dataloader**()

Implement one or multiple PyTorch DataLoaders for testing.

The dataloader you return will not be called every epoch unless you set *reload_dataloaders_every_epoch* to `True`.

It's recommended that all data downloads and preparation happen in *prepare_data()*.

- *fit()*
- ...
- *prepare_data()*
- *train_dataloader()*
- *val_dataloader()*
- *test_dataloader()*

**Note:** Lightning adds the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

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

**Returns** Single or multiple PyTorch DataLoaders.

### Example

```
def test_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=False, transform=transform,
                    download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=False
    )

    return loader
```

**Note:** If you don't need a test dataset and a `test_step()`, you don't need to implement this method.

**test_end**(*outputs*)

**Warning:** Deprecated in v0.7.0. Use `test_epoch_end()` instead. Will be removed in 1.0.0.

**test_epoch_end**(*outputs*)

Called at the end of a test epoch with the output of all test steps.

```
# the pseudocode for these calls
test_outs = []
for test_batch in test_data:
    out = test_step(test_batch)
    test_outs.append(out)
test_epoch_end(test_outs)
```

**Parameters** **outputs** (`Union`[`List`[`Dict`[`str`, `Tensor`]], `List`[`List`[`Dict`[`str`, `Tensor`]]]]) – List of outputs you defined in `test_step_end()`, or if there are multiple dataloaders, a list containing a list of outputs for each dataloader

**Returns**

Dict has the following optional keys:

- progress_bar -> Dict for progress bar display. Must have only tensors.
- log -> Dict of metrics to add to logger. Must have only tensors (no images, etc).

**Return type** Dict or OrderedDict

**Note:** If you didn't define a `test_step()`, this won't be called.

- The outputs here are strictly for logging or progress bar.
- If you don't need to display anything, don't return anything.
- If you want to manually set current step, specify it with the 'step' key in the 'log' Dict

### Examples

With a single dataloader:

```
def test_epoch_end(self, outputs):
    test_acc_mean = 0
    for output in outputs:
        test_acc_mean += output['test_acc']

    test_acc_mean /= len(outputs)
    tqdm_dict = {'test_acc': test_acc_mean.item()}

    # show test_loss and test_acc in progress bar but only log test_loss
    results = {
        'progress_bar': tqdm_dict,
        'log': {'test_acc': test_acc_mean.item()}
    }
    return results
```

With multiple dataloaders, *outputs* will be a list of lists. The outer list contains one entry per dataloader, while the inner list contains the individual outputs of each test step for that dataloader.

```
def test_epoch_end(self, outputs):
    test_acc_mean = 0
    i = 0
    for dataloader_outputs in outputs:
        for output in dataloader_outputs:
            test_acc_mean += output['test_acc']
            i += 1

    test_acc_mean /= i
    tqdm_dict = {'test_acc': test_acc_mean.item()}

    # show test_loss and test_acc in progress bar but only log test_loss
    results = {
        'progress_bar': tqdm_dict,
        'log': {'test_acc': test_acc_mean.item(), 'step': self.current_epoch}
    }
    return results
```

**test_step**(**args*, ***kwargs*)

Operates on a single batch of data from the test set. In this step you'd normally generate examples or calculate anything of interest such as accuracy.

```
# the pseudocode for these calls
test_outs = []
for test_batch in test_data:
    out = test_step(test_batch)
    test_outs.append(out)
test_epoch_end(test_outs)
```

**Parameters**

- **batch** (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (*int*) – The index of this batch.

- **dataloader_idx** (*int*) – The index of the dataloader that produced this batch (only if multiple test datasets used).

**Return type** Dict[str, Tensor]

**Returns** Dict or OrderedDict - passed to the *test_epoch_end()* method. If you defined *test_step_end()* it will go to that first.

```
# if you have one test dataloader:
def test_step(self, batch, batch_idx)

# if you have multiple test dataloaders:
def test_step(self, batch, batch_idx, dataloader_idx)
```

### Examples

```
# CASE 1: A single test dataset
def test_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # all optional...
    # return whatever you need for the collation function test_epoch_end
    output = OrderedDict({
        'val_loss': loss_val,
        'val_acc': torch.tensor(val_acc), # everything must be a tensor
    })

    # return an optional dict
    return output
```

If you pass in multiple validation datasets, *test_step()* will have an additional argument.

```
# CASE 2: multiple test datasets
def test_step(self, batch, batch_idx, dataset_idx):
    # dataset_idx tells you which dataset this is.
```

**Note:** If you don't need to validate you don't need to implement this method.

**Note:** When the *test_step()* is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of the test epoch, the model goes back to training mode and gradients are enabled.

---

**test_step_end**(**args*, ***kwargs*)

Use this when testing with dp or ddp2 because *test_step()* will operate on only part of the batch. However, this is still optional and only needed for things like softmax or NCE loss.

**Note:** If you later switch to ddp or some other mode, this will still be called so that you don't have to change your code.

```
# pseudocode
sub_batches = split_batches_for_dp(batch)
batch_parts_outputs = [test_step(sub_batch) for sub_batch in sub_batches]
test_step_end(batch_parts_outputs)
```

**Parameters** **batch_parts_outputs** – What you return in *test_step()* for each batch part.

**Return type** Dict[str, Tensor]

**Returns** Dict or OrderedDict - passed to the *test_epoch_end()*.

### Examples

```
# WITHOUT test_step_end
# if used in DP or DDP2, this batch is 1/num_gpus large
def test_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self(x)
    loss = self.softmax(out)
    loss = nce_loss(loss)
    return {'loss': loss}

# --------------
# with test_step_end to do softmax over the full batch
def test_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self(x)
    return {'out': out}

def test_step_end(self, outputs):
    # this out is now the full size of the batch
    out = outputs['out']

    # this softmax now uses the full batch size
    loss = nce_loss(loss)
    return {'loss': loss}
```

**See also:**

See the *Multi-GPU training* guide for more details.

**tng_dataloader**()

> **Warning:** Deprecated in v0.5.0. Use `train_dataloader()` instead. Will be removed in 1.0.0.

**train_dataloader**()

Implement a PyTorch DataLoader for training.

**Return type** `DataLoader`

**Returns** Single PyTorch `DataLoader`.

The dataloader you return will not be called every epoch unless you set `reload_dataloaders_every_epoch` to `True`.

It's recommended that all data downloads and preparation happen in `prepare_data()`.

- `fit()`
- ...
- `prepare_data()`
- `train_dataloader()`

**Note:** Lightning adds the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

### Example

```
def train_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=True, transform=transform,
                    download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=True
    )
    return loader
```

**training_end**(**args*, ***kwargs*)

> **Warning:** Deprecated in v0.7.0. Use `training_step_end()` instead.

**training_epoch_end**(*outputs*)

Called at the end of the training epoch with the outputs of all training steps.

```
# the pseudocode for these calls
train_outs = []
for train_batch in train_data:
```

(continues on next page)

(continued from previous page)

```
    out = training_step(train_batch)
    train_outs.append(out)
training_epoch_end(train_outs)
```

**Parameters** **outputs** (Union[List[Dict[str, Tensor]], List[List[Dict[str, Tensor]]]]) – List of outputs you defined in *training_step()*, or if there are multiple dataloaders, a list containing a list of outputs for each dataloader.

**Return type** Dict[str, Dict[str, Tensor]]

**Returns**

Dict or OrderedDict. May contain the following optional keys:

- log (metrics to be added to the logger; only tensors)
- progress_bar (dict for progress bar display)
- any metric used in a callback (e.g. early stopping).

**Note:** If this method is not overridden, this won't be called.

- The outputs here are strictly for logging or progress bar.
- If you don't need to display anything, don't return anything.
- If you want to manually set current step, you can specify the 'step' key in the 'log' dict.

## Examples

With a single dataloader:

```
def training_epoch_end(self, outputs):
    train_acc_mean = 0
    for output in outputs:
        train_acc_mean += output['train_acc']

    train_acc_mean /= len(outputs)

    # log training accuracy at the end of an epoch
    results = {
        'log': {'train_acc': train_acc_mean.item()},
        'progress_bar': {'train_acc': train_acc_mean},
    }
    return results
```

With multiple dataloaders, `outputs` will be a list of lists. The outer list contains one entry per dataloader, while the inner list contains the individual outputs of each training step for that dataloader.

```
def training_epoch_end(self, outputs):
    train_acc_mean = 0
    i = 0
    for dataloader_outputs in outputs:
        for output in dataloader_outputs:
            train_acc_mean += output['train_acc']
```

(continues on next page)

(continued from previous page)

```
            i += 1

    train_acc_mean /= i

    # log training accuracy at the end of an epoch
    results = {
        'log': {'train_acc': train_acc_mean.item(), 'step': self.current_
↪epoch}
        'progress_bar': {'train_acc': train_acc_mean},
    }
    return results
```

**training_step**(**args*, ***kwargs*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

**Parameters**

- **batch** (Tensor | (Tensor, . . . ) | [Tensor, . . . ]) – The output of your DataLoader. A tensor, tuple or list.
- **batch_idx** (*int*) – Integer displaying index of this batch
- **optimizer_idx** (*int*) – When using multiple optimizers, this argument will also be present.
- **hiddens** (Tensor) – Passed in if *truncated_bptt_steps* > 0.

**Return type** Union[int, Dict[str, Union[Tensor, Dict[str, Tensor]]]]

**Returns**

Dict with loss key and optional log or progress bar keys. When implementing *training_step()*, return whatever you need in that step:

- loss -> tensor scalar **REQUIRED**
- progress_bar -> Dict for progress bar display. Must have only tensors
- log -> Dict of metrics to add to logger. Must have only tensors (no images, etc)

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

### Examples

```
def training_step(self, batch, batch_idx):
    x, y, z = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, x)

    logger_logs = {'training_loss': loss} # optional (MUST ALL BE TENSORS)

    # if using TestTubeLogger or TensorBoardLogger you can nest scalars
    logger_logs = {'losses': logger_logs} # optional (MUST ALL BE TENSORS)

    output = {
```

(continues on next page)

(continued from previous page)

```
        'loss': loss, # required
        'progress_bar': {'training_loss': loss}, # optional (MUST ALL BE␣
↪TENSORS)
        'log': logger_logs
    }

    # return a dict
    return output
```

If you define multiple optimizers, this step will be called with an additional `optimizer_idx` parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
    if optimizer_idx == 1:
        # do training_step with decoder
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    ...
    out, hiddens = self.lstm(data, hiddens)
    ...

    return {
        "loss": ...,
        "hiddens": hiddens  # remember to detach() this
    }
```

### Notes

The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

**training_step_end**(**args*, ***kwargs*)
Use this when training with dp or ddp2 because `training_step()` will operate on only part of the batch. However, this is still optional and only needed for things like softmax or NCE loss.

**Note:** If you later switch to ddp or some other mode, this will still be called so that you don't have to change your code

```
# pseudocode
sub_batches = split_batches_for_dp(batch)
batch_parts_outputs = [training_step(sub_batch) for sub_batch in sub_batches]
training_step_end(batch_parts_outputs)
```

**Parameters** **batch_parts_outputs** – What you return in *training_step* for each batch part.

**Return type** `Dict`[`str`, `Union`[`Tensor`, `Dict`[`str`, `Tensor`]]]

**Returns**

Dict with loss key and optional log or progress bar keys.

- loss -> tensor scalar **REQUIRED**
- progress_bar -> Dict for progress bar display. Must have only tensors
- log -> Dict of metrics to add to logger. Must have only tensors (no images, etc)

### Examples

```
# WITHOUT training_step_end
# if used in DP or DDP2, this batch is 1/num_gpus large
def training_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self(x)
    loss = self.softmax(out)
    loss = nce_loss(loss)
    return {'loss': loss}

# --------------
# with training_step_end to do softmax over the full batch
def training_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self(x)
    return {'out': out}

def training_step_end(self, outputs):
    # this out is now the full size of the batch
    out = outputs['out']

    # this softmax now uses the full batch size
    loss = nce_loss(loss)
    return {'loss': loss}
```

**See also:**

See the *Multi-GPU training* guide for more details.

**unfreeze**()

Unfreeze all parameters for training.

```
model = MyLightningModule(...)
model.unfreeze()
```

**Return type** `None`

**val_dataloader**()

Implement one or multiple PyTorch DataLoaders for validation.

The dataloader you return will not be called every epoch unless you set *reload_dataloaders_every_epoch* to `True`.

It's recommended that all data downloads and preparation happen in *prepare_data()*.

- *fit()*
- ...
- *prepare_data()*
- *train_dataloader()*
- *val_dataloader()*
- *test_dataloader()*

**Note:** Lightning adds the correct sampler for distributed and arbitrary hardware There is no need to set it yourself.

**Return type** Union[DataLoader, List[DataLoader]]

**Returns** Single or multiple PyTorch DataLoaders.

### Examples

```
def val_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=False,
                    transform=transform, download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=False
    )

    return loader

# can also return multiple dataloaders
def val_dataloader(self):
    return [loader_a, loader_b, ..., loader_n]
```

**Note:** If you don't need a validation dataset and a *validation_step()*, you don't need to implement this method.

**Note:** In the case where you return multiple validation dataloaders, the *validation_step()* will have an argument `dataset_idx` which matches the order here.

**validation_end**(*outputs*)

**Warning:** Deprecated in v0.7.0. Use *validation_epoch_end()* instead. Will be removed in 1.0.0.

**validation_epoch_end**(*outputs*)
Called at the end of the validation epoch with the outputs of all validation steps.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

**Parameters** **outputs** (`Union`[`List`[`Dict`[`str`, `Tensor`]], `List`[`List`[`Dict`[`str`, `Tensor`]]]]) – List of outputs you defined in *validation_step()*, or if there are multiple dataloaders, a list containing a list of outputs for each dataloader.

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

**Returns**

Dict or OrderedDict. May have the following optional keys:

- progress_bar (dict for progress bar display; only tensors)
- log (dict of metrics to add to logger; only tensors).

---

**Note:** If you didn't define a *validation_step()*, this won't be called.

---

- The outputs here are strictly for logging or progress bar.
- If you don't need to display anything, don't return anything.
- If you want to manually set current step, you can specify the 'step' key in the 'log' dict.

### Examples

With a single dataloader:

```
def validation_epoch_end(self, outputs):
    val_acc_mean = 0
    for output in outputs:
        val_acc_mean += output['val_acc']

    val_acc_mean /= len(outputs)
    tqdm_dict = {'val_acc': val_acc_mean.item()}

    # show val_acc in progress bar but only log val_loss
    results = {
        'progress_bar': tqdm_dict,
        'log': {'val_acc': val_acc_mean.item()}
    }
    return results
```

With multiple dataloaders, *outputs* will be a list of lists. The outer list contains one entry per dataloader, while the inner list contains the individual outputs of each validation step for that dataloader.

```
def validation_epoch_end(self, outputs):
    val_acc_mean = 0
    i = 0
    for dataloader_outputs in outputs:
```

(continues on next page)

(continued from previous page)

```
        for output in dataloader_outputs:
            val_acc_mean += output['val_acc']
            i += 1

    val_acc_mean /= i
    tqdm_dict = {'val_acc': val_acc_mean.item()}

    # show val_loss and val_acc in progress bar but only log val_loss
    results = {
        'progress_bar': tqdm_dict,
        'log': {'val_acc': val_acc_mean.item(), 'step': self.current_epoch}
    }
    return results
```

**validation_step**(**args*, ***kwargs*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(train_batch)
    val_outs.append(out)
    validation_epoch_end(val_outs)
```

**Parameters**

- **batch** (Tensor | (Tensor, ...) | [Tensor, ...]) – The output of your DataLoader. A tensor, tuple or list.
- **batch_idx** (*int*) – The index of this batch
- **dataloader_idx** (*int*) – The index of the dataloader that produced this batch (only if multiple val datasets used)

**Return type** Dict[str, Tensor]

**Returns** Dict or OrderedDict - passed to *validation_epoch_end()*. If you defined *validation_step_end()* it will go to that first.

```
# pseudocode of order
out = validation_step()
if defined('validation_step_end'):
    out = validation_step_end(out)
out = validation_epoch_end(out)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx)

# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx)
```

### Examples

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # all optional...
    # return whatever you need for the collation function validation_epoch_end
    output = OrderedDict({
        'val_loss': loss_val,
        'val_acc': torch.tensor(val_acc), # everything must be a tensor
    })

    # return an optional dict
    return output
```

If you pass in multiple val datasets, validation_step will have an additional argument.

```
# CASE 2: multiple validation datasets
def validation_step(self, batch, batch_idx, dataset_idx):
    # dataset_idx tells you which dataset this is.
```

**Note:** If you don't need to validate you don't need to implement this method.

**Note:** When the `validation_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

**validation_step_end**(**args*, ***kwargs*)

Use this when validating with dp or ddp2 because `validation_step()` will operate on only part of the batch. However, this is still optional and only needed for things like softmax or NCE loss.

**Note:** If you later switch to ddp or some other mode, this will still be called so that you don't have to change your code.

```
# pseudocode
sub_batches = split_batches_for_dp(batch)
```

(continues on next page)

(continued from previous page)

```
batch_parts_outputs = [validation_step(sub_batch) for sub_batch in sub_
↪batches]
validation_step_end(batch_parts_outputs)
```

**Parameters** **batch_parts_outputs** – What you return in *validation_step()* for each batch part.

**Return type** Dict[str, Tensor]

**Returns** Dict or OrderedDict - passed to the *validation_epoch_end()* method.

### Examples

```
# WITHOUT validation_step_end
# if used in DP or DDP2, this batch is 1/num_gpus large
def validation_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self(x)
    loss = self.softmax(out)
    loss = nce_loss(loss)
    return {'loss': loss}

# --------------
# with validation_step_end to do softmax over the full batch
def validation_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self(x)
    return {'out': out}

def validation_epoch_end(self, outputs):
    # this out is now the full size of the batch
    out = outputs['out']

    # this softmax now uses the full batch size
    loss = nce_loss(loss)
    return {'loss': loss}
```

**See also:**

See the *Multi-GPU training* guide for more details.

**_device = None**
device reference

**_dtype = None**
Current dtype

**current_epoch = None**
The current epoch

**property example_input_array**

**Return type** Any

**global_step = None**
Total training batches seen across all epochs

**property hparams**

**Return type** Union[*AttributeDict*, str]

**logger = None**
Pointer to the logger object

**property on_gpu**
True if your model is currently running on GPUs. Useful to set flags around the LightningModule for different CPU vs GPU behavior.

**trainer = None**
Pointer to the trainer object

**use_amp = None**
True if using amp

**use_ddp = None**
True if using ddp

**use_ddp2 = None**
True if using ddp2

**use_dp = None**
True if using dp

pytorch_lightning.core.**data_loader**(*fn*)
Decorator to make any fx with this use the lazy property.

**Warning:** This decorator deprecated in v0.7.0 and it will be removed v0.9.0.

## 37.1.9 Submodules

### pytorch_lightning.core.decorators module

pytorch_lightning.core.decorators.**auto_move_data**(*fn*)
Decorator for *LightningModule* methods for which input arguments should be moved automatically to the correct device. It as no effect if applied to a method of an object that is not an instance of *LightningModule* and is typically applied to `__call__` or `forward`.

**Parameters** **fn** (Callable) – A LightningModule method for which the arguments should be moved to the device the parameters are on.

#### Example

```
# directly in the source code
class LitModel(LightningModule):

    @auto_move_data
    def forward(self, x):
        return x

# or outside
```

(continues on next page)

(continued from previous page)

```
LitModel.forward = auto_move_data(LitModel.forward)

model = LitModel()
model = model.to('cuda')
model(torch.zeros(1, 3))

# input gets moved to device
# tensor([[0., 0., 0.]], device='cuda:0')
```

**Return type** Callable

`pytorch_lightning.core.decorators.`**`data_loader`**`(`*fn*`)`

Decorator to make any fx with this use the lazy property.

> **Warning:** This decorator deprecated in v0.7.0 and it will be removed v0.9.0.

## pytorch_lightning.core.grads module

Module to describe gradients

**`class`** `pytorch_lightning.core.grads.`**`GradInformation`**`(`**args*, ***kwargs*`)`

Bases: `torch.nn.Module`

**`grad_norm`**`(`*norm_type*`)`

Compute each parameter's gradient's norm and their overall norm.

The overall norm is computed over all gradients together, as if they were concatenated into a single vector.

**Parameters** **`norm_type`** (`Union`[`float`, `int`, `str`]) – The type of the used p-norm, cast to float if necessary. Can be `'inf'` for infinity norm.

**Returns**

**The dictionary of p-norms of each parameter's gradient and** a special entry for the total p-norm of the gradients viewed as a single vector.

**Return type** norms

## pytorch_lightning.core.hooks module

**`class`** `pytorch_lightning.core.hooks.`**`ModelHooks`**`(`**args*, ***kwargs*`)`

Bases: `torch.nn.Module`

**`backward`**`(`*trainer*, *loss*, *optimizer*, *optimizer_idx*`)`

Override backward with your own implementation if you need to.

**Parameters**

- **`trainer`** – Pointer to the trainer
- **`loss`** (`Tensor`) – Loss is already scaled by accumulated grads
- **`optimizer`** (`Optimizer`) – Current optimizer being used
- **`optimizer_idx`** (`int`) – Index of the current optimizer being used

Called to perform backward step. Feel free to override as needed.

The loss passed in has already been scaled for accumulated gradients if requested.

Example:

```
def backward(self, use_amp, loss, optimizer):
    if use_amp:
        with amp.scale_loss(loss, optimizer) as scaled_loss:
            scaled_loss.backward()
    else:
        loss.backward()
```

**Return type** None

**on_after_backward**()

Called in the training loop after loss.backward() and before optimizers do anything. This is the ideal place to inspect or log gradient information.

Example:

```
def on_after_backward(self):
    # example to inspect gradient information in tensorboard
    if self.trainer.global_step % 25 == 0:  # don't make the tf file huge
        params = self.state_dict()
        for k, v in params.items():
            grads = v
            name = k
            self.logger.experiment.add_histogram(tag=name, values=grads,
                                                 global_step=self.trainer.
↪global_step)
```

**Return type** None

**on_batch_end**()

Called in the training loop after the batch.

**Return type** None

**on_batch_start**(*batch*)

Called in the training loop before anything happens for that batch.

If you return -1 here, you will skip training for the rest of the current epoch.

**Parameters** **batch** (Any) – The batched data as it is returned by the training DataLoader.

**Return type** None

**on_before_zero_grad**(*optimizer*)

Called after optimizer.step() and before optimizer.zero_grad().

Called in the training loop after taking an optimizer step and before zeroing grads. Good place to inspect weight information with weights updated.

This is where it is called:

```
for optimizer in optimizers:
    optimizer.step()
    model.on_before_zero_grad(optimizer) # < ---- called here
    optimizer.zero_grad
```

**Parameters** **optimizer** (Optimizer) – The optimizer for which grads should be zeroed.

**Return type** None

**on_epoch_end**()
Called in the training loop at the very end of the epoch.

**Return type** None

**on_epoch_start**()
Called in the training loop at the very beginning of the epoch.

**Return type** None

**on_fit_end**()
Called at the very end of fit. If on DDP it is called on every process

**on_fit_start**()
Called at the very beginning of fit. If on DDP it is called on every process

**on_post_performance_check**()
Called at the very end of the validation loop.

**Return type** None

**on_pre_performance_check**()
Called at the very beginning of the validation loop.

**Return type** None

**on_sanity_check_start**()
Called before starting evaluation.

> **Warning:** Deprecated. Will be removed in v0.9.0.

**on_train_end**()
Called at the end of training before logger experiment is closed.

**Return type** None

**on_train_start**()
Called at the beginning of training before sanity check.

**Return type** None

**setup**(*stage*)
Called at the beginning of fit and test. This is a good hook when you need to build models dynamically or adjust something about them. This hook is called on every process when using DDP.

**Parameters** **stage** (str) – either 'fit' or 'test'

Example:

```
class LitModel(...):
    def __init__(self):
        self.l1 = None

    def prepare_data(self):
        download_data()
        tokenize()
```

(continues on next page)

(continued from previous page)

```
        # don't do this
        self.something = else

    def setup(step):
        data = Load_data(...)
        self.l1 = nn.Linear(28, data.num_classes)
```

**teardown** (*stage*)
Called at the end of fit and test.

**Parameters** **stage** (`str`) – either 'fit' or 'test'

**transfer_batch_to_device** (*batch, device*)
Override this hook if your `DataLoader` returns tensors wrapped in a custom data structure.

The data types listed below (and any arbitrary nesting of them) are supported out of the box:

- `torch.Tensor`
- `list`
- `dict`
- `tuple`
- `torchtext.data.Batch` (COMING SOON)

For anything else, you need to define how the data is moved to the target device (CPU, GPU, TPU, . . . ).

Example:

```
def transfer_batch_to_device(self, batch, device)
    if isinstance(batch, CustomBatch):
        # move all tensors in your custom data structure to the device
        batch.samples = batch.samples.to(device)
        batch.targets = batch.targets.to(device)
    else:
        batch = super().transfer_batch_to_device(data, device)
    return batch
```

**Parameters**

- **batch** (`Any`) – A batch of data that needs to be transferred to a new device.
- **device** (`device`) – The target device as defined in PyTorch.

**Return type** `Any`

**Returns** A reference to the data on the new device.

**Note:** This hook should only transfer the data and not modify it, nor should it move the data to any other device than the one passed in as argument (unless you know what you are doing). The *`Trainer`* already takes care of splitting the batch and determines the target devices.

**See also:**

- *`move_data_to_device()`*
- *`apply_to_collection()`*

### pytorch_lightning.core.lightning module

**class** pytorch_lightning.core.lightning.**LightningModule**(**args*, ***kwargs*)

Bases: abc.ABC, *pytorch_lightning.utilities.device_dtype_mixin.DeviceDtypeModuleMixin*, *pytorch_lightning.core.grads.GradInformation*, *pytorch_lightning.core.saving.ModelIO*, *pytorch_lightning.core.hooks.ModelHooks*, torch.nn.Module

**_LightningModule__get_hparams_assignment_variable**()

looks at the code of the class to figure out what the user named self.hparams this only happens when the user explicitly sets self.hparams

**classmethod _auto_collect_arguments**(*frame=None*)

Collect all module arguments in the current constructor and all child constructors. The child constructors are all the `__init__` methods that reach the current class through (chained) `super().__init__()` calls.

**Parameters** **frame** – instance frame

**Returns** arguments dictionary of the first instance parents_arguments: arguments dictionary of the parent's instances

**Return type** self_arguments

**_init_slurm_connection**()

Sets up environment variables necessary for pytorch distributed communications based on slurm environment.

**Return type** None

**_set_hparams**(*hp*)

**Return type** None

**configure_apex**(*amp*, *model*, *optimizers*, *amp_level*)

Override to init AMP your own way. Must return a model and list of optimizers.

**Parameters**

- **amp** (object) – pointer to amp library object.
- **model** (*LightningModule*) – pointer to current *LightningModule*.
- **optimizers** (List[Optimizer]) – list of optimizers passed in *configure_optimizers()*.
- **amp_level** (str) – AMP mode chosen ('O1', 'O2', etc...)

**Return type** Tuple[*LightningModule*, List[Optimizer]]

**Returns** Apex wrapped model and optimizers

### Examples

```
# Default implementation used by Trainer.
def configure_apex(self, amp, model, optimizers, amp_level):
    model, optimizers = amp.initialize(
        model, optimizers, opt_level=amp_level,
    )

    return model, optimizers
```

**configure_ddp**(*model*, *device_ids*)

Override to init DDP in your own way or with your own wrapper. The only requirements are that:

1. On a validation batch the call goes to `model.validation_step`.
2. On a training batch the call goes to `model.training_step`.
3. On a testing batch, the call goes to `model.test_step`.+

**Parameters**

- **model** (*LightningModule*) – the *LightningModule* currently being optimized.
- **device_ids** (`List[int]`) – the list of GPU ids.

**Return type** `DistributedDataParallel`

**Returns** DDP wrapped model

### Examples

```
# default implementation used in Trainer
def configure_ddp(self, model, device_ids):
    # Lightning DDP simply routes to test_step, val_step, etc...
    model = LightningDistributedDataParallel(
        model,
        device_ids=device_ids,
        find_unused_parameters=True
    )
    return model
```

**configure_optimizers**()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

**Return type** `Union[Optimizer, Sequence[Optimizer], Dict, Sequence[Dict], Tuple[List, List], None]`

**Returns**

Any of these 6 options.

- Single optimizer.
- List or Tuple - List of optimizers.
- Two lists - The first list has multiple optimizers, the second a list of LR schedulers (or lr_dict).
- Dictionary, with an 'optimizer' key, and (optionally) a 'lr_scheduler' key which value is a single LR scheduler or lr_dict.

- Tuple of dictionaries as described, with an optional 'frequency' key.
- None - Fit will run without any optimizer.

**Note:** The 'frequency' value is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1: In the former case, all optimizers will operate on the given batch in each optimization step. In the latter, only one optimizer will operate on the given batch at every step.

The lr_dict is a dictionary which contains scheduler and its associated configuration. It has five keys. The default configuration is shown below.

```
{
    'scheduler': lr_scheduler, # The LR schduler
    'interval': 'epoch', # The unit of the scheduler's step size
    'frequency': 1, # The frequency of the scheduler
    'reduce_on_plateau': False, # For ReduceLROnPlateau scheduler
    'monitor': 'val_loss' # Metric to monitor
}
```

If user only provides LR schedulers, then their configuration will set to default as shown above.

### Examples

```
# most cases
def configure_optimizers(self):
    opt = Adam(self.parameters(), lr=1e-3)
    return opt

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    generator_opt = Adam(self.model_gen.parameters(), lr=0.01)
    disriminator_opt = Adam(self.model_disc.parameters(), lr=0.02)
    return generator_opt, disriminator_opt

# example with learning rate schedulers
def configure_optimizers(self):
    generator_opt = Adam(self.model_gen.parameters(), lr=0.01)
    disriminator_opt = Adam(self.model_disc.parameters(), lr=0.02)
    discriminator_sched = CosineAnnealing(discriminator_opt, T_max=10)
    return [generator_opt, disriminator_opt], [discriminator_sched]

# example with step-based learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_disc.parameters(), lr=0.02)
    gen_sched = {'scheduler': ExponentialLR(gen_opt, 0.99),
                 'interval': 'step'}  # called after each training step
    dis_sched = CosineAnnealing(discriminator_opt, T_max=10) # called every␣
↪epoch
    return [gen_opt, dis_opt], [gen_sched, dis_sched]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`,␣
↪Algorithm 1
```

(continues on next page)

(continued from previous page)

```
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_disc.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

**Note:** Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer and learning rate scheduler as needed.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers for you.
- If you use multiple optimizers, *`training_step()`* will have an additional `optimizer_idx` parameter.
- If you use LBFGS Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the *`optimizer_step()`* hook.
- If you only want to call a learning rate scheduler every `x` step or epoch, or want to monitor a custom metric, you can specify these in a lr_dict:

```
{
    'scheduler': lr_scheduler,
    'interval': 'step',  # or 'epoch'
    'monitor': 'val_f1',
    'frequency': x,
}
```

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

Same as `torch.nn.Module.forward()`, however in Lightning you want this to define the operations you want to use for prediction (i.e.: on a server or as a feature extractor).

Normally you'd call `self()` from your *`training_step()`* method. This makes it easy to write a complex system for training with the outputs you'd want in a prediction setting.

You may also find the *`auto_move_data()`* decorator useful when using the module outside Lightning in a production setting.

**Parameters**

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

**Returns** Predicted output

### Examples

```
# example if we were using this model as a feature extractor
def forward(self, x):
    feature_maps = self.convnet(x)
    return feature_maps

def training_step(self, batch, batch_idx):
    x, y = batch
    feature_maps = self(x)
    logits = self.classifier(feature_maps)

    # ...
    return loss

# splitting it this way allows model to be used a feature extractor
model = MyModelAbove()

inputs = server.get_request()
results = model(inputs)
server.write_results(results)

# -------------
# This is in stark contrast to torch.nn.Module where normally you would have␣
↪this:
def forward(self, batch):
    x, y = batch
    feature_maps = self.convnet(x)
    logits = self.classifier(feature_maps)
    return logits
```

**freeze**()
Freeze all params for inference.

### Example

```
model = MyLightningModule(...)
model.freeze()
```

**Return type** None

**get_progress_bar_dict**()
Additional items to be displayed in the progress bar.

**Return type** Dict[str, Union[int, str]]

**Returns** Dictionary with the items to be displayed in the progress bar.

**get_tqdm_dict**()
Additional items to be displayed in the progress bar.

**Return type** Dict[str, Union[int, str]]

**Returns** Dictionary with the items to be displayed in the progress bar.

**Warning:** Deprecated since v0.7.3. Use *get_progress_bar_dict()* instead.

**init_ddp_connection**(*global_rank*, *world_size*, *is_slurm_managing_tasks=True*)
Override to define your custom way of setting up a distributed environment.

Lightning's implementation uses env:// init by default and sets the first node as root for SLURM managed cluster.

**Parameters**

- **global_rank** (int) – The global process idx.
- **world_size** (int) – Number of GPUs being use across all nodes. (num_nodes * num_gpus).
- **is_slurm_managing_tasks** (bool) – is cluster managed by SLURM.

**Return type** None

**on_load_checkpoint**(*checkpoint*)
Called by Lightning to restore your model. If you saved something with *on_save_checkpoint()* this is your chance to restore this.

**Parameters checkpoint** (Dict[str, Any]) – Loaded checkpoint

### Example

```
def on_load_checkpoint(self, checkpoint):
    # 99% of the time you don't need to implement this method
    self.something_cool_i_want_to_save = checkpoint['something_cool_i_want_to_
↪save']
```

**Note:** Lightning auto-restores global step, epoch, and train state including amp scaling. There is no need for you to restore anything regarding training.

**Return type** None

**on_save_checkpoint**(*checkpoint*)
Called by Lightning when saving a checkpoint to give you a chance to store anything else you might want to save.

**Parameters checkpoint** (Dict[str, Any]) – Checkpoint to be saved

### Example

```
def on_save_checkpoint(self, checkpoint):
    # 99% of use cases you don't need to implement this method
    checkpoint['something_cool_i_want_to_save'] = my_cool_pickable_object
```

**Note:** Lightning saves all aspects of training (epoch, global step, etc...) including amp scaling. There is no need for you to store anything about training.

**Return type** None

**optimizer_step**(*epoch*, *batch_idx*, *optimizer*, *optimizer_idx*, *second_order_closure=None*)

Override this method to adjust the default way the *Trainer* calls each optimizer. By default, Lightning calls step() and zero_grad() as shown in the example once per optimizer.

**Parameters**

- **epoch** (int) – Current epoch
- **batch_idx** (int) – Index of current batch
- **optimizer** (Optimizer) – A PyTorch optimizer
- **optimizer_idx** (int) – If you used multiple optimizers this indexes into that list.
- **second_order_closure** (Optional[Callable]) – closure for second order methods

## Examples

```
# DEFAULT
def optimizer_step(self, current_epoch, batch_idx, optimizer, optimizer_idx,
                   second_order_closure=None):
    optimizer.step()
    optimizer.zero_grad()

# Alternating schedule for optimizer steps (i.e.: GANs)
def optimizer_step(self, current_epoch, batch_idx, optimizer, optimizer_idx,
                   second_order_closure=None):
    # update generator opt every 2 steps
    if optimizer_idx == 0:
        if batch_idx % 2 == 0 :
            optimizer.step()
            optimizer.zero_grad()

    # update discriminator opt every 4 steps
    if optimizer_idx == 1:
        if batch_idx % 4 == 0 :
            optimizer.step()
            optimizer.zero_grad()

    # ...
    # add as many optimizers as you want
```

Here's another example showing how to use this for more advanced things such as learning rate warm-up:

```
# learning rate warm-up
def optimizer_step(self, current_epoch, batch_idx, optimizer,
                    optimizer_idx, second_order_closure=None):
    # warm up lr
    if self.trainer.global_step < 500:
        lr_scale = min(1., float(self.trainer.global_step + 1) / 500.)
        for pg in optimizer.param_groups:
            pg['lr'] = lr_scale * self.learning_rate

    # update params
    optimizer.step()
    optimizer.zero_grad()
```

**Note:** If you also override the `on_before_zero_grad()` model hook don't forget to add the call to it before `optimizer.zero_grad()` yourself.

**Return type** None

**prepare_data**()

Use this to download and prepare data.

**Warning:** DO NOT set state to the model (use *setup* instead) since this is NOT called on every GPU in DDP/TPU

Example:

```
def prepare_data(self):
    # good
    download_data()
    tokenize()
    etc()

    # bad
    self.split = data_split
    self.some_state = some_other_state()
```

In DDP prepare_data can be called in two ways (using Trainer(prepare_data_per_node)):

1. Once per node. This is the default and is only called on LOCAL_RANK=0.
2. Once in total. Only called on GLOBAL_RANK=0.

Example:

```
# DEFAULT
# called once per node on LOCAL_RANK=0 of that node
Trainer(prepare_data_per_node=True)

# call on GLOBAL_RANK=0 (great for shared file systems)
Trainer(prepare_data_per_node=False)
```

This is called before requesting the dataloaders:

```
model.prepare_data()
    if ddp/tpu: init()
model.setup(step)
model.train_dataloader()
model.val_dataloader()
model.test_dataloader()
```

**Return type** None

**print**(**args*, ***kwargs*)

Prints only from process 0. Use this in any distributed mode to log only once.

**Parameters**

- ***args** – The thing to print. Will be passed to Python's built-in print function.

- **`**kwargs`** – Will be passed to Python's built-in print function.

### Example

```
def forward(self, x):
    self.print(x, 'in forward')
```

**Return type** `None`

**`save_hyperparameters`**(**args*, *frame=None*)

Save all model arguments.

**Parameters** **`args`** – single object of *dict*, *NameSpace* or *OmegaConf* or string names or argumenst from class *__init__*

```
>>> from collections import OrderedDict
>>> class ManuallyArgsModel(LightningModule):
...     def __init__(self, arg1, arg2, arg3):
...         super().__init__()
...         # manually assine arguments
...         self.save_hyperparameters('arg1', 'arg3')
...     def forward(self, *args, **kwargs):
...         ...
>>> model = ManuallyArgsModel(1, 'abc', 3.14)
>>> model.hparams
"arg1": 1
"arg3": 3.14
```

```
>>> class AutomaticArgsModel(LightningModule):
...     def __init__(self, arg1, arg2, arg3):
...         super().__init__()
...         # equivalent automatic
...         self.save_hyperparameters()
...     def forward(self, *args, **kwargs):
...         ...
>>> model = AutomaticArgsModel(1, 'abc', 3.14)
>>> model.hparams
"arg1": 1
"arg2": abc
"arg3": 3.14
```

```
>>> class SingleArgModel(LightningModule):
...     def __init__(self, params):
...         super().__init__()
...         # manually assign single argument
...         self.save_hyperparameters(params)
...     def forward(self, *args, **kwargs):
...         ...
>>> model = SingleArgModel(Namespace(p1=1, p2='abc', p3=3.14))
>>> model.hparams
"p1": 1
"p2": abc
"p3": 3.14
```

**Return type** `None`

**summarize**(*mode='top'*)

**Return type** *ModelSummary*

**tbptt_split_batch**(*batch, split_size*)

When using truncated backpropagation through time, each batch must be split along the time dimension. Lightning handles this by default, but for custom behavior override this function.

**Parameters**

- **batch** (Tensor) – Current batch
- **split_size** (int) – The size of the split

**Return type** list

**Returns** List of batch splits. Each split will be passed to *training_step()* to enable truncated back propagation through time. The default implementation splits root level Tensors and Sequences at dim=1 (i.e. time dim). It assumes that each time dim is the same length.

### Examples

```
def tbptt_split_batch(self, batch, split_size):
  splits = []
  for t in range(0, time_dims[0], split_size):
      batch_split = []
      for i, x in enumerate(batch):
          if isinstance(x, torch.Tensor):
              split_x = x[:, t:t + split_size]
          elif isinstance(x, collections.Sequence):
              split_x = [None] * len(x)
              for batch_idx in range(len(x)):
                  split_x[batch_idx] = x[batch_idx][t:t + split_size]

          batch_split.append(split_x)

      splits.append(batch_split)

  return splits
```

**Note:** Called in the training loop after *on_batch_start()* if *truncated_bptt_steps* > 0. Each returned batch split is passed separately to *training_step()*.

**test_dataloader**()

Implement one or multiple PyTorch DataLoaders for testing.

The dataloader you return will not be called every epoch unless you set *reload_dataloaders_every_epoch* to True.

It's recommended that all data downloads and preparation happen in *prepare_data()*.

- *fit()*
- ...
- *prepare_data()*
- *train_dataloader()*
- *val_dataloader()*

- *test_dataloader()*

**Note:** Lightning adds the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

**Return type** Union[DataLoader, List[DataLoader]]

**Returns** Single or multiple PyTorch DataLoaders.

### Example

```
def test_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=False, transform=transform,
                    download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=False
    )

    return loader
```

**Note:** If you don't need a test dataset and a *test_step()*, you don't need to implement this method.

**test_end**(*outputs*)

**Warning:** Deprecated in v0.7.0. Use *test_epoch_end()* instead. Will be removed in 1.0.0.

**test_epoch_end**(*outputs*)

Called at the end of a test epoch with the output of all test steps.

```
# the pseudocode for these calls
test_outs = []
for test_batch in test_data:
    out = test_step(test_batch)
    test_outs.append(out)
test_epoch_end(test_outs)
```

**Parameters outputs** (Union[List[Dict[str, Tensor]], List[List[Dict[str, Tensor]]]]) – List of outputs you defined in *test_step_end()*, or if there are multiple dataloaders, a list containing a list of outputs for each dataloader

**Returns**

Dict has the following optional keys:

- progress_bar -> Dict for progress bar display. Must have only tensors.
- log -> Dict of metrics to add to logger. Must have only tensors (no images, etc).

**Return type** Dict or OrderedDict

**Note:** If you didn't define a `test_step()`, this won't be called.

- The outputs here are strictly for logging or progress bar.
- If you don't need to display anything, don't return anything.
- If you want to manually set current step, specify it with the 'step' key in the 'log' Dict

### Examples

With a single dataloader:

```
def test_epoch_end(self, outputs):
    test_acc_mean = 0
    for output in outputs:
        test_acc_mean += output['test_acc']

    test_acc_mean /= len(outputs)
    tqdm_dict = {'test_acc': test_acc_mean.item()}

    # show test_loss and test_acc in progress bar but only log test_loss
    results = {
        'progress_bar': tqdm_dict,
        'log': {'test_acc': test_acc_mean.item()}
    }
    return results
```

With multiple dataloaders, *outputs* will be a list of lists. The outer list contains one entry per dataloader, while the inner list contains the individual outputs of each test step for that dataloader.

```
def test_epoch_end(self, outputs):
    test_acc_mean = 0
    i = 0
    for dataloader_outputs in outputs:
        for output in dataloader_outputs:
            test_acc_mean += output['test_acc']
            i += 1

    test_acc_mean /= i
    tqdm_dict = {'test_acc': test_acc_mean.item()}

    # show test_loss and test_acc in progress bar but only log test_loss
    results = {
        'progress_bar': tqdm_dict,
        'log': {'test_acc': test_acc_mean.item(), 'step': self.current_epoch}
    }
    return results
```

**test_step**(**args*, ***kwargs*)

Operates on a single batch of data from the test set. In this step you'd normally generate examples or calculate anything of interest such as accuracy.

```
# the pseudocode for these calls
test_outs = []
for test_batch in test_data:
    out = test_step(test_batch)
    test_outs.append(out)
test_epoch_end(test_outs)
```

**Parameters**

- **batch** (Tensor | (Tensor, ...) | [Tensor, ...]) – The output of your DataLoader. A tensor, tuple or list.
- **batch_idx** (*int*) – The index of this batch.
- **dataloader_idx** (*int*) – The index of the dataloader that produced this batch (only if multiple test datasets used).

**Return type** Dict[str, Tensor]

**Returns** Dict or OrderedDict - passed to the *test_epoch_end()* method. If you defined *test_step_end()* it will go to that first.

```
# if you have one test dataloader:
def test_step(self, batch, batch_idx)

# if you have multiple test dataloaders:
def test_step(self, batch, batch_idx, dataloader_idx)
```

### Examples

```
# CASE 1: A single test dataset
def test_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # all optional...
    # return whatever you need for the collation function test_epoch_end
    output = OrderedDict({
        'val_loss': loss_val,
        'val_acc': torch.tensor(val_acc), # everything must be a tensor
    })
```

(continues on next page)

(continued from previous page)

```
    # return an optional dict
    return output
```

If you pass in multiple validation datasets, `test_step()` will have an additional argument.

```
# CASE 2: multiple test datasets
def test_step(self, batch, batch_idx, dataset_idx):
    # dataset_idx tells you which dataset this is.
```

**Note:** If you don't need to validate you don't need to implement this method.

**Note:** When the `test_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of the test epoch, the model goes back to training mode and gradients are enabled.

**test_step_end**(**args*, ***kwargs*)

Use this when testing with dp or ddp2 because `test_step()` will operate on only part of the batch. However, this is still optional and only needed for things like softmax or NCE loss.

**Note:** If you later switch to ddp or some other mode, this will still be called so that you don't have to change your code.

```
# pseudocode
sub_batches = split_batches_for_dp(batch)
batch_parts_outputs = [test_step(sub_batch) for sub_batch in sub_batches]
test_step_end(batch_parts_outputs)
```

**Parameters** **batch_parts_outputs** – What you return in `test_step()` for each batch part.

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

**Returns** Dict or OrderedDict - passed to the `test_epoch_end()`.

### Examples

```
# WITHOUT test_step_end
# if used in DP or DDP2, this batch is 1/num_gpus large
def test_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self(x)
    loss = self.softmax(out)
    loss = nce_loss(loss)
    return {'loss': loss}

# --------------
# with test_step_end to do softmax over the full batch
```

(continues on next page)

(continued from previous page)

```
def test_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self(x)
    return {'out': out}

def test_step_end(self, outputs):
    # this out is now the full size of the batch
    out = outputs['out']

    # this softmax now uses the full batch size
    loss = nce_loss(loss)
    return {'loss': loss}
```

**See also:**

See the *Multi-GPU training* guide for more details.

**tng_dataloader**()

> **Warning:** Deprecated in v0.5.0. Use *train_dataloader()* instead. Will be removed in 1.0.0.

**train_dataloader**()

Implement a PyTorch DataLoader for training.

**Return type** `DataLoader`

**Returns** Single PyTorch `DataLoader`.

The dataloader you return will not be called every epoch unless you set *reload_dataloaders_every_epoch* to `True`.

It's recommended that all data downloads and preparation happen in *prepare_data()*.

- *fit()*
- ...
- *prepare_data()*
- *train_dataloader()*

> **Note:** Lightning adds the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

### Example

```
def train_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=True, transform=transform,
                    download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=True
    )
    return loader
```

**training_end**(**args*, ***kwargs*)

> **Warning:** Deprecated in v0.7.0. Use *training_step_end()* instead.

**training_epoch_end**(*outputs*)

Called at the end of the training epoch with the outputs of all training steps.

```
# the pseudocode for these calls
train_outs = []
for train_batch in train_data:
    out = training_step(train_batch)
    train_outs.append(out)
training_epoch_end(train_outs)
```

**Parameters** **outputs** (Union[List[Dict[str, Tensor]], List[List[Dict[str, Tensor]]]]) – List of outputs you defined in *training_step()*, or if there are multiple dataloaders, a list containing a list of outputs for each dataloader.

**Return type** Dict[str, Dict[str, Tensor]]

**Returns**

Dict or OrderedDict. May contain the following optional keys:

- log (metrics to be added to the logger; only tensors)
- progress_bar (dict for progress bar display)
- any metric used in a callback (e.g. early stopping).

> **Note:** If this method is not overridden, this won't be called.

- The outputs here are strictly for logging or progress bar.
- If you don't need to display anything, don't return anything.
- If you want to manually set current step, you can specify the 'step' key in the 'log' dict.

### Examples

With a single dataloader:

```
def training_epoch_end(self, outputs):
    train_acc_mean = 0
    for output in outputs:
        train_acc_mean += output['train_acc']

    train_acc_mean /= len(outputs)

    # log training accuracy at the end of an epoch
    results = {
        'log': {'train_acc': train_acc_mean.item()},
        'progress_bar': {'train_acc': train_acc_mean},
    }
    return results
```

With multiple dataloaders, `outputs` will be a list of lists. The outer list contains one entry per dataloader, while the inner list contains the individual outputs of each training step for that dataloader.

```
def training_epoch_end(self, outputs):
    train_acc_mean = 0
    i = 0
    for dataloader_outputs in outputs:
        for output in dataloader_outputs:
            train_acc_mean += output['train_acc']
            i += 1

    train_acc_mean /= i

    # log training accuracy at the end of an epoch
    results = {
        'log': {'train_acc': train_acc_mean.item(), 'step': self.current_
↪epoch}
        'progress_bar': {'train_acc': train_acc_mean},
    }
    return results
```

**training_step**(**args*, ***kwargs*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

**Parameters**

- **batch** (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (*int*) – Integer displaying index of this batch
- **optimizer_idx** (*int*) – When using multiple optimizers, this argument will also be present.
- **hiddens** (`Tensor`) – Passed in if *truncated_bptt_steps* > 0.

**Return type** `Union`[`int`, `Dict`[`str`, `Union`[`Tensor`, `Dict`[`str`, `Tensor`]]]]

**Returns**

Dict with loss key and optional log or progress bar keys. When implementing *training_step()*, return whatever you need in that step:

- loss -> tensor scalar **REQUIRED**
- progress_bar -> Dict for progress bar display. Must have only tensors
- log -> Dict of metrics to add to logger. Must have only tensors (no images, etc)

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

### Examples

```
def training_step(self, batch, batch_idx):
    x, y, z = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, x)

    logger_logs = {'training_loss': loss} # optional (MUST ALL BE TENSORS)

    # if using TestTubeLogger or TensorBoardLogger you can nest scalars
    logger_logs = {'losses': logger_logs} # optional (MUST ALL BE TENSORS)

    output = {
        'loss': loss, # required
        'progress_bar': {'training_loss': loss}, # optional (MUST ALL BE TENSORS)
        'log': logger_logs
    }

    # return a dict
    return output
```

If you define multiple optimizers, this step will be called with an additional `optimizer_idx` parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
    if optimizer_idx == 1:
        # do training_step with decoder
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    ...
    out, hiddens = self.lstm(data, hiddens)
    ...

    return {
        "loss": ...,
        "hiddens": hiddens  # remember to detach() this
    }
```

### Notes

The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

**training_step_end**(**args*, ***kwargs*)

Use this when training with dp or ddp2 because `training_step()` will operate on only part of the batch. However, this is still optional and only needed for things like softmax or NCE loss.

**Note:** If you later switch to ddp or some other mode, this will still be called so that you don't have to change your code

```
# pseudocode
sub_batches = split_batches_for_dp(batch)
batch_parts_outputs = [training_step(sub_batch) for sub_batch in sub_batches]
training_step_end(batch_parts_outputs)
```

**Parameters** **batch_parts_outputs** – What you return in *training_step* for each batch part.

**Return type** `Dict[str, Union[Tensor, Dict[str, Tensor]]]`

**Returns**

Dict with loss key and optional log or progress bar keys.

- loss -> tensor scalar **REQUIRED**
- progress_bar -> Dict for progress bar display. Must have only tensors
- log -> Dict of metrics to add to logger. Must have only tensors (no images, etc)

### Examples

```
# WITHOUT training_step_end
# if used in DP or DDP2, this batch is 1/num_gpus large
def training_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self(x)
    loss = self.softmax(out)
    loss = nce_loss(loss)
    return {'loss': loss}

# --------------
# with training_step_end to do softmax over the full batch
def training_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self(x)
    return {'out': out}

def training_step_end(self, outputs):
    # this out is now the full size of the batch
    out = outputs['out']
```

(continues on next page)

(continued from previous page)

```
    # this softmax now uses the full batch size
    loss = nce_loss(loss)
    return {'loss': loss}
```

**See also:**

See the *Multi-GPU training* guide for more details.

**unfreeze**()

Unfreeze all parameters for training.

```
model = MyLightningModule(...)
model.unfreeze()
```

**Return type** None

**val_dataloader**()

Implement one or multiple PyTorch DataLoaders for validation.

The dataloader you return will not be called every epoch unless you set *reload_dataloaders_every_epoch* to True.

It's recommended that all data downloads and preparation happen in *prepare_data()*.

- *fit()*
- ...
- *prepare_data()*
- *train_dataloader()*
- *val_dataloader()*
- *test_dataloader()*

**Note:** Lightning adds the correct sampler for distributed and arbitrary hardware There is no need to set it yourself.

**Return type** Union[DataLoader, List[DataLoader]]

**Returns** Single or multiple PyTorch DataLoaders.

### Examples

```
def val_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=False,
                    transform=transform, download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=False
    )
```

(continues on next page)

(continued from previous page)

```
    return loader

# can also return multiple dataloaders
def val_dataloader(self):
    return [loader_a, loader_b, ..., loader_n]
```

**Note:** If you don't need a validation dataset and a *validation_step()*, you don't need to implement this method.

**Note:** In the case where you return multiple validation dataloaders, the *validation_step()* will have an argument `dataset_idx` which matches the order here.

**validation_end**(*outputs*)

**Warning:** Deprecated in v0.7.0. Use *validation_epoch_end()* instead. Will be removed in 1.0.0.

**validation_epoch_end**(*outputs*)

Called at the end of the validation epoch with the outputs of all validation steps.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

**Parameters outputs** (Union[List[Dict[str, Tensor]], List[List[Dict[str, Tensor]]]]) – List of outputs you defined in *validation_step()*, or if there are multiple dataloaders, a list containing a list of outputs for each dataloader.

**Return type** Dict[str, Dict[str, Tensor]]

**Returns**

Dict or OrderedDict. May have the following optional keys:

- progress_bar (dict for progress bar display; only tensors)
- log (dict of metrics to add to logger; only tensors).

**Note:** If you didn't define a *validation_step()*, this won't be called.

- The outputs here are strictly for logging or progress bar.
- If you don't need to display anything, don't return anything.
- If you want to manually set current step, you can specify the 'step' key in the 'log' dict.

### Examples

With a single dataloader:

```
def validation_epoch_end(self, outputs):
    val_acc_mean = 0
    for output in outputs:
        val_acc_mean += output['val_acc']

    val_acc_mean /= len(outputs)
    tqdm_dict = {'val_acc': val_acc_mean.item()}

    # show val_acc in progress bar but only log val_loss
    results = {
        'progress_bar': tqdm_dict,
        'log': {'val_acc': val_acc_mean.item()}
    }
    return results
```

With multiple dataloaders, *outputs* will be a list of lists. The outer list contains one entry per dataloader, while the inner list contains the individual outputs of each validation step for that dataloader.

```
def validation_epoch_end(self, outputs):
    val_acc_mean = 0
    i = 0
    for dataloader_outputs in outputs:
        for output in dataloader_outputs:
            val_acc_mean += output['val_acc']
            i += 1

    val_acc_mean /= i
    tqdm_dict = {'val_acc': val_acc_mean.item()}

    # show val_loss and val_acc in progress bar but only log val_loss
    results = {
        'progress_bar': tqdm_dict,
        'log': {'val_acc': val_acc_mean.item(), 'step': self.current_epoch}
    }
    return results
```

**validation_step**(**args*, ***kwargs*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(train_batch)
    val_outs.append(out)
    validation_epoch_end(val_outs)
```

**Parameters**

- **batch** (Tensor | (Tensor, …) | [Tensor, …]) – The output of your DataLoader. A tensor, tuple or list.
- **batch_idx** (*int*) – The index of this batch

- **dataloader_idx** (*int*) – The index of the dataloader that produced this batch (only if multiple val datasets used)

**Return type** Dict[str, Tensor]

**Returns** Dict or OrderedDict - passed to *validation_epoch_end()*. If you defined *validation_step_end()* it will go to that first.

```
# pseudocode of order
out = validation_step()
if defined('validation_step_end'):
    out = validation_step_end(out)
out = validation_epoch_end(out)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx)

# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx)
```

### Examples

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # all optional...
    # return whatever you need for the collation function validation_epoch_end
    output = OrderedDict({
        'val_loss': loss_val,
        'val_acc': torch.tensor(val_acc), # everything must be a tensor
    })

    # return an optional dict
    return output
```

If you pass in multiple val datasets, validation_step will have an additional argument.

```
# CASE 2: multiple validation datasets
def validation_step(self, batch, batch_idx, dataset_idx):
    # dataset_idx tells you which dataset this is.
```

**Note:** If you don't need to validate you don't need to implement this method.

**Note:** When the `validation_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

**validation_step_end**(**args, **kwargs*)

Use this when validating with dp or ddp2 because `validation_step()` will operate on only part of the batch. However, this is still optional and only needed for things like softmax or NCE loss.

**Note:** If you later switch to ddp or some other mode, this will still be called so that you don't have to change your code.

```
# pseudocode
sub_batches = split_batches_for_dp(batch)
batch_parts_outputs = [validation_step(sub_batch) for sub_batch in sub_
↪batches]
validation_step_end(batch_parts_outputs)
```

**Parameters** **batch_parts_outputs** – What you return in `validation_step()` for each batch part.

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

**Returns** Dict or OrderedDict - passed to the `validation_epoch_end()` method.

### Examples

```
# WITHOUT validation_step_end
# if used in DP or DDP2, this batch is 1/num_gpus large
def validation_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self(x)
    loss = self.softmax(out)
    loss = nce_loss(loss)
    return {'loss': loss}

# --------------
# with validation_step_end to do softmax over the full batch
def validation_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self(x)
    return {'out': out}

def validation_epoch_end(self, outputs):
    # this out is now the full size of the batch
    out = outputs['out']
```

(continues on next page)

(continued from previous page)

```
    # this softmax now uses the full batch size
    loss = nce_loss(loss)
    return {'loss': loss}
```

**See also:**

See the *Multi-GPU training* guide for more details.

**_device = None**
device reference

**_dtype = None**
Current dtype

**current_epoch = None**
The current epoch

**property example_input_array**

**Return type** Any

**global_step = None**
Total training batches seen across all epochs

**property hparams**

**Return type** Union[*AttributeDict*, str]

**logger = None**
Pointer to the logger object

**property on_gpu**
True if your model is currently running on GPUs. Useful to set flags around the LightningModule for different CPU vs GPU behavior.

**trainer = None**
Pointer to the trainer object

**use_amp = None**
True if using amp

**use_ddp = None**
True if using ddp

**use_ddp2 = None**
True if using ddp2

**use_dp = None**
True if using dp

### pytorch_lightning.core.memory module

**class** pytorch_lightning.core.memory.**LayerSummary**(*module*)

Bases: object

Summary class for a single layer in a *LightningModule*. It collects the following information:

- Type of the layer (e.g. Linear, BatchNorm1d, . . . )
- Input shape
- Output shape
- Number of parameters

The input and output shapes are only known after the example input array was passed through the model.

Example:

```
>>> model = torch.nn.Conv2d(3, 8, 3)
>>> summary = LayerSummary(model)
>>> summary.num_parameters
224
>>> summary.layer_type
'Conv2d'
>>> output = model(torch.rand(1, 3, 5, 5))
>>> summary.in_size
[1, 3, 5, 5]
>>> summary.out_size
[1, 8, 3, 3]
```

**Parameters** **module** (Module) – A module to summarize

**_register_hook**()

Registers a hook on the module that computes the input- and output size(s) on the first forward pass. The hook will remove itself from the module, meaning that recursive models will only record their input- and output shapes once.

**property in_size**

**property layer_type**

Returns the class name of the module.

**Return type** str

**property num_parameters**

Returns the number of parameters in this module.

**Return type** int

**property out_size**

**class** pytorch_lightning.core.memory.**ModelSummary**(*model*, *mode='top'*)

Bases: object

Generates a summary of all layers in a *LightningModule*.

**Parameters**

- **model** – The model to summarize (also referred to as the root module)
- **mode** – Can be one of
  - *top* (default): only the top-level modules will be recorded (the children of the root module)

– *full*: summarizes all layers and their submodules in the root module

The string representation of this summary prints a table with columns containing the name, type and number of parameters for each layer.

The root module may also have an attribute `example_input_array` as shown in the example below. If present, the root module will be called with it as input to determine the intermediate input- and output shapes of all layers. Supported are tensors and nested lists and tuples of tensors. All other types of inputs will be skipped and show as *?* in the summary table. The summary will also display *?* for layers not used in the forward pass.

Example:

```
>>> class LitModel(pl.LightningModule):
...
...     def __init__(self):
...         super().__init__()
...         self.net = nn.Sequential(nn.Linear(256, 512), nn.BatchNorm1d(512))
...         self.example_input_array = torch.zeros(10, 256)  # optional
...
...     def forward(self, x):
...         return self.net(x)
...
>>> model = LitModel()
>>> ModelSummary(model, mode='top')
  | Name | Type       | Params | In sizes  | Out sizes
------------------------------------------------------------
0 | net  | Sequential | 132 K  | [10, 256] | [10, 512]
>>> ModelSummary(model, mode='full')
  | Name  | Type        | Params | In sizes  | Out sizes
--------------------------------------------------------------
0 | net   | Sequential  | 132 K  | [10, 256] | [10, 512]
1 | net.0 | Linear      | 131 K  | [10, 256] | [10, 512]
2 | net.1 | BatchNorm1d | 1 K    | [10, 512] | [10, 512]
```

**_forward_example_input**()

Run the example input through each layer to get input- and output sizes.

**Return type** None

**summarize**()

**Return type** Dict[str, *LayerSummary*]

**MODES = ['full', 'top']**

**MODE_DEFAULT = 'top'**

**MODE_FULL = 'full'**

**MODE_TOP = 'top'**

**property in_sizes**

**Return type** List

**property layer_names**

**Return type** List[str]

**property layer_types**

**Return type** List[str]

**property named_modules**

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

**property out_sizes**

**Return type** List

**property param_nums**

**Return type** List[int]

pytorch_lightning.core.memory.**_format_summary_table**(**cols*)

Takes in a number of arrays, each specifying a column in the summary table, and combines them all into one big string defining the summary table that are nicely formatted.

**Return type** str

pytorch_lightning.core.memory.**get_gpu_memory_map**()

Get the current gpu usage.

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

**Returns** A dictionary in which the keys are device ids as integers and values are memory usage as integers in MB.

pytorch_lightning.core.memory.**get_human_readable_count**(*number*)

Abbreviates an integer number with K, M, B, T for thousands, millions, billions and trillions, respectively.

### Examples

```
>>> get_human_readable_count(123)
'123  '
>>> get_human_readable_count(1234)  # (one thousand)
'1 K'
>>> get_human_readable_count(2e6)   # (two million)
'2 M'
>>> get_human_readable_count(3e9)   # (three billion)
'3 B'
>>> get_human_readable_count(4e12)  # (four trillion)
'4 T'
>>> get_human_readable_count(5e15)  # (more than trillion)
'5,000 T'
```

**Parameters** **number** (int) – a positive integer number

**Return type** str

**Returns** A string formatted according to the pattern described above.

pytorch_lightning.core.memory.**get_memory_profile**(*mode*)

Get a profile of the current memory usage.

**Parameters** **mode** (str) – There are two modes:

- 'all' means return memory for all gpus
- 'min_max' means return memory for max and min

**Return type** Union[Dict[str, int], Dict[int, int]]

**Returns**

A dictionary in which the keys are device ids as integers and values are memory usage as integers in MB. If mode is 'min_max', the dictionary will also contain two additional keys:

- 'min_gpu_mem': the minimum memory usage in MB
- 'max_gpu_mem': the maximum memory usage in MB

pytorch_lightning.core.memory.**parse_batch_shape**(*batch*)

**Return type** Union[str, List]

### pytorch_lightning.core.saving module

**class** pytorch_lightning.core.saving.**ModelIO**

Bases: object

**classmethod _load_model_state**(*checkpoint*, **args*, ***kwargs*)

**classmethod load_from_checkpoint**(*checkpoint_path*, **args*, *map_location=None*, *hparams_file=None*, *tags_csv=None*, ***kwargs*)

Primary way of loading a model from a checkpoint. When Lightning saves a checkpoint it stores the arguments passed to *__init__* in the checkpoint under *module_arguments*

Any arguments specified through *args and **kwargs will override args stored in *hparams*.

**Parameters**

- **checkpoint_path** (str) – Path to checkpoint. This can also be a URL.
- **args** – Any positional args needed to init the model.
- **map_location** (Union[Dict[str, str], str, device, int, Callable, None]) – If your checkpoint saved a GPU model and you now load on CPUs or a different number of GPUs, use this to map to the new setup. The behaviour is the same as in torch.load().
- **hparams_file** (Optional[str]) – Optional path to a .yaml file with hierarchical structure as in this example:

```
drop_prob: 0.2
dataloader:
    batch_size: 32
```

  You most likely won't need this since Lightning will always save the hyperparameters to the checkpoint. However, if your checkpoint weights don't have the hyperparameters saved, use this method to pass in a .yaml file with the hparams you'd like to use. These will be converted into a dict and passed into your LightningModule for use.

  If your model's *hparams* argument is Namespace and .yaml file has hierarchical structure, you need to refactor your model to treat *hparams* as dict.

  .csv files are acceptable here till v0.9.0, see tags_csv argument for detailed usage.

- **tags_csv** (Optional[str]) –

  > **Warning:** Deprecated since version 0.7.6.
  >
  > *tags_csv* argument is deprecated in v0.7.6. Will be removed v0.9.0.

  Optional path to a .csv file with two columns (key, value) as in this example:

```
key,value
drop_prob,0.2
batch_size,32
```

Use this method to pass in a .csv file with the hparams you'd like to use.

- **hparam_overrides** – A dictionary with keys to override in the hparams
- **kwargs** – Any keyword args needed to init the model.

**Returns** `LightningModule` with loaded weights and hyperparameters (if available).

### Example

```
# load weights without mapping ...
MyLightningModule.load_from_checkpoint('path/to/checkpoint.ckpt')

# or load weights mapping all weights from GPU 1 to GPU 0 ...
map_location = {'cuda:1':'cuda:0'}
MyLightningModule.load_from_checkpoint(
    'path/to/checkpoint.ckpt',
    map_location=map_location
)

# or load weights and hyperparameters from separate files.
MyLightningModule.load_from_checkpoint(
    'path/to/checkpoint.ckpt',
    hparams_file='/path/to/hparams_file.yaml'
)

# override some of the params with new values
MyLightningModule.load_from_checkpoint(
    PATH,
    num_layers=128,
    pretrained_ckpt_path: NEW_PATH,
)

# predict
pretrained_model.eval()
pretrained_model.freeze()
y_hat = pretrained_model(x)
```

**classmethod load_from_metrics**(*weights_path*, *tags_csv*, *map_location=None*)

> **Warning:** Deprecated in version 0.7.0. You should use *load_from_checkpoint()* instead. Will be removed in v0.9.0.

**on_hpc_load**(*checkpoint*)

Hook to do whatever you need right before Slurm manager loads the model.

**Parameters** **checkpoint** (`Dict[str, Any]`) – A dictionary with variables from the checkpoint.

**Return type** `None`

**on_hpc_save** (*checkpoint*)
Hook to do whatever you need right before Slurm manager saves the model.

**Parameters** **checkpoint** (`Dict`[`str`, `Any`]) – A dictionary in which you can save variables to save in a checkpoint. Contents need to be pickleable.

**Return type** `None`

**on_load_checkpoint** (*checkpoint*)
Do something with the checkpoint. Gives model a chance to load something before `state_dict` is restored.

**Parameters** **checkpoint** (`Dict`[`str`, `Any`]) – A dictionary with variables from the checkpoint.

**Return type** `None`

**on_save_checkpoint** (*checkpoint*)
Give the model a chance to add something to the checkpoint. `state_dict` is already there.

**Parameters** **checkpoint** (`Dict`[`str`, `Any`]) – A dictionary in which you can save variables to save in a checkpoint. Contents need to be pickleable.

**Return type** `None`

**CHECKPOINT_HYPER_PARAMS_KEY = 'hyper_parameters'**

**CHECKPOINT_HYPER_PARAMS_NAME = 'hparams_name'**

**CHECKPOINT_HYPER_PARAMS_TYPE = 'hparams_type'**

pytorch_lightning.core.saving.**convert** (*val*)

**Return type** `Union`[`int`, `float`, `bool`, `str`]

pytorch_lightning.core.saving.**load_hparams_from_tags_csv** (*tags_csv*)
Load hparams from a file.

```
>>> hparams = Namespace(batch_size=32, learning_rate=0.001, data_root='./any/path/
↪here')
>>> path_csv = './testing-hparams.csv'
>>> save_hparams_to_tags_csv(path_csv, hparams)
>>> hparams_new = load_hparams_from_tags_csv(path_csv)
>>> vars(hparams) == hparams_new
True
>>> os.remove(path_csv)
```

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

pytorch_lightning.core.saving.**load_hparams_from_yaml** (*config_yaml*)
Load hparams from a file.

```
>>> hparams = Namespace(batch_size=32, learning_rate=0.001, data_root='./any/path/
↪here')
>>> path_yaml = './testing-hparams.yaml'
>>> save_hparams_to_yaml(path_yaml, hparams)
>>> hparams_new = load_hparams_from_yaml(path_yaml)
>>> vars(hparams) == hparams_new
True
>>> os.remove(path_yaml)
```

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

pytorch_lightning.core.saving.**save_hparams_to_tags_csv**(*tags_csv*, *hparams*)

**Return type** None

pytorch_lightning.core.saving.**save_hparams_to_yaml**(*config_yaml*, *hparams*)

**Parameters**

- **config_yaml** – path to new YAML file
- **hparams** (Union[dict, Namespace]) – parameters to be saved

**Return type** None

pytorch_lightning.core.saving.**update_hparams**(*hparams*, *updates*)

Overrides hparams with new values

```
>>> hparams = {'c': 4}
>>> update_hparams(hparams, {'a': {'b': 2}, 'c': 1})
>>> hparams['a']['b'], hparams['c']
(2, 1)
>>> update_hparams(hparams, {'a': {'b': 4}, 'c': 7})
>>> hparams['a']['b'], hparams['c']
(4, 7)
```

**Parameters**

- **hparams** (dict) – the original params and also target object
- **updates** (dict) – new params to be used as update

**Return type** None

## 37.2 pytorch_lightning.callbacks package

**class** pytorch_lightning.callbacks.**Callback**

Bases: abc.ABC

Abstract base class used to build new callbacks.

**on_batch_end**(*trainer*, *pl_module*)

Called when the training batch ends.

**on_batch_start**(*trainer*, *pl_module*)

Called when the training batch begins.

**on_epoch_end**(*trainer*, *pl_module*)

Called when the epoch ends.

**on_epoch_start**(*trainer*, *pl_module*)

Called when the epoch begins.

**on_fit_end**(*trainer*)

Called when fit ends

**on_fit_start**(*trainer*)

Called when fit begins

**on_init_end**(*trainer*)

Called when the trainer initialization ends, model has not yet been set.

**on_init_start**(*trainer*)
Called when the trainer initialization begins, model has not yet been set.

**on_keyboard_interrupt**(*trainer*, *pl_module*)
Called when the training is interrupted by KeyboardInterrupt.

**on_sanity_check_end**(*trainer*, *pl_module*)
Called when the validation sanity check ends.

**on_sanity_check_start**(*trainer*, *pl_module*)
Called when the validation sanity check starts.

**on_test_batch_end**(*trainer*, *pl_module*)
Called when the test batch ends.

**on_test_batch_start**(*trainer*, *pl_module*)
Called when the test batch begins.

**on_test_end**(*trainer*, *pl_module*)
Called when the test ends.

**on_test_start**(*trainer*, *pl_module*)
Called when the test begins.

**on_train_end**(*trainer*, *pl_module*)
Called when the train ends.

**on_train_start**(*trainer*, *pl_module*)
Called when the train begins.

**on_validation_batch_end**(*trainer*, *pl_module*)
Called when the validation batch ends.

**on_validation_batch_start**(*trainer*, *pl_module*)
Called when the validation batch begins.

**on_validation_end**(*trainer*, *pl_module*)
Called when the validation loop ends.

**on_validation_start**(*trainer*, *pl_module*)
Called when the validation loop begins.

**setup**(*trainer*, *stage*)
Called when fit or test begins

**teardown**(*trainer*, *stage*)
Called when fit or test ends

**class** pytorch_lightning.callbacks.**EarlyStopping**(*monitor='val_loss'*, *min_delta=0.0*, *patience=3*, *verbose=False*, *mode='auto'*, *strict=True*)

Bases: *pytorch_lightning.callbacks.base.Callback*

**Parameters**

- **monitor** (str) – quantity to be monitored. Default: `'val_loss'`.
- **min_delta** (float) – minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than *min_delta*, will count as no improvement. Default: `0`.
- **patience** (int) – number of validation epochs with no improvement after which training will be stopped. Default: `0`.
- **verbose** (bool) – verbosity mode. Default: `False`.

- **mode** (str) – one of {auto, min, max}. In *min* mode, training will stop when the quantity monitored has stopped decreasing; in *max* mode it will stop when the quantity monitored has stopped increasing; in *auto* mode, the direction is automatically inferred from the name of the monitored quantity. Default: `'auto'`.
- **strict** (bool) – whether to crash the training if *monitor* is not found in the validation metrics. Default: `True`.

Example:

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.callbacks import EarlyStopping
>>> early_stopping = EarlyStopping('val_loss')
>>> trainer = Trainer(early_stop_callback=early_stopping)
```

**_run_early_stopping_check** (*trainer*, *pl_module*)

**_validate_condition_metric** (*logs*)
Checks that the condition metric for early stopping is good :param _sphinx_paramlinks_pytorch_lightning.callbacks.EarlyStopping._validate_condition_metric.logs: :return:

**on_train_end** (*trainer*, *pl_module*)
Called when the train ends.

**on_train_start** (*trainer*, *pl_module*)
Called when the train begins.

**on_validation_end** (*trainer*, *pl_module*)
Called when the validation loop ends.

**mode_dict = {'max': torch.gt, 'min': torch.lt}**

**property monitor_op**

**class** pytorch_lightning.callbacks.**ModelCheckpoint** (*filepath=None*, *monitor='val_loss'*, *verbose=False*, *save_last=False*, *save_top_k=1*, *save_weights_only=False*, *mode='auto'*, *period=1*, *prefix=''*)

Bases: `pytorch_lightning.callbacks.base.Callback`

Save the model after every epoch if it improves.

After training finishes, use `best_model_path` to retrieve the path to the best checkpoint file and `best_model_score` to retrieve its score.

**Parameters**

- **filepath** (Optional[str]) – path to save the model file. Can contain named formatting options to be auto-filled.

  Example:

```
# custom path
# saves a file like: my/path/epoch_0.ckpt
>>> checkpoint_callback = ModelCheckpoint('my/path/')

# save any arbitrary metrics like `val_loss`, etc. in name
# saves a file like: my/path/epoch=2-val_loss=0.2_other_metric=0.3.
↪ckpt
>>> checkpoint_callback = ModelCheckpoint(
```

(continues on next page)

(continued from previous page)

```
...     filepath='my/path/{epoch}-{val_loss:.2f}-{other_metric:.2f}
↪'
... )
```

Can also be set to *None*, then it will be set to default location during trainer construction.

- **monitor** (str) – quantity to monitor.
- **verbose** (bool) – verbosity mode. Default: `False`.
- **save_last** (bool) – always saves the model at the end of the epoch. Default: `False`.
- **save_top_k** (int) – if *save_top_k* == *k*, the best k models according to the quantity monitored will be saved. if `save_top_k == 0`, no models are saved. if `save_top_k == -1`, all models are saved. Please note that the monitors are checked every *period* epochs. if `save_top_k >= 2` and the callback is called multiple times inside an epoch, the name of the saved file will be appended with a version count starting with *v0*.
- **mode** (str) – one of {auto, min, max}. If `save_top_k != 0`, the decision to overwrite the current save file is made based on either the maximization or the minimization of the monitored quantity. For *val_acc*, this should be *max*, for *val_loss* this should be *min*, etc. In *auto* mode, the direction is automatically inferred from the name of the monitored quantity.
- **save_weights_only** (bool) – if `True`, then only the model's weights will be saved (`model.save_weights(filepath)`), else the full model is saved (`model.save(filepath)`).
- **period** (int) – Interval (number of epochs) between checkpoints.

Example:

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.callbacks import ModelCheckpoint

# saves checkpoints to 'my/path/' whenever 'val_loss' has a new min
>>> checkpoint_callback = ModelCheckpoint(filepath='my/path/')
>>> trainer = Trainer(checkpoint_callback=checkpoint_callback)

# save epoch and val_loss in name
# saves a file like: my/path/sample-mnist_epoch=02_val_loss=0.32.ckpt
>>> checkpoint_callback = ModelCheckpoint(
...     filepath='my/path/sample-mnist_{epoch:02d}-{val_loss:.2f}'
... )

# retrieve the best checkpoint after training
checkpoint_callback = ModelCheckpoint(filepath='my/path/')
trainer = Trainer(checkpoint_callback=checkpoint_callback)
model = ...
trainer.fit(model)
checkpoint_callback.best_model_path
```

**_del_model**(*filepath*)

**_do_check_save**(*filepath, current, epoch*)

**_save_model**(*filepath*)

**check_monitor_top_k**(*current*)

**format_checkpoint_name**(*epoch, metrics, ver=None*)
Generate a filename according to the defined template.

Example:

```
>>> tmpdir = os.path.dirname(__file__)
>>> ckpt = ModelCheckpoint(os.path.join(tmpdir, '{epoch}'))
>>> os.path.basename(ckpt.format_checkpoint_name(0, {}))
'epoch=0.ckpt'
>>> ckpt = ModelCheckpoint(os.path.join(tmpdir, '{epoch:03d}'))
>>> os.path.basename(ckpt.format_checkpoint_name(5, {}))
'epoch=005.ckpt'
>>> ckpt = ModelCheckpoint(os.path.join(tmpdir, '{epoch}-{val_loss:.2f}'))
>>> os.path.basename(ckpt.format_checkpoint_name(2, dict(val_loss=0.123456)))
'epoch=2-val_loss=0.12.ckpt'
>>> ckpt = ModelCheckpoint(os.path.join(tmpdir, '{missing:d}'))
>>> os.path.basename(ckpt.format_checkpoint_name(0, {}))
'missing=0.ckpt'
```

**on_validation_end**(*trainer*, *pl_module*)
Called when the validation loop ends.

**property best**

**property kth_best_model**

**class** pytorch_lightning.callbacks.**GradientAccumulationScheduler**(*scheduling*)
Bases: *pytorch_lightning.callbacks.base.Callback*

Change gradient accumulation factor according to scheduling.

**Parameters** **scheduling** (dict) – scheduling in format {epoch: accumulation_factor}

> **Warning:** Epochs indexing starts from "1" until v0.6.x, but will start from "0" in v0.8.0.

Example:

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.callbacks import GradientAccumulationScheduler

# at epoch 5 start accumulating every 2 batches
>>> accumulator = GradientAccumulationScheduler(scheduling={5: 2})
>>> trainer = Trainer(callbacks=[accumulator])

# alternatively, pass the scheduling dict directly to the Trainer
>>> trainer = Trainer(accumulate_grad_batches={5: 2})
```

**on_epoch_start**(*trainer*, *pl_module*)
Called when the epoch begins.

**class** pytorch_lightning.callbacks.**LearningRateLogger**
Bases: *pytorch_lightning.callbacks.base.Callback*

Automatically logs learning rate for learning rate schedulers during training.

Example:

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.callbacks import LearningRateLogger
>>> lr_logger = LearningRateLogger()
>>> trainer = Trainer(callbacks=[lr_logger])
```

Logging names are automatically determined based on optimizer class name. In case of multiple optimizers of same type, they will be named *Adam*, *Adam-1* etc. If a optimizer has multiple parameter groups they will be named *Adam/pg1*, *Adam/pg2* etc. To control naming, pass in a *name* keyword in the construction of the learning rate schdulers

Example:

```
def configure_optimizer(self):
    optimizer = torch.optim.Adam(...)
    lr_scheduler = {'scheduler': torch.optim.lr_schedulers.LambdaLR(optimizer, ...
↪)
                    'name': 'my_logging_name'}
    return [optimizer], [lr_scheduler]
```

**_extract_lr** (*trainer*, *interval*)
Extracts learning rates for lr schedulers and saves information into dict structure.

**_find_names** (*lr_schedulers*)

**on_batch_start** (*trainer*, *pl_module*)
Called when the training batch begins.

**on_epoch_start** (*trainer*, *pl_module*)
Called when the epoch begins.

**on_train_start** (*trainer*, *pl_module*)
Called before training, determines unique names for all lr schedulers in the case of multiple of the same type or in the case of multiple parameter groups

**class** pytorch_lightning.callbacks.**ProgressBarBase**
Bases: *pytorch_lightning.callbacks.base.Callback*

The base class for progress bars in Lightning. It is a *Callback* that keeps track of the batch progress in the *Trainer*. You should implement your highly custom progress bars with this as the base class.

Example:

```
class LitProgressBar(ProgressBarBase):

    def __init__(self):
        super().__init__()  # don't forget this :)
        self.enable = True

    def disable(self):
        self.enable = False

    def on_batch_end(self, trainer, pl_module):
        super().on_batch_end(trainer, pl_module)  # don't forget this :)
        percent = (self.train_batch_idx / self.total_train_batches) * 100
        sys.stdout.flush()
        sys.stdout.write(f'{percent:.01f} percent complete \r')

bar = LitProgressBar()
trainer = Trainer(callbacks=[bar])
```

**disable** ()
You should provide a way to disable the progress bar. The *Trainer* will call this to disable the output on processes that have a rank different from 0, e.g., in multi-node training.

**enable** ()
You should provide a way to enable the progress bar. The *Trainer* will call this in e.g. pre-training

routines like the learning rate finder to temporarily enable and disable the main progress bar.

**on_batch_end**(*trainer*, *pl_module*)
Called when the training batch ends.

**on_epoch_start**(*trainer*, *pl_module*)
Called when the epoch begins.

**on_init_end**(*trainer*)
Called when the trainer initialization ends, model has not yet been set.

**on_test_batch_end**(*trainer*, *pl_module*)
Called when the test batch ends.

**on_test_start**(*trainer*, *pl_module*)
Called when the test begins.

**on_train_start**(*trainer*, *pl_module*)
Called when the train begins.

**on_validation_batch_end**(*trainer*, *pl_module*)
Called when the validation batch ends.

**on_validation_start**(*trainer*, *pl_module*)
Called when the validation loop begins.

**property test_batch_idx**
The current batch index being processed during testing. Use this to update your progress bar.

**Return type** `int`

**property total_test_batches**
The total number of training batches during testing, which may change from epoch to epoch. Use this to set the total number of iterations in the progress bar. Can return `inf` if the test dataloader is of infinite size.

**Return type** `int`

**property total_train_batches**
The total number of training batches during training, which may change from epoch to epoch. Use this to set the total number of iterations in the progress bar. Can return `inf` if the training dataloader is of infinite size.

**Return type** `int`

**property total_val_batches**
The total number of training batches during validation, which may change from epoch to epoch. Use this to set the total number of iterations in the progress bar. Can return `inf` if the validation dataloader is of infinite size.

**Return type** `int`

**property train_batch_idx**
The current batch index being processed during training. Use this to update your progress bar.

**Return type** `int`

**property trainer**

**property val_batch_idx**
The current batch index being processed during validation. Use this to update your progress bar.

**Return type** `int`

**class** pytorch_lightning.callbacks.**ProgressBar**(*refresh_rate=1*, *process_position=0*)
Bases: *pytorch_lightning.callbacks.progress.ProgressBarBase*

This is the default progress bar used by Lightning. It prints to *stdout* using the `tqdm` package and shows up to four different bars:

- **sanity check progress:** the progress during the sanity check run
- **main progress:** shows training + validation progress combined. It also accounts for multiple validation runs during training when *val_check_interval* is used.
- **validation progress:** only visible during validation; shows total progress over all validation datasets.
- **test progress:** only active when testing; shows total progress over all test datasets.

For infinite datasets, the progress bar never ends.

If you want to customize the default `tqdm` progress bars used by Lightning, you can override specific methods of the callback class and pass your custom implementation to the *Trainer*:

Example:

```
class LitProgressBar(ProgressBar):

    def init_validation_tqdm(self):
        bar = super().init_validation_tqdm()
        bar.set_description('running validation ...')
        return bar

bar = LitProgressBar()
trainer = Trainer(callbacks=[bar])
```

**Parameters**

- **refresh_rate** (`int`) – Determines at which rate (in number of batches) the progress bars get updated. Set it to `0` to disable the display. By default, the *Trainer* uses this implementation of the progress bar and sets the refresh rate to the value provided to the *progress_bar_refresh_rate* argument in the *Trainer*.
- **process_position** (`int`) – Set this to a value greater than `0` to offset the progress bars by this many lines. This is useful when you have progress bars defined elsewhere and want to show all of them together. This corresponds to *process_position* in the *Trainer*.

**disable**()
You should provide a way to disable the progress bar. The *Trainer* will call this to disable the output on processes that have a rank different from 0, e.g., in multi-node training.

**Return type** `None`

**enable**()
You should provide a way to enable the progress bar. The *Trainer* will call this in e.g. pre-training routines like the learning rate finder to temporarily enable and disable the main progress bar.

**Return type** `None`

**init_sanity_tqdm**()
Override this to customize the tqdm bar for the validation sanity run.

**Return type** `tqdm`

**init_test_tqdm**()
Override this to customize the tqdm bar for testing.

**Return type** `tqdm`

**init_train_tqdm**()
Override this to customize the tqdm bar for training.

**Return type** `tqdm`

**init_validation_tqdm**()
Override this to customize the tqdm bar for validation.

**Return type** `tqdm`

**on_batch_end**(*trainer*, *pl_module*)
Called when the training batch ends.

**on_epoch_start**(*trainer*, *pl_module*)
Called when the epoch begins.

**on_sanity_check_end**(*trainer*, *pl_module*)
Called when the validation sanity check ends.

**on_sanity_check_start**(*trainer*, *pl_module*)
Called when the validation sanity check starts.

**on_test_batch_end**(*trainer*, *pl_module*)
Called when the test batch ends.

**on_test_end**(*trainer*, *pl_module*)
Called when the test ends.

**on_test_start**(*trainer*, *pl_module*)
Called when the test begins.

**on_train_end**(*trainer*, *pl_module*)
Called when the train ends.

**on_train_start**(*trainer*, *pl_module*)
Called when the train begins.

**on_validation_batch_end**(*trainer*, *pl_module*)
Called when the validation batch ends.

**on_validation_end**(*trainer*, *pl_module*)
Called when the validation loop ends.

**on_validation_start**(*trainer*, *pl_module*)
Called when the validation loop begins.

**property is_disabled**

**Return type** `bool`

**property is_enabled**

**Return type** `bool`

**property process_position**

**Return type** `int`

**property refresh_rate**

**Return type** `int`

### 37.2.1 Submodules

#### pytorch_lightning.callbacks.base module

#### Callback Base

Abstract base class used to build new callbacks.

**class** pytorch_lightning.callbacks.base.**Callback**
Bases: abc.ABC

Abstract base class used to build new callbacks.

**on_batch_end**(*trainer*, *pl_module*)
Called when the training batch ends.

**on_batch_start**(*trainer*, *pl_module*)
Called when the training batch begins.

**on_epoch_end**(*trainer*, *pl_module*)
Called when the epoch ends.

**on_epoch_start**(*trainer*, *pl_module*)
Called when the epoch begins.

**on_fit_end**(*trainer*)
Called when fit ends

**on_fit_start**(*trainer*)
Called when fit begins

**on_init_end**(*trainer*)
Called when the trainer initialization ends, model has not yet been set.

**on_init_start**(*trainer*)
Called when the trainer initialization begins, model has not yet been set.

**on_keyboard_interrupt**(*trainer*, *pl_module*)
Called when the training is interrupted by KeyboardInterrupt.

**on_sanity_check_end**(*trainer*, *pl_module*)
Called when the validation sanity check ends.

**on_sanity_check_start**(*trainer*, *pl_module*)
Called when the validation sanity check starts.

**on_test_batch_end**(*trainer*, *pl_module*)
Called when the test batch ends.

**on_test_batch_start**(*trainer*, *pl_module*)
Called when the test batch begins.

**on_test_end**(*trainer*, *pl_module*)
Called when the test ends.

**on_test_start**(*trainer*, *pl_module*)
Called when the test begins.

**on_train_end**(*trainer*, *pl_module*)
Called when the train ends.

**on_train_start**(*trainer*, *pl_module*)
Called when the train begins.

**on_validation_batch_end**(*trainer*, *pl_module*)
Called when the validation batch ends.

**on_validation_batch_start**(*trainer*, *pl_module*)
Called when the validation batch begins.

**on_validation_end**(*trainer*, *pl_module*)
Called when the validation loop ends.

**on_validation_start**(*trainer*, *pl_module*)
Called when the validation loop begins.

**setup**(*trainer*, *stage*)
Called when fit or test begins

**teardown**(*trainer*, *stage*)
Called when fit or test ends

## pytorch_lightning.callbacks.early_stopping module

### Early Stopping

Monitor a validation metric and stop training when it stops improving.

**class** pytorch_lightning.callbacks.early_stopping.**EarlyStopping**(*monitor='val_loss'*, *min_delta=0.0*, *patience=3*, *verbose=False*, *mode='auto'*, *strict=True*)

Bases: `pytorch_lightning.callbacks.base.Callback`

**Parameters**

- **monitor** (`str`) – quantity to be monitored. Default: `'val_loss'`.
- **min_delta** (`float`) – minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than *min_delta*, will count as no improvement. Default: `0`.
- **patience** (`int`) – number of validation epochs with no improvement after which training will be stopped. Default: `0`.
- **verbose** (`bool`) – verbosity mode. Default: `False`.
- **mode** (`str`) – one of {auto, min, max}. In *min* mode, training will stop when the quantity monitored has stopped decreasing; in *max* mode it will stop when the quantity monitored has stopped increasing; in *auto* mode, the direction is automatically inferred from the name of the monitored quantity. Default: `'auto'`.
- **strict** (`bool`) – whether to crash the training if *monitor* is not found in the validation metrics. Default: `True`.

Example:

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.callbacks import EarlyStopping
>>> early_stopping = EarlyStopping('val_loss')
>>> trainer = Trainer(early_stop_callback=early_stopping)
```

**_run_early_stopping_check**(*trainer*, *pl_module*)

**_validate_condition_metric**(*logs*)
Checks that the condition metric for early stopping is good :param _sphinx_paramlinks_pytorch_lightning.callbacks.early_stopping.EarlyStopping._validate_condition_metric.logs: :return:

**on_train_end**(*trainer*, *pl_module*)
Called when the train ends.

**on_train_start**(*trainer*, *pl_module*)
Called when the train begins.

**on_validation_end**(*trainer*, *pl_module*)
Called when the validation loop ends.

**mode_dict = {'max': torch.gt, 'min': torch.lt}**

**property monitor_op**

### pytorch_lightning.callbacks.gradient_accumulation_scheduler module

#### Gradient Accumulator

Change gradient accumulation factor according to scheduling.

**class** pytorch_lightning.callbacks.gradient_accumulation_scheduler.**GradientAccumulationSche**
Bases: *pytorch_lightning.callbacks.base.Callback*

Change gradient accumulation factor according to scheduling.

**Parameters** **scheduling** (dict) – scheduling in format {epoch: accumulation_factor}

> **Warning:** Epochs indexing starts from "1" until v0.6.x, but will start from "0" in v0.8.0.

Example:

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.callbacks import GradientAccumulationScheduler

# at epoch 5 start accumulating every 2 batches
>>> accumulator = GradientAccumulationScheduler(scheduling={5: 2})
>>> trainer = Trainer(callbacks=[accumulator])

# alternatively, pass the scheduling dict directly to the Trainer
>>> trainer = Trainer(accumulate_grad_batches={5: 2})
```

**on_epoch_start**(*trainer*, *pl_module*)
Called when the epoch begins.

### pytorch_lightning.callbacks.lr_logger module

#### Learning Rate Logger

Log learning rate for lr schedulers during training

**class** pytorch_lightning.callbacks.lr_logger.**LearningRateLogger**

Bases: *pytorch_lightning.callbacks.base.Callback*

Automatically logs learning rate for learning rate schedulers during training.

Example:

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.callbacks import LearningRateLogger
>>> lr_logger = LearningRateLogger()
>>> trainer = Trainer(callbacks=[lr_logger])
```

Logging names are automatically determined based on optimizer class name. In case of multiple optimizers of same type, they will be named *Adam*, *Adam-1* etc. If a optimizer has multiple parameter groups they will be named *Adam/pg1*, *Adam/pg2* etc. To control naming, pass in a *name* keyword in the construction of the learning rate schdulers

Example:

```
def configure_optimizer(self):
    optimizer = torch.optim.Adam(...)
    lr_scheduler = {'scheduler': torch.optim.lr_schedulers.LambdaLR(optimizer, ...
↪)
                    'name': 'my_logging_name'}
    return [optimizer], [lr_scheduler]
```

**_extract_lr**(*trainer*, *interval*)

Extracts learning rates for lr schedulers and saves information into dict structure.

**_find_names**(*lr_schedulers*)

**on_batch_start**(*trainer*, *pl_module*)

Called when the training batch begins.

**on_epoch_start**(*trainer*, *pl_module*)

Called when the epoch begins.

**on_train_start**(*trainer*, *pl_module*)

Called before training, determines unique names for all lr schedulers in the case of multiple of the same type or in the case of multiple parameter groups

### pytorch_lightning.callbacks.model_checkpoint module

#### Model Checkpointing

Automatically save model checkpoints during training.

**class** pytorch_lightning.callbacks.model_checkpoint.**ModelCheckpoint**(*filepath=None*, *monitor='val_loss'*, *verbose=False*, *save_last=False*, *save_top_k=1*, *save_weights_only=False*, *mode='auto'*, *period=1*, *prefix=''*)

Bases: *pytorch_lightning.callbacks.base.Callback*

Save the model after every epoch if it improves.

After training finishes, use `best_model_path` to retrieve the path to the best checkpoint file and `best_model_score` to retrieve its score.

**Parameters**

- **filepath** (`Optional[str]`) – path to save the model file. Can contain named formatting options to be auto-filled.

  Example:

  ```
  # custom path
  # saves a file like: my/path/epoch_0.ckpt
  >>> checkpoint_callback = ModelCheckpoint('my/path/')

  # save any arbitrary metrics like `val_loss`, etc. in name
  # saves a file like: my/path/epoch=2-val_loss=0.2_other_metric=0.3.
  ↪ckpt
  >>> checkpoint_callback = ModelCheckpoint(
  ...     filepath='my/path/{epoch}-{val_loss:.2f}-{other_metric:.2f}
  ↪'
  ... )
  ```

  Can also be set to *None*, then it will be set to default location during trainer construction.

- **monitor** (`str`) – quantity to monitor.
- **verbose** (`bool`) – verbosity mode. Default: `False`.
- **save_last** (`bool`) – always saves the model at the end of the epoch. Default: `False`.
- **save_top_k** (`int`) – if *save_top_k == k*, the best k models according to the quantity monitored will be saved. if `save_top_k == 0`, no models are saved. if `save_top_k == -1`, all models are saved. Please note that the monitors are checked every *period* epochs. if `save_top_k >= 2` and the callback is called multiple times inside an epoch, the name of the saved file will be appended with a version count starting with *v0*.
- **mode** (`str`) – one of {auto, min, max}. If `save_top_k != 0`, the decision to overwrite the current save file is made based on either the maximization or the minimization of the monitored quantity. For *val_acc*, this should be *max*, for *val_loss* this should be *min*, etc. In *auto* mode, the direction is automatically inferred from the name of the monitored quantity.
- **save_weights_only** (`bool`) – if `True`, then only the model's weights will be saved (`model.save_weights(filepath)`), else the full model is saved (`model.save(filepath)`).
- **period** (`int`) – Interval (number of epochs) between checkpoints.

Example:

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.callbacks import ModelCheckpoint

# saves checkpoints to 'my/path/' whenever 'val_loss' has a new min
>>> checkpoint_callback = ModelCheckpoint(filepath='my/path/')
>>> trainer = Trainer(checkpoint_callback=checkpoint_callback)

# save epoch and val_loss in name
# saves a file like: my/path/sample-mnist_epoch=02_val_loss=0.32.ckpt
>>> checkpoint_callback = ModelCheckpoint(
...     filepath='my/path/sample-mnist_{epoch:02d}-{val_loss:.2f}'
... )

# retrieve the best checkpoint after training
checkpoint_callback = ModelCheckpoint(filepath='my/path/')
trainer = Trainer(checkpoint_callback=checkpoint_callback)
model = ...
trainer.fit(model)
checkpoint_callback.best_model_path
```

**_del_model** (*filepath*)

**_do_check_save** (*filepath, current, epoch*)

**_save_model** (*filepath*)

**check_monitor_top_k** (*current*)

**format_checkpoint_name** (*epoch, metrics, ver=None*)
Generate a filename according to the defined template.

Example:

```
>>> tmpdir = os.path.dirname(__file__)
>>> ckpt = ModelCheckpoint(os.path.join(tmpdir, '{epoch}'))
>>> os.path.basename(ckpt.format_checkpoint_name(0, {}))
'epoch=0.ckpt'
>>> ckpt = ModelCheckpoint(os.path.join(tmpdir, '{epoch:03d}'))
>>> os.path.basename(ckpt.format_checkpoint_name(5, {}))
'epoch=005.ckpt'
>>> ckpt = ModelCheckpoint(os.path.join(tmpdir, '{epoch}-{val_loss:.2f}'))
>>> os.path.basename(ckpt.format_checkpoint_name(2, dict(val_loss=0.123456)))
'epoch=2-val_loss=0.12.ckpt'
>>> ckpt = ModelCheckpoint(os.path.join(tmpdir, '{missing:d}'))
>>> os.path.basename(ckpt.format_checkpoint_name(0, {}))
'missing=0.ckpt'
```

**on_validation_end** (*trainer, pl_module*)
Called when the validation loop ends.

**property best**

**property kth_best_model**

### pytorch_lightning.callbacks.progress module

#### Progress Bars

Use or override one of the progress bar callbacks.

**class** pytorch_lightning.callbacks.progress.**ProgressBar**(*refresh_rate=1*, *process_position=0*)

Bases: *pytorch_lightning.callbacks.progress.ProgressBarBase*

This is the default progress bar used by Lightning. It prints to *stdout* using the `tqdm` package and shows up to four different bars:

- **sanity check progress:** the progress during the sanity check run
- **main progress:** shows training + validation progress combined. It also accounts for multiple validation runs during training when *val_check_interval* is used.
- **validation progress:** only visible during validation; shows total progress over all validation datasets.
- **test progress:** only active when testing; shows total progress over all test datasets.

For infinite datasets, the progress bar never ends.

If you want to customize the default `tqdm` progress bars used by Lightning, you can override specific methods of the callback class and pass your custom implementation to the *Trainer*:

Example:

```
class LitProgressBar(ProgressBar):

    def init_validation_tqdm(self):
        bar = super().init_validation_tqdm()
        bar.set_description('running validation ...')
        return bar

bar = LitProgressBar()
trainer = Trainer(callbacks=[bar])
```

**Parameters**

- **refresh_rate** (`int`) – Determines at which rate (in number of batches) the progress bars get updated. Set it to `0` to disable the display. By default, the *Trainer* uses this implementation of the progress bar and sets the refresh rate to the value provided to the *progress_bar_refresh_rate* argument in the *Trainer*.
- **process_position** (`int`) – Set this to a value greater than `0` to offset the progress bars by this many lines. This is useful when you have progress bars defined elsewhere and want to show all of them together. This corresponds to *process_position* in the *Trainer*.

**disable**()

You should provide a way to disable the progress bar. The *Trainer* will call this to disable the output on processes that have a rank different from 0, e.g., in multi-node training.

**Return type** `None`

**enable**()

You should provide a way to enable the progress bar. The *Trainer* will call this in e.g. pre-training routines like the learning rate finder to temporarily enable and disable the main progress bar.

**Return type** `None`

**init_sanity_tqdm**()
Override this to customize the tqdm bar for the validation sanity run.

**Return type** tqdm

**init_test_tqdm**()
Override this to customize the tqdm bar for testing.

**Return type** tqdm

**init_train_tqdm**()
Override this to customize the tqdm bar for training.

**Return type** tqdm

**init_validation_tqdm**()
Override this to customize the tqdm bar for validation.

**Return type** tqdm

**on_batch_end**(*trainer*, *pl_module*)
Called when the training batch ends.

**on_epoch_start**(*trainer*, *pl_module*)
Called when the epoch begins.

**on_sanity_check_end**(*trainer*, *pl_module*)
Called when the validation sanity check ends.

**on_sanity_check_start**(*trainer*, *pl_module*)
Called when the validation sanity check starts.

**on_test_batch_end**(*trainer*, *pl_module*)
Called when the test batch ends.

**on_test_end**(*trainer*, *pl_module*)
Called when the test ends.

**on_test_start**(*trainer*, *pl_module*)
Called when the test begins.

**on_train_end**(*trainer*, *pl_module*)
Called when the train ends.

**on_train_start**(*trainer*, *pl_module*)
Called when the train begins.

**on_validation_batch_end**(*trainer*, *pl_module*)
Called when the validation batch ends.

**on_validation_end**(*trainer*, *pl_module*)
Called when the validation loop ends.

**on_validation_start**(*trainer*, *pl_module*)
Called when the validation loop begins.

**property is_disabled**

**Return type** bool

**property is_enabled**

**Return type** bool

**property process_position**

**Return type** int

**property refresh_rate**

**Return type** int

**class** pytorch_lightning.callbacks.progress.**ProgressBarBase**

Bases: *pytorch_lightning.callbacks.base.Callback*

The base class for progress bars in Lightning. It is a *Callback* that keeps track of the batch progress in the *Trainer*. You should implement your highly custom progress bars with this as the base class.

Example:

```
class LitProgressBar(ProgressBarBase):

    def __init__(self):
        super().__init__()  # don't forget this :)
        self.enable = True

    def disable(self):
        self.enable = False

    def on_batch_end(self, trainer, pl_module):
        super().on_batch_end(trainer, pl_module)  # don't forget this :)
        percent = (self.train_batch_idx / self.total_train_batches) * 100
        sys.stdout.flush()
        sys.stdout.write(f'{percent:.01f} percent complete \r')

bar = LitProgressBar()
trainer = Trainer(callbacks=[bar])
```

**disable**()

You should provide a way to disable the progress bar. The *Trainer* will call this to disable the output on processes that have a rank different from 0, e.g., in multi-node training.

**enable**()

You should provide a way to enable the progress bar. The *Trainer* will call this in e.g. pre-training routines like the learning rate finder to temporarily enable and disable the main progress bar.

**on_batch_end**(*trainer*, *pl_module*)

Called when the training batch ends.

**on_epoch_start**(*trainer*, *pl_module*)

Called when the epoch begins.

**on_init_end**(*trainer*)

Called when the trainer initialization ends, model has not yet been set.

**on_test_batch_end**(*trainer*, *pl_module*)

Called when the test batch ends.

**on_test_start**(*trainer*, *pl_module*)

Called when the test begins.

**on_train_start**(*trainer*, *pl_module*)

Called when the train begins.

**on_validation_batch_end**(*trainer*, *pl_module*)

Called when the validation batch ends.

**on_validation_start**(*trainer*, *pl_module*)

Called when the validation loop begins.

**property test_batch_idx**
The current batch index being processed during testing. Use this to update your progress bar.

**Return type** int

**property total_test_batches**
The total number of training batches during testing, which may change from epoch to epoch. Use this to set the total number of iterations in the progress bar. Can return `inf` if the test dataloader is of infinite size.

**Return type** int

**property total_train_batches**
The total number of training batches during training, which may change from epoch to epoch. Use this to set the total number of iterations in the progress bar. Can return `inf` if the training dataloader is of infinite size.

**Return type** int

**property total_val_batches**
The total number of training batches during validation, which may change from epoch to epoch. Use this to set the total number of iterations in the progress bar. Can return `inf` if the validation dataloader is of infinite size.

**Return type** int

**property train_batch_idx**
The current batch index being processed during training. Use this to update your progress bar.

**Return type** int

**property trainer**

**property val_batch_idx**
The current batch index being processed during validation. Use this to update your progress bar.

**Return type** int

pytorch_lightning.callbacks.progress.**convert_inf**(*x*)
The tqdm doesn't support inf values. We have to convert it to None.

## 37.3 pytorch_lightning.loggers package

**class** pytorch_lightning.loggers.**LightningLoggerBase**(*agg_key_funcs=None*, *agg_default_func=numpy.mean*)
Bases: abc.ABC

Base class for experiment loggers.

**Parameters**

- **agg_key_funcs** (Optional[Mapping[str, Callable[[Sequence[float]], float]]]) – Dictionary which maps a metric name to a function, which will aggregate the metric values for the same steps.
- **agg_default_func** (Callable[[Sequence[float]], float]) – Default function to aggregate metric values. If some metric name is not presented in the *agg_key_funcs* dictionary, then the *agg_default_func* will be used for aggregation.

**Note:** The *agg_key_funcs* and *agg_default_func* arguments are used only when one logs metrics with the `agg_and_log_metrics()` method.

**`_aggregate_metrics`**(*metrics*, *step=None*)
Aggregates metrics.

**Parameters**

- **`metrics`** (`Dict`[`str`, `float`]) – Dictionary with metric names as keys and measured quantities as values
- **`step`** (`Optional`[`int`]) – Step number at which the metrics should be recorded

**Return type** `Tuple`[`int`, `Optional`[`Dict`[`str`, `float`]]]

**Returns** Step and aggregated metrics. The return value could be `None`. In such case, metrics are added to the aggregation list, but not aggregated yet.

**`static _convert_params`**(*params*)

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

**`_finalize_agg_metrics`**()
This shall be called before save/close.

**`static _flatten_dict`**(*params*, *delimiter='/'*)
Flatten hierarchical dict, e.g. `{'a': {'b': 'c'}} -> {'a/b': 'c'}`.

**Parameters**

- **`params`** (`Dict`[`str`, `Any`]) – Dictionary containing the hyperparameters
- **`delimiter`** (`str`) – Delimiter to express the hierarchy. Defaults to `'/'`.

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

**Returns** Flattened dict.

### Examples

```
>>> LightningLoggerBase._flatten_dict({'a': {'b': 'c'}})
{'a/b': 'c'}
>>> LightningLoggerBase._flatten_dict({'a': {'b': 123}})
{'a/b': 123}
```

**`_reduce_agg_metrics`**()
Aggregate accumulated metrics.

**`static _sanitize_params`**(*params*)
Returns params with non-primitvies converted to strings for logging.

```
>>> params = {"float": 0.3,
...           "int": 1,
...           "string": "abc",
...           "bool": True,
...           "list": [1, 2, 3],
...           "namespace": Namespace(foo=3),
...           "layer": torch.nn.BatchNorm1d}
>>> import pprint
```

(continues on next page)

(continued from previous page)

```
>>> pprint.pprint(LightningLoggerBase._sanitize_params(params))
{'bool': True,
 'float': 0.3,
 'int': 1,
 'layer': "<class 'torch.nn.modules.batchnorm.BatchNorm1d'>",
 'list': '[1, 2, 3]',
 'namespace': 'Namespace(foo=3)',
 'string': 'abc'}
```

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

**agg_and_log_metrics**(*metrics*, *step=None*)
Aggregates and records metrics. This method doesn't log the passed metrics instantaneously, but instead it aggregates them and logs only if metrics are ready to be logged.

**Parameters**

- **metrics** (Dict[str, float]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded

**close**()
Do any cleanup that is necessary to close an experiment.

**Return type** None

**finalize**(*status*)
Do any processing that is necessary to finalize an experiment.

**Parameters** **status** (str) – Status that the experiment finished with (e.g. success, failed, aborted)

**Return type** None

**abstract log_hyperparams**(*params*)
Record hyperparameters.

**Parameters** **params** (Namespace) – Namespace containing the hyperparameters

**abstract log_metrics**(*metrics*, *step=None*)
Records metrics. This method logs metrics as as soon as it received them. If you want to aggregate metrics for one specific *step*, use the *agg_and_log_metrics()* method.

**Parameters**

- **metrics** (Dict[str, float]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded

**save**()
Save log data.

**Return type** None

**update_agg_funcs**(*agg_key_funcs=None*, *agg_default_func=numpy.mean*)
Update aggregation methods.

**Parameters**

- **agg_key_funcs** (Optional[Mapping[str, Callable[[Sequence[float]], float]]]) – Dictionary which maps a metric name to a function, which will aggregate the metric values for the same steps.
- **agg_default_func** (Callable[[Sequence[float]], float]) – Default function to aggregate metric values. If some metric name is not presented in the *agg_key_funcs* dictionary, then the *agg_default_func* will be used for aggregation.

**abstract property experiment**
Return the experiment object associated with this logger.

**Return type** Any

**abstract property name**
Return the experiment name.

**Return type** str

**abstract property version**
Return the experiment version.

**Return type** Union[int, str]

**class** pytorch_lightning.loggers.**LoggerCollection**(*logger_iterable*)
Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

The *LoggerCollection* class is used to iterate all logging actions over the given *logger_iterable*.

**Parameters** **logger_iterable** (Iterable[*LightningLoggerBase*]) – An iterable collection of loggers

**close**()
Do any cleanup that is necessary to close an experiment.

**Return type** None

**finalize**(*status*)
Do any processing that is necessary to finalize an experiment.

**Parameters** **status** (str) – Status that the experiment finished with (e.g. success, failed, aborted)

**Return type** None

**log_hyperparams**(*params*)
Record hyperparameters.

**Parameters** **params** (Union[Dict[str, Any], Namespace]) – Namespace containing the hyperparameters

**Return type** None

**log_metrics**(*metrics*, *step=None*)
Records metrics. This method logs metrics as as soon as it received them. If you want to aggregate metrics for one specific *step*, use the *agg_and_log_metrics()* method.

**Parameters**

- **metrics** (Dict[str, float]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded

**Return type** None

**save**()
Save log data.

**Return type** None

**property experiment**
Return the experiment object associated with this logger.

**Return type** List[Any]

**property name**
Return the experiment name.

**Return type** str

**property version**
Return the experiment version.

**Return type** str

**class** pytorch_lightning.loggers.**TensorBoardLogger**(*save_dir*, *name='default'*, *version=None*, ***kwargs*)
Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Log to local file system in TensorBoard format. Implemented using `SummaryWriter`. Logs are saved to `os.path.join(save_dir, name, version)`. This is the default logger in Lightning, it comes preinstalled.

### Example

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.loggers import TensorBoardLogger
>>> logger = TensorBoardLogger("tb_logs", name="my_model")
>>> trainer = Trainer(logger=logger)
```

**Parameters**

- **save_dir** (str) – Save directory
- **name** (Optional[str]) – Experiment name. Defaults to `'default'`. If it is the empty string then no per-experiment subdirectory is used.
- **version** (Union[int, str, None]) – Experiment version. If version is not specified the logger inspects the save directory for existing versions, then automatically assigns the next available version. If it is a string then it is used as the run-specific subdirectory name, otherwise `'version_${version}'` is used.
- ****kwargs** – Other arguments are passed directly to the `SummaryWriter` constructor.

**_get_next_version**()

**finalize**(*status*)
Do any processing that is necessary to finalize an experiment.

**Parameters status** (str) – Status that the experiment finished with (e.g. success, failed, aborted)

**Return type** None

**log_hyperparams**(*params*, *metrics=None*)
Record hyperparameters.

**Parameters** **params** (Union[Dict[str, Any], Namespace]) – Namespace containing the hyperparameters

**Return type** None

**log_metrics** (*metrics*, *step=None*)

Records metrics. This method logs metrics as as soon as it received them. If you want to aggregate metrics for one specific *step*, use the *agg_and_log_metrics()* method.

**Parameters**

- **metrics** (Dict[str, float]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded

**Return type** None

**save** ()

Save log data.

**Return type** None

**NAME_HPARAMS_FILE = 'hparams.yaml'**

**property experiment**

Actual tensorboard object. To use TensorBoard features in your *LightningModule* do the following.

Example:

```
self.logger.experiment.some_tensorboard_function()
```

**Return type** SummaryWriter

**property log_dir**

The directory for this run's tensorboard checkpoint. By default, it is named `'version_${self.version}'` but it can be overridden by passing a string value for the constructor's version parameter instead of `None` or an int.

**Return type** str

**property name**

Return the experiment name.

**Return type** str

**property root_dir**

Parent directory for all tensorboard checkpoint subdirectories. If the experiment name parameter is `None` or the empty string, no experiment subdirectory is used and the checkpoint will be saved in "save_dir/version_dir"

**Return type** str

**property version**

Return the experiment version.

**Return type** int

**class** pytorch_lightning.loggers.**CometLogger** (*api_key=None*, *save_dir=None*, *workspace=None*, *project_name=None*, *rest_api_key=None*, *experiment_name=None*, *experiment_key=None*, ***kwargs*)

Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Log using Comet.ml. Install it with pip:

```
pip install comet-ml
```

Comet requires either an API Key (online mode) or a local directory path (offline mode).

**ONLINE MODE**

### Example

```
>>> import os
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.loggers import CometLogger
>>> # arguments made to CometLogger are passed on to the comet_ml.Experiment class
>>> comet_logger = CometLogger(
...     api_key=os.environ.get('COMET_API_KEY'),
...     workspace=os.environ.get('COMET_WORKSPACE'),  # Optional
...     save_dir='.',  # Optional
...     project_name='default_project',  # Optional
...     rest_api_key=os.environ.get('COMET_REST_API_KEY'),  # Optional
...     experiment_name='default'  # Optional
... )
>>> trainer = Trainer(logger=comet_logger)
```

**OFFLINE MODE**

### Example

```
>>> from pytorch_lightning.loggers import CometLogger
>>> # arguments made to CometLogger are passed on to the comet_ml.Experiment class
>>> comet_logger = CometLogger(
...     save_dir='.',
...     workspace=os.environ.get('COMET_WORKSPACE'),  # Optional
...     project_name='default_project',  # Optional
...     rest_api_key=os.environ.get('COMET_REST_API_KEY'),  # Optional
...     experiment_name='default'  # Optional
... )
>>> trainer = Trainer(logger=comet_logger)
```

**Parameters**

- **api_key** (Optional[str]) – Required in online mode. API key, found on Comet.ml
- **save_dir** (Optional[str]) – Required in offline mode. The path for the directory to save local comet logs
- **workspace** (Optional[str]) – Optional. Name of workspace for this user
- **project_name** (Optional[str]) – Optional. Send your experiment to a specific project. Otherwise will be sent to Uncategorized Experiments. If the project name does not already exist, Comet.ml will create a new project.
- **rest_api_key** (Optional[str]) – Optional. Rest API key found in Comet.ml settings. This is used to determine version number

- **experiment_name** (`Optional`[`str`]) – Optional. String representing the name for this particular experiment on Comet.ml.
- **experiment_key** (`Optional`[`str`]) – Optional. If set, restores from existing experiment.

**finalize** (*status*)

When calling `self.experiment.end()`, that experiment won't log any more data to Comet. That's why, if you need to log any more data, you need to create an ExistingCometExperiment. For example, to log data when testing your model after training, because when training is finalized *`CometLogger.finalize()`* is called.

This happens automatically in the *`experiment()`* property, when `self._experiment` is set to `None`, i.e. `self.reset_experiment()`.

**Return type** `None`

**log_hyperparams** (*params*)

Record hyperparameters.

**Parameters params** (`Union`[`Dict`[`str`, `Any`], `Namespace`]) – `Namespace` containing the hyperparameters

**Return type** `None`

**log_metrics** (*metrics*, *step=None*)

Records metrics. This method logs metrics as as soon as it received them. If you want to aggregate metrics for one specific *step*, use the *`agg_and_log_metrics()`* method.

**Parameters**

- **metrics** (`Dict`[`str`, `Union`[`Tensor`, `float`]]) – Dictionary with metric names as keys and measured quantities as values
- **step** (`Optional`[`int`]) – Step number at which the metrics should be recorded

**Return type** `None`

**reset_experiment** ()

**property experiment**

Actual Comet object. To use Comet features in your *`LightningModule`* do the following.

Example:

```
self.logger.experiment.some_comet_function()
```

**Return type** `BaseExperiment`

**property name**

Return the experiment name.

**Return type** `str`

**property version**

Return the experiment version.

**Return type** `str`

**class** pytorch_lightning.loggers.**MLFlowLogger** (*experiment_name='default'*, *tracking_uri=None*, *tags=None*, *save_dir=None*)

Bases: *`pytorch_lightning.loggers.base.LightningLoggerBase`*

Log using MLflow. Install it with pip:

```
pip install mlflow
```

### Example

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.loggers import MLFlowLogger
>>> mlf_logger = MLFlowLogger(
...     experiment_name="default",
...     tracking_uri="file:./ml-runs"
... )
>>> trainer = Trainer(logger=mlf_logger)
```

Use the logger anywhere in you *LightningModule* as follows:

```
>>> from pytorch_lightning import LightningModule
>>> class LitModel(LightningModule):
...     def training_step(self, batch, batch_idx):
...         # example
...         self.logger.experiment.whatever_ml_flow_supports(...)
...
...     def any_lightning_module_function_or_hook(self):
...         self.logger.experiment.whatever_ml_flow_supports(...)
```

**Parameters**

- **experiment_name** (str) – The name of the experiment
- **tracking_uri** (Optional[str]) – Address of local or remote tracking server. If not provided, defaults to the service set by mlflow.tracking.set_tracking_uri.
- **tags** (Optional[Dict[str, Any]]) – A dictionary tags for the experiment.

**finalize**(*status='FINISHED'*)
Do any processing that is necessary to finalize an experiment.

**Parameters status** (str) – Status that the experiment finished with (e.g. success, failed, aborted)

**Return type** None

**log_hyperparams**(*params*)
Record hyperparameters.

**Parameters params** (Union[Dict[str, Any], Namespace]) – Namespace containing the hyperparameters

**Return type** None

**log_metrics**(*metrics*, *step=None*)
Records metrics. This method logs metrics as as soon as it received them. If you want to aggregate metrics for one specific *step*, use the *agg_and_log_metrics()* method.

**Parameters**

- **metrics** (Dict[str, float]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded

**Return type** None

**property experiment**

Actual MLflow object. To use mlflow features in your `LightningModule` do the following.

Example:

```
self.logger.experiment.some_mlflow_function()
```

**Return type** MlflowClient

**property name**

Return the experiment name.

**Return type** str

**property run_id**

**property version**

Return the experiment version.

**Return type** str

**class** pytorch_lightning.loggers.**NeptuneLogger**(*api_key=None*, *project_name=None*, *close_after_fit=True*, *offline_mode=False*, *experiment_name=None*, *upload_source_files=None*, *params=None*, *properties=None*, *tags=None*, ***kwargs*)

Bases: `pytorch_lightning.loggers.base.LightningLoggerBase`

Log using Neptune. Install it with pip:

```
pip install neptune-client
```

The Neptune logger can be used in the online mode or offline (silent) mode. To log experiment data in online mode, `NeptuneLogger` requires an API key. In offline mode, the logger does not connect to Neptune.

**ONLINE MODE**

## Example

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.loggers import NeptuneLogger
>>> # arguments made to NeptuneLogger are passed on to the neptune.experiments.
↪Experiment class
>>> # We are using an api_key for the anonymous user "neptuner" but you can use␣
↪your own.
>>> neptune_logger = NeptuneLogger(
...     api_key='ANONYMOUS',
...     project_name='shared/pytorch-lightning-integration',
...     experiment_name='default',  # Optional,
...     params={'max_epochs': 10},  # Optional,
...     tags=['pytorch-lightning', 'mlp']  # Optional,
... )
>>> trainer = Trainer(max_epochs=10, logger=neptune_logger)
```

**OFFLINE MODE**

### Example

```
>>> from pytorch_lightning.loggers import NeptuneLogger
>>> # arguments made to NeptuneLogger are passed on to the neptune.experiments.
↪Experiment class
>>> neptune_logger = NeptuneLogger(
...     offline_mode=True,
...     project_name='USER_NAME/PROJECT_NAME',
...     experiment_name='default',  # Optional,
...     params={'max_epochs': 10},  # Optional,
...     tags=['pytorch-lightning', 'mlp']  # Optional,
... )
>>> trainer = Trainer(max_epochs=10, logger=neptune_logger)
```

Use the logger anywhere in you `LightningModule` as follows:

```
>>> from pytorch_lightning import LightningModule
>>> class LitModel(LightningModule):
...     def training_step(self, batch, batch_idx):
...         # log metrics
...         self.logger.experiment.log_metric('acc_train', ...)
...         # log images
...         self.logger.experiment.log_image('worse_predictions', ...)
...         # log model checkpoint
...         self.logger.experiment.log_artifact('model_checkpoint.pt', ...)
...         self.logger.experiment.whatever_neptune_supports(...)
...
...     def any_lightning_module_function_or_hook(self):
...         self.logger.experiment.log_metric('acc_train', ...)
...         self.logger.experiment.log_image('worse_predictions', ...)
...         self.logger.experiment.log_artifact('model_checkpoint.pt', ...)
...         self.logger.experiment.whatever_neptune_supports(...)
```

If you want to log objects after the training is finished use `close_after_fit=False`:

```
neptune_logger = NeptuneLogger(
    ...
    close_after_fit=False,
    ...
)
trainer = Trainer(logger=neptune_logger)
trainer.fit()

# Log test metrics
trainer.test(model)

# Log additional metrics
from sklearn.metrics import accuracy_score

accuracy = accuracy_score(y_true, y_pred)
neptune_logger.experiment.log_metric('test_accuracy', accuracy)

# Log charts
from scikitplot.metrics import plot_confusion_matrix
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(16, 12))
```

(continues on next page)

(continued from previous page)

```
plot_confusion_matrix(y_true, y_pred, ax=ax)
neptune_logger.experiment.log_image('confusion_matrix', fig)

# Save checkpoints folder
neptune_logger.experiment.log_artifact('my/checkpoints')

# When you are done, stop the experiment
neptune_logger.experiment.stop()
```

**See also:**

- An Example experiment showing the UI of Neptune.
- Tutorial on how to use Pytorch Lightning with Neptune.

**Parameters**

- **api_key** (Optional[str]) – Required in online mode. Neptune API token, found on https://neptune.ai. Read how to get your API key. It is recommended to keep it in the *NEPTUNE_API_TOKEN* environment variable and then you can leave `api_key=None`.
- **project_name** (Optional[str]) – Required in online mode. Qualified name of a project in a form of "namespace/project_name" for example "tom/minst-classification". If `None`, the value of *NEPTUNE_PROJECT* environment variable will be taken. You need to create the project in https://neptune.ai first.
- **offline_mode** (bool) – Optional default `False`. If `True` no logs will be sent to Neptune. Usually used for debug purposes.
- **close_after_fit** (Optional[bool]) – Optional default `True`. If `False` the experiment will not be closed after training and additional metrics, images or artifacts can be logged. Also, remember to close the experiment explicitly by running `neptune_logger.experiment.stop()`.
- **experiment_name** (Optional[str]) – Optional. Editable name of the experiment. Name is displayed in the experiment's Details (Metadata section) and in experiments view as a column.
- **upload_source_files** (Optional[List[str]]) – Optional. List of source files to be uploaded. Must be list of str or single str. Uploaded sources are displayed in the experiment's Source code tab. If `None` is passed, the Python file from which the experiment was created will be uploaded. Pass an empty list (`[]`) to upload no files. Unix style pathname pattern expansion is supported. For example, you can pass `'\*.py'` to upload all python source files from the current directory. For recursion lookup use `'\**/\*.py'` (for Python 3.5 and later). For more information see `glob` library.
- **params** (Optional[Dict[str, Any]]) – Optional. Parameters of the experiment. After experiment creation params are read-only. Parameters are displayed in the experiment's Parameters section and each key-value pair can be viewed in the experiments view as a column.
- **properties** (Optional[Dict[str, Any]]) – Optional. Default is `{}`. Properties of the experiment. They are editable after the experiment is created. Properties are displayed in the experiment's Details section and each key-value pair can be viewed in the experiments view as a column.
- **tags** (Optional[List[str]]) – Optional. Default is `[]`. Must be list of str. Tags of the experiment. They are editable after the experiment is created (see: `append_tag()`

and `remove_tag()`). Tags are displayed in the experiment's Details section and can be viewed in the experiments view as a column.

**_create_or_get_experiment**()

**append_tags**(*tags*)
Appends tags to the neptune experiment.

**Parameters** **tags** (`Union`[`str`, `Iterable`[`str`]]) – Tags to add to the current experiment. If str is passed, a single tag is added. If multiple - comma separated - str are passed, all of them are added as tags. If list of str is passed, all elements of the list are added as tags.

**Return type** `None`

**finalize**(*status*)
Do any processing that is necessary to finalize an experiment.

**Parameters** **status** (`str`) – Status that the experiment finished with (e.g. success, failed, aborted)

**Return type** `None`

**log_artifact**(*artifact*, *destination=None*)
Save an artifact (file) in Neptune experiment storage.

**Parameters**

- **artifact** (`str`) – A path to the file in local filesystem.
- **destination** (`Optional`[`str`]) – Optional. Default is `None`. A destination path. If `None` is passed, an artifact file name will be used.

**Return type** `None`

**log_hyperparams**(*params*)
Record hyperparameters.

**Parameters** **params** (`Union`[`Dict`[`str`, `Any`], `Namespace`]) – `Namespace` containing the hyperparameters

**Return type** `None`

**log_image**(*log_name*, *image*, *step=None*)
Log image data in Neptune experiment

**Parameters**

- **log_name** (`str`) – The name of log, i.e. bboxes, visualisations, sample_images.
- **image** (`Union`[`str`, `Image`, `Any`]) – The value of the log (data-point). Can be one of the following types: PIL image, *matplotlib.figure.Figure*, path to image file (str)
- **step** (`Optional`[`int`]) – Step number at which the metrics should be recorded, must be strictly increasing

**Return type** `None`

**log_metric**(*metric_name*, *metric_value*, *step=None*)
Log metrics (numeric values) in Neptune experiments.

**Parameters**

- **metric_name** (`str`) – The name of log, i.e. mse, loss, accuracy.
- **metric_value** (`Union`[`Tensor`, `float`, `str`]) – The value of the log (data-point).

- **step** (Optional[int]) – Step number at which the metrics should be recorded, must be strictly increasing

**Return type** None

**log_metrics** (*metrics*, *step=None*)
Log metrics (numeric values) in Neptune experiments.

**Parameters**

- **metrics** (Dict[str, Union[Tensor, float]]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded, must be strictly increasing

**Return type** None

**log_text** (*log_name*, *text*, *step=None*)
Log text data in Neptune experiments.

**Parameters**

- **log_name** (str) – The name of log, i.e. mse, my_text_data, timing_info.
- **text** (str) – The value of the log (data-point).
- **step** (Optional[int]) – Step number at which the metrics should be recorded, must be strictly increasing

**Return type** None

**set_property** (*key*, *value*)
Set key-value pair as Neptune experiment property.

**Parameters**

- **key** (str) – Property key.
- **value** (Any) – New value of a property.

**Return type** None

**property experiment**
Actual Neptune object. To use neptune features in your *LightningModule* do the following.

Example:

```
self.logger.experiment.some_neptune_function()
```

**Return type** Experiment

**property name**
Return the experiment name.

**Return type** str

**property version**
Return the experiment version.

**Return type** str

**class** pytorch_lightning.loggers.**TestTubeLogger**(*save_dir*, *name='default'*, *description=None*, *debug=False*, *version=None*, *create_git_tag=False*)

Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Log to local file system in TensorBoard format but using a nicer folder structure (see full docs). Install it with pip:

```
pip install test_tube
```

### Example

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.loggers import TestTubeLogger
>>> logger = TestTubeLogger("tt_logs", name="my_exp_name")
>>> trainer = Trainer(logger=logger)
```

Use the logger anywhere in your *LightningModule* as follows:

```
>>> from pytorch_lightning import LightningModule
>>> class LitModel(LightningModule):
...     def training_step(self, batch, batch_idx):
...         # example
...         self.logger.experiment.whatever_method_summary_writer_supports(...)
...
...     def any_lightning_module_function_or_hook(self):
...         self.logger.experiment.add_histogram(...)
```

**Parameters**

- **save_dir** (str) – Save directory
- **name** (str) – Experiment name. Defaults to 'default'.
- **description** (Optional[str]) – A short snippet about this experiment
- **debug** (bool) – If True, it doesn't log anything.
- **version** (Optional[int]) – Experiment version. If version is not specified the logger inspects the save directory for existing versions, then automatically assigns the next available version.
- **create_git_tag** (bool) – If True creates a git tag to save the code used in this experiment.

**close**()

Do any cleanup that is necessary to close an experiment.

**Return type** None

**finalize**(*status*)

Do any processing that is necessary to finalize an experiment.

**Parameters** **status** (str) – Status that the experiment finished with (e.g. success, failed, aborted)

**Return type** None

**log_hyperparams**(*params*)

Record hyperparameters.

**Parameters** **params** (Union[Dict[str, Any], Namespace]) – Namespace containing the hyperparameters

**Return type** None

**log_metrics** (*metrics*, *step=None*)
Records metrics. This method logs metrics as as soon as it received them. If you want to aggregate metrics for one specific *step*, use the *agg_and_log_metrics()* method.

**Parameters**

- **metrics** (Dict[str, float]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded

**Return type** None

**save** ()
Save log data.

**Return type** None

**property experiment**
Actual TestTube object. To use TestTube features in your *LightningModule* do the following.

Example:

```
self.logger.experiment.some_test_tube_function()
```

**Return type** Experiment

**property name**
Return the experiment name.

**Return type** str

**property version**
Return the experiment version.

**Return type** int

**class** pytorch_lightning.loggers.**WandbLogger** (*name=None*, *save_dir=None*, *offline=False*, *id=None*, *anonymous=False*, *version=None*, *project=None*, *tags=None*, *log_model=False*, *experiment=None*, *entity=None*, *group=None*)
Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Log using Weights and Biases. Install it with pip:

```
pip install wandb
```

**Parameters**

- **name** (Optional[str]) – Display name for the run.
- **save_dir** (Optional[str]) – Path where data is saved.
- **offline** (bool) – Run offline (data can be streamed later to wandb servers).
- **id** (Optional[str]) – Sets the version, mainly used to resume a previous run.
- **anonymous** (bool) – Enables or explicitly disables anonymous logging.

- **version** (Optional[str]) – Sets the version, mainly used to resume a previous run.
- **project** (Optional[str]) – The name of the project to which this run will belong.
- **tags** (Optional[List[str]]) – Tags associated with this run.
- **log_model** (bool) – Save checkpoints in wandb dir to upload on W&B servers.
- **experiment** – WandB experiment object
- **entity** – The team posting this run (default: your username or your default team)
- **group** (Optional[str]) – A unique string shared by all runs in a given group

### Example

```
>>> from pytorch_lightning.loggers import WandbLogger
>>> from pytorch_lightning import Trainer
>>> wandb_logger = WandbLogger()
>>> trainer = Trainer(logger=wandb_logger)
```

**See also:**

- Tutorial on how to use W&B with Pytorch Lightning.

**log_hyperparams** (*params*)
Record hyperparameters.

**Parameters params** (Union[Dict[str, Any], Namespace]) – Namespace containing the hyperparameters

**Return type** None

**log_metrics** (*metrics*, *step=None*)
Records metrics. This method logs metrics as as soon as it received them. If you want to aggregate metrics for one specific *step*, use the *agg_and_log_metrics()* method.

**Parameters**

- **metrics** (Dict[str, float]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded

**Return type** None

**watch** (*model*, *log='gradients'*, *log_freq=100*)

**property experiment**
Actual wandb object. To use wandb features in your *LightningModule* do the following.

Example:

```
self.logger.experiment.some_wandb_function()
```

**Return type** Run

**property name**
Return the experiment name.

**Return type** str

**property version**
Return the experiment version.

**Return type** str

**class** pytorch_lightning.loggers.**TrainsLogger**(*project_name=None*, *task_name=None*, *task_type='training'*, *reuse_last_task_id=True*, *output_uri=None*, *auto_connect_arg_parser=True*, *auto_connect_frameworks=True*, *auto_resource_monitoring=True*)

Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Log using allegro.ai TRAINS. Install it with pip:

```
pip install trains
```

### Example

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.loggers import TrainsLogger
>>> trains_logger = TrainsLogger(
...     project_name='pytorch lightning',
...     task_name='default',
...     output_uri='.',
... )
TRAINS Task: ...
TRAINS results page: ...
>>> trainer = Trainer(logger=trains_logger)
```

Use the logger anywhere in your *LightningModule* as follows:

```
>>> from pytorch_lightning import LightningModule
>>> class LitModel(LightningModule):
...     def training_step(self, batch, batch_idx):
...         # example
...         self.logger.experiment.whatever_trains_supports(...)
...
...     def any_lightning_module_function_or_hook(self):
...         self.logger.experiment.whatever_trains_supports(...)
```

**Parameters**

- **project_name** (Optional[str]) – The name of the experiment's project. Defaults to None.
- **task_name** (Optional[str]) – The name of the experiment. Defaults to None.
- **task_type** (str) – The name of the experiment. Defaults to 'training'.
- **reuse_last_task_id** (bool) – Start with the previously used task id. Defaults to True.
- **output_uri** (Optional[str]) – Default location for output models. Defaults to None.
- **auto_connect_arg_parser** (bool) – Automatically grab the ArgumentParser and connect it with the task. Defaults to True.

- **auto_connect_frameworks** (bool) – If True, automatically patch to trains backend. Defaults to True.
- **auto_resource_monitoring** (bool) – If True, machine vitals will be sent along side the task scalars. Defaults to True.

### Examples

```
>>> logger = TrainsLogger("pytorch lightning", "default", output_uri=".")
TRAINS Task: ...
TRAINS results page: ...
>>> logger.log_metrics({"val_loss": 1.23}, step=0)
>>> logger.log_text("sample test")
sample test
>>> import numpy as np
>>> logger.log_artifact("confusion matrix", np.ones((2, 3)))
>>> logger.log_image("passed", "Image 1", np.random.randint(0, 255, (200, 150, 3),
↪ dtype=np.uint8))
```

**classmethod bypass_mode**()

Returns the bypass mode state.

**Note:** *GITHUB_ACTIONS* env will automatically set bypass_mode to True unless overridden specifically with TrainsLogger.set_bypass_mode(False).

**Return type** bool

**Returns** If True, all outside communication is skipped.

**finalize**(*status=None*)

Do any processing that is necessary to finalize an experiment.

**Parameters** **status** (Optional[str]) – Status that the experiment finished with (e.g. success, failed, aborted)

**Return type** None

**log_artifact**(*name*, *artifact*, *metadata=None*, *delete_after_upload=False*)

Save an artifact (file/object) in TRAINS experiment storage.

**Parameters**

- **name** (str) – Artifact name. Notice! it will override the previous artifact if the name already exists.
- **artifact** (Union[str, Path, Dict[str, Any], ndarray, Image]) – Artifact object to upload. Currently supports:
  - string / pathlib.Path are treated as path to artifact file to upload If a wildcard or a folder is passed, a zip file containing the local files will be created and uploaded.
  - dict will be stored as .json file and uploaded
  - pandas.DataFrame will be stored as .csv.gz (compressed CSV file) and uploaded
  - numpy.ndarray will be stored as .npz and uploaded
  - PIL.Image.Image will be stored to .png file and uploaded

- **metadata** (Optional[Dict[str, Any]]) – Simple key/value dictionary to store on the artifact. Defaults to None.
- **delete_after_upload** (bool) – If True, the local artifact will be deleted (only applies if artifact is a local file). Defaults to False.

**Return type** None

**log_hyperparams** (*params*)
Log hyperparameters (numeric values) in TRAINS experiments.

**Parameters** **params** (Union[Dict[str, Any], Namespace]) – The hyperparameters that passed through the model.

**Return type** None

**log_image** (*title*, *series*, *image*, *step=None*)
Log Debug image in TRAINS experiment

**Parameters**

- **title** (str) – The title of the debug image, i.e. "failed", "passed".
- **series** (str) – The series name of the debug image, i.e. "Image 0", "Image 1".
- **image** (Union[str, ndarray, Image, Tensor]) – Debug image to log. If numpy.ndarray or torch.Tensor, the image is assumed to be the following:
  - shape: CHW
  - color space: RGB
  - value range: [0., 1.] (float) or [0, 255] (uint8)
- **step** (Optional[int]) – Step number at which the metrics should be recorded. Defaults to None.

**Return type** None

**log_metric** (*title*, *series*, *value*, *step=None*)
Log metrics (numeric values) in TRAINS experiments. This method will be called by the users.

**Parameters**

- **title** (str) – The title of the graph to log, e.g. loss, accuracy.
- **series** (str) – The series name in the graph, e.g. classification, localization.
- **value** (float) – The value to log.
- **step** (Optional[int]) – Step number at which the metrics should be recorded. Defaults to None.

**Return type** None

**log_metrics** (*metrics*, *step=None*)
Log metrics (numeric values) in TRAINS experiments. This method will be called by Trainer.

**Parameters**

- **metrics** (Dict[str, float]) – The dictionary of the metrics. If the key contains "/", it will be split by the delimiter, then the elements will be logged as "title" and "series" respectively.
- **step** (Optional[int]) – Step number at which the metrics should be recorded. Defaults to None.

**Return type** None

**log_text**(*text*)
Log console text data in TRAINS experiment.

**Parameters** **text** (str) – The value of the log (data-point).

**Return type** None

**classmethod set_bypass_mode**(*bypass*)
Will bypass all outside communication, and will drop all logs. Should only be used in "standalone mode", when there is no access to the *trains-server*.

**Parameters** **bypass** (bool) – If True, all outside communication is skipped.

**Return type** None

**classmethod set_credentials**(*api_host=None*, *web_host=None*, *files_host=None*, *key=None*, *secret=None*)
Set new default TRAINS-server host and credentials. These configurations could be overridden by either OS environment variables or trains.conf configuration file.

**Note:** Credentials need to be set *prior* to Logger initialization.

**Parameters**

- **api_host** (Optional[str]) – Trains API server url, example: host='http://localhost:8008'
- **web_host** (Optional[str]) – Trains WEB server url, example: host='http://localhost:8080'
- **files_host** (Optional[str]) – Trains Files server url, example: host='http://localhost:8081'
- **key** (Optional[str]) – user key/secret pair, example: key='thisisakey123'
- **secret** (Optional[str]) – user key/secret pair, example: secret='thisisseceret123'

**Return type** None

**_bypass = None**

**property experiment**
Actual TRAINS object. To use TRAINS features in your *LightningModule* do the following.

Example:

```
self.logger.experiment.some_trains_function()
```

**Return type** Task

**property id**
ID is a uuid (string) representing this specific experiment in the entire system.

**Return type** Optional[str]

**property name**
Name is a human readable non-unique name (str) of the experiment.

**Return type** Optional[str]

**property version**
Return the experiment version.

**Return type** Optional[str]

### 37.3.1 Submodules

#### pytorch_lightning.loggers.base module

**class** pytorch_lightning.loggers.base.**DummyExperiment**
Bases: object

Dummy experiment

**nop**(***kw*)

**class** pytorch_lightning.loggers.base.**DummyLogger**
Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Dummy logger for internal use. Is usefull if we want to disable users logger for a feature, but still secure that users code can run

**log_hyperparams**(*params*)
Record hyperparameters.

**Parameters params** – Namespace containing the hyperparameters

**log_metrics**(*metrics, step*)
Records metrics. This method logs metrics as as soon as it received them. If you want to aggregate metrics for one specific *step*, use the *agg_and_log_metrics()* method.

**Parameters**

- **metrics** – Dictionary with metric names as keys and measured quantities as values
- **step** – Step number at which the metrics should be recorded

**property experiment**
Return the experiment object associated with this logger.

**property name**
Return the experiment name.

**property version**
Return the experiment version.

**class** pytorch_lightning.loggers.base.**LightningLoggerBase**(*agg_key_funcs=None, agg_default_func=numpy.mean*)
Bases: abc.ABC

Base class for experiment loggers.

**Parameters**

- **agg_key_funcs** (Optional[Mapping[str, Callable[[Sequence[float]], float]]]) – Dictionary which maps a metric name to a function, which will aggregate the metric values for the same steps.
- **agg_default_func** (Callable[[Sequence[float]], float]) – Default function to aggregate metric values. If some metric name is not presented in the *agg_key_funcs* dictionary, then the *agg_default_func* will be used for aggregation.

**Note:** The *agg_key_funcs* and *agg_default_func* arguments are used only when one logs metrics with the `agg_and_log_metrics()` method.

**`_aggregate_metrics`**(*metrics*, *step=None*)
Aggregates metrics.

**Parameters**

- **metrics** (`Dict`[`str`, `float`]) – Dictionary with metric names as keys and measured quantities as values
- **step** (`Optional`[`int`]) – Step number at which the metrics should be recorded

**Return type** `Tuple`[`int`, `Optional`[`Dict`[`str`, `float`]]]

**Returns** Step and aggregated metrics. The return value could be `None`. In such case, metrics are added to the aggregation list, but not aggregated yet.

**static `_convert_params`**(*params*)

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

**`_finalize_agg_metrics`**()
This shall be called before save/close.

**static `_flatten_dict`**(*params*, *delimiter='/'*)
Flatten hierarchical dict, e.g. `{'a': {'b': 'c'}} -> {'a/b': 'c'}`.

**Parameters**

- **params** (`Dict`[`str`, `Any`]) – Dictionary containing the hyperparameters
- **delimiter** (`str`) – Delimiter to express the hierarchy. Defaults to `'/'`.

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

**Returns** Flattened dict.

### Examples

```
>>> LightningLoggerBase._flatten_dict({'a': {'b': 'c'}})
{'a/b': 'c'}
>>> LightningLoggerBase._flatten_dict({'a': {'b': 123}})
{'a/b': 123}
```

**`_reduce_agg_metrics`**()
Aggregate accumulated metrics.

**static `_sanitize_params`**(*params*)
Returns params with non-primitvies converted to strings for logging.

```
>>> params = {"float": 0.3,
...           "int": 1,
...           "string": "abc",
...           "bool": True,
...           "list": [1, 2, 3],
...           "namespace": Namespace(foo=3),
...           "layer": torch.nn.BatchNorm1d}
>>> import pprint
```

(continues on next page)

(continued from previous page)

```
>>> pprint.pprint(LightningLoggerBase._sanitize_params(params))
{'bool': True,
 'float': 0.3,
 'int': 1,
 'layer': "<class 'torch.nn.modules.batchnorm.BatchNorm1d'>",
 'list': '[1, 2, 3]',
 'namespace': 'Namespace(foo=3)',
 'string': 'abc'}
```

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

**agg_and_log_metrics**(*metrics*, *step=None*)
Aggregates and records metrics. This method doesn't log the passed metrics instantaneously, but instead it aggregates them and logs only if metrics are ready to be logged.

**Parameters**

- **metrics** (Dict[str, float]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded

**close**()
Do any cleanup that is necessary to close an experiment.

**Return type** None

**finalize**(*status*)
Do any processing that is necessary to finalize an experiment.

**Parameters** **status** (str) – Status that the experiment finished with (e.g. success, failed, aborted)

**Return type** None

**abstract log_hyperparams**(*params*)
Record hyperparameters.

**Parameters** **params** (Namespace) – Namespace containing the hyperparameters

**abstract log_metrics**(*metrics*, *step=None*)
Records metrics. This method logs metrics as as soon as it received them. If you want to aggregate metrics for one specific *step*, use the *agg_and_log_metrics()* method.

**Parameters**

- **metrics** (Dict[str, float]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded

**save**()
Save log data.

**Return type** None

**update_agg_funcs**(*agg_key_funcs=None*, *agg_default_func=numpy.mean*)
Update aggregation methods.

**Parameters**

- **agg_key_funcs** (Optional[Mapping[str, Callable[[Sequence[float]], float]]]) – Dictionary which maps a metric name to a function, which will aggregate the metric values for the same steps.
- **agg_default_func** (Callable[[Sequence[float]], float]) – Default function to aggregate metric values. If some metric name is not presented in the *agg_key_funcs* dictionary, then the *agg_default_func* will be used for aggregation.

**abstract property experiment**
Return the experiment object associated with this logger.

**Return type** Any

**abstract property name**
Return the experiment name.

**Return type** str

**abstract property version**
Return the experiment version.

**Return type** Union[int, str]

**class** pytorch_lightning.loggers.base.**LoggerCollection**(*logger_iterable*)
Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

The *LoggerCollection* class is used to iterate all logging actions over the given *logger_iterable*.

**Parameters logger_iterable** (Iterable[*LightningLoggerBase*]) – An iterable collection of loggers

**close**()
Do any cleanup that is necessary to close an experiment.

**Return type** None

**finalize**(*status*)
Do any processing that is necessary to finalize an experiment.

**Parameters status** (str) – Status that the experiment finished with (e.g. success, failed, aborted)

**Return type** None

**log_hyperparams**(*params*)
Record hyperparameters.

**Parameters params** (Union[Dict[str, Any], Namespace]) – Namespace containing the hyperparameters

**Return type** None

**log_metrics**(*metrics*, *step=None*)
Records metrics. This method logs metrics as as soon as it received them. If you want to aggregate metrics for one specific *step*, use the *agg_and_log_metrics()* method.

**Parameters**

- **metrics** (Dict[str, float]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded

**Return type** None

**save**()
Save log data.

**Return type** None

**property experiment**
Return the experiment object associated with this logger.

**Return type** List[Any]

**property name**
Return the experiment name.

**Return type** str

**property version**
Return the experiment version.

**Return type** str

pytorch_lightning.loggers.base.**merge_dicts**(*dicts*, *agg_key_funcs=None*, *default_func=numpy.mean*)
Merge a sequence with dictionaries into one dictionary by aggregating the same keys with some given function.

**Parameters**

- **dicts** (Sequence[Mapping]) – Sequence of dictionaries to be merged.
- **agg_key_funcs** (Optional[Mapping[str, Callable[[Sequence[float]], float]]]) – Mapping from key name to function. This function will aggregate a list of values, obtained from the same key of all dictionaries. If some key has no specified aggregation function, the default one will be used. Default is: None (all keys will be aggregated by the default function).
- **default_func** (Callable[[Sequence[float]], float]) – Default function to aggregate keys, which are not presented in the *agg_key_funcs* map.

**Return type** Dict

**Returns** Dictionary with merged values.

### Examples

```
>>> import pprint
>>> d1 = {'a': 1.7, 'b': 2.0, 'c': 1, 'd': {'d1': 1, 'd3': 3}}
>>> d2 = {'a': 1.1, 'b': 2.2, 'v': 1, 'd': {'d1': 2, 'd2': 3}}
>>> d3 = {'a': 1.1, 'v': 2.3, 'd': {'d3': 3, 'd4': {'d5': 1}}}
>>> dflt_func = min
>>> agg_funcs = {'a': np.mean, 'v': max, 'd': {'d1': sum}}
>>> pprint.pprint(merge_dicts([d1, d2, d3], agg_funcs, dflt_func))
{'a': 1.3,
 'b': 2.0,
 'c': 1,
 'd': {'d1': 3, 'd2': 3, 'd3': 3, 'd4': {'d5': 1}},
 'v': 2.3}
```

### pytorch_lightning.loggers.comet module

#### Comet

**class** pytorch_lightning.loggers.comet.**CometLogger**(*api_key=None*, *save_dir=None*, *workspace=None*, *project_name=None*, *rest_api_key=None*, *experiment_name=None*, *experiment_key=None*, ***kwargs*)

Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Log using Comet.ml. Install it with pip:

```
pip install comet-ml
```

Comet requires either an API Key (online mode) or a local directory path (offline mode).

**ONLINE MODE**

#### Example

```
>>> import os
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.loggers import CometLogger
>>> # arguments made to CometLogger are passed on to the comet_ml.Experiment class
>>> comet_logger = CometLogger(
...     api_key=os.environ.get('COMET_API_KEY'),
...     workspace=os.environ.get('COMET_WORKSPACE'),  # Optional
...     save_dir='.',  # Optional
...     project_name='default_project',  # Optional
...     rest_api_key=os.environ.get('COMET_REST_API_KEY'),  # Optional
...     experiment_name='default'  # Optional
... )
>>> trainer = Trainer(logger=comet_logger)
```

**OFFLINE MODE**

#### Example

```
>>> from pytorch_lightning.loggers import CometLogger
>>> # arguments made to CometLogger are passed on to the comet_ml.Experiment class
>>> comet_logger = CometLogger(
...     save_dir='.',
...     workspace=os.environ.get('COMET_WORKSPACE'),  # Optional
...     project_name='default_project',  # Optional
...     rest_api_key=os.environ.get('COMET_REST_API_KEY'),  # Optional
...     experiment_name='default'  # Optional
... )
>>> trainer = Trainer(logger=comet_logger)
```

**Parameters**

- **api_key** (Optional[str]) – Required in online mode. API key, found on Comet.ml

- **save_dir** (Optional[str]) – Required in offline mode. The path for the directory to save local comet logs
- **workspace** (Optional[str]) – Optional. Name of workspace for this user
- **project_name** (Optional[str]) – Optional. Send your experiment to a specific project. Otherwise will be sent to Uncategorized Experiments. If the project name does not already exist, Comet.ml will create a new project.
- **rest_api_key** (Optional[str]) – Optional. Rest API key found in Comet.ml settings. This is used to determine version number
- **experiment_name** (Optional[str]) – Optional. String representing the name for this particular experiment on Comet.ml.
- **experiment_key** (Optional[str]) – Optional. If set, restores from existing experiment.

**finalize** (*status*)

When calling `self.experiment.end()`, that experiment won't log any more data to Comet. That's why, if you need to log any more data, you need to create an ExistingCometExperiment. For example, to log data when testing your model after training, because when training is finalized *CometLogger.finalize()* is called.

This happens automatically in the *experiment()* property, when `self._experiment` is set to `None`, i.e. `self.reset_experiment()`.

**Return type** None

**log_hyperparams** (*params*)

Record hyperparameters.

**Parameters params** (Union[Dict[str, Any], Namespace]) – Namespace containing the hyperparameters

**Return type** None

**log_metrics** (*metrics*, *step=None*)

Records metrics. This method logs metrics as as soon as it received them. If you want to aggregate metrics for one specific *step*, use the *agg_and_log_metrics()* method.

**Parameters**

- **metrics** (Dict[str, Union[Tensor, float]]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded

**Return type** None

**reset_experiment** ()

**property experiment**

Actual Comet object. To use Comet features in your *LightningModule* do the following.

Example:

```
self.logger.experiment.some_comet_function()
```

**Return type** BaseExperiment

**property name**

Return the experiment name.

**Return type** str

**property version**
Return the experiment version.

**Return type** str

### pytorch_lightning.loggers.mlflow module

### MLflow

**class** pytorch_lightning.loggers.mlflow.**MLFlowLogger**(*experiment_name='default'*, *tracking_uri=None*, *tags=None*, *save_dir=None*)
Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Log using MLflow. Install it with pip:

```
pip install mlflow
```

#### Example

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.loggers import MLFlowLogger
>>> mlf_logger = MLFlowLogger(
...     experiment_name="default",
...     tracking_uri="file:./ml-runs"
... )
>>> trainer = Trainer(logger=mlf_logger)
```

Use the logger anywhere in you *LightningModule* as follows:

```
>>> from pytorch_lightning import LightningModule
>>> class LitModel(LightningModule):
...     def training_step(self, batch, batch_idx):
...         # example
...         self.logger.experiment.whatever_ml_flow_supports(...)
...
...     def any_lightning_module_function_or_hook(self):
...         self.logger.experiment.whatever_ml_flow_supports(...)
```

**Parameters**

- **experiment_name** (str) – The name of the experiment
- **tracking_uri** (Optional[str]) – Address of local or remote tracking server. If not provided, defaults to the service set by mlflow.tracking.set_tracking_uri.
- **tags** (Optional[Dict[str, Any]]) – A dictionary tags for the experiment.

**finalize**(*status='FINISHED'*)
Do any processing that is necessary to finalize an experiment.

**Parameters status** (str) – Status that the experiment finished with (e.g. success, failed, aborted)

**Return type** None

**log_hyperparams** (*params*)
Record hyperparameters.

**Parameters** **params** (Union[Dict[str, Any], Namespace]) – Namespace containing the hyperparameters

**Return type** None

**log_metrics** (*metrics*, *step=None*)
Records metrics. This method logs metrics as as soon as it received them. If you want to aggregate metrics for one specific *step*, use the agg_and_log_metrics() method.

**Parameters**

- **metrics** (Dict[str, float]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded

**Return type** None

**property experiment**
Actual MLflow object. To use mlflow features in your LightningModule do the following.

Example:

```
self.logger.experiment.some_mlflow_function()
```

**Return type** MlflowClient

**property name**
Return the experiment name.

**Return type** str

**property run_id**

**property version**
Return the experiment version.

**Return type** str

## pytorch_lightning.loggers.neptune module

### Neptune

**class** pytorch_lightning.loggers.neptune.**NeptuneLogger** (*api_key=None*, *project_name=None*, *close_after_fit=True*, *offline_mode=False*, *experiment_name=None*, *upload_source_files=None*, *params=None*, *properties=None*, *tags=None*, ***kwargs*)

Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Log using Neptune. Install it with pip:

```
pip install neptune-client
```

The Neptune logger can be used in the online mode or offline (silent) mode. To log experiment data in online mode, `NeptuneLogger` requires an API key. In offline mode, the logger does not connect to Neptune.

**ONLINE MODE**

### Example

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.loggers import NeptuneLogger
>>> # arguments made to NeptuneLogger are passed on to the neptune.experiments.
↪Experiment class
>>> # We are using an api_key for the anonymous user "neptuner" but you can use␣
↪your own.
>>> neptune_logger = NeptuneLogger(
...     api_key='ANONYMOUS',
...     project_name='shared/pytorch-lightning-integration',
...     experiment_name='default',  # Optional,
...     params={'max_epochs': 10},  # Optional,
...     tags=['pytorch-lightning', 'mlp']  # Optional,
... )
>>> trainer = Trainer(max_epochs=10, logger=neptune_logger)
```

**OFFLINE MODE**

### Example

```
>>> from pytorch_lightning.loggers import NeptuneLogger
>>> # arguments made to NeptuneLogger are passed on to the neptune.experiments.
↪Experiment class
>>> neptune_logger = NeptuneLogger(
...     offline_mode=True,
...     project_name='USER_NAME/PROJECT_NAME',
...     experiment_name='default',  # Optional,
...     params={'max_epochs': 10},  # Optional,
...     tags=['pytorch-lightning', 'mlp']  # Optional,
... )
>>> trainer = Trainer(max_epochs=10, logger=neptune_logger)
```

Use the logger anywhere in you `LightningModule` as follows:

```
>>> from pytorch_lightning import LightningModule
>>> class LitModel(LightningModule):
...     def training_step(self, batch, batch_idx):
...         # log metrics
...         self.logger.experiment.log_metric('acc_train', ...)
...         # log images
...         self.logger.experiment.log_image('worse_predictions', ...)
...         # log model checkpoint
...         self.logger.experiment.log_artifact('model_checkpoint.pt', ...)
...         self.logger.experiment.whatever_neptune_supports(...)
...
...     def any_lightning_module_function_or_hook(self):
```

(continues on next page)

(continued from previous page)

```
...         self.logger.experiment.log_metric('acc_train', ...)
...         self.logger.experiment.log_image('worse_predictions', ...)
...         self.logger.experiment.log_artifact('model_checkpoint.pt', ...)
...         self.logger.experiment.whatever_neptune_supports(...)
```

If you want to log objects after the training is finished use `close_after_fit=False`:

```
neptune_logger = NeptuneLogger(
    ...
    close_after_fit=False,
    ...
)
trainer = Trainer(logger=neptune_logger)
trainer.fit()

# Log test metrics
trainer.test(model)

# Log additional metrics
from sklearn.metrics import accuracy_score

accuracy = accuracy_score(y_true, y_pred)
neptune_logger.experiment.log_metric('test_accuracy', accuracy)

# Log charts
from scikitplot.metrics import plot_confusion_matrix
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(16, 12))
plot_confusion_matrix(y_true, y_pred, ax=ax)
neptune_logger.experiment.log_image('confusion_matrix', fig)

# Save checkpoints folder
neptune_logger.experiment.log_artifact('my/checkpoints')

# When you are done, stop the experiment
neptune_logger.experiment.stop()
```

**See also:**

- An Example experiment showing the UI of Neptune.
- Tutorial on how to use Pytorch Lightning with Neptune.

**Parameters**

- **api_key** (`Optional[str]`) – Required in online mode. Neptune API token, found on https://neptune.ai. Read how to get your API key. It is recommended to keep it in the *NEPTUNE_API_TOKEN* environment variable and then you can leave `api_key=None`.
- **project_name** (`Optional[str]`) – Required in online mode. Qualified name of a project in a form of "namespace/project_name" for example "tom/minst-classification". If `None`, the value of *NEPTUNE_PROJECT* environment variable will be taken. You need to create the project in https://neptune.ai first.
- **offline_mode** (`bool`) – Optional default `False`. If `True` no logs will be sent to Neptune. Usually used for debug purposes.

- **close_after_fit** (Optional[bool]) – Optional default `True`. If `False` the experiment will not be closed after training and additional metrics, images or artifacts can be logged. Also, remember to close the experiment explicitly by running `neptune_logger.experiment.stop()`.
- **experiment_name** (Optional[str]) – Optional. Editable name of the experiment. Name is displayed in the experiment's Details (Metadata section) and in experiments view as a column.
- **upload_source_files** (Optional[List[str]]) – Optional. List of source files to be uploaded. Must be list of str or single str. Uploaded sources are displayed in the experiment's Source code tab. If `None` is passed, the Python file from which the experiment was created will be uploaded. Pass an empty list (`[]`) to upload no files. Unix style pathname pattern expansion is supported. For example, you can pass `'\*.py'` to upload all python source files from the current directory. For recursion lookup use `'\**/\*.py'` (for Python 3.5 and later). For more information see `glob` library.
- **params** (Optional[Dict[str, Any]]) – Optional. Parameters of the experiment. After experiment creation params are read-only. Parameters are displayed in the experiment's Parameters section and each key-value pair can be viewed in the experiments view as a column.
- **properties** (Optional[Dict[str, Any]]) – Optional. Default is `{}`. Properties of the experiment. They are editable after the experiment is created. Properties are displayed in the experiment's Details section and each key-value pair can be viewed in the experiments view as a column.
- **tags** (Optional[List[str]]) – Optional. Default is `[]`. Must be list of str. Tags of the experiment. They are editable after the experiment is created (see: `append_tag()` and `remove_tag()`). Tags are displayed in the experiment's Details section and can be viewed in the experiments view as a column.

**_create_or_get_experiment**()

**append_tags**(*tags*)

Appends tags to the neptune experiment.

**Parameters** **tags** (Union[str, Iterable[str]]) – Tags to add to the current experiment. If str is passed, a single tag is added. If multiple - comma separated - str are passed, all of them are added as tags. If list of str is passed, all elements of the list are added as tags.

**Return type** None

**finalize**(*status*)

Do any processing that is necessary to finalize an experiment.

**Parameters** **status** (str) – Status that the experiment finished with (e.g. success, failed, aborted)

**Return type** None

**log_artifact**(*artifact*, *destination=None*)

Save an artifact (file) in Neptune experiment storage.

**Parameters**

- **artifact** (str) – A path to the file in local filesystem.
- **destination** (Optional[str]) – Optional. Default is `None`. A destination path. If `None` is passed, an artifact file name will be used.

**Return type** None

**log_hyperparams** (*params*)
Record hyperparameters.

**Parameters** **params** (Union[Dict[str, Any], Namespace]) – Namespace containing the hyperparameters

**Return type** None

**log_image** (*log_name*, *image*, *step=None*)
Log image data in Neptune experiment

**Parameters**

- **log_name** (str) – The name of log, i.e. bboxes, visualisations, sample_images.
- **image** (Union[str, Image, Any]) – The value of the log (data-point). Can be one of the following types: PIL image, *matplotlib.figure.Figure*, path to image file (str)
- **step** (Optional[int]) – Step number at which the metrics should be recorded, must be strictly increasing

**Return type** None

**log_metric** (*metric_name*, *metric_value*, *step=None*)
Log metrics (numeric values) in Neptune experiments.

**Parameters**

- **metric_name** (str) – The name of log, i.e. mse, loss, accuracy.
- **metric_value** (Union[Tensor, float, str]) – The value of the log (data-point).
- **step** (Optional[int]) – Step number at which the metrics should be recorded, must be strictly increasing

**Return type** None

**log_metrics** (*metrics*, *step=None*)
Log metrics (numeric values) in Neptune experiments.

**Parameters**

- **metrics** (Dict[str, Union[Tensor, float]]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded, must be strictly increasing

**Return type** None

**log_text** (*log_name*, *text*, *step=None*)
Log text data in Neptune experiments.

**Parameters**

- **log_name** (str) – The name of log, i.e. mse, my_text_data, timing_info.
- **text** (str) – The value of the log (data-point).
- **step** (Optional[int]) – Step number at which the metrics should be recorded, must be strictly increasing

**Return type** None

**set_property** (*key*, *value*)
Set key-value pair as Neptune experiment property.

**Parameters**

- **key** (str) – Property key.
- **value** (Any) – New value of a property.

**Return type** None

**property experiment**

Actual Neptune object. To use neptune features in your *LightningModule* do the following.

Example:

```
self.logger.experiment.some_neptune_function()
```

**Return type** Experiment

**property name**

Return the experiment name.

**Return type** str

**property version**

Return the experiment version.

**Return type** str

## pytorch_lightning.loggers.tensorboard module

### TensorBoard

**class** pytorch_lightning.loggers.tensorboard.**TensorBoardLogger**(*save_dir*, *name='default'*, *version=None*, ***kwargs*)

Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Log to local file system in TensorBoard format. Implemented using SummaryWriter. Logs are saved to os.path.join(save_dir, name, version). This is the default logger in Lightning, it comes preinstalled.

#### Example

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.loggers import TensorBoardLogger
>>> logger = TensorBoardLogger("tb_logs", name="my_model")
>>> trainer = Trainer(logger=logger)
```

**Parameters**

- **save_dir** (str) – Save directory
- **name** (Optional[str]) – Experiment name. Defaults to 'default'. If it is the empty string then no per-experiment subdirectory is used.

- **version** (Union[int, str, None]) – Experiment version. If version is not specified the logger inspects the save directory for existing versions, then automatically assigns the next available version. If it is a string then it is used as the run-specific subdirectory name, otherwise `'version_${version}'` is used.
- ****kwargs** – Other arguments are passed directly to the `SummaryWriter` constructor.

**_get_next_version**()

**finalize**(*status*)
Do any processing that is necessary to finalize an experiment.

**Parameters** **status** (str) – Status that the experiment finished with (e.g. success, failed, aborted)

**Return type** None

**log_hyperparams**(*params*, *metrics=None*)
Record hyperparameters.

**Parameters** **params** (Union[Dict[str, Any], Namespace]) – Namespace containing the hyperparameters

**Return type** None

**log_metrics**(*metrics*, *step=None*)
Records metrics. This method logs metrics as as soon as it received them. If you want to aggregate metrics for one specific *step*, use the *agg_and_log_metrics()* method.

**Parameters**

- **metrics** (Dict[str, float]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded

**Return type** None

**save**()
Save log data.

**Return type** None

**NAME_HPARAMS_FILE = 'hparams.yaml'**

**property experiment**
Actual tensorboard object. To use TensorBoard features in your *LightningModule* do the following.

Example:

```
self.logger.experiment.some_tensorboard_function()
```

**Return type** SummaryWriter

**property log_dir**
The directory for this run's tensorboard checkpoint. By default, it is named `'version_${self.version}'` but it can be overridden by passing a string value for the constructor's version parameter instead of `None` or an int.

**Return type** str

**property name**
Return the experiment name.

**Return type** `str`

**property root_dir**
Parent directory for all tensorboard checkpoint subdirectories. If the experiment name parameter is `None` or the empty string, no experiment subdirectory is used and the checkpoint will be saved in "save_dir/version_dir"

**Return type** `str`

**property version**
Return the experiment version.

**Return type** `int`

### pytorch_lightning.loggers.test_tube module

### Test Tube

**class** pytorch_lightning.loggers.test_tube.**TestTubeLogger**(*save_dir, name='default', description=None, debug=False, version=None, create_git_tag=False*)

Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Log to local file system in TensorBoard format but using a nicer folder structure (see full docs). Install it with pip:

```
pip install test_tube
```

#### Example

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.loggers import TestTubeLogger
>>> logger = TestTubeLogger("tt_logs", name="my_exp_name")
>>> trainer = Trainer(logger=logger)
```

Use the logger anywhere in your *LightningModule* as follows:

```
>>> from pytorch_lightning import LightningModule
>>> class LitModel(LightningModule):
...     def training_step(self, batch, batch_idx):
...         # example
...         self.logger.experiment.whatever_method_summary_writer_supports(...)
...
...     def any_lightning_module_function_or_hook(self):
...         self.logger.experiment.add_histogram(...)
```

**Parameters**

- **save_dir** (`str`) – Save directory
- **name** (`str`) – Experiment name. Defaults to `'default'`.
- **description** (`Optional[str]`) – A short snippet about this experiment
- **debug** (`bool`) – If `True`, it doesn't log anything.

- **version** (Optional[int]) – Experiment version. If version is not specified the logger inspects the save directory for existing versions, then automatically assigns the next available version.
- **create_git_tag** (bool) – If True creates a git tag to save the code used in this experiment.

**close**()
Do any cleanup that is necessary to close an experiment.

**Return type** None

**finalize**(*status*)
Do any processing that is necessary to finalize an experiment.

**Parameters** **status** (str) – Status that the experiment finished with (e.g. success, failed, aborted)

**Return type** None

**log_hyperparams**(*params*)
Record hyperparameters.

**Parameters** **params** (Union[Dict[str, Any], Namespace]) – Namespace containing the hyperparameters

**Return type** None

**log_metrics**(*metrics*, *step=None*)
Records metrics. This method logs metrics as as soon as it received them. If you want to aggregate metrics for one specific *step*, use the *agg_and_log_metrics()* method.

**Parameters**

- **metrics** (Dict[str, float]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded

**Return type** None

**save**()
Save log data.

**Return type** None

**property experiment**
Actual TestTube object. To use TestTube features in your *LightningModule* do the following.

Example:

```
self.logger.experiment.some_test_tube_function()
```

**Return type** Experiment

**property name**
Return the experiment name.

**Return type** str

**property version**
Return the experiment version.

**Return type** int

### pytorch_lightning.loggers.trains module

#### TRAINS

**class** pytorch_lightning.loggers.trains.**TrainsLogger**(*project_name=None, task_name=None, task_type='training', reuse_last_task_id=True, output_uri=None, auto_connect_arg_parser=True, auto_connect_frameworks=True, auto_resource_monitoring=True*)

Bases: `pytorch_lightning.loggers.base.LightningLoggerBase`

Log using allegro.ai TRAINS. Install it with pip:

```
pip install trains
```

#### Example

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.loggers import TrainsLogger
>>> trains_logger = TrainsLogger(
...     project_name='pytorch lightning',
...     task_name='default',
...     output_uri='.',
... )
TRAINS Task: ...
TRAINS results page: ...
>>> trainer = Trainer(logger=trains_logger)
```

Use the logger anywhere in your `LightningModule` as follows:

```
>>> from pytorch_lightning import LightningModule
>>> class LitModel(LightningModule):
...     def training_step(self, batch, batch_idx):
...         # example
...         self.logger.experiment.whatever_trains_supports(...)
...
...     def any_lightning_module_function_or_hook(self):
...         self.logger.experiment.whatever_trains_supports(...)
```

**Parameters**

- **project_name** (`Optional[str]`) – The name of the experiment's project. Defaults to `None`.
- **task_name** (`Optional[str]`) – The name of the experiment. Defaults to `None`.
- **task_type** (`str`) – The name of the experiment. Defaults to `'training'`.
- **reuse_last_task_id** (`bool`) – Start with the previously used task id. Defaults to `True`.
- **output_uri** (`Optional[str]`) – Default location for output models. Defaults to `None`.

- **auto_connect_arg_parser** (bool) – Automatically grab the ArgumentParser and connect it with the task. Defaults to True.
- **auto_connect_frameworks** (bool) – If True, automatically patch to trains backend. Defaults to True.
- **auto_resource_monitoring** (bool) – If True, machine vitals will be sent along side the task scalars. Defaults to True.

### Examples

```
>>> logger = TrainsLogger("pytorch lightning", "default", output_uri=".")
TRAINS Task: ...
TRAINS results page: ...
>>> logger.log_metrics({"val_loss": 1.23}, step=0)
>>> logger.log_text("sample test")
sample test
>>> import numpy as np
>>> logger.log_artifact("confusion matrix", np.ones((2, 3)))
>>> logger.log_image("passed", "Image 1", np.random.randint(0, 255, (200, 150, 3),
↪ dtype=np.uint8))
```

**classmethod bypass_mode**()

Returns the bypass mode state.

**Note:** *GITHUB_ACTIONS* env will automatically set bypass_mode to True unless overridden specifically with TrainsLogger.set_bypass_mode(False).

**Return type** bool

**Returns** If True, all outside communication is skipped.

**finalize**(*status=None*)

Do any processing that is necessary to finalize an experiment.

**Parameters status** (Optional[str]) – Status that the experiment finished with (e.g. success, failed, aborted)

**Return type** None

**log_artifact**(*name, artifact, metadata=None, delete_after_upload=False*)

Save an artifact (file/object) in TRAINS experiment storage.

**Parameters**

- **name** (str) – Artifact name. Notice! it will override the previous artifact if the name already exists.
- **artifact** (Union[str, Path, Dict[str, Any], ndarray, Image]) – Artifact object to upload. Currently supports:
  - string / pathlib.Path are treated as path to artifact file to upload If a wildcard or a folder is passed, a zip file containing the local files will be created and uploaded.
  - dict will be stored as .json file and uploaded
  - pandas.DataFrame will be stored as .csv.gz (compressed CSV file) and uploaded
  - numpy.ndarray will be stored as .npz and uploaded

– `PIL.Image.Image` will be stored to .png file and uploaded

- **metadata** (`Optional`[`Dict`[`str`, `Any`]]) – Simple key/value dictionary to store on the artifact. Defaults to `None`.
- **delete_after_upload** (`bool`) – If `True`, the local artifact will be deleted (only applies if `artifact` is a local file). Defaults to `False`.

**Return type** `None`

**log_hyperparams** (*params*)
Log hyperparameters (numeric values) in TRAINS experiments.

**Parameters** **params** (`Union`[`Dict`[`str`, `Any`], `Namespace`]) – The hyperparameters that passed through the model.

**Return type** `None`

**log_image** (*title*, *series*, *image*, *step=None*)
Log Debug image in TRAINS experiment

**Parameters**

- **title** (`str`) – The title of the debug image, i.e. "failed", "passed".
- **series** (`str`) – The series name of the debug image, i.e. "Image 0", "Image 1".
- **image** (`Union`[`str`, `ndarray`, `Image`, `Tensor`]) – Debug image to log. If `numpy.ndarray` or `torch.Tensor`, the image is assumed to be the following:
  – shape: CHW
  – color space: RGB
  – value range: [0., 1.] (float) or [0, 255] (uint8)
- **step** (`Optional`[`int`]) – Step number at which the metrics should be recorded. Defaults to None.

**Return type** `None`

**log_metric** (*title*, *series*, *value*, *step=None*)
Log metrics (numeric values) in TRAINS experiments. This method will be called by the users.

**Parameters**

- **title** (`str`) – The title of the graph to log, e.g. loss, accuracy.
- **series** (`str`) – The series name in the graph, e.g. classification, localization.
- **value** (`float`) – The value to log.
- **step** (`Optional`[`int`]) – Step number at which the metrics should be recorded. Defaults to `None`.

**Return type** `None`

**log_metrics** (*metrics*, *step=None*)
Log metrics (numeric values) in TRAINS experiments. This method will be called by Trainer.

**Parameters**

- **metrics** (`Dict`[`str`, `float`]) – The dictionary of the metrics. If the key contains "/", it will be split by the delimiter, then the elements will be logged as "title" and "series" respectively.

- **step** (Optional[int]) – Step number at which the metrics should be recorded. Defaults to None.

**Return type** None

**log_text** (*text*)
Log console text data in TRAINS experiment.

**Parameters** **text** (str) – The value of the log (data-point).

**Return type** None

**classmethod set_bypass_mode** (*bypass*)
Will bypass all outside communication, and will drop all logs. Should only be used in "standalone mode", when there is no access to the *trains-server*.

**Parameters** **bypass** (bool) – If True, all outside communication is skipped.

**Return type** None

**classmethod set_credentials** (*api_host=None, web_host=None, files_host=None, key=None, secret=None*)
Set new default TRAINS-server host and credentials. These configurations could be overridden by either OS environment variables or trains.conf configuration file.

**Note:** Credentials need to be set *prior* to Logger initialization.

**Parameters**

- **api_host** (Optional[str]) – Trains API server url, example: host='http://localhost:8008'
- **web_host** (Optional[str]) – Trains WEB server url, example: host='http://localhost:8080'
- **files_host** (Optional[str]) – Trains Files server url, example: host='http://localhost:8081'
- **key** (Optional[str]) – user key/secret pair, example: key='thisisakey123'
- **secret** (Optional[str]) – user key/secret pair, example: secret='thisisseceret123'

**Return type** None

**_bypass = None**

**property experiment**
Actual TRAINS object. To use TRAINS features in your *LightningModule* do the following.

Example:

```
self.logger.experiment.some_trains_function()
```

**Return type** Task

**property id**
ID is a uuid (string) representing this specific experiment in the entire system.

**Return type** Optional[str]

**property name**
Name is a human readable non-unique name (str) of the experiment.

**Return type** Optional[str]

**property version**
Return the experiment version.

**Return type** Optional[str]

## pytorch_lightning.loggers.wandb module

### Weights and Biases

**class** pytorch_lightning.loggers.wandb.**WandbLogger**(*name=None*, *save_dir=None*, *offline=False*, *id=None*, *anonymous=False*, *version=None*, *project=None*, *tags=None*, *log_model=False*, *experiment=None*, *entity=None*, *group=None*)

Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Log using Weights and Biases. Install it with pip:

```
pip install wandb
```

**Parameters**

- **name** (Optional[str]) – Display name for the run.
- **save_dir** (Optional[str]) – Path where data is saved.
- **offline** (bool) – Run offline (data can be streamed later to wandb servers).
- **id** (Optional[str]) – Sets the version, mainly used to resume a previous run.
- **anonymous** (bool) – Enables or explicitly disables anonymous logging.
- **version** (Optional[str]) – Sets the version, mainly used to resume a previous run.
- **project** (Optional[str]) – The name of the project to which this run will belong.
- **tags** (Optional[List[str]]) – Tags associated with this run.
- **log_model** (bool) – Save checkpoints in wandb dir to upload on W&B servers.
- **experiment** – WandB experiment object
- **entity** – The team posting this run (default: your username or your default team)
- **group** (Optional[str]) – A unique string shared by all runs in a given group

### Example

```
>>> from pytorch_lightning.loggers import WandbLogger
>>> from pytorch_lightning import Trainer
>>> wandb_logger = WandbLogger()
>>> trainer = Trainer(logger=wandb_logger)
```

**See also:**

- Tutorial on how to use W&B with Pytorch Lightning.

**log_hyperparams** (*params*)
Record hyperparameters.

**Parameters params** (Union[Dict[str, Any], Namespace]) – Namespace containing the hyperparameters

**Return type** None

**log_metrics** (*metrics, step=None*)
Records metrics. This method logs metrics as as soon as it received them. If you want to aggregate metrics for one specific *step*, use the *agg_and_log_metrics()* method.

**Parameters**

- **metrics** (Dict[str, float]) – Dictionary with metric names as keys and measured quantities as values
- **step** (Optional[int]) – Step number at which the metrics should be recorded

**Return type** None

**watch** (*model, log='gradients', log_freq=100*)

**property experiment**
Actual wandb object. To use wandb features in your *LightningModule* do the following.

Example:

```
self.logger.experiment.some_wandb_function()
```

**Return type** Run

**property name**
Return the experiment name.

**Return type** str

**property version**
Return the experiment version.

**Return type** str

## 37.4 pytorch_lightning.metrics package

**class** pytorch_lightning.metrics.**MSE**(*reduction='elementwise_mean'*)

Bases: *pytorch_lightning.metrics.metric.Metric*

Computes the mean squared loss.

**Parameters** **reduction** (str) – a method for reducing mse over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements

### Example

```
>>> pred = torch.tensor([0., 1, 2, 3])
>>> target = torch.tensor([0., 1, 2, 2])
>>> metric = MSE()
>>> metric(pred, target)
tensor(0.2500)
```

**forward**(*pred*, *target*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the mse loss.

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.**RMSE**(*reduction='elementwise_mean'*)

Bases: *pytorch_lightning.metrics.metric.Metric*

Computes the root mean squared loss.

**Parameters** **reduction** (str) – a method for reducing mse over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements

### Example

```
>>> pred = torch.tensor([0., 1, 2, 3])
>>> target = torch.tensor([0., 1, 2, 2])
>>> metric = RMSE()
>>> metric(pred, target)
tensor(0.5000)
```

**forward**(*pred*, *target*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels

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

**Return type** Tensor

**Returns** A Tensor with the rmse loss.

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.**MAE**(*reduction='elementwise_mean'*)

Bases: *pytorch_lightning.metrics.metric.Metric*

Computes the root mean absolute loss or L1-loss.

**Parameters** **reduction** (str) – a method for reducing mse over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements

### Example

```
>>> pred = torch.tensor([0., 1, 2, 3])
>>> target = torch.tensor([0., 1, 2, 2])
>>> metric = MAE()
>>> metric(pred, target)
tensor(0.2500)
```

**forward**(*pred*, *target*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the mae loss.

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.**RMSLE**(*reduction='elementwise_mean'*)

Bases: *pytorch_lightning.metrics.metric.Metric*

Computes the root mean squared log loss.

**Parameters** **reduction** (str) – a method for reducing mse over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements

### Example

```
>>> pred = torch.tensor([0., 1, 2, 3])
>>> target = torch.tensor([0., 1, 2, 2])
>>> metric = RMSLE()
>>> metric(pred, target)
tensor(0.0207)
```

**forward**(*pred, target*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the rmsle loss.

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.**AUC**(*reduce_group=torch.distributed.group.WORLD, reduce_op=torch.distributed.ReduceOp.SUM*)
Bases: *pytorch_lightning.metrics.sklearns.SklearnMetric*

Calculates the Area Under the Curve using the trapoezoidal rule

**Warning:** Every metric call will cause a GPU synchronization, which may slow down your code

### Example

```
>>> y_pred = torch.tensor([0, 1, 2, 3])
>>> y_true = torch.tensor([0, 1, 2, 2])
>>> metric = AUC()
>>> metric(y_pred, y_true)
tensor([4.])
```

**Parameters**

- **reduce_group** (Any) – the process group for DDP reduces (only needed for DDP training). Defaults to all processes (world)
- **reduce_op** (Any) – the operation to perform during reduction within DDP (only needed for DDP training). Defaults to sum.

**forward**(*x, y*)
Computes the AUC

**Parameters**

- **x** (ndarray) – x coordinates.
- **y** (ndarray) – y coordinates.

**Return type** float

**Returns** AUC calculated with trapezoidal rule

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.**AUROC**(*pos_label=1*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the area under curve (AUC) of the receiver operator characteristic (ROC)

## Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = AUROC()
>>> metric(pred, target)
tensor(0.3333)
```

**Parameters**

- **pos_label** (int) – positive label indicator
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward**(*pred*, *target*, *sample_weight=None*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – groundtruth labels
- **sample_weight** (Optional[Sequence]) – the weights per sample

**Returns** classification score

**Return type** torch.Tensor

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.**Accuracy**(*num_classes=None*, *reduction='elementwise_mean'*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the accuracy classification score

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = Accuracy()
>>> metric(pred, target)
tensor(0.7500)
```

**Parameters**

- **num_classes** (Optional[int]) – number of classes
- **reduction** (str) – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward**(*pred, target*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the classification score.

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.**AveragePrecision**(*pos_label=1, reduce_group=None, reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the average precision score

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = AveragePrecision()
>>> metric(pred, target)
tensor(0.3333)
```

**Parameters**

- **pos_label** (int) – positive label indicator
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward** (*pred*, *target*, *sample_weight=None*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – groundtruth labels
- **sample_weight** (Optional[Sequence]) – the weights per sample

**Returns** classification score

**Return type** torch.Tensor

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.**ConfusionMatrix** (*normalize=False*, *reduce_group=None*, *reduce_op=None*)
Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the confusion matrix C where each entry C_{i,j} is the number of observations in group i that were predicted in group j.

### Example

```
>>> pred = torch.tensor([0, 1, 2, 2])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = ConfusionMatrix()
>>> metric(pred, target)
tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 2.]])
```

**Parameters**

- **normalize** (bool) – whether to compute a normalized confusion matrix
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward** (*pred*, *target*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the confusion matrix.

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.**DiceCoefficient**(*include_background=False, nan_score=0.0, no_fg_score=0.0, reduction='elementwise_mean', reduce_group=None, reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the dice coefficient

### Example

```
>>> pred = torch.tensor([[0.85, 0.05, 0.05, 0.05],
...                      [0.05, 0.85, 0.05, 0.05],
...                      [0.05, 0.05, 0.85, 0.05],
...                      [0.05, 0.05, 0.05, 0.85]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> metric = DiceCoefficient()
>>> metric(pred, target)
tensor(0.3333)
```

**Parameters**

- **include_background** (bool) – whether to also compute dice for the background
- **nan_score** (float) – score to return, if a NaN occurs during computation (denom zero)
- **no_fg_score** (float) – score to return, if no foreground pixel was found in target
- **reduction** (str) – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward**(*pred, target*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted probability for each label
- **target** (Tensor) – groundtruth labels

**Returns** the calculated dice coefficient

**Return type** torch.Tensor

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.**F1**(*num_classes=None, reduction='elementwise_mean', reduce_group=None, reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the F1 score, which is the harmonic mean of the precision and recall. It ranges between 1 and 0, where 1 is perfect and the worst value is 0.

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = F1()
>>> metric(pred, target)
tensor(0.6667)
```

**Parameters**

- **num_classes** (Optional[int]) – number of classes
- **reduction** (str) – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward** (*pred*, *target*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – groundtruth labels

**Returns** classification score

**Return type** torch.Tensor

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.**FBeta** (*beta*, *num_classes=None*, *reduction='elementwise_mean'*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

**Computes the FBeta Score, which is the weighted harmonic mean of precision and recall.** It ranges between 1 and 0, where 1 is perfect and the worst value is 0.

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = FBeta(0.25)
>>> metric(pred, target)
tensor(0.7361)
```

**Parameters**

- **beta** (float) – determines the weight of recall in the combined score.
- **num_classes** (Optional[int]) – number of classes

- **reduction** (str) – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for DDP reduction

**forward** (*pred*, *target*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – groundtruth labels

**Returns** classification score

**Return type** torch.Tensor

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.**MulticlassPrecisionRecall** (*num_classes=None*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorCollectionMetric*

Computes the multiclass PR Curve

### Example

```
>>> pred = torch.tensor([[0.85, 0.05, 0.05, 0.05],
...                      [0.05, 0.85, 0.05, 0.05],
...                      [0.05, 0.05, 0.85, 0.05],
...                      [0.05, 0.05, 0.05, 0.85]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> metric = MulticlassPrecisionRecall()
>>> metric(pred, target)
((tensor([1., 1.]), tensor([1., 0.]), tensor([0.8500])),
 (tensor([1., 1.]), tensor([1., 0.]), tensor([0.8500])),
 (tensor([0.2500, 0.0000, 1.0000]), tensor([1., 0., 0.]), tensor([0.0500, 0.
↪8500])),
 (tensor([0.2500, 0.0000, 1.0000]), tensor([1., 0., 0.]), tensor([0.0500, 0.
↪8500])))
```

**Parameters**

- **num_classes** (Optional[int]) – number of classes
- **reduction** – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward** (*pred*, *target*, *sample_weight=None*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted probability for each label
- **target** (Tensor) – groundtruth labels
- **sample_weight** (Optional[Sequence]) – Weights for each sample defining the sample's impact on the score

**Returns** A tuple consisting of one tuple per class, holding precision, recall and thresholds

**Return type** tuple

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.**MulticlassROC** (*num_classes=None*, *reduce_group=None*, *reduce_op=None*)
Bases: *pytorch_lightning.metrics.metric.TensorCollectionMetric*

Computes the multiclass ROC

## Example

```
>>> pred = torch.tensor([[0.85, 0.05, 0.05, 0.05],
...                      [0.05, 0.85, 0.05, 0.05],
...                      [0.05, 0.05, 0.85, 0.05],
...                      [0.05, 0.05, 0.05, 0.85]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> metric = MulticlassROC()
>>> classes_roc = metric(pred, target)
>>> metric(pred, target)
((tensor([0., 0., 1.]), tensor([0., 1., 1.]), tensor([1.8500, 0.8500, 0.0500])),
 (tensor([0., 0., 1.]), tensor([0., 1., 1.]), tensor([1.8500, 0.8500, 0.0500])),
 (tensor([0.0000, 0.3333, 1.0000]), tensor([0., 0., 1.]), tensor([1.8500, 0.8500,␣
↪0.0500])),
 (tensor([0.0000, 0.3333, 1.0000]), tensor([0., 0., 1.]), tensor([1.8500, 0.8500,␣
↪0.0500])))
```

**Parameters**

- **num_classes** (Optional[int]) – number of classes
- **reduction** – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward** (*pred*, *target*, *sample_weight=None*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted probability for each label

- **target** (Tensor) – groundtruth labels
- **sample_weight** (Optional[Sequence]) – Weights for each sample defining the sample's impact on the score

**Returns** A tuple consisting of one tuple per class, holding false positive rate, true positive rate and thresholds

**Return type** tuple

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.**Precision**(*num_classes=None, reduction='elementwise_mean', reduce_group=None, reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the precision score

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = Precision(num_classes=4)
>>> metric(pred, target)
tensor(0.7500)
```

**Parameters**

- **num_classes** (Optional[int]) – number of classes
- **reduction** (str) – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward**(*pred, target*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the classification score.

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.**PrecisionRecall**(*pos_label=1, reduce_group=None, reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorCollectionMetric*

Computes the precision recall curve

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = PrecisionRecall()
>>> prec, recall, thr = metric(pred, target)
>>> prec
tensor([0.3333, 0.0000, 0.0000, 1.0000])
>>> recall
tensor([1., 0., 0., 0.])
>>> thr
tensor([1., 2., 3.])
```

**Parameters**

- **pos_label** (`int`) – positive label indicator
- **reduce_group** (`Optional[Any]`) – the process group to reduce metric results from DDP
- **reduce_op** (`Optional[Any]`) – the operation to perform for ddp reduction

**forward**(*pred*, *target*, *sample_weight=None*)

Actual metric computation

**Parameters**

- **pred** (`Tensor`) – predicted labels
- **target** (`Tensor`) – groundtruth labels
- **sample_weight** (`Optional[Sequence]`) – the weights per sample

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

**Returns**

- precision values
- recall values
- threshold values

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.**PrecisionRecallCurve**(*pos_label=1*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*)

Bases: *pytorch_lightning.metrics.sklearns.SklearnMetric*

Compute precision-recall pairs for different probability thresholds

**Note:** This implementation is restricted to the binary classification task.

The precision is the ratio `tp / (tp + fp)` where `tp` is the number of true positives and `fp` the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative. The recall is the ratio `tp / (tp + fn)` where `tp` is the number of true positives and `fn` the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples. The last precision and recall values are 1. and 0. respectively and do not have a corresponding threshold. This ensures that the graph starts on the x axis.

**Parameters**

- **pos_label** (`Union`[`str`, `int`]) – The class to report if `average='binary'`.
- **reduce_group** (`Any`) – the process group for DDP reduces (only needed for DDP training). Defaults to all processes (world)
- **reduce_op** (`Any`) – the operation to perform during reduction within DDP (only needed for DDP training). Defaults to sum.

**forward**(*probas_pred*, *y_true*, *sample_weight=None*)

**Parameters**

- **probas_pred** (`ndarray`) – Estimated probabilities or decision function.
- **y_true** (`ndarray`) – Ground truth (correct) target values.
- **sample_weight** (`Optional`[`ndarray`]) – Sample weights.

**Returns**

**Precision values such that element i is the precision of** predictions with score >= thresholds[i] and the last element is 1.

**recall:** Decreasing recall values such that element i is the recall of predictions with score >= thresholds[i] and the last element is 0.

**thresholds:** Increasing thresholds on the decision function used to compute precision and recall.

**Return type** precision

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.**ROC**(*pos_label=1*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorCollectionMetric*

Computes the Receiver Operator Characteristic (ROC)

## Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = ROC()
>>> fps, tps, thresholds = metric(pred, target)
>>> fps
tensor([0.0000, 0.3333, 0.6667, 0.6667, 1.0000])
>>> tps
tensor([0., 0., 0., 1., 1.])
>>> thresholds
tensor([4., 3., 2., 1., 0.])
```

**Parameters**

- **pos_label** (int) – positive label indicator
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward**(*pred*, *target*, *sample_weight=None*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – groundtruth labels
- **sample_weight** (Optional[Sequence]) – the weights per sample

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

**Returns**

- false positive rate
- true positive rate
- thresholds

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.**Recall**(*num_classes=None*, *reduction='elementwise_mean'*, *reduce_group=None*, *reduce_op=None*)
Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the recall score

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = Recall()
>>> metric(pred, target)
tensor(0.6250)
```

**Parameters**

- **num_classes** (Optional[int]) – number of classes
- **reduction** (str) – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward**(*pred*, *target*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the classification score.

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.**IoU**(*remove_bg=False*, *reduction='elementwise_mean'*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the intersection over union.

### Example

```
>>> pred = torch.tensor([[0, 0, 0, 0, 0, 0, 0, 0],
...                      [0, 0, 1, 1, 1, 0, 0, 0],
...                      [0, 0, 0, 0, 0, 0, 0, 0]])
>>> target = torch.tensor([[0, 0, 0, 0, 0, 0, 0, 0],
...                        [0, 0, 0, 1, 1, 1, 0, 0],
...                        [0, 0, 0, 0, 0, 0, 0, 0]])
>>> metric = IoU()
>>> metric(pred, target)
tensor(0.7045)
```

**Parameters**

- **remove_bg** (bool) – Flag to state whether a background class has been included within input parameters. If true, will remove background class. If false, return IoU over all classes. Assumes that background is '0' class in input tensor
- **reduction** (str) – a method for reducing IoU over labels (default: takes the mean) Available reduction methods:
  - elementwise_mean: takes the mean
  - none: pass array
  - sum: add elements

**forward**(*y_pred*, *y_true*, *sample_weight=None*)

Actual metric calculation.

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.**SklearnMetric**(*metric_name*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*, ***kwargs*)

Bases: *pytorch_lightning.metrics.metric.NumpyMetric*

Bridge between PyTorch Lightning and scikit-learn metrics

**Warning:** Every metric call will cause a GPU synchronization, which may slow down your code

**Note:** The order of targets and predictions may be different from the order typically used in PyTorch

**Parameters**

- **metric_name** (str) – the metric name to import and compute from scikit-learn.metrics
- **reduce_group** (Any) – the process group for DDP reduces (only needed for DDP training). Defaults to all processes (world)
- **reduce_op** (Any) – the operation to perform during reduction within DDP (only needed for DDP training). Defaults to sum.
- ****kwargs** – additonal keyword arguments (will be forwarded to metric call)

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

Carries the actual metric computation

**Parameters**

- ***args** – Positional arguments forwarded to metric call (should be already converted to numpy)
- ****kwargs** – keyword arguments forwarded to metric call (should be already converted to numpy)

**Return type** Union[ndarray, int, float]

**Returns** the metric value (will be converted to tensor by baseclass)

**_device = None**

**_dtype = None**

**property metric_fn**

### 37.4.1 Subpackages

#### pytorch_lightning.metrics.functional package

#### Submodules

#### pytorch_lightning.metrics.functional.classification module

pytorch_lightning.metrics.functional.classification.**_binary_clf_curve**(*pred*, *target*, *sample_weight=None*, *pos_label=1.0*)

adapted from https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/metrics/_ranking.py

**Return type** Tuple[Tensor, Tensor, Tensor] `pytorch_lightning.metrics.functional.classification.`**`accuracy`**(*pred*, *target*, *num_classes=None*, *reduction='elementwise_mean'*)

Computes the accuracy classification score

**Parameters**

- **`pred`** (`Tensor`) – predicted labels
- **`target`** (`Tensor`) – ground truth labels
- **`num_classes`** (`Optional[int]`) – number of classes
- **`reduction`** – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods:
  - elementwise_mean: takes the mean
  - none: pass array
  - sum: add elements

**Return type** `Tensor`

**Returns** A Tensor with the classification score.

### Example

```
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> accuracy(x, y)
tensor(0.7500)
```

`pytorch_lightning.metrics.functional.classification.`**`auc`**(*x*, *y*, *reorder=True*)

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

**Parameters**

- **`x`** (`Tensor`) – x-coordinates
- **`y`** (`Tensor`) – y-coordinates
- **`reorder`** (`bool`) – reorder coordinates, so they are increasing.

**Return type** `Tensor`

**Returns** Tensor containing AUC score (float)

### Example

```
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> auc(x, y)
tensor(4.)
```

`pytorch_lightning.metrics.functional.classification.`**`auc_decorator`**(*reorder=True*)

**Return type** `Callable`

pytorch_lightning.metrics.functional.classification.**auroc**(*pred*, *target*, *sample_weight=None*, *pos_label=1.0*)

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

**Parameters**

- **pred** (Tensor) – estimated probabilities
- **target** (Tensor) – ground-truth labels
- **sample_weight** (Optional[Sequence]) – sample weights
- **pos_label** (int) – the label for the positive class (default: 1.)

## Example

```
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> auroc(x, y)
tensor(0.3333)
```

**Return type** Tensor

pytorch_lightning.metrics.functional.classification.**average_precision**(*pred*, *target*, *sample_weight=None*, *pos_label=1.0*)

**Parameters**

- **pred** (Tensor) – estimated probabilities
- **target** (Tensor) – ground-truth labels
- **sample_weight** (Optional[Sequence]) – sample weights
- **pos_label** (int) – the label for the positive class (default: 1.)

## Example

```
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> average_precision(x, y)
tensor(0.3333)
```

**Return type** Tensor

pytorch_lightning.metrics.functional.classification.**confusion_matrix**(*pred*, *target*, *normalize=False*)

Computes the confusion matrix C where each entry C_{i,j} is the number of observations in group i that were predicted in group j.

**Parameters**

- **pred** (Tensor) – estimated targets
- **target** (Tensor) – ground truth labels
- **normalize** (bool) – normalizes confusion matrix

**Return type** Tensor

**Returns** Tensor, confusion matrix C [num_classes, num_classes ]

### Example

```
>>> x = torch.tensor([1, 2, 3])
>>> y = torch.tensor([0, 2, 3])
>>> confusion_matrix(x, y)
tensor([[0., 1., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.]])
```

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

**Parameters**

- **pred** (Tensor) – estimated probabilities
- **target** (Tensor) – ground-truth labels
- **bg** (bool) – whether to also compute dice for the background
- **nan_score** (float) – score to return, if a NaN occurs during computation (denom zero)
- **no_fg_score** (float) – score to return, if no foreground pixel was found in target
- **reduction** (str) – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods:
  - elementwise_mean: takes the mean
  - none: pass array
  - sum: add elements

### Example

```
>>> pred = torch.tensor([[0.85, 0.05, 0.05, 0.05],
...                      [0.05, 0.85, 0.05, 0.05],
...                      [0.05, 0.05, 0.85, 0.05],
...                      [0.05, 0.05, 0.05, 0.85]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> average_precision(pred, target)
tensor(0.2500)
```

**Return type** Tensor

pytorch_lightning.metrics.functional.classification.**f1_score**(*pred*, *target*, *num_classes=None*, *reduction='elementwise_mean'*)

Computes the F1-score (a.k.a F-measure), which is the harmonic mean of the precision and recall. It ranges between 1 and 0, where 1 is perfect and the worst value is 0.

**Parameters**

- **pred** (Tensor) – estimated probabilities
- **target** (Tensor) – ground-truth labels
- **num_classes** (Optional[int]) – number of classes
- **reduction** – method for reducing F1-score (default: takes the mean) Available reduction methods:
  - elementwise_mean: takes the mean
  - none: pass array
  - sum: add elements.

**Return type** Tensor

**Returns** Tensor containing F1-score

### Example

```
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> f1_score(x, y)
tensor(0.6667)
```

pytorch_lightning.metrics.functional.classification.**fbeta_score**(*pred*, *target*, *beta*, *num_classes=None*, *reduction='elementwise_mean'*)

Computes the F-beta score which is a weighted harmonic mean of precision and recall. It ranges between 1 and 0, where 1 is perfect and the worst value is 0.

**Parameters**

- **pred** (Tensor) – estimated probabilities
- **target** (Tensor) – ground-truth labels
- **beta** (float) – weights recall when combining the score. beta < 1: more weight to precision. beta > 1 more weight to recall beta = 0: only precision beta -> inf: only recall
- **num_classes** (Optional[int]) – number of classes
- **reduction** (str) – method for reducing F-score (default: takes the mean) Available reduction methods:
  - elementwise_mean: takes the mean
  - none: pass array

– sum: add elements.

**Return type** Tensor

**Returns** Tensor with the value of F-score. It is a value between 0-1.

### Example

```
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> fbeta_score(x, y, 0.2)
tensor(0.7407)
```

pytorch_lightning.metrics.functional.classification.**get_num_classes**(*pred*, *target*, *num_classes*)

Returns the number of classes for a given prediction and target tensor.

**Args:** pred: predicted values target: true labels num_classes: number of classes if known (default: None)

**Return:** An integer that represents the number of classes.

**Return type** int

pytorch_lightning.metrics.functional.classification.**iou**(*pred*, *target*, *num_classes=None*, *remove_bg=False*, *reduction='elementwise_mean'*)

Intersection over union, or Jaccard index calculation.

**Parameters**

- **pred** (Tensor) – Tensor containing predictions
- **target** (Tensor) – Tensor containing targets
- **num_classes** (Optional[int]) – Optionally specify the number of classes
- **remove_bg** (bool) – Flag to state whether a background class has been included within input parameters. If true, will remove background class. If false, return IoU over all classes. Assumes that background is '0' class in input tensor
- **reduction** (str) – a method for reducing IoU over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements

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

**Return type** IoU score

### Example

```
>>> target = torch.randint(0, 1, (10, 25, 25))
>>> pred = torch.tensor(target)
>>> pred[2:5, 7:13, 9:15] = 1 - pred[2:5, 7:13, 9:15]
>>> iou(pred, target)
tensor(0.4914)
```

pytorch_lightning.metrics.functional.classification.**multiclass_auc_decorator**(*reorder=True*)

**Return type** Callable

pytorch_lightning.metrics.functional.classification.**multiclass_precision_recall_curve**(*pred, target, sample_weight, num_c*

Computes precision-recall pairs for different thresholds given a multiclass scores.

**Parameters**

- **pred** (Tensor) – estimated probabilities
- **target** (Tensor) – ground-truth labels
- **sample_weight** (Optional[Sequence]) – sample weight
- **num_classes** (Optional[int]) – number of classes

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

**Returns** number of classes, precision, recall, thresholds

### Example

```
>>> pred = torch.tensor([[0.85, 0.05, 0.05, 0.05],
...                      [0.05, 0.85, 0.05, 0.05],
...                      [0.05, 0.05, 0.85, 0.05],
...                      [0.05, 0.05, 0.05, 0.85]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> nb_classes, precision, recall, thresholds = multiclass_precision_recall_
↪curve(pred, target)
>>> nb_classes
(tensor([1., 1.]), tensor([1., 0.]), tensor([0.8500]))
>>> precision
(tensor([1., 1.]), tensor([1., 0.]), tensor([0.8500]))
>>> recall
(tensor([0.2500, 0.0000, 1.0000]), tensor([1., 0., 0.]), tensor([0.0500, 0.8500]))
>>> thresholds
(tensor([0.2500, 0.0000, 1.0000]), tensor([1., 0., 0.]), tensor([0.0500, 0.8500]))
```

pytorch_lightning.metrics.functional.classification.**multiclass_roc**(*pred, target, sample_weight=None, num_classes=None*)

Computes the Receiver Operating Characteristic (ROC) for multiclass predictors.

**Parameters**

- **pred** (Tensor) – estimated probabilities
- **target** (Tensor) – ground-truth labels
- **sample_weight** (Optional[Sequence]) – sample weights
- **num_classes** (Optional[int]) – number of classes (default: None, computes automatically from data)

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

**Returns** returns roc for each class. Number of classes, false-positive rate (fpr), true-positive rate (tpr), thresholds

### Example

```
>>> pred = torch.tensor([[0.85, 0.05, 0.05, 0.05],
...                      [0.05, 0.85, 0.05, 0.05],
...                      [0.05, 0.05, 0.85, 0.05],
...                      [0.05, 0.05, 0.05, 0.85]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> multiclass_roc(pred, target)
((tensor([0., 0., 1.]), tensor([0., 1., 1.]), tensor([1.8500, 0.8500, 0.0500])),
 (tensor([0., 0., 1.]), tensor([0., 1., 1.]), tensor([1.8500, 0.8500, 0.0500])),
 (tensor([0.0000, 0.3333, 1.0000]), tensor([0., 0., 1.]), tensor([1.8500, 0.8500, 
↪0.0500])),
 (tensor([0.0000, 0.3333, 1.0000]), tensor([0., 0., 1.]), tensor([1.8500, 0.8500, 
↪0.0500])))
```

pytorch_lightning.metrics.functional.classification.**precision**(*pred*, *target*, *num_classes=None*, *reduction='elementwise_mean'*)

Computes precision score.

**Parameters**

- **pred** (Tensor) – estimated probabilities
- **target** (Tensor) – ground-truth labels
- **num_classes** (Optional[int]) – number of classes
- **reduction** (str) – method for reducing precision values (default: takes the mean) Available reduction methods:
  - elementwise_mean: takes the mean
  - none: pass array
  - sum: add elements

**Return type** Tensor

**Returns** Tensor with precision.

### Example

```
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> precision(x, y)
tensor(0.7500)
```

pytorch_lightning.metrics.functional.classification.**precision_recall**(*pred*, *target*, *num_classes=None*, *reduction='elementwise_mean'*)

Computes precision and recall for different thresholds

**Parameters**

- **pred** (Tensor) – estimated probabilities
- **target** (Tensor) – ground-truth labels
- **num_classes** (Optional[int]) – number of classes
- **reduction** (str) – method for reducing precision-recall values (default: takes the mean) Available reduction methods:
  - elementwise_mean: takes the mean
  - none: pass array
  - sum: add elements

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

**Returns** Tensor with precision and recall

### Example

```
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> precision_recall(x, y)
(tensor(0.7500), tensor(0.6250))
```

pytorch_lightning.metrics.functional.classification.**precision_recall_curve**(*pred*, *target*, *sample_weight=None*, *pos_label=1.0*)

Computes precision-recall pairs for different thresholds.

**Parameters**

- **pred** (Tensor) – estimated probabilities
- **target** (Tensor) – ground-truth labels
- **sample_weight** (Optional[Sequence]) – sample weights
- **pos_label** (int) – the label for the positive class (default: 1.)

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

**Returns** precision, recall, thresholds

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> precision, recall, thresholds = precision_recall_curve(pred, target)
>>> precision
tensor([0.3333, 0.0000, 0.0000, 1.0000])
>>> recall
tensor([1., 0., 0., 0.])
>>> thresholds
tensor([1, 2, 3])
```

`pytorch_lightning.metrics.functional.classification.`**`recall`**(*pred*, *target*, *num_classes=None*, *reduction='elementwise_mean'*)

Computes recall score.

**Parameters**

- **pred** (`Tensor`) – estimated probabilities
- **target** (`Tensor`) – ground-truth labels
- **num_classes** (`Optional[int]`) – number of classes
- **reduction** (`str`) – method for reducing recall values (default: takes the mean) Available reduction methods:
  - elementwise_mean: takes the mean
  - none: pass array
  - sum: add elements

**Return type** `Tensor`

**Returns** Tensor with recall.

### Example

```
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> recall(x, y)
tensor(0.6250)
```

`pytorch_lightning.metrics.functional.classification.`**`roc`**(*pred*, *target*, *sample_weight=None*, *pos_label=1.0*)

Computes the Receiver Operating Characteristic (ROC). It assumes classifier is binary.

**Parameters**

- **pred** (`Tensor`) – estimated probabilities
- **target** (`Tensor`) – ground-truth labels
- **sample_weight** (`Optional[Sequence]`) – sample weights

- **pos_label** (int) – the label for the positive class (default: 1)

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

**Returns** false-positive rate (fpr), true-positive rate (tpr), thresholds

### Example

```
>>> x = torch.tensor([0, 1, 2, 3])
>>> y = torch.tensor([0, 1, 2, 2])
>>> fpr, tpr, thresholds = roc(x, y)
>>> fpr
tensor([0.0000, 0.3333, 0.6667, 0.6667, 1.0000])
>>> tpr
tensor([0., 0., 0., 1., 1.])
>>> thresholds
tensor([4, 3, 2, 1, 0])
```

pytorch_lightning.metrics.functional.classification.**stat_scores**(*pred*, *target*, *class_index*, *argmax_dim=1*)

Calculates the number of true positive, false positive, true negative and false negative for a specific class

**Parameters**

- **pred** (Tensor) – prediction tensor
- **target** (Tensor) – target tensor
- **class_index** (int) – class to calculate over
- **argmax_dim** (int) – if pred is a tensor of probabilities, this indicates the axis the argmax transformation will be applied over

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

**Returns** True Positive, False Positive, True Negative, False Negative

### Example

```
>>> x = torch.tensor([1, 2, 3])
>>> y = torch.tensor([0, 2, 3])
>>> tp, fp, tn, fn, sup = stat_scores(x, y, class_index=1)
>>> tp, fp, tn, fn, sup
(tensor(0), tensor(1), tensor(2), tensor(0), tensor(0))
```

pytorch_lightning.metrics.functional.classification.**stat_scores_multiple_classes**(*pred*, *target*, *num_classes=*, *argmax_dim=*

Calls the stat_scores function iteratively for all classes, thus calculating the number of true postive, false postive, true negative and false negative for each class

**Parameters**

- **pred** (Tensor) – prediction tensor
- **target** (Tensor) – target tensor

- **class_index** – class to calculate over
- **argmax_dim** (int) – if pred is a tensor of probabilities, this indicates the axis the argmax transformation will be applied over

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

**Returns** True Positive, False Positive, True Negative, False Negative

### Example

```
>>> x = torch.tensor([1, 2, 3])
>>> y = torch.tensor([0, 2, 3])
>>> tps, fps, tns, fns, sups = stat_scores_multiple_classes(x, y)
>>> tps
tensor([0., 0., 1., 1.])
>>> fps
tensor([0., 1., 0., 0.])
>>> tns
tensor([2., 2., 2., 2.])
>>> fns
tensor([1., 0., 0., 0.])
>>> sups
tensor([1., 0., 1., 1.])
```

pytorch_lightning.metrics.functional.classification.**to_categorical**(*tensor*, *argmax_dim=1*)

Converts a tensor of probabilities to a dense label tensor

**Parameters**

- **tensor** (Tensor) – probabilities to get the categorical label [N, d1, d2, . . . ]
- **argmax_dim** (int) – dimension to apply (default: 1)

**Return type** Tensor

**Returns** A tensor with categorical labels [N, d2, . . . ]

### Example

```
>>> x = torch.tensor([[0.2, 0.5], [0.9, 0.1]])
>>> to_categorical(x)
tensor([1, 0])
```

pytorch_lightning.metrics.functional.classification.**to_onehot**(*tensor*, *n_classes=None*)

Converts a dense label tensor to one-hot format

**Parameters**

- **tensor** (Tensor) – dense label tensor, with shape [N, d1, d2, . . . ]
- **n_classes** (Optional[int]) – number of classes C

**Output:** A sparse label tensor with shape [N, C, d1, d2, . . . ]

#### Example

```
>>> x = torch.tensor([1, 2, 3])
>>> to_onehot(x)
tensor([[0, 1, 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 1]])
```

**Return type** `Tensor`

### pytorch_lightning.metrics.functional.reduction module

`pytorch_lightning.metrics.functional.reduction.`**`reduce`**(*to_reduce*, *reduction*)

Reduces a given tensor by a given reduction method

**Parameters** **`to_reduce`** (`Tensor`) – the tensor, which shall be reduced

reduction : a string specifying the reduction method ('elementwise_mean', 'none', 'sum')

**Return type** `Tensor`

**Returns** reduced Tensor

**Raise:** ValueError if an invalid reduction parameter was given

## 37.4.2 Submodules

### pytorch_lightning.metrics.classification module

**`class`** `pytorch_lightning.metrics.classification.`**`AUROC`**(*pos_label=1*, *reduce_group=None*, *reduce_op=None*)

Bases: *`pytorch_lightning.metrics.metric.TensorMetric`*

Computes the area under curve (AUC) of the receiver operator characteristic (ROC)

#### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = AUROC()
>>> metric(pred, target)
tensor(0.3333)
```

**Parameters**

- **`pos_label`** (`int`) – positive label indicator
- **`reduce_group`** (`Optional[Any]`) – the process group to reduce metric results from DDP
- **`reduce_op`** (`Optional[Any]`) – the operation to perform for ddp reduction

**`forward`**(*pred*, *target*, *sample_weight=None*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – groundtruth labels
- **sample_weight** (Optional[Sequence]) – the weights per sample

**Returns** classification score

**Return type** torch.Tensor

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.classification.**Accuracy**(*num_classes=None, reduction='elementwise_mean', reduce_group=None, reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the accuracy classification score

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = Accuracy()
>>> metric(pred, target)
tensor(0.7500)
```

**Parameters**

- **num_classes** (Optional[int]) – number of classes
- **reduction** (str) – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward**(*pred, target*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the classification score.

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.classification.**AveragePrecision**(*pos_label=1*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the average precision score

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = AveragePrecision()
>>> metric(pred, target)
tensor(0.3333)
```

**Parameters**

- **pos_label** (int) – positive label indicator
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward**(*pred*, *target*, *sample_weight=None*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – groundtruth labels
- **sample_weight** (Optional[Sequence]) – the weights per sample

**Returns** classification score

**Return type** torch.Tensor

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.classification.**ConfusionMatrix**(*normalize=False*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the confusion matrix C where each entry C_{i,j} is the number of observations in group i that were predicted in group j.

### Example

```
>>> pred = torch.tensor([0, 1, 2, 2])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = ConfusionMatrix()
>>> metric(pred, target)
tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 2.]])
```

**Parameters**

- **normalize** (bool) – whether to compute a normalized confusion matrix
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward** (*pred, target*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the confusion matrix.

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.classification.**DiceCoefficient** (*include_background=False, nan_score=0.0, no_fg_score=0.0, reduction='elementwise_mean', reduce_group=None, reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the dice coefficient

### Example

```
>>> pred = torch.tensor([[0.85, 0.05, 0.05, 0.05],
...                      [0.05, 0.85, 0.05, 0.05],
...                      [0.05, 0.05, 0.85, 0.05],
...                      [0.05, 0.05, 0.05, 0.85]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> metric = DiceCoefficient()
>>> metric(pred, target)
tensor(0.3333)
```

**Parameters**

- **include_background** (bool) – whether to also compute dice for the background
- **nan_score** (float) – score to return, if a NaN occurs during computation (denom zero)
- **no_fg_score** (float) – score to return, if no foreground pixel was found in target
- **reduction** (str) – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward** (*pred*, *target*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted probability for each label
- **target** (Tensor) – groundtruth labels

**Returns** the calculated dice coefficient

**Return type** torch.Tensor

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.classification.**F1** (*num_classes=None*, *reduction='elementwise_mean'*, *reduce_group=None*, *reduce_op=None*)
Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the F1 score, which is the harmonic mean of the precision and recall. It ranges between 1 and 0, where 1 is perfect and the worst value is 0.

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = F1()
>>> metric(pred, target)
tensor(0.6667)
```

**Parameters**

- **num_classes** (Optional[int]) – number of classes
- **reduction** (str) – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward** (*pred, target*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – groundtruth labels

**Returns** classification score

**Return type** torch.Tensor

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.classification.**FBeta** (*beta, num_classes=None, reduction='elementwise_mean', reduce_group=None, reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

**Computes the FBeta Score, which is the weighted harmonic mean of precision and recall.** It ranges between 1 and 0, where 1 is perfect and the worst value is 0.

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = FBeta(0.25)
>>> metric(pred, target)
tensor(0.7361)
```

**Parameters**

- **beta** (float) – determines the weight of recall in the combined score.
- **num_classes** (Optional[int]) – number of classes
- **reduction** (str) – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for DDP reduction

**forward** (*pred, target*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – groundtruth labels

**Returns** classification score

**Return type** torch.Tensor

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.classification.**IoU**(*remove_bg=False*, *reduction='elementwise_mean'*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the intersection over union.

### Example

```
>>> pred = torch.tensor([[0, 0, 0, 0, 0, 0, 0, 0],
...                      [0, 0, 1, 1, 1, 0, 0, 0],
...                      [0, 0, 0, 0, 0, 0, 0, 0]])
>>> target = torch.tensor([[0, 0, 0, 0, 0, 0, 0, 0],
...                        [0, 0, 0, 1, 1, 1, 0, 0],
...                        [0, 0, 0, 0, 0, 0, 0, 0]])
>>> metric = IoU()
>>> metric(pred, target)
tensor(0.7045)
```

**Parameters**

- **remove_bg** (bool) – Flag to state whether a background class has been included within input parameters. If true, will remove background class. If false, return IoU over all classes. Assumes that background is '0' class in input tensor
- **reduction** (str) – a method for reducing IoU over labels (default: takes the mean) Available reduction methods:
  - elementwise_mean: takes the mean
  - none: pass array
  - sum: add elements

**forward**(*y_pred*, *y_true*, *sample_weight=None*)

Actual metric calculation.

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.classification.**MulticlassPrecisionRecall**(*num_classes=None*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorCollectionMetric*

Computes the multiclass PR Curve

### Example

```
>>> pred = torch.tensor([[0.85, 0.05, 0.05, 0.05],
...                      [0.05, 0.85, 0.05, 0.05],
...                      [0.05, 0.05, 0.85, 0.05],
...                      [0.05, 0.05, 0.05, 0.85]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> metric = MulticlassPrecisionRecall()
>>> metric(pred, target)
((tensor([1., 1.]), tensor([1., 0.]), tensor([0.8500])),
 (tensor([1., 1.]), tensor([1., 0.]), tensor([0.8500])),
 (tensor([0.2500, 0.0000, 1.0000]), tensor([1., 0., 0.]), tensor([0.0500, 0.
↪8500])),
 (tensor([0.2500, 0.0000, 1.0000]), tensor([1., 0., 0.]), tensor([0.0500, 0.
↪8500])))
```

**Parameters**

- **num_classes** (Optional[int]) – number of classes
- **reduction** – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward**(*pred, target, sample_weight=None*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted probability for each label
- **target** (Tensor) – groundtruth labels
- **sample_weight** (Optional[Sequence]) – Weights for each sample defining the sample's impact on the score

**Returns** A tuple consisting of one tuple per class, holding precision, recall and thresholds

**Return type** tuple

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.classification.**MulticlassROC**(*num_classes=None, reduce_group=None, reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorCollectionMetric*

Computes the multiclass ROC

### Example

```
>>> pred = torch.tensor([[0.85, 0.05, 0.05, 0.05],
...                      [0.05, 0.85, 0.05, 0.05],
...                      [0.05, 0.05, 0.85, 0.05],
...                      [0.05, 0.05, 0.05, 0.85]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> metric = MulticlassROC()
>>> classes_roc = metric(pred, target)
>>> metric(pred, target)
((tensor([0., 0., 1.]), tensor([0., 1., 1.]), tensor([1.8500, 0.8500, 0.0500])),
 (tensor([0., 0., 1.]), tensor([0., 1., 1.]), tensor([1.8500, 0.8500, 0.0500])),
 (tensor([0.0000, 0.3333, 1.0000]), tensor([0., 0., 1.]), tensor([1.8500, 0.8500,␣
↪0.0500])),
 (tensor([0.0000, 0.3333, 1.0000]), tensor([0., 0., 1.]), tensor([1.8500, 0.8500,␣
↪0.0500])))
```

**Parameters**

- **num_classes** (Optional[int]) – number of classes
- **reduction** – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward**(*pred*, *target*, *sample_weight=None*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted probability for each label
- **target** (Tensor) – groundtruth labels
- **sample_weight** (Optional[Sequence]) – Weights for each sample defining the sample's impact on the score

**Returns** A tuple consisting of one tuple per class, holding false positive rate, true positive rate and thresholds

**Return type** tuple

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.classification.**Precision**(*num_classes=None*, *reduction='elementwise_mean'*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the precision score

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = Precision(num_classes=4)
>>> metric(pred, target)
tensor(0.7500)
```

**Parameters**

- **num_classes** (Optional[int]) – number of classes
- **reduction** (str) – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward**(*pred*, *target*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the classification score.

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.classification.**PrecisionRecall**(*pos_label=1*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorCollectionMetric*

Computes the precision recall curve

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = PrecisionRecall()
>>> prec, recall, thr = metric(pred, target)
>>> prec
tensor([0.3333, 0.0000, 0.0000, 1.0000])
>>> recall
tensor([1., 0., 0., 0.])
>>> thr
tensor([1., 2., 3.])
```

**Parameters**

- **pos_label** (int) – positive label indicator
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward** (*pred, target, sample_weight=None*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – groundtruth labels
- **sample_weight** (Optional[Sequence]) – the weights per sample

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

**Returns**

- precision values
- recall values
- threshold values

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.classification.**ROC** (*pos_label=1, reduce_group=None, reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorCollectionMetric*

Computes the Receiver Operator Characteristic (ROC)

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = ROC()
>>> fps, tps, thresholds = metric(pred, target)
>>> fps
tensor([0.0000, 0.3333, 0.6667, 0.6667, 1.0000])
>>> tps
tensor([0., 0., 0., 1., 1.])
>>> thresholds
tensor([4., 3., 2., 1., 0.])
```

**Parameters**

- **pos_label** (int) – positive label indicator
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward** (*pred*, *target*, *sample_weight=None*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – groundtruth labels
- **sample_weight** (Optional[Sequence]) – the weights per sample

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

**Returns**

- false positive rate
- true positive rate
- thresholds

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.classification.**Recall** (*num_classes=None*, *reduction='elementwise_mean'*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.TensorMetric*

Computes the recall score

### Example

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 2, 2])
>>> metric = Recall()
>>> metric(pred, target)
tensor(0.6250)
```

**Parameters**

- **num_classes** (Optional[int]) – number of classes
- **reduction** (str) – a method for reducing accuracies over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements
- **reduce_group** (Optional[Any]) – the process group to reduce metric results from DDP
- **reduce_op** (Optional[Any]) – the operation to perform for ddp reduction

**forward** (*pred*, *target*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the classification score.

**_device = None**

**_dtype = None**

## pytorch_lightning.metrics.converters module

This file provides functions and decorators for automated input and output conversion to/from `numpy.ndarray` and `torch.Tensor` as well as utilities to sync tensors between different processes in a DDP scenario, when needed.

`pytorch_lightning.metrics.converters.`**`_apply_to_inputs`**(*func_to_apply*, **dec_args*, ***dec_kwargs*)

Decorator function to apply a function to all inputs of a function.

**Parameters**

- **func_to_apply** (`Callable`) – the function to apply to the inputs
- ***dec_args** – positional arguments for the function to be applied
- ****dec_kwargs** – keyword arguments for the function to be applied

**Return type** `Callable`

**Returns** the decorated function

`pytorch_lightning.metrics.converters.`**`_apply_to_outputs`**(*func_to_apply*, **dec_args*, ***dec_kwargs*)

Decorator function to apply a function to all outputs of a function.

**Parameters**

- **func_to_apply** (`Callable`) – the function to apply to the outputs
- ***dec_args** – positional arguments for the function to be applied
- ****dec_kwargs** – keyword arguments for the function to be applied

**Return type** `Callable`

**Returns** the decorated function

`pytorch_lightning.metrics.converters.`**`_convert_to_numpy`**(*data*)

Convert all tensors and numpy arrays to numpy arrays.

**Parameters** **data** (`Union`[`Tensor`, `ndarray`, `Number`]) – the tensor or array to convert to numpy

**Return type** `ndarray`

**Returns** the resulting numpy array

`pytorch_lightning.metrics.converters.`**`_convert_to_tensor`**(*data*)

Maps all kind of collections and numbers to tensors.

**Parameters** **data** (`Any`) – the data to convert to tensor

**Return type** `Any`

**Returns** the converted data

`pytorch_lightning.metrics.converters.`**`_numpy_metric_conversion`**(*func_to_decorate*)

Decorator handling the argument conversion for metrics working on numpy. All inputs of the decorated function will be converted to numpy and all outputs will be converted to tensors.

**Parameters** **func_to_decorate** (Callable) – the function whose inputs and outputs shall be converted

**Return type** Callable

**Returns** the decorated function

pytorch_lightning.metrics.converters.**_numpy_metric_input_conversion**(*func_to_decorate*)
Decorator converting all inputs of a function to numpy

**Parameters** **func_to_decorate** (Callable) – the function whose inputs shall be converted

**Returns** the decorated function

**Return type** Callable

pytorch_lightning.metrics.converters.**_sync_ddp_if_available**(*result*, *group=None*, *reduce_op=None*)
Function to reduce the tensors from several ddp processes to one master process

**Parameters**

- **result** (Tensor) – the value to sync and reduce (typically tensor or number)
- **group** (Optional[Any]) – the process group to gather results from. Defaults to all processes (world)
- **reduce_op** (Optional[ReduceOp]) – the reduction operation. Defaults to sum.

**Return type** Tensor

**Returns** reduced value

pytorch_lightning.metrics.converters.**_tensor_collection_metric_conversion**(*func_to_decorate*)
Decorator Handling the argument conversion for metrics working on tensors. All inputs of the decorated function and all numpy arrays and numbers in it's outputs will be converted to tensors

**Parameters** **func_to_decorate** (Callable) – the function whose inputs and outputs shall be converted

**Return type** Callable

**Returns** the decorated function

pytorch_lightning.metrics.converters.**_tensor_collection_metric_output_conversion**(*func_to_decor*
Decorator converting all numpy arrays and numbers occuring in the outputs of a function to tensors

**Parameters** **func_to_decorate** (Callable) – the function whose outputs shall be converted

**Returns** the decorated function

**Return type** Callable

pytorch_lightning.metrics.converters.**_tensor_metric_conversion**(*func_to_decorate*)
Decorator Handling the argument conversion for metrics working on tensors. All inputs and outputs of the decorated function will be converted to tensors

**Parameters** **func_to_decorate** (Callable) – the function whose inputs and outputs shall be converted

**Return type** Callable

**Returns** the decorated function

pytorch_lightning.metrics.converters.**_tensor_metric_input_conversion**(*func_to_decorate*)
Decorator converting all inputs of a function to tensors

**Parameters** **func_to_decorate** (Callable) – the function whose inputs shall be converted

**Returns** the decorated function

**Return type** Callable

`pytorch_lightning.metrics.converters._tensor_metric_output_conversion`(*func_to_decorate*)

Decorator converting all outputs of a function to tensors

**Parameters** **func_to_decorate** (`Callable`) – the function whose outputs shall be converted

**Returns** the decorated function

**Return type** Callable

`pytorch_lightning.metrics.converters.numpy_metric`(*group=None*, *reduce_op=None*)

This decorator shall be used on all function metrics working on numpy arrays. It handles the argument conversion and DDP reduction for metrics working on numpy. All inputs of the decorated function will be converted to numpy and all outputs will be converted to tensors. In DDP Training all output tensors will be reduced according to the given rules.

**Parameters**

- **group** (`Optional[Any]`) – the process group to gather results from. Defaults to all processes (world)
- **reduce_op** (`Optional[ReduceOp]`) – the reduction operation. Defaults to sum

**Return type** `Callable`

**Returns** the decorated function

`pytorch_lightning.metrics.converters.sync_ddp`(*group=None*, *reduce_op=None*)

This decorator syncs a functions outputs across different processes for DDP.

**Parameters**

- **group** (`Optional[Any]`) – the process group to gather results from. Defaults to all processes (world)
- **reduce_op** (`Optional[ReduceOp]`) – the reduction operation. Defaults to sum

**Return type** `Callable`

**Returns** the decorated function

`pytorch_lightning.metrics.converters.tensor_collection_metric`(*group=None*, *reduce_op=None*)

This decorator shall be used on all function metrics working on tensors and returning collections that cannot be converted to tensors. It handles the argument conversion and DDP reduction for metrics working on tensors. All inputs and outputs of the decorated function will be converted to tensors. In DDP Training all output tensors will be reduced according to the given rules.

**Parameters**

- **group** (`Optional[Any]`) – the process group to gather results from. Defaults to all processes (world)
- **reduce_op** (`Optional[ReduceOp]`) – the reduction operation. Defaults to sum

**Return type** `Callable`

**Returns** the decorated function

`pytorch_lightning.metrics.converters.tensor_metric`(*group=None*, *reduce_op=None*)

This decorator shall be used on all function metrics working on tensors. It handles the argument conversion and DDP reduction for metrics working on tensors. All inputs and outputs of the decorated function will be converted to tensors. In DDP Training all output tensors will be reduced according to the given rules.

**Parameters**

- **group** (Optional[Any]) – the process group to gather results from. Defaults to all processes (world)
- **reduce_op** (Optional[ReduceOp]) – the reduction operation. Defaults to sum

**Return type** Callable

**Returns** the decorated function

### pytorch_lightning.metrics.metric module

**class** pytorch_lightning.metrics.metric.**Metric**(*name*)

Bases: *pytorch_lightning.utilities.device_dtype_mixin.DeviceDtypeModuleMixin*, torch.nn.Module, abc.ABC

Abstract base class for metric implementation.

Should be used to implement metrics that 1. Return multiple Outputs 2. Handle their own DDP sync

**Parameters** **name** (str) – the metric's name

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

Implements the actual metric computation.

**Return type** Tensor

**Returns** metric value

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.metric.**NumpyMetric**(*name*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.Metric*

Base class for metric implementation operating on numpy arrays. All inputs will be casted to numpy if necessary and all outputs will be casted to tensors if necessary. Already handles DDP sync and input/output conversions.

**Parameters**

- **name** (str) – the metric's name
- **reduce_group** (Optional[Any]) – the process group for DDP reduces (only needed for DDP training). Defaults to all processes (world)
- **reduce_op** (Optional[Any]) – the operation to perform during reduction within DDP (only needed for DDP training). Defaults to sum.

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

Call self as a function.

**Return type** Tensor

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.metric.**TensorCollectionMetric**(*name*, *reduce_group=None*, *reduce_op=None*)

Bases: *pytorch_lightning.metrics.metric.Metric*

Base class for metric implementation operating directly on tensors. All inputs will be casted to tensors if necessary. Outputs won't be casted. Already handles DDP sync and input conversions.

This class differs from *TensorMetric*, as it assumes all outputs to be collections of tensors and does not explicitly convert them. This is necessary, since some collections (like for ROC, Precision-Recall Curve etc.) cannot be converted to tensors at the highest level. All numpy arrays and numbers occuring in these outputs will still be converted.

Use this class as a baseclass, whenever you want to ensure inputs are tensors and outputs cannot be converted to tensors automatically

**Parameters**

- **name** (`str`) – the metric's name
- **reduce_group** (`Optional[Any]`) – the process group for DDP reduces (only needed for DDP training). Defaults to all processes (world)
- **reduce_op** (`Optional[Any]`) – the operation to perform during reduction within DDP (only needed for DDP training). Defaults to sum.

**`__call__`**(**args*, ***kwargs*)
Call self as a function.

**Return type** `Tensor`

**`_device = None`**

**`_dtype = None`**

**class** `pytorch_lightning.metrics.metric.`**`TensorMetric`**(*name*, *reduce_group=None*, *reduce_op=None*)
Bases: *pytorch_lightning.metrics.metric.Metric*

Base class for metric implementation operating directly on tensors. All inputs and outputs will be casted to tensors if necessary. Already handles DDP sync and input/output conversions.

**Parameters**

- **name** (`str`) – the metric's name
- **reduce_group** (`Optional[Any]`) – the process group for DDP reduces (only needed for DDP training). Defaults to all processes (world)
- **reduce_op** (`Optional[Any]`) – the operation to perform during reduction within DDP (only needed for DDP training). Defaults to sum.

**`__call__`**(**args*, ***kwargs*)
Call self as a function.

**Return type** `Tensor`

**`_device = None`**

**`_dtype = None`**

### pytorch_lightning.metrics.regression module

**class** pytorch_lightning.metrics.regression.**MSE**(*reduction='elementwise_mean'*)
Bases: *pytorch_lightning.metrics.metric.Metric*

Computes the mean squared loss.

**Parameters** **reduction** (str) – a method for reducing mse over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements

#### Example

```
>>> pred = torch.tensor([0., 1, 2, 3])
>>> target = torch.tensor([0., 1, 2, 2])
>>> metric = MSE()
>>> metric(pred, target)
tensor(0.2500)
```

**forward**(*pred*, *target*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the mse loss.

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.regression.**RMSE**(*reduction='elementwise_mean'*)
Bases: *pytorch_lightning.metrics.metric.Metric*

Computes the root mean squared loss.

**Parameters** **reduction** (str) – a method for reducing mse over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements

#### Example

```
>>> pred = torch.tensor([0., 1, 2, 3])
>>> target = torch.tensor([0., 1, 2, 2])
>>> metric = RMSE()
>>> metric(pred, target)
tensor(0.5000)
```

**forward**(*pred*, *target*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels

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

**Return type** Tensor

**Returns** A Tensor with the rmse loss.

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.regression.**MAE**(*reduction='elementwise_mean'*)

Bases: *pytorch_lightning.metrics.metric.Metric*

Computes the root mean absolute loss or L1-loss.

**Parameters** **reduction** (str) – a method for reducing mse over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements

### Example

```
>>> pred = torch.tensor([0., 1, 2, 3])
>>> target = torch.tensor([0., 1, 2, 2])
>>> metric = MAE()
>>> metric(pred, target)
tensor(0.2500)
```

**forward**(*pred, target*)

Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the mae loss.

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.regression.**RMSLE**(*reduction='elementwise_mean'*)

Bases: *pytorch_lightning.metrics.metric.Metric*

Computes the root mean squared log loss.

**Parameters** **reduction** (str) – a method for reducing mse over labels (default: takes the mean) Available reduction methods: - elementwise_mean: takes the mean - none: pass array - sum: add elements

### Example

```
>>> pred = torch.tensor([0., 1, 2, 3])
>>> target = torch.tensor([0., 1, 2, 2])
>>> metric = RMSLE()
>>> metric(pred, target)
tensor(0.0207)
```

**forward** (*pred*, *target*)
Actual metric computation

**Parameters**

- **pred** (Tensor) – predicted labels
- **target** (Tensor) – ground truth labels

**Return type** Tensor

**Returns** A Tensor with the rmsle loss.

**_device = None**

**_dtype = None**

## pytorch_lightning.metrics.sklearns module

**class** pytorch_lightning.metrics.sklearns.**AUC** (*reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*)
Bases: *pytorch_lightning.metrics.sklearns.SklearnMetric*

Calculates the Area Under the Curve using the trapoezoidal rule

**Warning:** Every metric call will cause a GPU synchronization, which may slow down your code

### Example

```
>>> y_pred = torch.tensor([0, 1, 2, 3])
>>> y_true = torch.tensor([0, 1, 2, 2])
>>> metric = AUC()
>>> metric(y_pred, y_true)
tensor([4.])
```

**Parameters**

- **reduce_group** (Any) – the process group for DDP reduces (only needed for DDP training). Defaults to all processes (world)
- **reduce_op** (Any) – the operation to perform during reduction within DDP (only needed for DDP training). Defaults to sum.

**forward** (*x*, *y*)
Computes the AUC

**Parameters**

- **x** (ndarray) – x coordinates.

- **y** (ndarray) – y coordinates.

**Return type** float

**Returns** AUC calculated with trapezoidal rule

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.sklearns.**AUROC**(*average='macro'*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*)

Bases: *pytorch_lightning.metrics.sklearns.SklearnMetric*

Compute Area Under the Curve (AUC) from prediction scores

**Note:** this implementation is restricted to the binary classification task or multilabel classification task in label indicator format.

**Parameters**

- **average** (Optional[str]) – If None, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:
  - If 'micro': Calculate metrics globally by considering each element of the label indicator matrix as a label.
  - If 'macro': Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.
  - If 'weighted': Calculate metrics for each label, and find their average, weighted by support (the number of true instances for each label).
  - If 'samples': Calculate metrics for each instance, and find their average.
- **reduce_group** (Any) – the process group for DDP reduces (only needed for DDP training). Defaults to all processes (world)
- **reduce_op** (Any) – the operation to perform during reduction within DDP (only needed for DDP training). Defaults to sum.

**forward**(*y_score*, *y_true*, *sample_weight=None*)

**Parameters**

- **y_score** (ndarray) – Target scores, can either be probability estimates of the positive class, confidence values, or binary decisions.
- **y_true** (ndarray) – True binary labels in binary label indicators.
- **sample_weight** (Optional[ndarray]) – Sample weights.

**Return type** float

**Returns** Area Under Receiver Operating Characteristic Curve

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.sklearns.**Accuracy**(*normalize=True*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*)

Bases: *pytorch_lightning.metrics.sklearns.SklearnMetric*

Calculates the Accuracy Score

**Warning:** Every metric call will cause a GPU synchronization, which may slow down your code

### Example

```
>>> y_pred = torch.tensor([0, 1, 2, 3])
>>> y_true = torch.tensor([0, 1, 2, 2])
>>> metric = Accuracy()
>>> metric(y_pred, y_true)
tensor([0.7500])
```

**Parameters**

- **normalize** (bool) – If False, return the number of correctly classified samples. Otherwise, return the fraction of correctly classified samples.
- **reduce_group** (Any) – the process group for DDP reduces (only needed for DDP training). Defaults to all processes (world)
- **reduce_op** (Any) – the operation to perform during reduction within DDP (only needed for DDP training). Defaults to sum.

**forward**(*y_pred*, *y_true*, *sample_weight=None*)

Computes the accuracy

**Parameters**

- **y_pred** (ndarray) – the array containing the predictions (already in categorical form)
- **y_true** (ndarray) – the array containing the targets (in categorical form)
- **sample_weight** (Optional[ndarray]) – Sample weights.

**Return type** float

**Returns** Accuracy Score

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.sklearns.**AveragePrecision**(*average='macro'*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*)

Bases: *pytorch_lightning.metrics.sklearns.SklearnMetric*

Calculates the average precision (AP) score.

**Parameters**

- **average** (Optional[str]) – If None, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

- If 'micro': Calculate metrics globally by considering each element of the label indicator matrix as a label.
- If 'macro': Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.
- If 'weighted': Calculate metrics for each label, and find their average, weighted by support (the number of true instances for each label).
- If 'samples': Calculate metrics for each instance, and find their average.

- **reduce_group** (Any) – the process group for DDP reduces (only needed for DDP training). Defaults to all processes (world)
- **reduce_op** (Any) – the operation to perform during reduction within DDP (only needed for DDP training). Defaults to sum.

**forward** (*y_score*, *y_true*, *sample_weight=None*)

**Parameters**

- **y_score** (ndarray) – Target scores, can either be probability estimates of the positive class, confidence values, or binary decisions.
- **y_true** (ndarray) – True binary labels in binary label indicators.
- **sample_weight** (Optional[ndarray]) – Sample weights.

**Return type** float

**Returns** average precision score

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.sklearns.**ConfusionMatrix** (*labels=None*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*)

Bases: *pytorch_lightning.metrics.sklearns.SklearnMetric*

Compute confusion matrix to evaluate the accuracy of a classification By definition a confusion matrix $C$ is such that $C_{i,j}$ is equal to the number of observations known to be in group $i$ but predicted to be in group $j$.

### Example

```
>>> y_pred = torch.tensor([0, 1, 2, 1])
>>> y_true = torch.tensor([0, 1, 2, 2])
>>> metric = ConfusionMatrix()
>>> metric(y_pred, y_true)
tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 1., 1.]])
```

**Parameters**

- **labels** (Optional[Sequence]) – List of labels to index the matrix. This may be used to reorder or select a subset of labels. If none is given, those that appear at least once in y_true or y_pred are used in sorted order.

- **reduce_group** (Any) – the process group for DDP reduces (only needed for DDP training). Defaults to all processes (world)
- **reduce_op** (Any) – the operation to perform during reduction within DDP (only needed for DDP training). Defaults to sum.

**forward** (*y_pred*, *y_true*)

**Parameters**

- **y_pred** (ndarray) – Estimated targets as returned by a classifier.
- **y_true** (ndarray) – Ground truth (correct) target values.

**Return type** ndarray

**Returns** Confusion matrix (array of shape [n_classes, n_classes])

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.sklearns.**F1** (*labels=None*, *pos_label=1*, *average='macro'*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*)

Bases: *pytorch_lightning.metrics.sklearns.SklearnMetric*

Compute the F1 score, also known as balanced F-score or F-measure The F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0. The relative contribution of precision and recall to the F1 score are equal. The formula for the F1 score is:

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

In the multi-class and multi-label case, this is the weighted average of the F1 score of each class.

## Example

```
>>> y_pred = torch.tensor([0, 1, 2, 3])
>>> y_true = torch.tensor([0, 1, 2, 2])
>>> metric = F1()
>>> metric(y_pred, y_true)
tensor([0.6667])
```

**References**

- [1] Wikipedia entry for the F1-score

**Parameters**

- **labels** (Optional[Sequence]) – Integer array of labels.
- **pos_label** (Union[str, int]) – The class to report if `average='binary'`.
- **average** (Optional[str]) – This parameter is required for multiclass/multilabel targets. If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:
  - `'binary'`: Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.

– `'micro'`: Calculate metrics globally by counting the total true positives, false negatives and false positives.
– `'macro'`: Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.
– `'weighted'`: Calculate metrics for each label, and find their average, weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.
– `'samples'`: Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from `accuracy_score()`).

Note that if `pos_label` is given in binary classification with *average != 'binary'*, only that positive class is reported. This behavior is deprecated and will change in version 0.18.

- **reduce_group** (`Any`) – the process group for DDP reduces (only needed for DDP training). Defaults to all processes (world)
- **reduce_op** (`Any`) – the operation to perform during reduction within DDP (only needed for DDP training). Defaults to sum.

**forward** (*y_pred*, *y_true*, *sample_weight=None*)

**Parameters**

- **y_pred** (`ndarray`) – Estimated targets as returned by a classifier.
- **y_true** (`ndarray`) – Ground truth (correct) target values.
- **sample_weight** (`Optional[ndarray]`) – Sample weights.

**Return type** `Union[ndarray, float]`

**Returns** F1 score of the positive class in binary classification or weighted average of the F1 scores of each class for the multiclass task.

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.sklearns.**FBeta** (*beta*, *labels=None*, *pos_label=1*, *average='macro'*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*)

Bases: *pytorch_lightning.metrics.sklearns.SklearnMetric*

Compute the F-beta score. The *beta* parameter determines the weight of precision in the combined score. `beta < 1` lends more weight to precision, while `beta > 1` favors recall (`beta -> 0` considers only precision, `beta -> inf` only recall).

## Example

```
>>> y_pred = torch.tensor([0, 1, 2, 3])
>>> y_true = torch.tensor([0, 1, 2, 2])
>>> metric = FBeta(beta=0.25)
>>> metric(y_pred, y_true)
tensor([0.7361])
```

### References

- [1] R. Baeza-Yates and B. Ribeiro-Neto (2011). Modern Information Retrieval. Addison Wesley, pp. 327-328.
- [2] Wikipedia entry for the F1-score

**Parameters**

- **beta** (float) – Weight of precision in harmonic mean.
- **labels** (Optional[Sequence]) – Integer array of labels.
- **pos_label** (Union[str, int]) – The class to report if `average='binary'`.
- **average** (Optional[str]) – This parameter is required for multiclass/multilabel targets. If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:
  - `'binary'`: Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.
  - `'micro'`: Calculate metrics globally by counting the total true positives, false negatives and false positives.
  - `'macro'`: Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.
  - `'weighted'`: Calculate metrics for each label, and find their average, weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.
  - `'samples'`: Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from `accuracy_score()`).

  Note that if `pos_label` is given in binary classification with *average != 'binary'*, only that positive class is reported. This behavior is deprecated and will change in version 0.18.
- **reduce_group** (Any) – the process group for DDP reduces (only needed for DDP training). Defaults to all processes (world)
- **reduce_op** (Any) – the operation to perform during reduction within DDP (only needed for DDP training). Defaults to sum.

**forward**(*y_pred*, *y_true*, *sample_weight=None*)

**Parameters**

- **y_pred** (ndarray) – Estimated targets as returned by a classifier.
- **y_true** (ndarray) – Ground truth (correct) target values.
- **sample_weight** (Optional[ndarray]) – Sample weights.

**Return type** Union[ndarray, float]

**Returns** FBeta score of the positive class in binary classification or weighted average of the FBeta scores of each class for the multiclass task.

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.sklearns.**Precision**(*labels=None*, *pos_label=1*, *average='macro'*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*)

Bases: `pytorch_lightning.metrics.sklearns.SklearnMetric`

Compute the precision The precision is the ratio `tp / (tp + fp)` where `tp` is the number of true positives and `fp` the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative. The best value is 1 and the worst value is 0.

### Example

```
>>> y_pred = torch.tensor([0, 1, 2, 3])
>>> y_true = torch.tensor([0, 1, 2, 2])
>>> metric = Precision()
>>> metric(y_pred, y_true)
tensor([0.7500])
```

**Parameters**

- **labels** (`Optional[Sequence]`) – Integer array of labels.
- **pos_label** (`Union[str, int]`) – The class to report if `average='binary'`.
- **average** (`Optional[str]`) – This parameter is required for multiclass/multilabel targets. If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:
  - `'binary'`: Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.
  - `'micro'`: Calculate metrics globally by counting the total true positives, false negatives and false positives.
  - `'macro'`: Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.
  - `'weighted'`: Calculate metrics for each label, and find their average, weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.
  - `'samples'`: Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from `accuracy_score()`).

  Note that if `pos_label` is given in binary classification with *average != 'binary'*, only that positive class is reported. This behavior is deprecated and will change in version 0.18.
- **reduce_group** (`Any`) – the process group for DDP reduces (only needed for DDP training). Defaults to all processes (world)
- **reduce_op** (`Any`) – the operation to perform during reduction within DDP (only needed for DDP training). Defaults to sum.

**forward**(*y_pred*, *y_true*, *sample_weight=None*)

**Parameters**

- **y_pred** (`ndarray`) – Estimated targets as returned by a classifier.
- **y_true** (`ndarray`) – Ground truth (correct) target values.

- **sample_weight** (Optional[ndarray]) – Sample weights.

**Return type** Union[ndarray, float]

**Returns** Precision of the positive class in binary classification or weighted average of the precision of each class for the multiclass task.

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.sklearns.**PrecisionRecallCurve**(*pos_label=1*, *reduce_group=torch.distributed.group.WOR*, *reduce_op=torch.distributed.ReduceOp.SUM*)

Bases: *pytorch_lightning.metrics.sklearns.SklearnMetric*

Compute precision-recall pairs for different probability thresholds

**Note:** This implementation is restricted to the binary classification task.

The precision is the ratio `tp / (tp + fp)` where `tp` is the number of true positives and `fp` the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative. The recall is the ratio `tp / (tp + fn)` where `tp` is the number of true positives and `fn` the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples. The last precision and recall values are 1. and 0. respectively and do not have a corresponding threshold. This ensures that the graph starts on the x axis.

**Parameters**

- **pos_label** (Union[str, int]) – The class to report if `average='binary'`.
- **reduce_group** (Any) – the process group for DDP reduces (only needed for DDP training). Defaults to all processes (world)
- **reduce_op** (Any) – the operation to perform during reduction within DDP (only needed for DDP training). Defaults to sum.

**forward**(*probas_pred*, *y_true*, *sample_weight=None*)

**Parameters**

- **probas_pred** (ndarray) – Estimated probabilities or decision function.
- **y_true** (ndarray) – Ground truth (correct) target values.
- **sample_weight** (Optional[ndarray]) – Sample weights.

**Returns**

**Precision values such that element i is the precision of** predictions with score >= thresholds[i] and the last element is 1.

**recall:** Decreasing recall values such that element i is the recall of predictions with score >= thresholds[i] and the last element is 0.

**thresholds:** Increasing thresholds on the decision function used to compute precision and recall.

**Return type** precision

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.sklearns.**ROC**(*pos_label=1*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*)

Bases: *pytorch_lightning.metrics.sklearns.SklearnMetric*

Compute Receiver operating characteristic (ROC)

**Note:** this implementation is restricted to the binary classification task.

### Example

```
>>> y_pred = torch.tensor([0, 1, 2, 3])
>>> y_true = torch.tensor([0, 1, 2, 2])
>>> metric = ROC()
>>> fps, tps = metric(y_pred, y_true)
>>> fps
tensor([0.0000, 0.3333, 0.6667, 0.6667, 1.0000])
>>> tps
tensor([0., 0., 0., 1., 1.])
```

### References

- [1] Wikipedia entry for the Receiver operating characteristic

**Parameters**

- **pos_labels** – The class to report if `average='binary'`.
- **reduce_group** (`Any`) – the process group for DDP reduces (only needed for DDP training). Defaults to all processes (world)
- **reduce_op** (`Any`) – the operation to perform during reduction within DDP (only needed for DDP training). Defaults to sum.

**forward**(*y_score*, *y_true*, *sample_weight=None*)

**Parameters**

- **y_score** (`ndarray`) – Target scores, can either be probability estimates of the positive class or confidence values.
- **y_true** (`ndarray`) – Ground truth (correct) target values.
- **sample_weight** (`Optional[ndarray]`) – Sample weights.

**Returns**

**Increasing false positive rates such that element i is the false** positive rate of predictions with score >= thresholds[i].

**tpr:** Increasing true positive rates such that element i is the true positive rate of predictions with score >= thresholds[i].

**thresholds:** Decreasing thresholds on the decision function used to compute fpr and tpr. *thresholds[0]* represents no instances being predicted and is arbitrarily set to *max(y_score) + 1*.

**Return type** fpr

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.sklearns.**Recall**(*labels=None*, *pos_label=1*, *average='macro'*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*)

Bases: *pytorch_lightning.metrics.sklearns.SklearnMetric*

Compute the recall The recall is the ratio `tp / (tp + fn)` where `tp` is the number of true positives and `fn` the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples. The best value is 1 and the worst value is 0.

### Example

```
>>> y_pred = torch.tensor([0, 1, 2, 3])
>>> y_true = torch.tensor([0, 1, 2, 2])
>>> metric = Recall()
>>> metric(y_pred, y_true)
tensor([0.6250])
```

**Parameters**

- **labels** (Optional[Sequence]) – Integer array of labels.
- **pos_label** (Union[str, int]) – The class to report if `average='binary'`.
- **average** (Optional[str]) – This parameter is required for multiclass/multilabel targets. If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:
  - `'binary'`: Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.
  - `'micro'`: Calculate metrics globally by counting the total true positives, false negatives and false positives.
  - `'macro'`: Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.
  - `'weighted'`: Calculate metrics for each label, and find their average, weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.
  - `'samples'`: Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from `accuracy_score()`).

  Note that if `pos_label` is given in binary classification with *average != 'binary'*, only that positive class is reported. This behavior is deprecated and will change in version 0.18.
- **reduce_group** (Any) – the process group for DDP reduces (only needed for DDP training). Defaults to all processes (world)
- **reduce_op** (Any) – the operation to perform during reduction within DDP (only needed for DDP training). Defaults to sum.

**forward**(*y_pred*, *y_true*, *sample_weight=None*)

**Parameters**

- **y_pred** (ndarray) – Estimated targets as returned by a classifier.
- **y_true** (ndarray) – Ground truth (correct) target values.
- **sample_weight** (Optional[ndarray]) – Sample weights.

**Return type** Union[ndarray, float]

**Returns** Recall of the positive class in binary classification or weighted average of the recall of each class for the multiclass task.

**_device = None**

**_dtype = None**

**class** pytorch_lightning.metrics.sklearns.**SklearnMetric**(*metric_name*, *reduce_group=torch.distributed.group.WORLD*, *reduce_op=torch.distributed.ReduceOp.SUM*, ***kwargs*)

Bases: *pytorch_lightning.metrics.metric.NumpyMetric*

Bridge between PyTorch Lightning and scikit-learn metrics

**Warning:** Every metric call will cause a GPU synchronization, which may slow down your code

**Note:** The order of targets and predictions may be different from the order typically used in PyTorch

**Parameters**

- **metric_name** (str) – the metric name to import and compute from scikit-learn.metrics
- **reduce_group** (Any) – the process group for DDP reduces (only needed for DDP training). Defaults to all processes (world)
- **reduce_op** (Any) – the operation to perform during reduction within DDP (only needed for DDP training). Defaults to sum.
- ****kwargs** – additonal keyword arguments (will be forwarded to metric call)

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

Carries the actual metric computation

**Parameters**

- ***args** – Positional arguments forwarded to metric call (should be already converted to numpy)
- ****kwargs** – keyword arguments forwarded to metric call (should be already converted to numpy)

**Return type** Union[ndarray, int, float]

**Returns** the metric value (will be converted to tensor by baseclass)

**_device = None**

**_dtype = None**

**property metric_fn**

## 37.5 pytorch_lightning.overrides package

### 37.5.1 Submodules

#### pytorch_lightning.overrides.data_parallel module

**class** pytorch_lightning.overrides.data_parallel.**LightningDataParallel**(**args*, ***kwargs*)

Bases: `torch.nn.DataParallel`

Override the forward call in lightning so it goes to training and validation step respectively

**forward**(**inputs*, ***kwargs*)

**parallel_apply**(*replicas*, *inputs*, *kwargs*)

**class** pytorch_lightning.overrides.data_parallel.**LightningDistributedDataParallel**(**args*, ***kwargs*)

Bases: `torch.nn.parallel.DistributedDataParallel`

Override the forward call in lightning so it goes to training and validation step respectively

**forward**(**inputs*, ***kwargs*)

**parallel_apply**(*replicas*, *inputs*, *kwargs*)

pytorch_lightning.overrides.data_parallel.**_find_tensors**(*obj*)

Recursively find all tensors contained in the specified object.

pytorch_lightning.overrides.data_parallel.**auto_squeeze_dim_zeros**(*output*)

In DP or DDP2 we need to unsqueeze dim 0 :param _sphinx_paramlinks_pytorch_lightning.overrides.data_parallel.auto_squeeze_
:return:

pytorch_lightning.overrides.data_parallel.**get_a_var**(*obj*)

pytorch_lightning.overrides.data_parallel.**parallel_apply**(*modules*, *inputs*, *kwargs_tup=None*, *devices=None*)

Applies each *module* in `modules` in parallel on arguments contained in `inputs` (positional) and `kwargs_tup` (keyword) on each of `devices`.

**Parameters**

- **modules** (*Module*) – modules to be parallelized
- **inputs** (*tensor*) – inputs to the modules
- **devices** (*list of int or torch.device*) – CUDA devices

`modules`, `inputs`, `kwargs_tup` (if given), and `devices` (if given) should all have same length. Moreover, each element of `inputs` can either be a single object as the only argument to a module, or a collection of positional arguments.

## 37.6 pytorch_lightning.profiler package

Profiling your training run can help you understand if there are any bottlenecks in your code.

### 37.6.1 Built-in checks

PyTorch Lightning supports profiling standard actions in the training loop out of the box, including:

- on_epoch_start
- on_epoch_end
- on_batch_start
- tbptt_split_batch
- model_forward
- model_backward
- on_after_backward
- optimizer_step
- on_batch_end
- training_step_end
- on_training_end

### 37.6.2 Enable simple profiling

If you only wish to profile the standard actions, you can set *profiler=True* when constructing your *Trainer* object.

```
trainer = Trainer(..., profiler=True)
```

The profiler's results will be printed at the completion of a training *fit()*.

```
Profiler Report

Action                  |  Mean duration (s)    |  Total time (s)
-----------------------------------------------------------------
on_epoch_start          |  5.993e-06            |  5.993e-06
get_train_batch         |  0.0087412            |  16.398
on_batch_start          |  5.0865e-06           |  0.0095372
model_forward           |  0.0017818            |  3.3408
model_backward          |  0.0018283            |  3.4282
on_after_backward       |  4.2862e-06           |  0.0080366
optimizer_step          |  0.0011072            |  2.0759
on_batch_end            |  4.5202e-06           |  0.0084753
on_epoch_end            |  3.919e-06            |  3.919e-06
on_train_end            |  5.449e-06            |  5.449e-06
```

### 37.6.3 Advanced Profiling

If you want more information on the functions called during each event, you can use the *AdvancedProfiler*. This option uses Python's cProfiler to provide a report of time spent on *each* function called within your code.

```
profiler = AdvancedProfiler()
trainer = Trainer(..., profiler=profiler)
```

The profiler's results will be printed at the completion of a training *fit()*. This profiler report can be quite long, so you can also specify an *output_filename* to save the report instead of logging it to the output in your terminal. The output below shows the profiling for the action *get_train_batch*.

```
Profiler Report

Profile stats for: get_train_batch
        4869394 function calls (4863767 primitive calls) in 18.893 seconds
Ordered by: cumulative time
List reduced from 76 to 10 due to restriction <10>
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
3752/1876    0.011    0.000   18.887    0.010 {built-in method builtins.next}
    1876     0.008    0.000   18.877    0.010 dataloader.py:344(__next__)
    1876     0.074    0.000   18.869    0.010 dataloader.py:383(_next_data)
    1875     0.012    0.000   18.721    0.010 fetch.py:42(fetch)
    1875     0.084    0.000   18.290    0.010 fetch.py:44(<listcomp>)
    60000    1.759    0.000   18.206    0.000 mnist.py:80(__getitem__)
    60000    0.267    0.000   13.022    0.000 transforms.py:68(__call__)
    60000    0.182    0.000    7.020    0.000 transforms.py:93(__call__)
    60000    1.651    0.000    6.839    0.000 functional.py:42(to_tensor)
    60000    0.260    0.000    5.734    0.000 transforms.py:167(__call__)
```

You can also reference this profiler in your LightningModule to profile specific actions of interest. If you don't want to always have the profiler turned on, you can optionally pass a *PassThroughProfiler* which will allow you to skip profiling without having to make any code changes. Each profiler has a method *profile()* which returns a context handler. Simply pass in the name of your action that you want to track and the profiler will record performance for code executed within this context.

```
from pytorch_lightning.profiler import Profiler, PassThroughProfiler

class MyModel(LightningModule):
    def __init__(self, profiler=None):
        self.profiler = profiler or PassThroughProfiler()

    def custom_processing_step(self, data):
        with profiler.profile('my_custom_action'):
            # custom processing step
        return data

profiler = Profiler()
model = MyModel(profiler)
trainer = Trainer(profiler=profiler, max_epochs=1)
```

**class** pytorch_lightning.profiler.**BaseProfiler**(*output_streams=None*)

Bases: abc.ABC

If you wish to write a custom profiler, you should inhereit from this class.

**Params:** stream_out: callable

**describe**()
Logs a profile report after the conclusion of the training run.

**Return type** None

**profile**(*action_name*)
Yields a context manager to encapsulate the scope of a profiled action.

Example:

```
with self.profile('load training data'):
    # load training data code
```

The profiler will start once you've entered the context and will automatically stop once you exit the code block.

**Return type** None

**profile_iterable**(*iterable*, *action_name*)

**Return type** None

**abstract start**(*action_name*)
Defines how to start recording an action.

**Return type** None

**abstract stop**(*action_name*)
Defines how to record the duration once an action is complete.

**Return type** None

**abstract summary**()
Create profiler summary in text format.

**Return type** str

**class** pytorch_lightning.profiler.**SimpleProfiler**(*output_filename=None*)
Bases: *pytorch_lightning.profiler.profilers.BaseProfiler*

This profiler simply records the duration of actions (in seconds) and reports the mean duration of each action and the total time spent over the entire training run.

**Params:**

**output_filename (str): optionally save profile results to file instead of printing** to std out when training is finished.

**describe**()
Logs a profile report after the conclusion of the training run.

**start**(*action_name*)
Defines how to start recording an action.

**Return type** None

**stop**(*action_name*)
Defines how to record the duration once an action is complete.

**Return type** None

**summary**()
Create profiler summary in text format.

**Return type** str

**class** pytorch_lightning.profiler.**AdvancedProfiler**(*output_filename=None*, *line_count_restriction=1.0*)

Bases: *pytorch_lightning.profiler.profilers.BaseProfiler*

This profiler uses Python's cProfiler to record more detailed information about time spent in each function call recorded during a given action. The output is quite verbose and you should only use this if you want very detailed reports.

**Parameters**

- **output_filename** (Optional[str]) – optionally save profile results to file instead of printing to std out when training is finished.
- **line_count_restriction** (float) – this can be used to limit the number of functions reported for each action. either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines)

**describe**()

Logs a profile report after the conclusion of the training run.

**start**(*action_name*)

Defines how to start recording an action.

**Return type** None

**stop**(*action_name*)

Defines how to record the duration once an action is complete.

**Return type** None

**summary**()

Create profiler summary in text format.

**Return type** str

**class** pytorch_lightning.profiler.**PassThroughProfiler**

Bases: *pytorch_lightning.profiler.profilers.BaseProfiler*

This class should be used when you don't want the (small) overhead of profiling. The Trainer uses this class by default.

Params: stream_out: callable

**start**(*action_name*)

Defines how to start recording an action.

**Return type** None

**stop**(*action_name*)

Defines how to record the duration once an action is complete.

**Return type** None

**summary**()

Create profiler summary in text format.

**Return type** str

### 37.6.4 Submodules

#### pytorch_lightning.profiler.profilers module

**class** pytorch_lightning.profiler.profilers.**AdvancedProfiler**(*output_filename=None*, *line_count_restriction=1.0*)

Bases: *pytorch_lightning.profiler.profilers.BaseProfiler*

This profiler uses Python's cProfiler to record more detailed information about time spent in each function call recorded during a given action. The output is quite verbose and you should only use this if you want very detailed reports.

**Parameters**

- **output_filename** (Optional[str]) – optionally save profile results to file instead of printing to std out when training is finished.
- **line_count_restriction** (float) – this can be used to limit the number of functions reported for each action. either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines)

**describe**()

Logs a profile report after the conclusion of the training run.

**start**(*action_name*)

Defines how to start recording an action.

**Return type** None

**stop**(*action_name*)

Defines how to record the duration once an action is complete.

**Return type** None

**summary**()

Create profiler summary in text format.

**Return type** str

**class** pytorch_lightning.profiler.profilers.**BaseProfiler**(*output_streams=None*)

Bases: abc.ABC

If you wish to write a custom profiler, you should inhereit from this class.

**Params:** stream_out: callable

**describe**()

Logs a profile report after the conclusion of the training run.

**Return type** None

**profile**(*action_name*)

Yields a context manager to encapsulate the scope of a profiled action.

Example:

```
with self.profile('load training data'):
    # load training data code
```

The profiler will start once you've entered the context and will automatically stop once you exit the code block.

**Return type** None

**profile_iterable**(*iterable*, *action_name*)

**Return type** None

**abstract start**(*action_name*)
Defines how to start recording an action.

**Return type** None

**abstract stop**(*action_name*)
Defines how to record the duration once an action is complete.

**Return type** None

**abstract summary**()
Create profiler summary in text format.

**Return type** str

**class** pytorch_lightning.profiler.profilers.**PassThroughProfiler**
Bases: *pytorch_lightning.profiler.profilers.BaseProfiler*

This class should be used when you don't want the (small) overhead of profiling. The Trainer uses this class by default.

Params: stream_out: callable

**start**(*action_name*)
Defines how to start recording an action.

**Return type** None

**stop**(*action_name*)
Defines how to record the duration once an action is complete.

**Return type** None

**summary**()
Create profiler summary in text format.

**Return type** str

**class** pytorch_lightning.profiler.profilers.**SimpleProfiler**(*output_filename=None*)
Bases: *pytorch_lightning.profiler.profilers.BaseProfiler*

This profiler simply records the duration of actions (in seconds) and reports the mean duration of each action and the total time spent over the entire training run.

**Params:**

**output_filename (str): optionally save profile results to file instead of printing** to std out when training is finished.

**describe**()
Logs a profile report after the conclusion of the training run.

**start**(*action_name*)
Defines how to start recording an action.

**Return type** None

**stop**(*action_name*)
Defines how to record the duration once an action is complete.

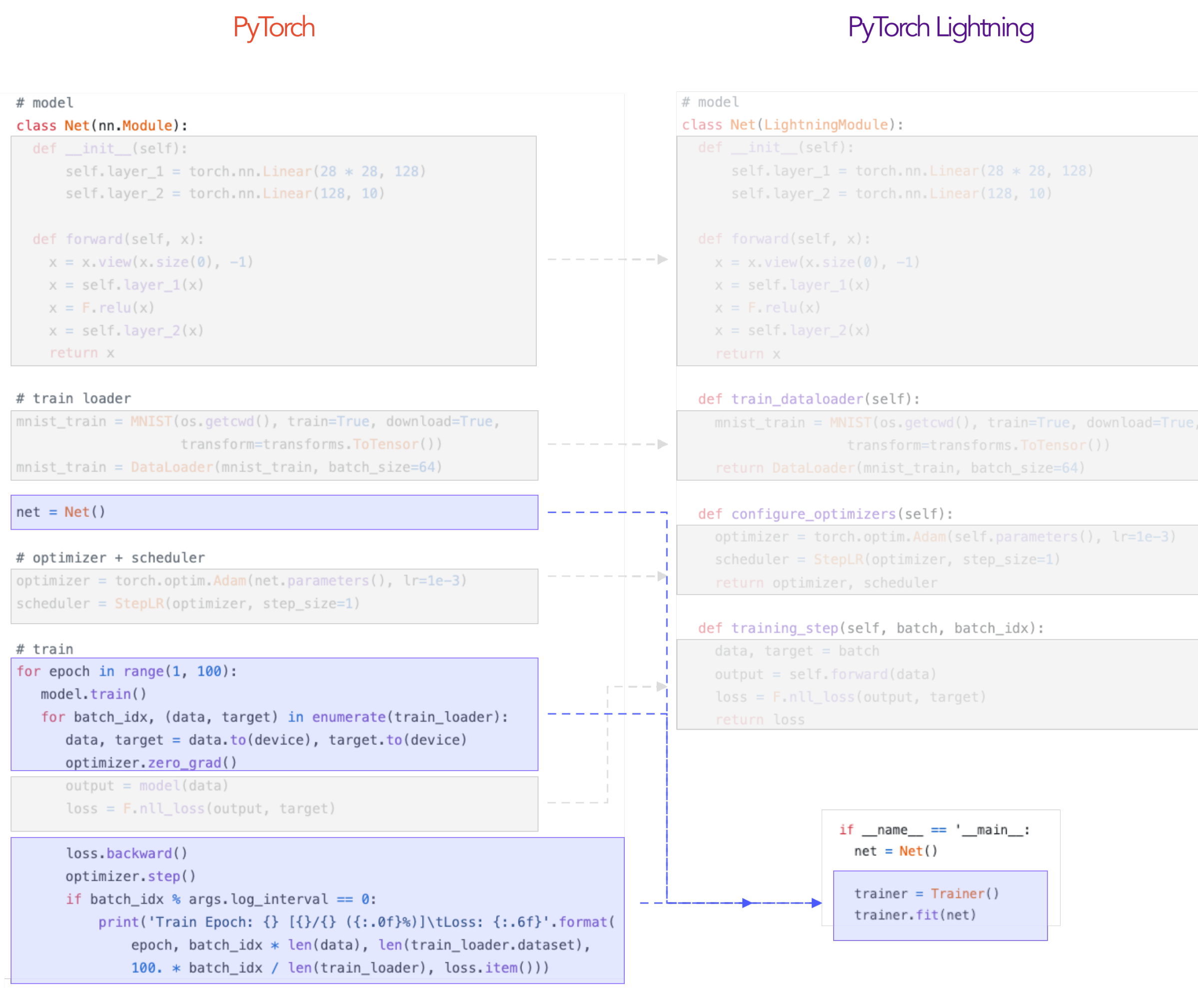
**Return type** None

**summary**()
Create profiler summary in text format.

**Return type** `str`

## 37.7 pytorch_lightning.trainer package

Once you've organized your PyTorch code into a LightningModule, the Trainer automates everything else.




This abstraction achieves the following:

1. You maintain control over all aspects via PyTorch code without an added abstraction.
2. The trainer uses best practices embedded by contributors and users from top AI labs such as Facebook AI Research, NYU, MIT, Stanford, etc. . .
3. The trainer allows overriding any key part that you don't want automated.

### 37.7.1 Basic use

This is the basic use of the trainer:

```
model = MyLightningModule()

trainer = Trainer()
trainer.fit(model)
```

---

### 37.7.2 Best Practices

For cluster computing, it's recommended you structure your main.py file this way

```
from argparse import ArgumentParser

def main(hparams):
    model = LightningModule()
    trainer = Trainer(gpus=hparams.gpus)
    trainer.fit(model)

if __name__ == '__main__':
    parser = ArgumentParser()
    parser.add_argument('--gpus', default=None)
    args = parser.parse_args()

    main(args)
```

So you can run it like so:

```
python main.py --gpus 2
```

**Note:** If you want to stop a training run early, you can press "Ctrl + C" on your keyboard. The trainer will catch the *KeyboardInterrupt* and attempt a graceful shutdown, including running callbacks such as *on_train_end*. The trainer object will also set an attribute *interrupted* to *True* in such cases. If you have a callback which shuts down compute resources, for example, you can conditionally run the shutdown logic for only uninterrupted runs.

---

### 37.7.3 Testing

Once you're done training, feel free to run the test set! (Only right before publishing your paper or pushing to production)

```
trainer.test()
```

---

### 37.7.4 Deployment / prediction

You just trained a LightningModule which is also just a torch.nn.Module. Use it to do whatever!

```
# load model
pretrained_model = LightningModule.load_from_checkpoint(PATH)
pretrained_model.freeze()

# use it for finetuning
def forward(self, x):
    features = pretrained_model(x)
    classes = classifier(features)

# or for prediction
out = pretrained_model(x)
api_write({'response': out}
```

You may wish to run the model on a variety of devices. Instead of moving the data manually to the correct device, decorate the forward method (or any other method you use for inference) with `auto_move_data()` and Lightning will take care of the rest.

---

### 37.7.5 Reproducibility

To ensure full reproducibility from run to run you need to set seeds for pseudo-random generators, and set `deterministic`` flag in `Trainer`.

Example:

```
from pytorch_lightning import Trainer, seed_everything

seed_everything(42)
# sets seeds for numpy, torch, python.random and PYTHONHASHSEED.
model = Model()
trainer = Trainer(deterministic=True)
```

---

### 37.7.6 Trainer flags

#### accumulate_grad_batches

Accumulates grads every k batches or as set up in the dict.

```
# default used by the Trainer (no accumulation)
trainer = Trainer(accumulate_grad_batches=1)
```

Example:

```
# accumulate every 4 batches (effective batch size is batch*4)
trainer = Trainer(accumulate_grad_batches=4)

# no accumulation for epochs 1-4. accumulate 3 for epochs 5-10. accumulate 20 after␣
↪that
trainer = Trainer(accumulate_grad_batches={5: 3, 10: 20})
```

### amp_level

The optimization level to use (O1, O2, etc. . . ) for 16-bit GPU precision (using NVIDIA apex under the hood).

Check NVIDIA apex docs for level

Example:

```
# default used by the Trainer
trainer = Trainer(amp_level='O1')
```

### auto_scale_batch_size

Automatically tries to find the largest batch size that fits into memory, before any training.

```
# default used by the Trainer (no scaling of batch size)
trainer = Trainer(auto_scale_batch_size=None)

# run batch size scaling, result overrides hparams.batch_size
trainer = Trainer(auto_scale_batch_size='binsearch')
```

### auto_lr_find

Runs a learning rate finder algorithm (see this paper) before any training, to find optimal initial learning rate.

```
# default used by the Trainer (no learning rate finder)
trainer = Trainer(auto_lr_find=False)
```

Example:

```
# run learning rate finder, results override hparams.learning_rate
trainer = Trainer(auto_lr_find=True)

# run learning rate finder, results override hparams.my_lr_arg
trainer = Trainer(auto_lr_find='my_lr_arg')
```

**Note:** See the learning rate finder guide

### benchmark

If true enables cudnn.benchmark. This flag is likely to increase the speed of your system if your input sizes don't change. However, if it does, then it will likely make your system slower.

The speedup comes from allowing the cudnn auto-tuner to find the best algorithm for the hardware [see discussion here].

Example:

```
# default used by the Trainer
trainer = Trainer(benchmark=False)
```

### deterministic

If true enables cudnn.deterministic. Might make your system slower, but ensures reproducibility. Also sets `$HOROVOD_FUSION_THRESHOLD=0`.

For more info check [pytorch docs].

Example:

```
# default used by the Trainer
trainer = Trainer(deterministic=False)
```

### callbacks

Add a list of user defined callbacks. These callbacks DO NOT replace the explicit callbacks (loggers, EarlyStopping or ModelCheckpoint).

**Note:** Only user defined callbacks (ie: Not EarlyStopping or ModelCheckpoint)

```
# a list of callbacks
callbacks = [PrintCallback()]
trainer = Trainer(callbacks=callbacks)
```

Example:

```
from pytorch_lightning.callbacks import Callback

class PrintCallback(Callback):
    def on_train_start(self, trainer, pl_module):
        print("Training is started!")
    def on_train_end(self, trainer, pl_module):
        print("Training is done.")
```

### check_val_every_n_epoch

Check val every n train epochs.

Example:

```
# default used by the Trainer
trainer = Trainer(check_val_every_n_epoch=1)

# run val loop every 10 training epochs
trainer = Trainer(check_val_every_n_epoch=10)
```

### checkpoint_callback

Callback for checkpointing.

```
from pytorch_lightning.callbacks import ModelCheckpoint
trainer = Trainer(checkpoint_callback=ModelCheckpoint())
```

Example:

```
from pytorch_lightning.callbacks import ModelCheckpoint

# default used by the Trainer
checkpoint_callback = ModelCheckpoint(
    filepath=os.getcwd(),
    save_top_k=True,
    verbose=True,
    monitor='val_loss',
    mode='min',
    prefix=''
)
```

### default_root_dir

Default path for logs and weights when no logger or *pytorch_lightning.callbacks.ModelCheckpoint* callback passed. On certain clusters you might want to separate where logs and checkpoints are stored. If you don't then use this method for convenience.

Example:

```
# default used by the Trainer
trainer = Trainer(default_root_path=os.getcwd())
```

### distributed_backend

The distributed backend to use.

- (`dp`) is DataParallel (split batch among GPUs of same machine)
- (`ddp`) is DistributedDataParallel (each gpu on each node trains, and syncs grads)
- (`ddp_cpu`) is DistributedDataParallel on CPU (same as *ddp*, but does not use GPUs. Useful for multi-node CPU training or single-node debugging. Note that this will **not** give a speedup on a single node, since Torch already makes effient use of multiple CPUs on a single machine.)
- **(`ddp2`) dp on node, ddp across nodes. Useful for things like increasing** the number of negative samples

```
# default used by the Trainer
trainer = Trainer(distributed_backend=None)
```

Example:

```
# dp = DataParallel
trainer = Trainer(gpus=2, distributed_backend='dp')

# ddp = DistributedDataParallel
trainer = Trainer(gpus=2, num_nodes=2, distributed_backend='ddp')
```

(continues on next page)

(continued from previous page)

```
# ddp2 = DistributedDataParallel + dp
trainer = Trainer(gpus=2, num_nodes=2, distributed_backend='ddp2')
```

**Note:** this option does not apply to TPU. TPUs use `` `ddp` `` by default (over each core)

**See also:**

- Multi-GPU training guide
- Multi-node (SLURM) guide

### early_stop_callback

Callback for early stopping. early_stop_callback (*pytorch_lightning.callbacks.EarlyStopping*)

- **True: A default callback monitoring 'val_loss' is created.** Will raise an error if `'val_loss'` is not found.
- `False`: Early stopping will be disabled.
- `None`: The default callback monitoring `'val_loss'` is created.
- Default: `None`.

```
from pytorch_lightning.callbacks import EarlyStopping

# default used by the Trainer
early_stop = EarlyStopping(
    monitor='val_loss',
    patience=3,
    strict=False,
    verbose=False,
    mode='min'
)
trainer = Trainer(early_stop_callback=early_stop)
```

**Note:** If `'val_loss'` is not found will work as if early stopping is disabled.

### fast_dev_run

Runs 1 batch of train, test and val to find any bugs (ie: a sort of unit test).

Under the hood the pseudocode looks like this:

```
# loading
__init__()
prepare_data

# test training step
training_batch = next(train_dataloader)
training_step(training_batch)
```

(continues on next page)

(continued from previous page)

```
# test val step
val_batch = next(val_dataloader)
out = validation_step(val_batch)
validation_epoch_end([out])
```

```
# default used by the Trainer
trainer = Trainer(fast_dev_run=False)

# runs 1 train, val, test  batch and program ends
trainer = Trainer(fast_dev_run=True)
```

### gpus

- Number of GPUs to train on
- or Which GPUs to train on
- can handle strings

```
# default used by the Trainer (ie: train on CPU)
trainer = Trainer(gpus=None)
```

Example:

```
# int: train on 2 gpus
trainer = Trainer(gpus=2)

# list: train on GPUs 1, 4 (by bus ordering)
trainer = Trainer(gpus=[1, 4])
trainer = Trainer(gpus='1, 4') # equivalent

# -1: train on all gpus
trainer = Trainer(gpus=-1)
trainer = Trainer(gpus='-1') # equivalent

# combine with num_nodes to train on multiple GPUs across nodes
# uses 8 gpus in total
trainer = Trainer(gpus=2, num_nodes=4)
```

**See also:**

- Multi-GPU training guide

### gradient_clip_val

Gradient clipping value

- 0 means don't clip.

```
# default used by the Trainer
trainer = Trainer(gradient_clip_val=0.0)
```

### limit_test_batches

How much of test dataset to check.

```
# default used by the Trainer
trainer = Trainer(limit_test_batches=1.0)

# run through only 25% of the test set each epoch
trainer = Trainer(limit_test_batches=0.25)

# run for only 10 batches
trainer = Trainer(limit_test_batches=10)
```

### limit_val_batches

How much of validation dataset to check. Useful when debugging or testing something that happens at the end of an epoch.

```
# default used by the Trainer
trainer = Trainer(limit_val_batches=1.0)

# run through only 25% of the validation set each epoch
trainer = Trainer(limit_val_batches=0.25)

# run for only 10 batches
trainer = Trainer(limit_val_batches=10)
```

### log_gpu_memory

Options:

- None
- 'min_max'
- 'all'

```
# default used by the Trainer
trainer = Trainer(log_gpu_memory=None)

# log all the GPUs (on master node only)
trainer = Trainer(log_gpu_memory='all')

# log only the min and max memory on the master node
trainer = Trainer(log_gpu_memory='min_max')
```

**Note:** Might slow performance because it uses the output of nvidia-smi.

### log_save_interval

Writes logs to disk this often.

```
# default used by the Trainer
trainer = Trainer(log_save_interval=100)
```

### logger

Logger (or iterable collection of loggers) for experiment tracking.

```
from pytorch_lightning.loggers import TensorBoardLogger

# default logger used by trainer
logger = TensorBoardLogger(
    save_dir=os.getcwd(),
    version=1,
    name='lightning_logs'
)
Trainer(logger=logger)
```

### max_epochs

Stop training once this number of epochs is reached

```
# default used by the Trainer
trainer = Trainer(max_epochs=1000)
```

### min_epochs

Force training for at least these many epochs

```
# default used by the Trainer
trainer = Trainer(min_epochs=1)
```

### max_steps

Stop training after this number of steps Training will stop if max_steps or max_epochs have reached (earliest).

```
# Default (disabled)
trainer = Trainer(max_steps=None)

# Stop after 100 steps
trainer = Trainer(max_steps=100)
```

### min_steps

Force training for at least these number of steps. Trainer will train model for at least min_steps or min_epochs (latest).

```
# Default (disabled)
trainer = Trainer(min_steps=None)

# Run at least for 100 steps (disable min_epochs)
trainer = Trainer(min_steps=100, min_epochs=0)
```

### num_nodes

Number of GPU nodes for distributed training.

```
# default used by the Trainer
trainer = Trainer(num_nodes=1)

# to train on 8 nodes
trainer = Trainer(num_nodes=8)
```

### num_processes

Number of processes to train with. Automatically set to the number of GPUs when using `distrbuted_backend="ddp"`. Set to a number greater than 1 when using `distributed_backend="ddp_cpu"` to mimic distributed training on a machine without GPUs. This is useful for debugging, but **will not** provide any speedup, since single-process Torch already makes effient use of multiple CPUs.

```
# Simulate DDP for debugging on your GPU-less laptop
trainer = Trainer(distributed_backend="ddp_cpu", num_processes=2)
```

### num_sanity_val_steps

Sanity check runs n batches of val before starting the training routine. This catches any bugs in your validation without having to wait for the first validation check. The Trainer uses 5 steps by default. Turn it off or modify it here.

```
# default used by the Trainer
trainer = Trainer(num_sanity_val_steps=5)

# turn it off
trainer = Trainer(num_sanity_val_steps=0)
```

### num_tpu_cores

> **Warning:** Deprecated since version 0.7.6.
>
> Use *tpu_cores* instead. Will remove 0.9.0.

Example:

```
python -m torch_xla.distributed.xla_dist
--tpu=$TPU_POD_NAME
--conda-env=torch-xla-nightly
--env=XLA_USE_BF16=1
-- python your_trainer_file.py
```

### prepare_data_per_node

If True will call *prepare_data()* on LOCAL_RANK=0 for every node. If False will only call from NODE_RANK=0, LOCAL_RANK=0

```
# default
Trainer(prepare_data_per_node=True)

# use only NODE_RANK=0, LOCAL_RANK=0
Trainer(prepare_data_per_node=False)
```

### tpu_cores

- How many TPU cores to train on (1 or 8).
- Which TPU core to train on [1-8]

A single TPU v2 or v3 has 8 cores. A TPU pod has up to 2048 cores. A slice of a POD means you get as many cores as you request.

Your effective batch size is batch_size * total tpu cores.

**Note:** No need to add a DistributedDataSampler, Lightning automatically does it for you.

This parameter can be either 1 or 8.

```
# your_trainer_file.py

# default used by the Trainer (ie: train on CPU)
trainer = Trainer(tpu_cores=None)

# int: train on a single core
trainer = Trainer(tpu_cores=1)

# list: train on a single selected core
trainer = Trainer(tpu_cores=[2])

# int: train on all cores few cores
trainer = Trainer(tpu_cores=8)

# for 8+ cores must submit via xla script with
# a max of 8 cores specified. The XLA script
# will duplicate script onto each TPU in the POD
trainer = Trainer(tpu_cores=8)
```

To train on more than 8 cores (ie: a POD), submit this script using the xla_dist script.

Example:

```
python -m torch_xla.distributed.xla_dist
--tpu=$TPU_POD_NAME
--conda-env=torch-xla-nightly
--env=XLA_USE_BF16=1
-- python your_trainer_file.py
```

### overfit_pct

**Warning:** Deprecated since version 0.8.0..

Use *overfit_batches*. Will remove 1.0.0.

### overfit_batches

Uses this much data of the training set. If will use the same training set for validation and testing. If the training Dataloaders(shuffle=True), Lightning will automatically disable it.

Useful for quickly debugging or trying to overfit on purpose.

```
# default used by the Trainer
trainer = Trainer(overfit_batches=0.0)

# use only 1% of the train set (and use the train set for val and test)
trainer = Trainer(overfit_batches=0.01)

# overfit on 10 of the same batches
trainer = Trainer(overfit_batches=10)
```

### precision

Full precision (32), half precision (16). Can be used on CPU, GPU or TPUs.

If used on TPU will use torch.bfloat16 but tensor printing will still show torch.float32.

```
# default used by the Trainer
trainer = Trainer(precision=32)

# 16-bit precision
trainer = Trainer(precision=16)
```

Example:

```
# one day
trainer = Trainer(precision=8|4|2)
```

### print_nan_grads

**Warning:** Deprecated since version 0.7.2..

Has no effect. When detected, NaN grads will be printed automatically. Will remove 0.9.0.

### process_position

Orders the progress bar. Useful when running multiple trainers on the same node.

```
# default used by the Trainer
trainer = Trainer(process_position=0)
```

**Note:** This argument is ignored if a custom callback is passed to `callbacks`.

### profiler

To profile individual steps during training and assist in identifying bottlenecks.

See the profiler documentation. for more details.

```
from pytorch_lightning.profiler import SimpleProfiler, AdvancedProfiler

# default used by the Trainer
trainer = Trainer(profiler=None)

# to profile standard training events
trainer = Trainer(profiler=True)

# equivalent to profiler=True
trainer = Trainer(profiler=SimpleProfiler())

# advanced profiler for function-level stats
trainer = Trainer(profiler=AdvancedProfiler())
```

### progress_bar_refresh_rate

How often to refresh progress bar (in steps). In notebooks, faster refresh rates (lower number) is known to crash them because of their screen refresh rates, so raise it to 50 or more.

```
# default used by the Trainer
trainer = Trainer(progress_bar_refresh_rate=1)

# disable progress bar
trainer = Trainer(progress_bar_refresh_rate=0)
```

**Note:** This argument is ignored if a custom callback is passed to `callbacks`.

### reload_dataloaders_every_epoch

Set to True to reload dataloaders every epoch.

```
# if False (default)
train_loader = model.train_dataloader()
for epoch in epochs:
    for batch in train_loader:
        ...

# if True
for epoch in epochs:
    train_loader = model.train_dataloader()
    for batch in train_loader:
```

### replace_sampler_ddp

Enables auto adding of distributed sampler.

```
# default used by the Trainer
trainer = Trainer(replace_sampler_ddp=True)
```

By setting to False, you have to add your own distributed sampler:

```
# default used by the Trainer
sampler = torch.utils.data.distributed.DistributedSampler(dataset, shuffle=True)
dataloader = DataLoader(dataset, batch_size=32, sampler=sampler)
```

### resume_from_checkpoint

To resume training from a specific checkpoint pass in the path here.

```
# default used by the Trainer
trainer = Trainer(resume_from_checkpoint=None)

# resume from a specific checkpoint
trainer = Trainer(resume_from_checkpoint='some/path/to/my_checkpoint.ckpt')
```

### row_log_interval

How often to add logging rows (does not write to disk)

```
# default used by the Trainer
trainer = Trainer(row_log_interval=50)
```

use_amp:

> **Warning:** Deprecated since version 0.7.0.
>
> Use *precision* instead. Will remove 0.9.0.

### show_progress_bar

**Warning:** Deprecated since version 0.7.2.

Set *progress_bar_refresh_rate* to 0 instead. Will remove 0.9.0.

### val_percent_check

**Warning:** deprecated in v0.8.0 please use *limit_val_batches*. Will remove in 0.10.0

### test_percent_check

**Warning:** deprecated in v0.8.0 please use *limit_test_batches*. Will remove in 0.10.0

### train_percent_check

**Warning:** deprecated in v0.8.0 please use *limit_train_batches*. Will remove in 0.10.0

### track_grad_norm

- no tracking (-1)
- Otherwise tracks that norm (2 for 2-norm)

```
# default used by the Trainer
trainer = Trainer(track_grad_norm=-1)

# track the 2-norm
trainer = Trainer(track_grad_norm=2)
```

### limit_train_batches

How much of training dataset to check. Useful when debugging or testing something that happens at the end of an epoch.

```
# default used by the Trainer
trainer = Trainer(limit_train_batches=1.0)
```

Example:

```
# default used by the Trainer
trainer = Trainer(limit_train_batches=1.0)

# run through only 25% of the training set each epoch
trainer = Trainer(limit_train_batches=0.25)

# run through only 10 batches of the training set each epoch
trainer = Trainer(limit_train_batches=10)
```

### truncated_bptt_steps

Truncated back prop breaks performs backprop every k steps of a much longer sequence.

If this is enabled, your batches will automatically get truncated and the trainer will apply Truncated Backprop to it.

(Williams et al. "An efficient gradient-based algorithm for on-line training of recurrent network trajectories.")

```
# default used by the Trainer (ie: disabled)
trainer = Trainer(truncated_bptt_steps=None)

# backprop every 5 steps in a batch
trainer = Trainer(truncated_bptt_steps=5)
```

---

**Note:** Make sure your batches have a sequence dimension.

---

Lightning takes care to split your batch along the time-dimension.

```
# we use the second as the time dimension
# (batch, time, ...)
sub_batch = batch[0, 0:t, ...]
```

Using this feature requires updating your LightningModule's `pytorch_lightning.core.LightningModule.training_step()` to include a *hiddens* arg with the hidden

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hiddens from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)

    return {
        "loss": ...,
        "hiddens": hiddens  # remember to detach() this
    }
```

To modify how the batch is split, override `pytorch_lightning.core.LightningModule.tbptt_split_batch()`:

```
class LitMNIST(LightningModule):
    def tbptt_split_batch(self, batch, split_size):
        # do your own splitting on the batch
        return splits
```

### val_check_interval

How often within one training epoch to check the validation set. Can specify as float or int.

- use (float) to check within a training epoch
- use (int) to check every n steps (batches)

```
# default used by the Trainer
trainer = Trainer(val_check_interval=1.0)

# check validation set 4 times during a training epoch
trainer = Trainer(val_check_interval=0.25)
```

(continues on next page)

(continued from previous page)

```
# check validation set every 1000 training batches
# use this when using iterableDataset and your dataset has no length
# (ie: production cases with streaming data)
trainer = Trainer(val_check_interval=1000)
```

### weights_save_path

Directory of where to save weights if specified.

```
# default used by the Trainer
trainer = Trainer(weights_save_path=os.getcwd())

# save to your custom path
trainer = Trainer(weights_save_path='my/path')
```

Example:

```
# if checkpoint callback used, then overrides the weights path
# **NOTE: this saves weights to some/path NOT my/path
checkpoint = ModelCheckpoint(filepath='some/path')
trainer = Trainer(
    checkpoint_callback=checkpoint,
    weights_save_path='my/path'
)
```

### weights_summary

Prints a summary of the weights when training begins. Options: 'full', 'top', None.

```
# default used by the Trainer (ie: print summary of top level modules)
trainer = Trainer(weights_summary='top')

# print full summary of all modules and submodules
trainer = Trainer(weights_summary='full')

# don't print a summary
trainer = Trainer(weights_summary=None)
```

### 37.7.7 Trainer class

**class** pytorch_lightning.trainer.**Trainer**(*logger=True, checkpoint_callback=True, early_stop_callback=False, callbacks=None, default_root_dir=None, gradient_clip_val=0, process_position=0, num_nodes=1, num_processes=1, gpus=None, auto_select_gpus=False, tpu_cores=None, log_gpu_memory=None, progress_bar_refresh_rate=1, overfit_batches=0.0, track_grad_norm=-1, check_val_every_n_epoch=1, fast_dev_run=False, accumulate_grad_batches=1, max_epochs=1000, min_epochs=1, max_steps=None, min_steps=None, limit_train_batches=1.0, limit_val_batches=1.0, limit_test_batches=1.0, val_check_interval=1.0, log_save_interval=100, row_log_interval=50, distributed_backend=None, precision=32, print_nan_grads=False, weights_summary='top', weights_save_path=None, num_sanity_val_steps=2, truncated_bptt_steps=None, resume_from_checkpoint=None, profiler=None, benchmark=False, deterministic=False, reload_dataloaders_every_epoch=False, auto_lr_find=False, replace_sampler_ddp=True, terminate_on_nan=False, auto_scale_batch_size=False, prepare_data_per_node=True, amp_level='O1', num_tpu_cores=None, use_amp=None, show_progress_bar=None, val_percent_check=None, test_percent_check=None, train_percent_check=None, overfit_pct=None*)

Bases: *pytorch_lightning.trainer.training_io.TrainerIOMixin*, *pytorch_lightning.trainer.optimizers.TrainerOptimizersMixin*, *pytorch_lightning.trainer.auto_mix_precision.TrainerAMPMixin*, *pytorch_lightning.trainer.distrib_parts.TrainerDPMixin*, *pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin*, *pytorch_lightning.trainer.logging.TrainerLoggingMixin*, *pytorch_lightning.trainer.model_hooks.TrainerModelHooksMixin*, *pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixin*, *pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin*, *pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin*, *pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin*, *pytorch_lightning.trainer.callback_config.TrainerCallbackConfigMixin*, *pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin*, *pytorch_lightning.trainer.lr_finder.TrainerLRFinderMixin*, *pytorch_lightning.trainer.deprecated_api.TrainerDeprecatedAPITillVer0_9*, *pytorch_lightning.trainer.deprecated_api.TrainerDeprecatedAPITillVer0_10*

Customize every aspect of training via flags

**Parameters**

- **logger** (`Union`[*LightningLoggerBase*, `Iterable`[*LightningLoggerBase*], `bool`]) – Logger (or iterable collection of loggers) for experiment tracking.
- **checkpoint_callback** (`Union`[*ModelCheckpoint*, `bool`]) – Callback for checkpointing.
- **early_stop_callback** (*pytorch_lightning.callbacks.EarlyStopping*) –
- **callbacks** (`Optional`[`List`[*Callback*]]) – Add a list of callbacks.
- **default_root_dir** (`Optional`[`str`]) – Default path for logs and weights when no logger/ckpt_callback passed
- **gradient_clip_val** (`float`) – 0 means don't clip.
- **gradient_clip** –

  **Warning:** Deprecated since version 0.7.0.

  Use *gradient_clip_val* instead. Will remove 0.9.0.

- **process_position** (`int`) – orders the progress bar when running multiple models on same machine.
- **num_nodes** (`int`) – number of GPU nodes for distributed training.
- **nb_gpu_nodes** –

  **Warning:** Deprecated since version 0.7.0.

  Use *num_nodes* instead. Will remove 0.9.0.

- **gpus** (`Union`[`int`, `str`, `List`[`int`], `None`]) – Which GPUs to train on.
- **auto_select_gpus** (`bool`) – If enabled and *gpus* is an integer, pick available gpus automatically. This is especially useful when GPUs are configured to be in "exclusive mode", such that only one process at a time can access them.
- **tpu_cores** (`Union`[`List`[`int`], `int`, `None`]) – How many TPU cores to train on (1 or 8) / Single TPU to train on [1]
- **num_tpu_cores** (`Optional`[`int`]) – How many TPU cores to train on (1 or 8) .. warning:: .. deprecated:: 0.7.6. Will remove 0.9.0.
- **log_gpu_memory** (`Optional`[`str`]) – None, 'min_max', 'all'. Might slow performance
- **show_progress_bar** –

  **Warning:** Deprecated since version 0.7.2.

  Set *progress_bar_refresh_rate* to positive integer to enable. Will remove 0.9.0.

- **progress_bar_refresh_rate** (`int`) – How often to refresh progress bar (in steps). Value `0` disables progress bar. Ignored when a custom callback is passed to *callbacks*.

- **overfit_batches** (Union[int, float]) – Overfit a percent of training data (float) or a set number of batches (int).
- **overfit_pct** (Optional[float]) –

> **Warning:** Deprecated since version 0.8.0.
>
> Use *overfit_batches* instead. Will remove 0.10.0.

- **track_grad_norm** (Union[int, float, str]) – -1 no tracking. Otherwise tracks that p-norm. May be set to 'inf' infinity-norm.
- **check_val_every_n_epoch** (int) – Check val every n train epochs.
- **fast_dev_run** (bool) – runs 1 batch of train, test and val to find any bugs (ie: a sort of unit test).
- **accumulate_grad_batches** (Union[int, Dict[int, int], List[list]]) – Accumulates grads every k batches or as set up in the dict.
- **max_epochs** (int) – Stop training once this number of epochs is reached.
- **max_nb_epochs** –

> **Warning:** Deprecated since version 0.7.0.
>
> Use *max_epochs* instead. Will remove 0.9.0.

- **min_epochs** (int) – Force training for at least these many epochs
- **min_nb_epochs** –

> **Warning:** Deprecated since version 0.7.0.
>
> Use *min_epochs* instead. Will remove 0.9.0.

- **max_steps** (Optional[int]) – Stop training after this number of steps. Disabled by default (None).
- **min_steps** (Optional[int]) – Force training for at least these number of steps. Disabled by default (None).
- **limit_train_batches** (Union[int, float]) – How much of training dataset to check.
- **limit_val_batches** (Union[int, float]) – How much of validation dataset to check (floats = percent, int = num_batches)
- **limit_test_batches** (Union[int, float]) – How much of test dataset to check (floats = percent, int = num_batches)
- **train_percent_check** (Optional[float]) –

> **Warning:** Deprecated since version 0.8.0.
>
> Use *limit_train_batches* instead. Will remove v0.10.0.

- **val_percent_check** (Optional[float]) –

  > **Warning:** Deprecated since version 0.8.0.
  >
  > Use *limit_val_batches* instead. Will remove v0.10.0.

- **test_percent_check** (Optional[float]) –

  > **Warning:** Deprecated since version 0.8.0.
  >
  > Use *limit_test_batches* instead. Will remove v0.10.0.

- **val_check_interval** (Union[int, float]) – How often within one training epoch to check the validation set
- **log_save_interval** (int) – Writes logs to disk this often
- **row_log_interval** (int) – How often to add logging rows (does not write to disk)
- **add_row_log_interval** –

  > **Warning:** Deprecated since version 0.7.0.
  >
  > Use *row_log_interval* instead. Will remove 0.9.0.

- **distributed_backend** (Optional[str]) – The distributed backend to use (dp, ddp, ddp2, ddp_spawn)
- **use_amp** –

  > **Warning:** Deprecated since version 0.7.0.
  >
  > Use *precision* instead. Will remove 0.9.0.

- **precision** (int) – Full precision (32), half precision (16).
- **print_nan_grads** (bool) –

  > **Warning:** Deprecated since version 0.7.2.
  >
  > Has no effect. When detected, NaN grads will be printed automatically. Will remove 0.9.0.

- **weights_summary** (Optional[str]) – Prints a summary of the weights when training begins.
- **weights_save_path** (Optional[str]) – Where to save weights if specified. Will override default_root_dir for checkpoints only. Use this if for whatever reason you need the checkpoints stored in a different place than the logs written in *default_root_dir*.
- **amp_level** (str) – The optimization level to use (O1, O2, etc…).
- **num_sanity_val_steps** (int) – Sanity check runs n batches of val before starting the training routine.

- **truncated_bptt_steps** (Optional[int]) – Truncated back prop breaks performs backprop every k steps of
- **resume_from_checkpoint** (Optional[str]) – To resume training from a specific checkpoint pass in the path here. This can be a URL.
- **profiler** (Union[*BaseProfiler*, bool, None]) – To profile individual steps during training and assist in
- **reload_dataloaders_every_epoch** (bool) – Set to True to reload dataloaders every epoch
- **auto_lr_find** (Union[bool, str]) – If set to True, will *initially* run a learning rate finder, trying to optimize initial learning for faster convergence. Sets learning rate in self.lr or self.learning_rate in the LightningModule. To use a different key, set a string instead of True with the key name.
- **replace_sampler_ddp** (bool) – Explicitly enables or disables sampler replacement. If not specified this will toggled automatically ddp is used
- **benchmark** (bool) – If true enables cudnn.benchmark.
- **deterministic** (bool) – If true enables cudnn.deterministic
- **terminate_on_nan** (bool) – If set to True, will terminate training (by raising a *ValueError*) at the end of each training batch, if any of the parameters or the loss are NaN or +/-inf.
- **auto_scale_batch_size** (Union[str, bool]) – If set to True, will *initially* run a batch size finder trying to find the largest batch size that fits into memory. The result will be stored in self.batch_size in the LightningModule. Additionally, can be set to either *power* that estimates the batch size through a power search or *binsearch* that estimates the batch size through a binary search.
- **prepare_data_per_node** (bool) – If True, each LOCAL_RANK=0 will call prepare data. Otherwise only NODE_RANK=0, LOCAL_RANK=0 will prepare data

**_Trainer__attach_dataloaders**(*model*, *train_dataloader=None*, *val_dataloaders=None*, *test_dataloaders=None*)

**_allowed_type**()

**Return type** Union[int, str]

**_arg_default**()

**Return type** Union[int, str]

**classmethod add_argparse_args**(*parent_parser*)

Extends existing argparse by default *Trainer* attributes.

**Parameters parent_parser** (ArgumentParser) – The custom cli arguments parser, which will be extended by the Trainer default arguments.

Only arguments of the allowed types (str, float, int, bool) will extend the *parent_parser*.

### Examples

```
>>> import argparse
>>> import pprint
>>> parser = argparse.ArgumentParser()
>>> parser = Trainer.add_argparse_args(parser)
>>> args = parser.parse_args([])
>>> pprint.pprint(vars(args))
{...
 'check_val_every_n_epoch': 1,
 'checkpoint_callback': True,
 'default_root_dir': None,
 'deterministic': False,
 'distributed_backend': None,
 'early_stop_callback': False,
 ...
 'logger': True,
 'max_epochs': 1000,
 'max_steps': None,
 'min_epochs': 1,
 'min_steps': None,
 ...
 'profiler': None,
 'progress_bar_refresh_rate': 1,
 ...}
```

**Return type** `ArgumentParser`

**`barrier`**(*name*)

**`can_prepare_data`**()

**`check_model_configuration`**(*model*)
Checks that the model is configured correctly before training or testing is started.

**Parameters** **`model`** (*`LightningModule`*) – The model to check the configuration.

**`classmethod default_attributes`**()

**`fit`**(*model*, *train_dataloader=None*, *val_dataloaders=None*)
Runs the full optimization routine.

**Parameters**

- **`model`** (*`LightningModule`*) – Model to fit.
- **`train_dataloader`** (`Optional[DataLoader]`) – A Pytorch DataLoader with training samples. If the model has a predefined train_dataloader method this will be skipped.
- **`val_dataloaders`** (`Union[DataLoader, List[DataLoader], None]`) – Either a single Pytorch Dataloader or a list of them, specifying validation samples. If the model has a predefined val_dataloaders method this will be skipped

Example:

```
# Option 1,
# Define the train_dataloader() and val_dataloader() fxs
# in the lightningModule
# RECOMMENDED FOR MOST RESEARCH AND APPLICATIONS TO MAINTAIN READABILITY
```

(continues on next page)

(continued from previous page)

```
trainer = Trainer()
model = LightningModule()
trainer.fit(model)

# Option 2
# in production cases we might want to pass different datasets to the same␣
↪model
# Recommended for PRODUCTION SYSTEMS
train, val = DataLoader(...), DataLoader(...)
trainer = Trainer()
model = LightningModule()
trainer.fit(model, train_dataloader=train, val_dataloaders=val)

# Option 1 & 2 can be mixed, for example the training set can be
# defined as part of the model, and validation can then be feed to .fit()
```

**classmethod from_argparse_args**(*args*, ***kwargs*)

Create an instance from CLI arguments.

**Parameters**

- **args** (Union[Namespace, ArgumentParser]) – The parser or namespace to take arguments from. Only known arguments will be parsed and passed to the *Trainer*.
- ****kwargs** – Additional keyword arguments that may override ones in the parser or namespace. These must be valid Trainer arguments.

### Example

```
>>> parser = ArgumentParser(add_help=False)
>>> parser = Trainer.add_argparse_args(parser)
>>> parser.add_argument('--my_custom_arg', default='something')
>>> args = Trainer.parse_argparser(parser.parse_args(""))
>>> trainer = Trainer.from_argparse_args(args, logger=False)
```

**Return type** *Trainer*

**classmethod get_deprecated_arg_names**()

Returns a list with deprecated Trainer arguments.

**Return type** List

**classmethod get_init_arguments_and_types**()

Scans the Trainer signature and returns argument names, types and default values.

**Returns** (argument name, set with argument types, argument default value).

**Return type** List with tuples of 3 values

### Examples

```
>>> args = Trainer.get_init_arguments_and_types()
>>> import pprint
>>> pprint.pprint(sorted(args))
[('accumulate_grad_batches',
  (<class 'int'>, typing.Dict[int, int], typing.List[list]),
  1),
 ...
 ('callbacks',
  (typing.List[pytorch_lightning.callbacks.base.Callback],
   <class 'NoneType'>),
   None),
 ('check_val_every_n_epoch', (<class 'int'>,), 1),
 ...
 ('max_epochs', (<class 'int'>,), 1000),
 ...
 ('precision', (<class 'int'>,), 32),
 ('prepare_data_per_node', (<class 'bool'>,), True),
 ('print_nan_grads', (<class 'bool'>,), False),
 ('process_position', (<class 'int'>,), 0),
 ('profiler',
  (<class 'pytorch_lightning.profiler.profilers.BaseProfiler'>,
   <class 'bool'>,
   <class 'NoneType'>),
  None),
 ...
```

**static parse_argparser**(*arg_parser*)

Parse CLI arguments, required for custom bool types.

**Return type** `Namespace`

**run_pretrain_routine**(*model*)

Sanity check a few things before starting actual training.

**Parameters** **model** (*LightningModule*) – The model to run sanity test on.

**test**(*model=None*, *test_dataloaders=None*, *ckpt_path='best'*)

Separates from fit to make sure you never run on your test set until you want to.

**Parameters**

- **model** (`Optional`[*LightningModule*]) – The model to test.
- **test_dataloaders** (`Union`[`DataLoader`, `List`[`DataLoader`], `None`]) – Either a single Pytorch Dataloader or a list of them, specifying validation samples.
- **ckpt_path** (`Optional`[`str`]) – Either `best` or path to the checkpoint you wish to test. If `None`, use the weights from the last epoch to test. Default to `best`.

Example:

```
# Option 1
# run test with the best checkpoint from ``ModelCheckpoint`` after fitting.
test = DataLoader(...)
trainer = Trainer()
model = LightningModule()

trainer.fit(model)
```

(continues on next page)

(continued from previous page)

```
trainer.test(test_dataloaders=test)

# Option 2
# run test with the specified checkpoint after fitting
test = DataLoader(...)
trainer = Trainer()
model = LightningModule()

trainer.fit(model)
trainer.test(test_dataloaders=test, ckpt_path='path/to/checkpoint.ckpt')

# Option 3
# run test with the weights from the end of training after fitting
test = DataLoader(...)
trainer = Trainer()
model = LightningModule()

trainer.fit(model)
trainer.test(test_dataloaders=test, ckpt_path=None)

# Option 4
# run test from a loaded model. ``ckpt_path`` is ignored in this case.
test = DataLoader(...)
model = LightningModule.load_from_checkpoint('path/to/checkpoint.ckpt')
trainer = Trainer()
trainer.test(model, test_dataloaders=test)
```

**DEPRECATED_IN_0_9 = ('use_amp', 'show_progress_bar', 'training_tqdm_dict', 'num_tpu_co**

**accumulate_grad_batches = None**

**checkpoint_callback = None**

**property data_parallel**

**Return type** `bool`

**early_stop_callback = None**

**global_rank = None**

**property is_global_zero**

this is just empty shell for code implemented in other class.

**Type** Warning

**logger = None**

**lr_schedulers = None**

**model = None**

**property num_gpus**

this is just empty shell for code implemented in other class.

**Type** Warning

**Return type** `int`

**num_training_batches = None**

**on_gpu = None**

**on_tpu = None**

**optimizers = None**

**property progress_bar_callback**

**property progress_bar_dict**
Read-only for progress bar metrics.

**Return type** dict

**resume_from_checkpoint = None**

**root_gpu = None**

**scaler = None**

**property slurm_job_id**
this is just empty shell for code implemented in other class.

**Type** Warning

**Return type** Optional[int]

**use_ddp = None**

**use_ddp2 = None**

**use_horovod = None**

**use_native_amp = None**

**weights_save_path = None**

pytorch_lightning.trainer.**seed_everything**(*seed=None*)
Function that sets seed for pseudo-random number generators in: pytorch, numpy, python.random and sets PYTHONHASHSEED environment variable.

**Return type** int

### 37.7.8 Submodules

#### pytorch_lightning.trainer.auto_mix_precision module

**class** pytorch_lightning.trainer.auto_mix_precision.**TrainerAMPMixin**
Bases: abc.ABC

**init_amp**(*use_amp*)

**precision: int = None**

**property use_amp**

**Return type** bool

**use_native_amp: bool = None**

### pytorch_lightning.trainer.callback_config module

**class** pytorch_lightning.trainer.callback_config.**TrainerCallbackConfigMixin**
Bases: abc.ABC

**configure_checkpoint_callback**()
Weight path set in this priority: Checkpoint_callback's path (if passed in). User provided weights_saved_path Otherwise use os.getcwd()

**configure_early_stopping**(*early_stop_callback*)

**configure_progress_bar**(*refresh_rate=1*, *process_position=0*)

**abstract is_overridden**(**args*)
Warning: this is just empty shell for code implemented in other class.

**abstract save_checkpoint**(**args*)
Warning: this is just empty shell for code implemented in other class.

**callbacks: List[Callback] = None**

**checkpoint_callback: Optional[ModelCheckpoint] = None**

**ckpt_path: str = None**

**default_root_dir: str = None**

**logger: LightningLoggerBase = None**

**abstract property slurm_job_id**
this is just empty shell for code implemented in other class.

**Type** Warning

**Return type** int

**weights_save_path: Optional[str] = None**

### pytorch_lightning.trainer.callback_hook module

**class** pytorch_lightning.trainer.callback_hook.**TrainerCallbackHookMixin**
Bases: abc.ABC

**on_batch_end**()
Called when the training batch ends.

**on_batch_start**()
Called when the training batch begins.

**on_epoch_end**()
Called when the epoch ends.

**on_epoch_start**()
Called when the epoch begins.

**on_fit_end**()
Called when the trainer initialization begins, model has not yet been set.

**on_fit_start**()
Called when the trainer initialization begins, model has not yet been set.

**on_init_end**()
Called when the trainer initialization ends, model has not yet been set.

**on_init_start**()
Called when the trainer initialization begins, model has not yet been set.

**on_keyboard_interrupt**()
Called when the training is interrupted by KeyboardInterrupt.

**on_sanity_check_end**()
Called when the validation sanity check ends.

**on_sanity_check_start**()
Called when the validation sanity check starts.

**on_test_batch_end**()
Called when the test batch ends.

**on_test_batch_start**()
Called when the test batch begins.

**on_test_end**()
Called when the test ends.

**on_test_start**()
Called when the test begins.

**on_train_end**()
Called when the train ends.

**on_train_start**()
Called when the train begins.

**on_validation_batch_end**()
Called when the validation batch ends.

**on_validation_batch_start**()
Called when the validation batch begins.

**on_validation_end**()
Called when the validation loop ends.

**on_validation_start**()
Called when the validation loop begins.

**setup**(*stage*)
Called in the beginning of fit and test

**teardown**(*stage*)
Called at the end of fit and test

**callbacks: List[pytorch_lightning.callbacks.base.Callback] = []**

**get_model: Callable = Ellipsis**

## pytorch_lightning.trainer.data_loading module

**class** pytorch_lightning.trainer.data_loading.**TrainerDataLoadingMixin**
Bases: abc.ABC

**_check_batch_limits**(*name*)
**Return type** None

**_get_distributed_sampler**(*dataloader*)

**_reset_eval_dataloader**(*model*, *mode*)
Generic method to reset a dataloader for evaluation.

**Parameters**

- **model** (*LightningModule*) – The current *LightningModule*
- **mode** (str) – Either *'val'* or *'test'*

**Return type** Tuple[List[Union[int, float]], List[DataLoader]]

**Returns** Tuple (num_batches, dataloaders)

**_worker_check**(*dataloader*, *name*)
**Return type** None

**auto_add_sampler**(*dataloader*, *train*)
**Return type** DataLoader

**determine_data_use_amount**(*overfit_batches*)
Use less data for debugging purposes

**Return type** None

**abstract is_overridden**(**args*)
Warning: this is just empty shell for code implemented in other class.

**replace_sampler**(*dataloader*, *sampler*)

**request_dataloader**(*dataloader_fx*)
Handles downloading data in the GPU or TPU case.

**Parameters dataloader_fx** (Callable) – The bound dataloader getter

**Return type** DataLoader

**Returns** The dataloader

**reset_test_dataloader**(*model*)
Resets the validation dataloader and determines the number of batches.

**Parameters model** – The current *LightningModule*

**Return type** None

**reset_train_dataloader**(*model*)
Resets the train dataloader and initialises required variables (number of batches, when to validate, etc.).

**Parameters model** (*LightningModule*) – The current *LightningModule*

**Return type** None

**reset_val_dataloader**(*model*)
Resets the validation dataloader and determines the number of batches.

**Parameters** **model** (*LightningModule*) – The current *LightningModule*

**Return type** None

**distributed_backend: Optional[str] = None**

**global_rank: int = None**

**limit_test_batches: Union[int, float] = None**

**limit_train_batches: Union[int, float] = None**

**limit_val_batches: Union[int, float] = None**

**num_nodes: int = None**

**num_processes: int = None**

**num_test_batches: List[Union[int, float]] = None**

**num_training_batches: Union[int, float] = None**

**num_val_batches: List[Union[int, float]] = None**

**replace_sampler_ddp: bool = None**

**shown_warnings: ... = None**

**test_dataloaders: List[DataLoader] = None**

**tpu_local_core_rank: int = None**

**train_dataloader: DataLoader = None**

**use_ddp: bool = None**

**use_ddp2: bool = None**

**use_horovod: bool = None**

**use_tpu: bool = None**

**val_check_batch: ... = None**

**val_check_interval: float = None**

**val_dataloaders: List[DataLoader] = None**

pytorch_lightning.trainer.data_loading.**_has_len**(*dataloader*)

Checks if a given Dataloader has __len__ method implemented i.e. if it is a finite dataloader or infinite dataloader

**Return type** bool

### pytorch_lightning.trainer.deprecated_api module

Mirroring deprecated API

**class** pytorch_lightning.trainer.deprecated_api.**TrainerDeprecatedAPITillVer0_10**

Bases: abc.ABC

**limit_test_batches: Union[int, float] = None**

**limit_train_batches: Union[int, float] = None**

**limit_val_batches: Union[int, float] = None**

**overfit_batches: Union[int, float] = None**

**property overfit_pct**
Back compatibility, will be removed in v0.10.0

**Return type** Union[int, float]

**property test_percent_check**
Back compatibility, will be removed in v0.10.0

**Return type** Union[int, float]

**property train_percent_check**
Back compatibility, will be removed in v0.10.0

**Return type** Union[int, float]

**property val_percent_check**
Back compatibility, will be removed in v0.10.0

**Return type** Union[int, float]

**class** pytorch_lightning.trainer.deprecated_api.**TrainerDeprecatedAPITillVer0_9**
Bases: abc.ABC

**property num_tpu_cores**
Back compatibility, will be removed in v0.9.0

**progress_bar_callback: ... = None**

**progress_bar_dict: ... = None**

**property show_progress_bar**
Back compatibility, will be removed in v0.9.0

**property training_tqdm_dict**
Back compatibility, will be removed in v0.9.0

### pytorch_lightning.trainer.distrib_data_parallel module

Lightning supports model training on a cluster managed by SLURM in the following cases:

1. Training on a single cpu or single GPU.
2. Train on multiple GPUs on the same node using DataParallel or DistributedDataParallel
3. Training across multiple GPUs on multiple different nodes via DistributedDataParallel.

**Note:** A node means a machine with multiple GPUs

#### Running grid search on a cluster

To use lightning to run a hyperparameter search (grid-search or random-search) on a cluster do 4 things:

(1). Define the parameters for the grid search

```
from test_tube import HyperOptArgumentParser

# subclass of argparse
parser = HyperOptArgumentParser(strategy='random_search')
parser.add_argument('--learning_rate', default=0.002, type=float, help='the learning␣
↪rate')
```

(continues on next page)

(continued from previous page)

```
# let's enable optimizing over the number of layers in the network
parser.opt_list('--nb_layers', default=2, type=int, tunable=True, options=[2, 4, 8])

hparams = parser.parse_args()
```

**Note:** You must set *Tunable=True* for that argument to be considered in the permutation set. Otherwise test-tube will use the default value. This flag is useful when you don't want to search over an argument and want to use the default instead.

**(2). Define the cluster options in the** SlurmCluster object (over 5 nodes and 8 gpus)

```
from test_tube.hpc import SlurmCluster

# hyperparameters is a test-tube hyper params object
# see https://williamfalcon.github.io/test-tube/hyperparameter_optimization/
↪HyperOptArgumentParser/
hyperparams = args.parse()

# init cluster
cluster = SlurmCluster(
    hyperparam_optimizer=hyperparams,
    log_path='/path/to/log/results/to',
    python_cmd='python3'
)

# let the cluster know where to email for a change in job status (ie: complete, fail,␣
↪etc...)
cluster.notify_job_status(email='some@email.com', on_done=True, on_fail=True)

# set the job options. In this instance, we'll run 20 different models
# each with its own set of hyperparameters giving each one 1 GPU (ie: taking up 20␣
↪GPUs)
cluster.per_experiment_nb_gpus = 8
cluster.per_experiment_nb_nodes = 5

# we'll request 10GB of memory per node
cluster.memory_mb_per_node = 10000

# set a walltime of 10 minues
cluster.job_time = '10:00'
```

(3). Make a main function with your model and trainer. Each job will call this function with a particular hparams configuration.:

```
from pytorch_lightning import Trainer

def train_fx(trial_hparams, cluster_manager, _):
    # hparams has a specific set of hyperparams

    my_model = MyLightningModel()

    # give the trainer the cluster object
    trainer = Trainer()
```

(continues on next page)

(continued from previous page)

```
    trainer.fit(my_model)

`
```

(4). Start the grid/random search:

```
# run the models on the cluster
cluster.optimize_parallel_cluster_gpu(
    train_fx,
    nb_trials=20,
    job_name='my_grid_search_exp_name',
    job_display_name='my_exp')
```

**Note:** *nb_trials* specifies how many of the possible permutations to use. If using *grid_search* it will use the depth first ordering. If using *random_search* it will use the first k shuffled options. FYI, random search has been shown to be just as good as any Bayesian optimization method when using a reasonable number of samples (60), see this paper for more information.

### Walltime auto-resubmit

Lightning automatically resubmits jobs when they reach the walltime. Make sure to set the SIGUSR1 signal in your SLURM script.:

```
# 90 seconds before training ends
#SBATCH --signal=SIGUSR1@90
```

When lightning receives the SIGUSR1 signal it will: 1. save a checkpoint with 'hpc_ckpt' in the name. 2. resubmit the job using the SLURM_JOB_ID

When the script starts again, Lightning will: 1. search for a 'hpc_ckpt' checkpoint. 2. restore the model, optimizers, schedulers, epoch, etc...

**class** pytorch_lightning.trainer.distrib_data_parallel.**TrainerDDPMixin**

Bases: abc.ABC

**_set_horovod_backend**()

**check_horovod**()

Raises a *MisconfigurationException* if the Trainer is not configured correctly for Horovod.

**configure_slurm_ddp**(*num_gpu_nodes*)

**abstract copy_trainer_model_properties**(**args*)

Warning: this is just empty shell for code implemented in other class.

**ddp_train**(*process_idx*, *model*, *is_master=False*, *proc_offset=0*)

Entry point into a DP thread :param _sphinx_paramlinks_pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin.d
:param _sphinx_paramlinks_pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin.ddp_train.model:
:param _sphinx_paramlinks_pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin.ddp_train.cluster_obj:
:return:

**determine_ddp_node_rank**()

**determine_local_rank**()

**static has_horovodrun**()
Returns True if running with *horovodrun* using Gloo or OpenMPI.

**abstract init_optimizers**(**args*)
Warning: this is just empty shell for code implemented in other class.

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

**init_tpu**()

**load_spawn_weights**(*original_model*)
Load the temp weights saved in the process To recover the trained model from the ddp process we load the saved weights :param _sphinx_paramlinks_pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin.load_spawn_weights.model: :return:

**abstract reinit_scheduler_properties**(**args*)
Warning: this is just empty shell for code implemented in other class.

**resolve_root_node_address**(*root_node*)

**abstract run_pretrain_routine**(**args*)
Warning: this is just empty shell for code implemented in other class.

**abstract save_checkpoint**(**args*)
Warning: this is just empty shell for code implemented in other class.

**save_spawn_weights**(*model*)
Dump a temporary checkpoint after ddp ends to get weights out of the process :param _sphinx_paramlinks_pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin.save_spawn_weights.model: :return:

**set_distributed_mode**(*distributed_backend*)

**set_nvidia_flags**(*is_slurm_managing_tasks*, *data_parallel_device_ids*)

**set_random_port**()
When running DDP NOT managed by SLURM, the ports might collide

**spawn_ddp_children**(*model*)

**amp_level: str = None**

**checkpoint_callback: Union[ModelCheckpoint, bool] = None**

**data_parallel_device_ids: ... = None**

**default_root_dir: str = None**

**distributed_backend: Optional[str] = None**

**gpus: List[int] = None**

**property is_global_zero**
this is just empty shell for code implemented in other class.

**Type** Warning

**Return type** int

**logger: Union[LightningLoggerBase, bool] = None**

**node_rank: int = None**

**num_gpu_nodes: int = None**

**abstract property num_gpus**
this is just empty shell for code implemented in other class.

**Type** Warning

**Return type** `int`

**num_nodes: int = None**

**num_processes: int = None**

**on_gpu: bool = None**

**progress_bar_callback: ... = None**

**tpu_cores: int = None**

**abstract property use_amp**
this is just empty shell for code implemented in other class.

**Type** Warning

**Return type** `bool`

**use_native_amp: bool = None**

**use_tpu: bool = None**

## pytorch_lightning.trainer.distrib_parts module

Root module for all distributed operations in Lightning. Currently supports training on CPU, GPU (dp, ddp, ddp2, horovod) and TPU.

**class** `pytorch_lightning.trainer.distrib_parts.`**TrainerDPMixin**
Bases: `abc.ABC`

**_TrainerDPMixin__transfer_batch_to_device**(*batch*, *device*)

**copy_trainer_model_properties**(*model*)

**dp_train**(*model*)

**abstract get_model**()
Warning: this is just empty shell for code implemented in other class.

**Return type** *LightningModule*

**horovod_train**(*model*)

**abstract init_optimizers**(**args*)
Warning: this is just empty shell for code implemented in other class.

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

**abstract reinit_scheduler_properties**(**args*)
Warning: this is just empty shell for code implemented in other class.

**abstract run_pretrain_routine**(**args*)
Warning: this is just empty shell for code implemented in other class.

**single_gpu_train**(*model*)

**tpu_train**(*tpu_core_idx*, *model*)

**transfer_batch_to_gpu**(*batch*, *gpu_id=None*)
Transfers the data to the GPU.

**Parameters**

- **batch** (Any) – A tensor or collection of tensors.
- **gpu_id** (Optional[int]) – The id of the GPU device. If omitted, the first available GPU is chosen.

**Returns** the tensor on the GPU device.

**See also:**

- *move_data_to_device()*

**transfer_batch_to_tpu** (*batch*, *tpu_id=None*)
Transfers the data to the TPU.

**Parameters**

- **batch** (Any) – A tensor or collection of tensors.
- **tpu_id** (Optional[int]) – The id of the TPU core. If omitted, the first available core is chosen.

**Returns** the tensor on the TPU device.

**See also:**

- *move_data_to_device()*

**amp_level: str = None**

**data_parallel_device_ids: ... = None**

**global_rank: int = None**

**on_colab_kaggle: str = None**

**on_gpu: bool = None**

**precision: ... = None**

**progress_bar_callback: ... = None**

**root_gpu: ... = None**

**save_spawn_weights: Callable = None**

**single_gpu: bool = None**

**testing: bool = None**

**tpu_global_core_rank: int = None**

**tpu_id: Optional[int] = None**

**tpu_local_core_rank: int = None**

**abstract property use_amp**
this is just empty shell for code implemented in other class.

**Type** Warning

**Return type** bool

**use_ddp: bool = None**

**use_ddp2: bool = None**

**use_dp: bool = None**

**use_native_amp: bool = None**

**use_tpu: bool = None**

pytorch_lightning.trainer.distrib_parts.**check_gpus_data_type**(*gpus*)
Checks that the gpus argument is one of: None, Int, String or List. Raises a MisconfigurationException otherwise.

**Parameters** **gpus** (Any) – parameter as passed to the Trainer

**Return type** None

pytorch_lightning.trainer.distrib_parts.**determine_root_gpu_device**(*gpus*)

**Parameters** **gpus** (List[int]) – non-empty list of ints representing which gpus to use

**Return type** Optional[int]

**Returns** designated root GPU device id

pytorch_lightning.trainer.distrib_parts.**get_all_available_gpus**()

**Return type** List[int]

**Returns** a list of all available gpus

pytorch_lightning.trainer.distrib_parts.**normalize_parse_gpu_input_to_list**(*gpus*)

**Return type** Optional[List[int]]

pytorch_lightning.trainer.distrib_parts.**normalize_parse_gpu_string_input**(*s*)

**Return type** Union[int, List[int]]

pytorch_lightning.trainer.distrib_parts.**parse_gpu_ids**(*gpus*)
Parses the GPU ids given in the format as accepted by the *Trainer*.

**Parameters** **gpus** (Union[int, str, List[int], None]) – An int -1 or string '-1' indicate that all available GPUs should be used. A list of ints or a string containing list of comma separated integers indicates specific GPUs to use. An int 0 means that no GPUs should be used. Any int N > 0 indicates that GPUs [0..N) should be used.

**Return type** Optional[List[int]]

**Returns** a list of gpus to be used or None if no GPUs were requested

If no GPUs are available but the value of gpus variable indicates request for GPUs then a MisconfigurationException is raised.

pytorch_lightning.trainer.distrib_parts.**pick_multiple_gpus**(*nb*)

pytorch_lightning.trainer.distrib_parts.**pick_single_gpu**(*exclude_gpus*)

pytorch_lightning.trainer.distrib_parts.**retry_jittered_backoff**(*func*, *num_retries=5*, *cap_delay=1.0*, *base_delay=0.01*)

Retry jittered backoff.

Based on: https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/

**Parameters**

- **func** (Callable) – tested function
- **num_retries** (int) – number of tries

- **cap_delay** (float) – max sleep time
- **base_delay** (float) – initial sleep time is 10ms

pytorch_lightning.trainer.distrib_parts.**sanitize_gpu_ids**(*gpus*)
Checks that each of the GPUs in the list is actually available. Raises a MisconfigurationException if any of the GPUs is not available.

**Parameters** **gpus** (List[int]) – list of ints corresponding to GPU indices

**Return type** List[int]

**Returns** unmodified gpus variable

### pytorch_lightning.trainer.evaluation_loop module

### Validation loop

The lightning validation loop handles everything except the actual computations of your model. To decide what will happen in your validation loop, define the *validation_step* function. Below are all the things lightning automates for you in the validation loop.

**Note:** Lightning will run 5 steps of validation in the beginning of training as a sanity check so you don't have to wait until a full epoch to catch possible validation issues.

#### Check validation every n epochs

If you have a small dataset you might want to check validation every n epochs

```
# DEFAULT
trainer = Trainer(check_val_every_n_epoch=1)
```

#### Set how much of the validation set to check

If you don't want to check 100% of the validation set (for debugging or if it's huge), set this flag

limit_val_batches will be overwritten by overfit_batches if *overfit_batches > 0*

```
# DEFAULT
trainer = Trainer(limit_val_batches=1.0)

# check 10% only
trainer = Trainer(limit_val_batches=0.1)
```

#### Set how much of the test set to check

If you don't want to check 100% of the test set (for debugging or if it's huge), set this flag

limit_test_batches will be overwritten by overfit_batches if *overfit_batches > 0*

```
# DEFAULT
trainer = Trainer(limit_test_batches=1.0)

# check 10% only
trainer = Trainer(limit_test_batches=0.1)
```

#### Set validation check frequency within 1 training epoch

**For large datasets it's often desirable to check validation multiple times within a training loop.** Pass in a float to check that often within 1 training epoch. Pass in an int k to check every k training batches. Must use an int if using an IterableDataset.

```
# DEFAULT
trainer = Trainer(val_check_interval=0.95)

# check every .25 of an epoch
trainer = Trainer(val_check_interval=0.25)

# check every 100 train batches (ie: for IterableDatasets or fixed frequency)
trainer = Trainer(val_check_interval=100)
```

#### Set the number of validation sanity steps

**Lightning runs a few steps of validation in the beginning of training.** This avoids crashing in the validation loop sometime deep into a lengthy training loop.

```
# DEFAULT
trainer = Trainer(num_sanity_val_steps=5)
```

You can use *Trainer(num_sanity_val_steps=0)* to skip the sanity check.

# Testing loop

**To ensure you don't accidentally use test data to guide training decisions Lightning** makes running the test set deliberate.

**test**

You have two options to run the test set. First case is where you test right after a full training routine.

```
# run full training
trainer.fit(model)

# run test set
trainer.test()
```

Second case is where you load a model and run the test set

```
model = MyLightningModule.load_from_checkpoint(
    checkpoint_path='/path/to/pytorch_checkpoint.ckpt',
    hparams_file='/path/to/test_tube/experiment/version/hparams.yaml',
    map_location=None
)

# init trainer with whatever options
trainer = Trainer(...)

# test (pass in the model)
trainer.test(model)
```

**In this second case, the options you pass to trainer will be used when running** the test set (ie: 16-bit, dp, ddp, etc. . . )

**class** pytorch_lightning.trainer.evaluation_loop.**TrainerEvaluationLoopMixin**
Bases: abc.ABC

**_evaluate**(*model*, *dataloaders*, *max_batches*, *test_mode=False*)
Run evaluation code.

**Parameters**

- **model** (*LightningModule*) – PT model
- **dataloaders** – list of PT dataloaders
- **max_batches** (List[int]) – List of scalars
- **test_mode** (bool) –

**abstract add_progress_bar_metrics**(**args*)
Warning: this is just empty shell for code implemented in other class.

**abstract copy_trainer_model_properties**(**args*)
Warning: this is just empty shell for code implemented in other class.

**evaluation_forward**(*model*, *batch*, *batch_idx*, *dataloader_idx*, *test_mode=False*)

**abstract get_model**()
Warning: this is just empty shell for code implemented in other class.

**Return type** *LightningModule*

**abstract is_overridden**(**args*)
Warning: this is just empty shell for code implemented in other class.

**abstract log_metrics**(**args*)
Warning: this is just empty shell for code implemented in other class.

**abstract reset_test_dataloader**(**args*)
Warning: this is just empty shell for code implemented in other class.

**abstract reset_val_dataloader**(**args*)
Warning: this is just empty shell for code implemented in other class.

**run_evaluation**(*test_mode=False*)

**abstract transfer_batch_to_gpu**(**args*)
Warning: this is just empty shell for code implemented in other class.

**abstract transfer_batch_to_tpu**(**args*)
Warning: this is just empty shell for code implemented in other class.

**callback_metrics: ... = None**

**current_epoch: int = None**

**data_parallel_device_ids: ... = None**

**fast_dev_run: ... = None**

**global_rank: int = None**

**model: LightningModule = None**

**num_test_batches: List[int] = None**

**num_val_batches: int = None**

**on_gpu: bool = None**

**on_test_batch_end: Callable = None**

**on_test_batch_start: Callable = None**

**on_test_end: Callable = None**

**on_test_start: Callable = None**

**on_validation_batch_end: Callable = None**

**on_validation_batch_start: Callable = None**

**on_validation_end: Callable = None**

**on_validation_start: Callable = None**

**process_output: ... = None**

**progress_bar_dict: ... = None**

**reload_dataloaders_every_epoch: ... = None**

**single_gpu: bool = None**

**test_dataloaders: DataLoader = None**

**tpu_id: int = None**

**use_ddp: bool = None**

**use_ddp2: bool = None**

**use_dp: bool = None**

**use_horovod: bool = None**

**use_tpu: bool = None**

**val_dataloaders: DataLoader = None**

### pytorch_lightning.trainer.ignored_warnings module

`pytorch_lightning.trainer.ignored_warnings.`**`ignore_scalar_return_in_dp`**`()`

### pytorch_lightning.trainer.logging module

**class** `pytorch_lightning.trainer.logging.`**`TrainerLoggingMixin`**

Bases: `abc.ABC`

**`add_progress_bar_metrics`**(*metrics*)

**`configure_logger`**(*logger*)

**`log_metrics`**(*metrics*, *grad_norm_dic*, *step=None*)

Logs the metric dict passed in. If *step* parameter is None and *step* key is presented is metrics, uses metrics["step"] as a step

**Parameters**

- **metrics** (`dict`) – Metric values
- **grad_norm_dic** (`dict`) – Gradient norms
- **step** (`int`) – Step for which metrics should be logged. Default value corresponds to *self.global_step*

**`metrics_to_scalars`**(*metrics*)

**`process_output`**(*output*, *train=False*)

Reduces output according to the training mode.

Separates loss from logging and progress bar metrics

**`reduce_distributed_output`**(*output*, *num_gpus*)

**`current_epoch: int = None`**

**`default_root_dir: str = None`**

**`global_rank: int = None`**

**`global_step: int = None`**

**`log_gpu_memory: ... = None`**

**`logger: Union[LightningLoggerBase, bool] = None`**

**`num_gpus: int = None`**

**`on_gpu: bool = None`**

**`progress_bar_metrics: ... = None`**

**`slurm_job_id: int = None`**

**`use_ddp2: bool = None`**

**`use_dp: bool = None`**

### pytorch_lightning.trainer.lr_finder module

Trainer Learning Rate Finder

**class** pytorch_lightning.trainer.lr_finder.**TrainerLRFinderMixin**
Bases: abc.ABC

**_TrainerLRFinderMixin__lr_finder_dump_params**(*model*)

**_TrainerLRFinderMixin__lr_finder_restore_params**(*model*)

**_run_lr_finder_internally**(*model*)
Call lr finder internally during Trainer.fit()

**abstract fit**(**args*)
Warning: this is just empty shell for code implemented in other class.

**abstract init_optimizers**(**args*)
Warning: this is just empty shell for code implemented in other class.

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

**lr_find**(*model*, *train_dataloader=None*, *val_dataloaders=None*, *min_lr=1e-08*, *max_lr=1*, *num_training=100*, *mode='exponential'*, *early_stop_threshold=4.0*, *num_accumulation_steps=None*)
lr_find enables the user to do a range test of good initial learning rates, to reduce the amount of guesswork in picking a good starting learning rate.

**Parameters**

- **model** (*LightningModule*) – Model to do range testing for
- **train_dataloader** (Optional[DataLoader]) – A PyTorch DataLoader with training samples. If the model has a predefined train_dataloader method this will be skipped.
- **min_lr** (float) – minimum learning rate to investigate
- **max_lr** (float) – maximum learning rate to investigate
- **num_training** (int) – number of learning rates to test
- **mode** (str) – search strategy, either 'linear' or 'exponential'. If set to 'linear' the learning rate will be searched by linearly increasing after each batch. If set to 'exponential', will increase learning rate exponentially.
- **early_stop_threshold** (float) – threshold for stopping the search. If the loss at any point is larger than early_stop_threshold*best_loss then the search is stopped. To disable, set to None.
- **num_accumulation_steps** – deprepecated, number of batches to calculate loss over. Set trainer argument `accumulate_grad_batches` instead.

Example:

```
# Setup model and trainer
model = MyModelClass(hparams)
trainer = pl.Trainer()

# Run lr finder
lr_finder = trainer.lr_find(model, ...)

# Inspect results
```

(continues on next page)

(continued from previous page)

```
fig = lr_finder.plot(); fig.show()
suggested_lr = lr_finder.suggestion()

# Overwrite lr and create new model
hparams.lr = suggested_lr
model = MyModelClass(hparams)

# Ready to train with new learning rate
trainer.fit(model)
```

**abstract restore**(**args*)
Warning: this is just empty shell for code implemented in other class.

**abstract save_checkpoint**(**args*)
Warning: this is just empty shell for code implemented in other class.

**default_root_dir: str = None**

**global_step: int = None**

**on_gpu: bool = None**

**progress_bar_callback: ... = None**

**total_batch_idx: int = None**

**class** pytorch_lightning.trainer.lr_finder.**_ExponentialLR**(*optimizer*, *end_lr*, *num_iter*, *last_epoch=-1*)

Bases: `torch.optim.lr_scheduler._LRScheduler`

Exponentially increases the learning rate between two boundaries over a number of iterations.

**Parameters**

- **optimizer** (`Optimizer`) – wrapped optimizer.
- **end_lr** (`float`) – the final learning rate.
- **num_iter** (`int`) – the number of iterations over which the test occurs.
- **last_epoch** (`int`) – the index of last epoch. Default: -1.

**get_lr**()

**base_lrs: Sequence = None**

**last_epoch: int = None**

**property lr**

**class** pytorch_lightning.trainer.lr_finder.**_LRCallback**(*num_training*, *early_stop_threshold=4.0*, *progress_bar_refresh_rate=0*, *beta=0.98*)

Bases: *pytorch_lightning.callbacks.base.Callback*

Special callback used by the learning rate finder. This callbacks log the learning rate before each batch and log the corresponding loss after each batch.

**Parameters**

- **num_training** (`int`) – number of iterations done by the learning rate finder

- **early_stop_threshold** (`float`) – threshold for stopping the search. If the loss at any point is larger than `early_stop_threshold*best_loss` then the search is stopped. To disable, set to `None`.
- **progress_bar_refresh_rate** (`int`) – rate to refresh the progress bar for the learning rate finder
- **beta** (`float`) – smoothing value, the loss being logged is a running average of loss values logged until now. `beta` controls the forget rate i.e. if `beta=0` all past information is ignored.

**on_batch_end** (*trainer*, *pl_module*)
Called when the training batch ends, logs the calculated loss

**on_batch_start** (*trainer*, *pl_module*)
Called before each training batch, logs the lr that will be used

**class** pytorch_lightning.trainer.lr_finder.**_LRFinder** (*mode*, *lr_min*, *lr_max*, *num_training*)
Bases: `object`

LR finder object. This object stores the results of Trainer.lr_find().

**Parameters**

- **mode** (`str`) – either *linear* or *exponential*, how to increase lr after each step
- **lr_min** (`float`) – lr to start search from
- **lr_max** (`float`) – lr to stop search
- **num_training** (`int`) – number of steps to take between lr_min and lr_max

**Example::** # Run lr finder lr_finder = trainer.lr_find(model)

# Results stored in lr_finder.results

# Plot using lr_finder.plot()

# Get suggestion lr = lr_finder.suggestion()

**_get_new_optimizer** (*optimizer*)

**Construct a new *configure_optimizers()* method, that has a optimizer** with initial lr set to lr_min and a scheduler that will either linearly or exponentially increase the lr to lr_max in num_training steps.

**Parameters** **optimizer** (`Optimizer`) – instance of *torch.optim.Optimizer*

**plot** (*suggest=False*, *show=False*)
Plot results from lr_find run :type _sphinx_paramlinks_pytorch_lightning.trainer.lr_finder._LRFinder.plot.suggest: `bool` :param _sphinx_paramlinks_pytorch_lightning.trainer.lr_finder._LRFinder.plot.suggest: if True, will mark suggested lr to use with a red point :type _sphinx_paramlinks_pytorch_lightning.trainer.lr_finder._LRFinder.plot.show: `bool` :param _sphinx_paramlinks_pytorch_lightning.trainer.lr_finder._LRFinder.plot.show: if True, will show figure

**suggestion** (*skip_begin=10*, *skip_end=1*)
This will propose a suggestion for choice of initial learning rate as the point with the steepest negative gradient.

**Returns** suggested initial learning rate to use skip_begin: how many samples to skip in the beginning. Prevent too naive estimates skip_end: how many samples to skip in the end. Prevent too optimistic estimates

**Return type** lr

**class** pytorch_lightning.trainer.lr_finder.**_LinearLR**(*optimizer*, *end_lr*, *num_iter*, *last_epoch=-1*)

Bases: torch.optim.lr_scheduler._LRScheduler

Linearly increases the learning rate between two boundaries over a number of iterations. :type _sphinx_paramlinks_pytorch_lightning.trainer.lr_finder._LinearLR.optimizer: Optimizer :param _sphinx_paramlinks_pytorch_lightning.trainer.lr_finder._LinearLR.optimizer: wrapped optimizer. :type _sphinx_paramlinks_pytorch_lightning.trainer.lr_finder._LinearLR.end_lr: float :param _sphinx_paramlinks_pytorch_lightning.trainer.lr_finder._LinearLR.end_lr: the final learning rate. :type _sphinx_paramlinks_pytorch_lightning.trainer.lr_finder._LinearLR.num_iter: int :param _sphinx_paramlinks_pytorch_lightning.trainer.lr_finder._LinearLR.num_iter: the number of iterations over which the test occurs. :type _sphinx_paramlinks_pytorch_lightning.trainer.lr_finder._LinearLR.last_epoch: int :param _sphinx_paramlinks_pytorch_lightning.trainer.lr_finder._LinearLR.last_epoch: the index of last epoch. Default: -1.

**get_lr**()

**base_lrs: Sequence = None**

**last_epoch: int = None**

**property lr**

pytorch_lightning.trainer.lr_finder.**_nested_hasattr**(*obj*, *path*)

pytorch_lightning.trainer.lr_finder.**_nested_setattr**(*obj*, *path*, *val*)

### pytorch_lightning.trainer.model_hooks module

**class** pytorch_lightning.trainer.model_hooks.**TrainerModelHooksMixin**

Bases: abc.ABC

**abstract get_model**()

Warning: this is just empty shell for code implemented in other class.

**Return type** *LightningModule*

**has_arg**(*f_name*, *arg_name*)

**is_function_implemented**(*f_name*, *model=None*)

**is_overridden**(*method_name*, *model=None*)

**Return type** bool

### pytorch_lightning.trainer.optimizers module

**class** pytorch_lightning.trainer.optimizers.**TrainerOptimizersMixin**

Bases: abc.ABC

**configure_schedulers**(*schedulers*)

**init_optimizers**(*model*)

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

**reinit_scheduler_properties**(*optimizers*, *schedulers*)

**class** pytorch_lightning.trainer.optimizers.**_MockOptimizer**
Bases: torch.optim.optimizer.Optimizer

The *_MockOptimizer* will be used inplace of an optimizer in the event that *None* is returned from *configure_optimizers*.

**add_param_group**(*param_group*)

**load_state_dict**(*state_dict*)

**state_dict**()

**step**(*closure=None*)

**zero_grad**()

## pytorch_lightning.trainer.supporters module

**class** pytorch_lightning.trainer.supporters.**TensorRunningAccum**(*window_length*)
Bases: object

Tracks a running accumulation values (min, max, mean) without graph references.

### Examples

```
>>> accum = TensorRunningAccum(5)
>>> accum.last(), accum.mean()
(None, None)
>>> accum.append(torch.tensor(1.5))
>>> accum.last(), accum.mean()
(tensor(1.5000), tensor(1.5000))
>>> accum.append(torch.tensor(2.5))
>>> accum.last(), accum.mean()
(tensor(2.5000), tensor(2.))
>>> accum.reset()
>>> _= [accum.append(torch.tensor(i)) for i in range(13)]
>>> accum.last(), accum.mean(), accum.min(), accum.max()
(tensor(12.), tensor(10.), tensor(8.), tensor(12.))
```

**_agg_memory**(*how*)

**append**(*x*)
Add an element to the accumulator.

**last**()
Get the last added element.

**max**()
Get maximal value from stored elements.

**mean**()
Get mean value from stored elements.

**min**()
Get minimal value from stored elements.

**reset**()
Empty the accumulator.

**Return type** None

### pytorch_lightning.trainer.trainer module

**class** pytorch_lightning.trainer.trainer.**Trainer**(*logger=True, checkpoint_callback=True, early_stop_callback=False, callbacks=None, default_root_dir=None, gradient_clip_val=0, process_position=0, num_nodes=1, num_processes=1, gpus=None, auto_select_gpus=False, tpu_cores=None, log_gpu_memory=None, progress_bar_refresh_rate=1, overfit_batches=0.0, track_grad_norm=-1, check_val_every_n_epoch=1, fast_dev_run=False, accumulate_grad_batches=1, max_epochs=1000, min_epochs=1, max_steps=None, min_steps=None, limit_train_batches=1.0, limit_val_batches=1.0, limit_test_batches=1.0, val_check_interval=1.0, log_save_interval=100, row_log_interval=50, distributed_backend=None, precision=32, print_nan_grads=False, weights_summary='top', weights_save_path=None, num_sanity_val_steps=2, truncated_bptt_steps=None, resume_from_checkpoint=None, profiler=None, benchmark=False, deterministic=False, reload_dataloaders_every_epoch=False, auto_lr_find=False, replace_sampler_ddp=True, terminate_on_nan=False, auto_scale_batch_size=False, prepare_data_per_node=True, amp_level='O1', num_tpu_cores=None, use_amp=None, show_progress_bar=None, val_percent_check=None, test_percent_check=None, train_percent_check=None, overfit_pct=None*)

Bases: *pytorch_lightning.trainer.training_io.TrainerIOMixin*, *pytorch_lightning.trainer.optimizers.TrainerOptimizersMixin*, *pytorch_lightning.trainer.auto_mix_precision.TrainerAMPMixin*, *pytorch_lightning.trainer.distrib_parts.TrainerDPMixin*, *pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin*, *pytorch_lightning.trainer.logging.TrainerLoggingMixin*, *pytorch_lightning.trainer.model_hooks.*

*TrainerModelHooksMixin*, *pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixin*, *pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin*, *pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin*, *pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin*, *pytorch_lightning.trainer.callback_config.TrainerCallbackConfigMixin*, *pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin*, *pytorch_lightning.trainer.lr_finder.TrainerLRFinderMixin*, *pytorch_lightning.trainer.deprecated_api.TrainerDeprecatedAPITillVer0_9*, *pytorch_lightning.trainer.deprecated_api.TrainerDeprecatedAPITillVer0_10*

Customize every aspect of training via flags

**Parameters**

- **logger** (Union[*LightningLoggerBase*, Iterable[*LightningLoggerBase*], bool]) – Logger (or iterable collection of loggers) for experiment tracking.
- **checkpoint_callback** (Union[*ModelCheckpoint*, bool]) – Callback for checkpointing.
- **early_stop_callback** (*pytorch_lightning.callbacks.EarlyStopping*) –
- **callbacks** (Optional[List[*Callback*]]) – Add a list of callbacks.
- **default_root_dir** (Optional[str]) – Default path for logs and weights when no logger/ckpt_callback passed
- **gradient_clip_val** (float) – 0 means don't clip.
- **gradient_clip** –

  > **Warning:** Deprecated since version 0.7.0.
  >
  > Use *gradient_clip_val* instead. Will remove 0.9.0.

- **process_position** (int) – orders the progress bar when running multiple models on same machine.
- **num_nodes** (int) – number of GPU nodes for distributed training.
- **nb_gpu_nodes** –

  > **Warning:** Deprecated since version 0.7.0.
  >
  > Use *num_nodes* instead. Will remove 0.9.0.

- **gpus** (Union[int, str, List[int], None]) – Which GPUs to train on.
- **auto_select_gpus** (bool) – If enabled and *gpus* is an integer, pick available gpus automatically. This is especially useful when GPUs are configured to be in "exclusive mode", such that only one process at a time can access them.
- **tpu_cores** (Union[List[int], int, None]) – How many TPU cores to train on (1 or 8) / Single TPU to train on [1]
- **num_tpu_cores** (Optional[int]) – How many TPU cores to train on (1 or 8) .. warning:: .. deprecated:: 0.7.6. Will remove 0.9.0.

- **log_gpu_memory** (Optional[str]) – None, 'min_max', 'all'. Might slow performance
- **show_progress_bar** –

> **Warning:** Deprecated since version 0.7.2.
>
> Set *progress_bar_refresh_rate* to positive integer to enable. Will remove 0.9.0.

- **progress_bar_refresh_rate** (int) – How often to refresh progress bar (in steps). Value 0 disables progress bar. Ignored when a custom callback is passed to *callbacks*.
- **overfit_batches** (Union[int, float]) – Overfit a percent of training data (float) or a set number of batches (int).
- **overfit_pct** (Optional[float]) –

> **Warning:** Deprecated since version 0.8.0.
>
> Use *overfit_batches* instead. Will remove 0.10.0.

- **track_grad_norm** (Union[int, float, str]) – -1 no tracking. Otherwise tracks that p-norm. May be set to 'inf' infinity-norm.
- **check_val_every_n_epoch** (int) – Check val every n train epochs.
- **fast_dev_run** (bool) – runs 1 batch of train, test and val to find any bugs (ie: a sort of unit test).
- **accumulate_grad_batches** (Union[int, Dict[int, int], List[list]]) – Accumulates grads every k batches or as set up in the dict.
- **max_epochs** (int) – Stop training once this number of epochs is reached.
- **max_nb_epochs** –

> **Warning:** Deprecated since version 0.7.0.
>
> Use *max_epochs* instead. Will remove 0.9.0.

- **min_epochs** (int) – Force training for at least these many epochs
- **min_nb_epochs** –

> **Warning:** Deprecated since version 0.7.0.
>
> Use *min_epochs* instead. Will remove 0.9.0.

- **max_steps** (Optional[int]) – Stop training after this number of steps. Disabled by default (None).
- **min_steps** (Optional[int]) – Force training for at least these number of steps. Disabled by default (None).
- **limit_train_batches** (Union[int, float]) – How much of training dataset to check.

- **limit_val_batches** (Union[int, float]) – How much of validation dataset to check (floats = percent, int = num_batches)
- **limit_test_batches** (Union[int, float]) – How much of test dataset to check (floats = percent, int = num_batches)
- **train_percent_check** (Optional[float]) –

  > **Warning:** Deprecated since version 0.8.0.
  >
  > Use *limit_train_batches* instead. Will remove v0.10.0.

- **val_percent_check** (Optional[float]) –

  > **Warning:** Deprecated since version 0.8.0.
  >
  > Use *limit_val_batches* instead. Will remove v0.10.0.

- **test_percent_check** (Optional[float]) –

  > **Warning:** Deprecated since version 0.8.0.
  >
  > Use *limit_test_batches* instead. Will remove v0.10.0.

- **val_check_interval** (Union[int, float]) – How often within one training epoch to check the validation set
- **log_save_interval** (int) – Writes logs to disk this often
- **row_log_interval** (int) – How often to add logging rows (does not write to disk)
- **add_row_log_interval** –

  > **Warning:** Deprecated since version 0.7.0.
  >
  > Use *row_log_interval* instead. Will remove 0.9.0.

- **distributed_backend** (Optional[str]) – The distributed backend to use (dp, ddp, ddp2, ddp_spawn)
- **use_amp** –

  > **Warning:** Deprecated since version 0.7.0.
  >
  > Use *precision* instead. Will remove 0.9.0.

- **precision** (int) – Full precision (32), half precision (16).
- **print_nan_grads** (bool) –

> **Warning:** Deprecated since version 0.7.2.
>
> Has no effect. When detected, NaN grads will be printed automatically. Will remove 0.9.0.

- **weights_summary** (Optional[str]) – Prints a summary of the weights when training begins.
- **weights_save_path** (Optional[str]) – Where to save weights if specified. Will override default_root_dir for checkpoints only. Use this if for whatever reason you need the checkpoints stored in a different place than the logs written in *default_root_dir*.
- **amp_level** (str) – The optimization level to use (O1, O2, etc. . . ).
- **num_sanity_val_steps** (int) – Sanity check runs n batches of val before starting the training routine.
- **truncated_bptt_steps** (Optional[int]) – Truncated back prop breaks performs backprop every k steps of
- **resume_from_checkpoint** (Optional[str]) – To resume training from a specific checkpoint pass in the path here. This can be a URL.
- **profiler** (Union[*BaseProfiler*, bool, None]) – To profile individual steps during training and assist in
- **reload_dataloaders_every_epoch** (bool) – Set to True to reload dataloaders every epoch
- **auto_lr_find** (Union[bool, str]) – If set to True, will *initially* run a learning rate finder, trying to optimize initial learning for faster convergence. Sets learning rate in self.lr or self.learning_rate in the LightningModule. To use a different key, set a string instead of True with the key name.
- **replace_sampler_ddp** (bool) – Explicitly enables or disables sampler replacement. If not specified this will toggled automatically ddp is used
- **benchmark** (bool) – If true enables cudnn.benchmark.
- **deterministic** (bool) – If true enables cudnn.deterministic
- **terminate_on_nan** (bool) – If set to True, will terminate training (by raising a *ValueError*) at the end of each training batch, if any of the parameters or the loss are NaN or +/-inf.
- **auto_scale_batch_size** (Union[str, bool]) – If set to True, will *initially* run a batch size finder trying to find the largest batch size that fits into memory. The result will be stored in self.batch_size in the LightningModule. Additionally, can be set to either *power* that estimates the batch size through a power search or *binsearch* that estimates the batch size through a binary search.
- **prepare_data_per_node** (bool) – If True, each LOCAL_RANK=0 will call prepare data. Otherwise only NODE_RANK=0, LOCAL_RANK=0 will prepare data

**_Trainer__attach_dataloaders**(*model*, *train_dataloader=None*, *val_dataloaders=None*, *test_dataloaders=None*)

**_allowed_type**()

**Return type** Union[int, str]

**_arg_default**()

**Return type** Union[int, str]

**classmethod add_argparse_args**(*parent_parser*)

Extends existing argparse by default *Trainer* attributes.

**Parameters parent_parser** (ArgumentParser) – The custom cli arguments parser, which will be extended by the Trainer default arguments.

Only arguments of the allowed types (str, float, int, bool) will extend the *parent_parser*.

### Examples

```
>>> import argparse
>>> import pprint
>>> parser = argparse.ArgumentParser()
>>> parser = Trainer.add_argparse_args(parser)
>>> args = parser.parse_args([])
>>> pprint.pprint(vars(args))
{...
 'check_val_every_n_epoch': 1,
 'checkpoint_callback': True,
 'default_root_dir': None,
 'deterministic': False,
 'distributed_backend': None,
 'early_stop_callback': False,
 ...
 'logger': True,
 'max_epochs': 1000,
 'max_steps': None,
 'min_epochs': 1,
 'min_steps': None,
 ...
 'profiler': None,
 'progress_bar_refresh_rate': 1,
 ...}
```

**Return type** ArgumentParser

**barrier**(*name*)

**can_prepare_data**()

**check_model_configuration**(*model*)

Checks that the model is configured correctly before training or testing is started.

**Parameters model** (*LightningModule*) – The model to check the configuration.

**classmethod default_attributes**()

**fit**(*model*, *train_dataloader=None*, *val_dataloaders=None*)

Runs the full optimization routine.

**Parameters**

- **model** (*LightningModule*) – Model to fit.
- **train_dataloader** (Optional[DataLoader]) – A Pytorch DataLoader with training samples. If the model has a predefined train_dataloader method this will be skipped.

- **val_dataloaders** (Union[DataLoader, List[DataLoader], None]) – Either a single Pytorch Dataloader or a list of them, specifying validation samples. If the model has a predefined val_dataloaders method this will be skipped

Example:

```
# Option 1,
# Define the train_dataloader() and val_dataloader() fxs
# in the lightningModule
# RECOMMENDED FOR MOST RESEARCH AND APPLICATIONS TO MAINTAIN READABILITY
trainer = Trainer()
model = LightningModule()
trainer.fit(model)

# Option 2
# in production cases we might want to pass different datasets to the same␣
↪model
# Recommended for PRODUCTION SYSTEMS
train, val = DataLoader(...), DataLoader(...)
trainer = Trainer()
model = LightningModule()
trainer.fit(model, train_dataloader=train, val_dataloaders=val)

# Option 1 & 2 can be mixed, for example the training set can be
# defined as part of the model, and validation can then be feed to .fit()
```

**classmethod from_argparse_args**(*args*, ***kwargs*)

Create an instance from CLI arguments.

**Parameters**

- **args** (Union[Namespace, ArgumentParser]) – The parser or namespace to take arguments from. Only known arguments will be parsed and passed to the *Trainer*.
- ****kwargs** – Additional keyword arguments that may override ones in the parser or namespace. These must be valid Trainer arguments.

### Example

```
>>> parser = ArgumentParser(add_help=False)
>>> parser = Trainer.add_argparse_args(parser)
>>> parser.add_argument('--my_custom_arg', default='something')
>>> args = Trainer.parse_argparser(parser.parse_args(""))
>>> trainer = Trainer.from_argparse_args(args, logger=False)
```

**Return type** *Trainer*

**classmethod get_deprecated_arg_names**()

Returns a list with deprecated Trainer arguments.

**Return type** List

**classmethod get_init_arguments_and_types**()

Scans the Trainer signature and returns argument names, types and default values.

**Returns** (argument name, set with argument types, argument default value).

**Return type** List with tuples of 3 values

### Examples

```
>>> args = Trainer.get_init_arguments_and_types()
>>> import pprint
>>> pprint.pprint(sorted(args))
[('accumulate_grad_batches',
  (<class 'int'>, typing.Dict[int, int], typing.List[list]),
  1),
 ...
 ('callbacks',
  (typing.List[pytorch_lightning.callbacks.base.Callback],
   <class 'NoneType'>),
   None),
 ('check_val_every_n_epoch', (<class 'int'>,), 1),
 ...
 ('max_epochs', (<class 'int'>,), 1000),
 ...
 ('precision', (<class 'int'>,), 32),
 ('prepare_data_per_node', (<class 'bool'>,), True),
 ('print_nan_grads', (<class 'bool'>,), False),
 ('process_position', (<class 'int'>,), 0),
 ('profiler',
  (<class 'pytorch_lightning.profiler.profilers.BaseProfiler'>,
   <class 'bool'>,
   <class 'NoneType'>),
  None),
 ...
```

**static parse_argparser**(*arg_parser*)
Parse CLI arguments, required for custom bool types.

**Return type** `Namespace`

**run_pretrain_routine**(*model*)
Sanity check a few things before starting actual training.

**Parameters** **model** (*LightningModule*) – The model to run sanity test on.

**test**(*model=None*, *test_dataloaders=None*, *ckpt_path='best'*)
Separates from fit to make sure you never run on your test set until you want to.

**Parameters**

- **model** (`Optional`[*LightningModule*]) – The model to test.
- **test_dataloaders** (`Union`[`DataLoader`, `List`[`DataLoader`], `None`]) – Either a single Pytorch Dataloader or a list of them, specifying validation samples.
- **ckpt_path** (`Optional`[`str`]) – Either `best` or path to the checkpoint you wish to test. If `None`, use the weights from the last epoch to test. Default to `best`.

Example:

```
# Option 1
# run test with the best checkpoint from ``ModelCheckpoint`` after fitting.
test = DataLoader(...)
trainer = Trainer()
model = LightningModule()

trainer.fit(model)
```

(continues on next page)

(continued from previous page)

```
trainer.test(test_dataloaders=test)

# Option 2
# run test with the specified checkpoint after fitting
test = DataLoader(...)
trainer = Trainer()
model = LightningModule()

trainer.fit(model)
trainer.test(test_dataloaders=test, ckpt_path='path/to/checkpoint.ckpt')

# Option 3
# run test with the weights from the end of training after fitting
test = DataLoader(...)
trainer = Trainer()
model = LightningModule()

trainer.fit(model)
trainer.test(test_dataloaders=test, ckpt_path=None)

# Option 4
# run test from a loaded model. ``ckpt_path`` is ignored in this case.
test = DataLoader(...)
model = LightningModule.load_from_checkpoint('path/to/checkpoint.ckpt')
trainer = Trainer()
trainer.test(model, test_dataloaders=test)
```

**DEPRECATED_IN_0_9 = ('use_amp', 'show_progress_bar', 'training_tqdm_dict', 'num_tpu_co**

**accumulate_grad_batches = None**

**checkpoint_callback = None**

**property data_parallel**

**Return type** bool

**early_stop_callback = None**

**global_rank = None**

**property is_global_zero**

this is just empty shell for code implemented in other class.

**Type** Warning

**logger = None**

**lr_schedulers = None**

**model = None**

**property num_gpus**

this is just empty shell for code implemented in other class.

**Type** Warning

**Return type** int

**num_training_batches = None**

**on_gpu = None**

**on_tpu = None**

**optimizers = None**

**property progress_bar_callback**

**property progress_bar_dict**
Read-only for progress bar metrics.

**Return type** dict

**resume_from_checkpoint = None**

**root_gpu = None**

**scaler = None**

**property slurm_job_id**
this is just empty shell for code implemented in other class.

**Type** Warning

**Return type** Optional[int]

**use_ddp = None**

**use_ddp2 = None**

**use_horovod = None**

**use_native_amp = None**

**weights_save_path = None**

**class** pytorch_lightning.trainer.trainer.**_PatchDataLoader**(*dataloader*)
Bases: object

Callable object for patching dataloaders passed into trainer.fit(). Use this class to override model.*_dataloader() and be pickle-compatible.

**Parameters** **dataloader** (Union[List[DataLoader], DataLoader]) – Dataloader object to return when called.

**__call__**()
Call self as a function.

**Return type** Union[List[DataLoader], DataLoader]

pytorch_lightning.trainer.trainer.**_determine_limit_batches**(*batches*)

**Return type** Union[int, float]

### pytorch_lightning.trainer.training_io module

### Lightning can automate saving and loading checkpoints

Checkpointing is enabled by default to the current working directory. To change the checkpoint path pass in:

```
Trainer(default_root_dir='/your/path/to/save/checkpoints')
```

To modify the behavior of checkpointing pass in your own callback.

```
from pytorch_lightning.callbacks import ModelCheckpoint

# DEFAULTS used by the Trainer
checkpoint_callback = ModelCheckpoint(
    filepath=os.getcwd(),
    save_top_k=1,
    verbose=True,
    monitor='val_loss',
    mode='min',
    prefix=''
)

trainer = Trainer(checkpoint_callback=checkpoint_callback)
```

### Restoring training session

You might want to not only load a model but also continue training it. Use this method to restore the trainer state as well. This will continue from the epoch and global step you last left off. However, the dataloaders will start from the first batch again (if you shuffled it shouldn't matter).

Lightning will restore the session if you pass a logger with the same version and there's a saved checkpoint.

```
from pytorch_lightning import Trainer

trainer = Trainer(
    resume_from_checkpoint=PATH
)

# this fit call loads model weights and trainer state
# the trainer continues seamlessly from where you left off
# without having to do anything else.
trainer.fit(model)
```

The trainer restores:

- global_step
- current_epoch
- All optimizers
- All lr_schedulers
- Model weights

You can even change the logic of your model as long as the weights and "architecture" of the system isn't different. If you add a layer, for instance, it might not work.

At a rough level, here's what happens inside Trainer `pytorch_lightning.base_module.saving.py`:

```
self.global_step = checkpoint['global_step']
self.current_epoch = checkpoint['epoch']

# restore the optimizers
optimizer_states = checkpoint['optimizer_states']
for optimizer, opt_state in zip(self.optimizers, optimizer_states):
    optimizer.load_state_dict(opt_state)
```

(continues on next page)

(continued from previous page)

```
# restore the lr schedulers
lr_schedulers = checkpoint['lr_schedulers']
for scheduler, lrs_state in zip(self.lr_schedulers, lr_schedulers):
    scheduler['scheduler'].load_state_dict(lrs_state)

# uses the model you passed into trainer
model.load_state_dict(checkpoint['state_dict'])
```

**class** pytorch_lightning.trainer.training_io.**TrainerIOMixin**

Bases: `abc.ABC`

**_atomic_save** (*checkpoint*, *filepath*)

Saves a checkpoint atomically, avoiding the creation of incomplete checkpoints.

This will create a temporary checkpoint with a suffix of `.part`, then copy it to the final location once saving is finished.

**Parameters**

- **checkpoint** – The object to save. Built to be used with the `dump_checkpoint` method, but can deal with anything which `torch.save` accepts.
- **filepath** (`str`) – The path to which the checkpoint will be saved. This points to the file that the checkpoint will be stored in.

**dump_checkpoint** (*weights_only=False*)

Creating model checkpoint.

**Parameters** **weights_only** (`bool`) – saving model weights only

**Return type** `dict`

**Returns** structured dictionary

**get_model** ()

**hpc_load** (*folderpath*, *on_gpu*)

**hpc_save** (*folderpath*, *logger*)

**max_ckpt_in_folder** (*path*, *name_key='ckpt_'*)

**register_slurm_signal_handlers** ()

**restore** (*checkpoint_path*, *on_gpu*)

Restore training state from checkpoint. Also restores all training state like: - epoch - callbacks - schedulers - optimizer

**restore_hpc_weights_if_needed** (*model*)

If there is a set of hpc weights, use as signal to restore model.

**restore_training_state** (*checkpoint*)

Restore trainer state. Model will get its change to update :param _sphinx_paramlinks_pytorch_lightning.trainer.training_io.TrainerIOMixin.restore_training_state.checkpoint: :return:

**restore_weights** (*model*)

We attempt to restore weights in this order: 1. HPC weights. 2. if no HPC weights restore checkpoint_path weights 3. otherwise don't restore weights

**save_checkpoint** (*filepath*, *weights_only=False*)

**sig_handler** (*signum*, *frame*)

**term_handler** (*signum*, *frame*)

**accumulate_grad_batches: int = None**

**checkpoint_callback: ... = None**

**early_stop_callback: ... = None**

**global_rank: int = None**

**logger: LightningLoggerBase = None**

**lr_schedulers: ... = None**

**model: LightningModule = None**

**num_training_batches: int = None**

**on_gpu: bool = None**

**on_tpu: bool = None**

**optimizers: ... = None**

**resume_from_checkpoint: ... = None**

**root_gpu: ... = None**

**scaler: ... = None**

**use_amp: bool = None**

**use_ddp: bool = None**

**use_ddp2: bool = None**

**use_horovod: bool = None**

**use_native_amp: bool = None**

**weights_save_path: str = None**

## pytorch_lightning.trainer.training_loop module

**The lightning training loop handles everything except the actual computations of your model.** To decide what will happen in your training loop, define the *training_step* function.

Below are all the things lightning automates for you in the training loop.

### Accumulated gradients

**Accumulated gradients runs K small batches of size N before doing a backwards pass.** The effect is a large effective batch size of size KxN.

```
# DEFAULT (ie: no accumulated grads)
trainer = Trainer(accumulate_grad_batches=1)
```

### Force training for min or max epochs

It can be useful to force training for a minimum number of epochs or limit to a max number

```
# DEFAULT
trainer = Trainer(min_epochs=1, max_epochs=1000)
```

### Force disable early stop

To disable early stopping pass None to the early_stop_callback

```
# DEFAULT
trainer = Trainer(early_stop_callback=None)
```

### Gradient Clipping

**Gradient clipping may be enabled to avoid exploding gradients.** Specifically, this will clip the gradient norm computed over all model parameters `together.

```
# DEFAULT (ie: don't clip)
trainer = Trainer(gradient_clip_val=0)

# clip gradients with norm above 0.5
trainer = Trainer(gradient_clip_val=0.5)
```

### Inspect gradient norms

Looking at grad norms can help you figure out where training might be going wrong.

```
# DEFAULT (-1 doesn't track norms)
trainer = Trainer(track_grad_norm=-1)

# track the LP norm (P=2 here)
trainer = Trainer(track_grad_norm=2)
```

### Set how much of the training set to check

If you don't want to check 100% of the training set (for debugging or if it's huge), set this flag.

limit_train_batches will be overwritten by overfit_batches if *overfit_batches > 0*

```
# DEFAULT
trainer = Trainer(limit_train_batches=1.0)

# check 10% only
trainer = Trainer(limit_train_batches=0.1)

# check 10 batches only
trainer = Trainer(limit_train_batches=10)
```

### Packed sequences as inputs

When using PackedSequence, do 2 things: 1. return either a padded tensor in dataset or a list of variable length tensors in the dataloader collate_fn (example above shows the list implementation). 2. Pack the sequence in forward or training and validation steps depending on use case.

```
# For use in dataloader
def collate_fn(batch):
    x = [item[0] for item in batch]
    y = [item[1] for item in batch]
    return x, y

# In module
def training_step(self, batch, batch_idx):
    x = rnn.pack_sequence(batch[0], enforce_sorted=False)
    y = rnn.pack_sequence(batch[1], enforce_sorted=False)
```

### Truncated Backpropagation Through Time

**There are times when multiple backwards passes are needed for each batch.** For example, it may save memory to use Truncated Backpropagation Through Time when training RNNs.

**When this flag is enabled each batch is split into sequences of size truncated_bptt_steps** and passed to training_step(…) separately. A default splitting function is provided, however, you can override it for more flexibility. See *tbptt_split_batch*.

```
# DEFAULT (single backwards pass per batch)
trainer = Trainer(truncated_bptt_steps=None)

# (split batch into sequences of size 2)
trainer = Trainer(truncated_bptt_steps=2)
```

### NaN detection and intervention

When the *terminate_on_nan* flag is enabled, after every forward pass during training, Lightning will check that

1. the loss you return in *training_step* is finite (not NaN and not +/-inf)
2. the model parameters have finite values.

Lightning will terminate the training loop with an error message if NaN or infinite values are detected. If this happens, you should investigate numerically unstable operations in your model.

```
# DEFAULT (won't perform the NaN check)
trainer = Trainer(terminate_on_nan=False)

# (NaN check each batch and terminate on NaN or infinite values)
trainer = Trainer(terminate_on_nan=True)
```

**class** pytorch_lightning.trainer.training_loop.**TrainerTrainLoopMixin**
Bases: abc.ABC

**_get_optimizers_iterable**()

**abstract add_progress_bar_metrics**(**args*)
Warning: this is just empty shell for code implemented in other class.

**call_checkpoint_callback**()

abstract **clip_gradients**()
Warning: this is just empty shell for code implemented in other class.

abstract **detect_nan_tensors**(**args*)
Warning: this is just empty shell for code implemented in other class.

abstract **get_model**()
Warning: this is just empty shell for code implemented in other class.

**Return type** *LightningModule*

abstract **has_arg**(**args*)
Warning: this is just empty shell for code implemented in other class.

abstract **is_function_implemented**(**args*, ***kwargs*)
Warning: this is just empty shell for code implemented in other class.

abstract **is_overridden**(**args*)
Warning: this is just empty shell for code implemented in other class.

abstract **log_metrics**(**args*)
Warning: this is just empty shell for code implemented in other class.

abstract **process_output**(**args*)
Warning: this is just empty shell for code implemented in other class.

abstract **reset_train_dataloader**(**args*)
Warning: this is just empty shell for code implemented in other class.

abstract **reset_val_dataloader**(*model*)
Warning: this is just empty shell for code implemented in other class.

abstract **run_evaluation**(**args*)
Warning: this is just empty shell for code implemented in other class.

**run_training_batch**(*batch*, *batch_idx*)

**run_training_epoch**()

**run_training_teardown**()

**train**()

**training_forward**(*batch*, *batch_idx*, *opt_idx*, *hiddens*)
Handle forward for each training case (distributed, single gpu, etc...) :param _sphinx_paramlinks_pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin.training_forward.batch: :param _sphinx_paramlinks_pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin.training_forward.batch_idx: :return:

abstract **transfer_batch_to_gpu**(**args*)
Warning: this is just empty shell for code implemented in other class.

abstract **transfer_batch_to_tpu**(**args*)
Warning: this is just empty shell for code implemented in other class.

**update_learning_rates**(*interval*)
Update learning rates.

**Parameters** **interval** (str) – either 'epoch' or 'step'.

**accumulate_grad_batches: int = None**

**accumulation_scheduler: ... = None**

**batch_idx: int = None**

**callback_metrics: ... = None**

**callbacks: List[Callback] = None**

**check_val_every_n_epoch: ... = None**

**checkpoint_callback: ... = None**

**data_parallel_device_ids: ... = None**

**disable_validation: bool = None**

**early_stop_callback: ... = None**

**enable_early_stop: ... = None**

**fast_dev_run: ... = None**

**global_rank: int = None**

**global_step: int = None**

**interactive_ddp_procs: ... = None**

**interrupted: bool = None**

**log_save_interval: float = None**

**logger: Union[LightningLoggerBase, bool] = None**

**lr_schedulers: ... = None**

**max_epochs: int = None**

**max_steps: int = None**

**min_epochs: int = None**

**min_steps: int = None**

**model: LightningModule = None**

**num_training_batches: int = None**

**num_val_batches: int = None**

**on_batch_end: Callable = None**

**on_batch_start: Callable = None**

**on_epoch_end: Callable = None**

**on_epoch_start: Callable = None**

**on_gpu: bool = None**

**on_keyboard_interrupt: Callable = None**

**on_train_end: Callable = None**

**on_train_start: Callable = None**

**on_validation_end: Callable = None**

**optimizer_frequencies: ... = None**

**optimizers: ... = None**

**precision: ... = None**

**profiler: ... = None**

**progress_bar_dict: ... = None**

**reduce_lr_on_plateau_scheduler: ... = None**

**reload_dataloaders_every_epoch: bool = None**

**row_log_interval: float = None**

**running_loss: ... = None**

**single_gpu: bool = None**

**terminate_on_nan: bool = None**

**testing: bool = None**

**total_batch_idx: int = None**

**tpu_id: int = None**

**track_grad_norm: ... = None**

**train_dataloader: DataLoader = None**

**truncated_bptt_steps: ... = None**

**use_ddp: bool = None**

**use_ddp2: bool = None**

**use_dp: bool = None**

**use_horovod: bool = None**

**use_tpu: bool = None**

**val_check_batch: ... = None**

pytorch_lightning.trainer.training_loop.**_with_is_last**(*iterable*)
Pass through values from the given iterable with an added boolean indicating if this is the last item. See https://stackoverflow.com/a/1630350

### pytorch_lightning.trainer.training_tricks module

**class** pytorch_lightning.trainer.training_tricks.**TrainerTrainingTricksMixin**
Bases: abc.ABC

**_TrainerTrainingTricksMixin__scale_batch_dump_params**()

**_TrainerTrainingTricksMixin__scale_batch_reset_params**(*model*, *steps_per_trial*)

**_TrainerTrainingTricksMixin__scale_batch_restore_params**()

**clip_gradients**()

**configure_accumulated_gradients**(*accumulate_grad_batches*)

**detect_nan_tensors**(*loss*)

**Return type** None

**abstract fit**(**args*)
Warning: this is just empty shell for code implemented in other class.

**abstract get_model**()
Warning: this is just empty shell for code implemented in other class.

**Return type** *LightningModule*

**print_nan_gradients**()

**Return type** None

**abstract restore**(**args*)
Warning: this is just empty shell for code implemented in other class.

**abstract save_checkpoint**(**args*)
Warning: this is just empty shell for code implemented in other class.

**scale_batch_size**(*model*, *mode='power'*, *steps_per_trial=3*, *init_val=2*, *max_trials=25*, *batch_arg_name='batch_size'*)
Will iteratively try to find the largest batch size for a given model that does not give an out of memory (OOM) error.

**Parameters**

- **model** (*LightningModule*) – Model to fit.
- **mode** (str) – string setting the search mode. Either *power* or *binsearch*. If mode is *power* we keep multiplying the batch size by 2, until we get an OOM error. If mode is 'binsearch', we will initially also keep multiplying by 2 and after encountering an OOM error do a binary search between the last successful batch size and the batch size that failed.
- **steps_per_trial** (int) – number of steps to run with a given batch size. Idealy 1 should be enough to test if a OOM error occurs, however in practise a few are needed
- **init_val** (int) – initial batch size to start the search with
- **max_trials** (int) – max number of increase in batch size done before algorithm is terminated

**default_root_dir: str = None**

**gradient_clip_val: ... = None**

**on_gpu: bool = None**

**precision: int = None**

**progress_bar_callback: ... = None**

pytorch_lightning.trainer.training_tricks.**_adjust_batch_size**(*trainer*, *batch_arg_name='batch_size'*, *factor=1.0*, *value=None*, *desc=None*)

**Function for adjusting the batch size. It is expected that the user** has provided a model that has a hparam field called *batch_size* i.e. *model.hparams.batch_size* should exist.

**Parameters**

- **trainer** – instance of pytorch_lightning.Trainer
- **batch_arg_name** (str) – field where batch_size is stored in *model.hparams*
- **factor** (float) – value which the old batch size is multiplied by to get the new batch size
- **value** (Optional[int]) – if a value is given, will override the batch size with this value. Note that the value of *factor* will not have an effect in this case

- **desc** (Optional[str]) – either *succeeded* or *failed*. Used purely for logging

pytorch_lightning.trainer.training_tricks.**_run_binsearch_scaling**(*trainer*, *model*, *new_size*, *batch_arg_name*, *max_trials*)

Batch scaling mode where the size is initially is doubled at each iteration until an OOM error is encountered. Hereafter, the batch size is further refined using a binary search

pytorch_lightning.trainer.training_tricks.**_run_power_scaling**(*trainer*, *model*, *new_size*, *batch_arg_name*, *max_trials*)

Batch scaling mode where the size is doubled at each iteration until an OOM error is encountered.

## 37.8 pytorch_lightning.utilities package

General utilities

### 37.8.1 Submodules

#### pytorch_lightning.utilities.apply_func module

pytorch_lightning.utilities.apply_func.**apply_to_collection**(*data*, *dtype*, *function*, **args*, ***kwargs*)

Recursively applies a function to all elements of a certain dtype.

**Parameters**

- **data** (Any) – the collection to apply the function to
- **dtype** (Union[type, tuple]) – the given function will be applied to all elements of this dtype
- **function** (Callable) – the function to apply
- ***args** – positional arguments (will be forwarded to calls of `function`)
- ****kwargs** – keyword arguments (will be forwarded to calls of `function`)

**Return type** Any

**Returns** the resulting collection

pytorch_lightning.utilities.apply_func.**move_data_to_device**(*batch*, *device*)

Transfers a collection of tensors to the given device.

**Parameters**

- **batch** (Any) – A tensor or collection of tensors. See *apply_to_collection()* for a list of supported collection types.
- **device** (device) – The device to which tensors should be moved

**Returns** the same collection but with all contained tensors residing on the new device.

**See also:**

- `torch.Tensor.to()`
- `torch.device`

### pytorch_lightning.utilities.cloud_io module

`pytorch_lightning.utilities.cloud_io.load(path_or_url, map_location=None)`

### pytorch_lightning.utilities.device_dtype_mixin module

**class** `pytorch_lightning.utilities.device_dtype_mixin.DeviceDtypeModuleMixin(*args, **kwargs)`

Bases: `torch.nn.Module`

**cpu()**
Moves all model parameters and buffers to the CPU. :returns: self :rtype: Module

**cuda(device=None)**
Moves all model parameters and buffers to the GPU. This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

**Parameters device** (`Optional[int]`) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**double()**
Casts all floating point parameters and buffers to `double` datatype.

**Returns** self

**Return type** Module

**float()**
Casts all floating point parameters and buffers to float datatype.

**Returns** self

**Return type** Module

**half()**
Casts all floating point parameters and buffers to `half` datatype.

**Returns** self

**Return type** Module

**to(*args, **kwargs)**
Moves and/or casts the parameters and buffers.

This can be called as .. function:: to(device=None, dtype=None, non_blocking=False) .. function:: to(dtype, non_blocking=False) .. function:: to(tensor, non_blocking=False) Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired *`dtype`* s. In addition, this method will only cast the floating point parameters and buffers to *`dtype`* (if given). The integral parameters and buffers will be moved *`device`*, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices. See below for examples.

**Note:** This method modifies the module in-place.

**Parameters**

- **device** – the desired device of the parameters and buffers in this module
- **dtype** – the desired floating point type of the floating point parameters and buffers in this module
- **tensor** – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

**Returns** self

**Return type** Module

**Example::**

```
>>> class ExampleModule(DeviceDtypeModuleMixin):
...     def __init__(self, weight: torch.Tensor):
...         super().__init__()
...         self.register_buffer('weight', weight)
>>> _ = torch.manual_seed(0)
>>> module = ExampleModule(torch.rand(3, 4))
>>> module.weight
tensor([[...]])
>>> module.to(torch.double)
ExampleModule()
>>> module.weight
tensor([[...]], dtype=torch.float64)
>>> cpu = torch.device('cpu')
>>> module.to(cpu, dtype=torch.half, non_blocking=True)
ExampleModule()
>>> module.weight
tensor([[...]], dtype=torch.float16)
>>> module.to(cpu)
ExampleModule()
>>> module.weight
tensor([[...]], dtype=torch.float16)
```

**type** (*dst_type*)

Casts all parameters and buffers to `dst_type`.

**Parameters** **dst_type** (*type or string*) – the desired type

**Returns** self

**Return type** Module

**_device: ... = None**

**_dtype: Union[str, torch.dtype] = None**

**property device**

**Return type** `Union`[`str`, `device`]

**property dtype**

**Return type** `Union`[`str`, `dtype`]

### pytorch_lightning.utilities.distributed module

pytorch_lightning.utilities.distributed.**_info**(**args, **kwargs*)

pytorch_lightning.utilities.distributed.**_warn**(**args, **kwargs*)

pytorch_lightning.utilities.distributed.**rank_zero_info**(**args, **kwargs*)

pytorch_lightning.utilities.distributed.**rank_zero_only**(*fn*)

pytorch_lightning.utilities.distributed.**rank_zero_warn**(**args, **kwargs*)

### pytorch_lightning.utilities.exceptions module

**exception** pytorch_lightning.utilities.exceptions.**MisconfigurationException**
Bases: Exception

### pytorch_lightning.utilities.memory module

pytorch_lightning.utilities.memory.**garbage_collection_cuda**()
Garbage collection Torch (CUDA) memory.

pytorch_lightning.utilities.memory.**is_cuda_out_of_memory**(*exception*)

pytorch_lightning.utilities.memory.**is_cudnn_snafu**(*exception*)

pytorch_lightning.utilities.memory.**is_oom_error**(*exception*)

pytorch_lightning.utilities.memory.**is_out_of_cpu_memory**(*exception*)

pytorch_lightning.utilities.memory.**recursive_detach**(*in_dict*)
Detach all tensors in *in_dict*.

May operate recursively if some of the values in *in_dict* are dictionaries which contain instances of *torch.Tensor*. Other types in *in_dict* are not affected by this utility function.

**Parameters** **in_dict** (dict) –

**Returns**

**Return type** out_dict

### pytorch_lightning.utilities.parsing module

**class** pytorch_lightning.utilities.parsing.**AttributeDict**
Bases: dict

Extended dictionary accesisable with dot notation.

```
>>> ad = AttributeDict({'key1': 1, 'key2': 'abc'})
>>> ad.key1
1
>>> ad.update({'my-key': 3.14})
>>> ad.update(mew_key=42)
>>> ad.key1 = 2
>>> ad
"key1":    2
"key2":    abc
"mew_key": 42
"my-key":  3.14
```

pytorch_lightning.utilities.parsing.**clean_namespace**(*hparams*)
Removes all functions from hparams so we can pickle.

pytorch_lightning.utilities.parsing.**collect_init_args**(*frame*, *path_args*, *inside=False*)
Recursively collects the arguments passed to the child constructors in the inheritance tree.

**Parameters**

- **frame** – the current stack frame
- **path_args** (list) – a list of dictionaries containing the constructor args in all parent classes
- **inside** (bool) – track if we are inside inheritance path, avoid terminating too soon

**Return type** list

**Returns** A list of dictionaries where each dictionary contains the arguments passed to the constructor at that level. The last entry corresponds to the constructor call of the most specific class in the hierarchy.

pytorch_lightning.utilities.parsing.**get_init_args**(*frame*)

**Return type** dict

pytorch_lightning.utilities.parsing.**str_to_bool**(*val*)
Convert a string representation of truth to true (1) or false (0). Copied from the python implementation distutils.utils.strtobool

True values are 'y', 'yes', 't', 'true', 'on', and '1'; false values are 'n', 'no', 'f', 'false', 'off', and '0'. Raises ValueError if 'val' is anything else.

```
>>> str_to_bool('YES')
1
>>> str_to_bool('FALSE')
0
```

### pytorch_lightning.utilities.seed module

Helper functions to help with reproducibility of models.

pytorch_lightning.utilities.seed.**_select_seed_randomly**(*min_seed_value=0*, *max_seed_value=255*)

**Return type** int

pytorch_lightning.utilities.seed.**seed_everything**(*seed=None*)
Function that sets seed for pseudo-random number generators in: pytorch, numpy, python.random and sets PYTHONHASHSEED environment variable.

**Return type** int

# PYTHON MODULE INDEX

## p


pytorch_lightning.callbacks, 334
pytorch_lightning.callbacks.base, 343
pytorch_lightning.callbacks.early_stopping, 344
pytorch_lightning.callbacks.gradient_accumulation_scheduler, 345
pytorch_lightning.callbacks.lr_logger, 346
pytorch_lightning.callbacks.model_checkpoint, 346
pytorch_lightning.callbacks.progress, 349
pytorch_lightning.core, 267
pytorch_lightning.core.decorators, 298
pytorch_lightning.core.grads, 299
pytorch_lightning.core.hooks, 299
pytorch_lightning.core.lightning, 303
pytorch_lightning.core.memory, 328
pytorch_lightning.core.saving, 331
pytorch_lightning.loggers, 352
pytorch_lightning.loggers.base, 373
pytorch_lightning.loggers.comet, 378
pytorch_lightning.loggers.mlflow, 380
pytorch_lightning.loggers.neptune, 381
pytorch_lightning.loggers.tensorboard, 386
pytorch_lightning.loggers.test_tube, 388
pytorch_lightning.loggers.trains, 390
pytorch_lightning.loggers.wandb, 394
pytorch_lightning.metrics, 396
pytorch_lightning.metrics.classification, 423
pytorch_lightning.metrics.converters, 435
pytorch_lightning.metrics.functional, 411
pytorch_lightning.metrics.functional.classification, 411
pytorch_lightning.metrics.functional.reduction, 423
pytorch_lightning.metrics.metric, 438
pytorch_lightning.metrics.regression, 440
pytorch_lightning.metrics.sklearns, 442
pytorch_lightning.overrides, 454
pytorch_lightning.overrides.data_parallel, 454
pytorch_lightning.profiler, 455
pytorch_lightning.profiler.profilers, 459
pytorch_lightning.trainer, 461
pytorch_lightning.trainer.auto_mix_precision, 488
pytorch_lightning.trainer.callback_config, 489
pytorch_lightning.trainer.callback_hook, 489
pytorch_lightning.trainer.data_loading, 491
pytorch_lightning.trainer.deprecated_api, 492
pytorch_lightning.trainer.distrib_data_parallel, 493
pytorch_lightning.trainer.distrib_parts, 497
pytorch_lightning.trainer.evaluation_loop, 500
pytorch_lightning.trainer.ignored_warnings, 504
pytorch_lightning.trainer.logging, 504
pytorch_lightning.trainer.lr_finder, 505
pytorch_lightning.trainer.model_hooks, 508
pytorch_lightning.trainer.optimizers, 508
pytorch_lightning.trainer.supporters, 509
pytorch_lightning.trainer.trainer, 510
pytorch_lightning.trainer.training_io, 519
pytorch_lightning.trainer.training_loop, 522
pytorch_lightning.trainer.training_tricks,

527
pytorch_lightning.utilities, 529
pytorch_lightning.utilities.apply_func, 529
pytorch_lightning.utilities.cloud_io, 530
pytorch_lightning.utilities.device_dtype_mixin, 530
pytorch_lightning.utilities.distributed, 532
pytorch_lightning.utilities.exceptions, 532
pytorch_lightning.utilities.memory, 532
pytorch_lightning.utilities.parsing, 532
pytorch_lightning.utilities.seed, 533

# INDEX

## Symbols


_ExponentialLR (class in pytorch_lightning.trainer.lr_finder), 506
_LRCallback (class in pytorch_lightning.trainer.lr_finder), 506
_LRFinder (class in pytorch_lightning.trainer.lr_finder), 507
_LightningModule__get_hparams_assignment_variable() (pytorch_lightning.core.LightningModule method), 273
_LightningModule__get_hparams_assignment_variable() (pytorch_lightning.core.lightning.LightningModule method), 303
_LinearLR (class in pytorch_lightning.trainer.lr_finder), 508
_MockOptimizer (class in pytorch_lightning.trainer.optimizers), 508
_PatchDataLoader (class in pytorch_lightning.trainer.trainer), 519
_TrainerDPMixin__transfer_batch_to_device() (pytorch_lightning.trainer.distrib_parts.TrainerDPMixin method), 497
_TrainerLRFinderMixin__lr_finder_dump_params() (pytorch_lightning.trainer.lr_finder.TrainerLRFinderMixin method), 505
_TrainerLRFinderMixin__lr_finder_restore_params() (pytorch_lightning.trainer.lr_finder.TrainerLRFinderMixin method), 505
_TrainerTrainingTricksMixin__scale_batch_dump_params() (pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixin method), 527
_TrainerTrainingTricksMixin__scale_batch_reset_params() (pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixin method), 527
_TrainerTrainingTricksMixin__scale_batch_restore_params() (pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixin method), 527
_Trainer__attach_dataloaders() (pytorch_lightning.trainer.Trainer method), 483
_Trainer__attach_dataloaders() (pytorch_lightning.trainer.trainer.Trainer method), 514
__call__() (pytorch_lightning.metrics.metric.NumpyMetric method), 438
__call__() (pytorch_lightning.metrics.metric.TensorCollectionMetric method), 439
__call__() (pytorch_lightning.metrics.metric.TensorMetric method), 439
__call__() (pytorch_lightning.trainer.trainer._PatchDataLoader method), 519
_adjust_batch_size() (in module pytorch_lightning.trainer.training_tricks), 528
_agg_memory() (pytorch_lightning.trainer.supporters.TensorRunningAccum method), 509
_aggregate_metrics() (pytorch_lightning.loggers.LightningLoggerBase method), 353
_aggregate_metrics() (pytorch_lightning.loggers.base.LightningLoggerBase method), 374
_allowed_type() (pytorch_lightning.trainer.Trainer method), 483
_allowed_type() (pytorch_lightning.trainer.trainer.Trainer method), 514
_apply_to_inputs() (in module pytorch_lightning.metrics.converters), 435
_apply_to_outputs() (in module pytorch_lightning.metrics.converters), 435
_arg_default() (pytorch_lightning.trainer.Trainer method), 483
_arg_default() (pytorch_lightning.trainer.trainer.Trainer method), 514
_atomic_save() (pytorch_lightning.trainer.training_io.TrainerIOMixin method), 521
_auto_collect_arguments() (pytorch_lightning.core.LightningModule class method), 273
_auto_collect_arguments() (pytorch_lightning.core.lightning.LightningModule class method), 303

_binary_clf_curve() (in module pytorch_lightning.metrics.functional.classification), 411
_bypass (pytorch_lightning.loggers.TrainsLogger attribute), 372
_bypass (pytorch_lightning.loggers.trains.TrainsLogger attribute), 393
_check_batch_limits() (pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin method), 491
_convert_params() (pytorch_lightning.loggers.LightningLoggerBase static method), 353
_convert_params() (pytorch_lightning.loggers.base.LightningLoggerBase static method), 374
_convert_to_numpy() (in module pytorch_lightning.metrics.converters), 435
_convert_to_tensor() (in module pytorch_lightning.metrics.converters), 435
_create_or_get_experiment() (pytorch_lightning.loggers.NeptuneLogger method), 364
_create_or_get_experiment() (pytorch_lightning.loggers.neptune.NeptuneLogger method), 384
_del_model() (pytorch_lightning.callbacks.ModelCheckpoint method), 337
_del_model() (pytorch_lightning.callbacks.model_checkpoint.ModelCheckpoint method), 348
_determine_limit_batches() (in module pytorch_lightning.trainer.trainer), 519
_device (pytorch_lightning.core.LightningModule attribute), 297
_device (pytorch_lightning.core.lightning.LightningModule attribute), 327
_device (pytorch_lightning.metrics.AUC attribute), 399
_device (pytorch_lightning.metrics.AUROC attribute), 399
_device (pytorch_lightning.metrics.Accuracy attribute), 400
_device (pytorch_lightning.metrics.AveragePrecision attribute), 401
_device (pytorch_lightning.metrics.ConfusionMatrix attribute), 401
_device (pytorch_lightning.metrics.DiceCoefficient attribute), 402
_device (pytorch_lightning.metrics.F1 attribute), 403
_device (pytorch_lightning.metrics.FBeta attribute), 404
_device (pytorch_lightning.metrics.IoU attribute), 410
_device (pytorch_lightning.metrics.MAE attribute), 397
_device (pytorch_lightning.metrics.MSE attribute), 396
_device (pytorch_lightning.metrics.MulticlassPrecisionRecall attribute), 405
_device (pytorch_lightning.metrics.MulticlassROC attribute), 406
_device (pytorch_lightning.metrics.Precision attribute), 406
_device (pytorch_lightning.metrics.PrecisionRecall attribute), 407
_device (pytorch_lightning.metrics.PrecisionRecallCurve attribute), 408
_device (pytorch_lightning.metrics.RMSE attribute), 397
_device (pytorch_lightning.metrics.RMSLE attribute), 398
_device (pytorch_lightning.metrics.ROC attribute), 409
_device (pytorch_lightning.metrics.Recall attribute), 410
_device (pytorch_lightning.metrics.SklearnMetric attribute), 411
_device (pytorch_lightning.metrics.classification.AUROC attribute), 424
_device (pytorch_lightning.metrics.classification.Accuracy attribute), 424
_device (pytorch_lightning.metrics.classification.AveragePrecision attribute), 425
_device (pytorch_lightning.metrics.classification.ConfusionMatrix attribute), 426
_device (pytorch_lightning.metrics.classification.DiceCoefficient attribute), 427
_device (pytorch_lightning.metrics.classification.F1 attribute), 428
_device (pytorch_lightning.metrics.classification.FBeta attribute), 428
_device (pytorch_lightning.metrics.classification.IoU attribute), 429
_device (pytorch_lightning.metrics.classification.MulticlassPrecisionRec attribute), 430
_device (pytorch_lightning.metrics.classification.MulticlassROC attribute), 431
_device (pytorch_lightning.metrics.classification.Precision attribute), 432
_device (pytorch_lightning.metrics.classification.PrecisionRecall attribute), 433
_device (pytorch_lightning.metrics.classification.ROC attribute), 434
_device (pytorch_lightning.metrics.classification.Recall attribute), 435
_device (pytorch_lightning.metrics.metric.Metric attribute), 438
_device (pytorch_lightning.metrics.metric.NumpyMetric attribute), 438

_device (*pytorch_lightning.metrics.metric.TensorCollectionMetric attribute*), 439
_device (*pytorch_lightning.metrics.metric.TensorMetric attribute*), 439
_device (*pytorch_lightning.metrics.regression.MAE attribute*), 441
_device (*pytorch_lightning.metrics.regression.MSE attribute*), 440
_device (*pytorch_lightning.metrics.regression.RMSE attribute*), 441
_device (*pytorch_lightning.metrics.regression.RMSLE attribute*), 442
_device (*pytorch_lightning.metrics.sklearns.AUC attribute*), 443
_device (*pytorch_lightning.metrics.sklearns.AUROC attribute*), 443
_device (*pytorch_lightning.metrics.sklearns.Accuracy attribute*), 444
_device (*pytorch_lightning.metrics.sklearns.AveragePrecision attribute*), 445
_device (*pytorch_lightning.metrics.sklearns.ConfusionMatrix attribute*), 446
_device (*pytorch_lightning.metrics.sklearns.F1 attribute*), 447
_device (*pytorch_lightning.metrics.sklearns.FBeta attribute*), 448
_device (*pytorch_lightning.metrics.sklearns.Precision attribute*), 450
_device (*pytorch_lightning.metrics.sklearns.PrecisionRecallCurve attribute*), 450
_device (*pytorch_lightning.metrics.sklearns.ROC attribute*), 452
_device (*pytorch_lightning.metrics.sklearns.Recall attribute*), 453
_device (*pytorch_lightning.metrics.sklearns.SklearnMetric attribute*), 453
_device (*pytorch_lightning.utilities.device_dtype_mixin.DeviceDtypeModuleMixin attribute*), 531
_do_check_save() (*pytorch_lightning.callbacks.ModelCheckpoint method*), 337
_do_check_save() (*pytorch_lightning.callbacks.model_checkpoint.ModelCheckpoint method*), 348
_dtype (*pytorch_lightning.core.LightningModule attribute*), 297
_dtype (*pytorch_lightning.core.lightning.LightningModule attribute*), 327
_dtype (*pytorch_lightning.metrics.AUC attribute*), 399
_dtype (*pytorch_lightning.metrics.AUROC attribute*), 399
_dtype (*pytorch_lightning.metrics.Accuracy attribute*), 400
_dtype (*pytorch_lightning.metrics.AveragePrecision attribute*), 401
_dtype (*pytorch_lightning.metrics.ConfusionMatrix attribute*), 401
_dtype (*pytorch_lightning.metrics.DiceCoefficient attribute*), 402
_dtype (*pytorch_lightning.metrics.F1 attribute*), 403
_dtype (*pytorch_lightning.metrics.FBeta attribute*), 404
_dtype (*pytorch_lightning.metrics.IoU attribute*), 410
_dtype (*pytorch_lightning.metrics.MAE attribute*), 397
_dtype (*pytorch_lightning.metrics.MSE attribute*), 396
_dtype (*pytorch_lightning.metrics.MulticlassPrecisionRecall attribute*), 405
_dtype (*pytorch_lightning.metrics.MulticlassROC attribute*), 406
_dtype (*pytorch_lightning.metrics.Precision attribute*), 406
_dtype (*pytorch_lightning.metrics.PrecisionRecall attribute*), 407
_dtype (*pytorch_lightning.metrics.PrecisionRecallCurve attribute*), 408
_dtype (*pytorch_lightning.metrics.RMSE attribute*), 397
_dtype (*pytorch_lightning.metrics.RMSLE attribute*), 398
_dtype (*pytorch_lightning.metrics.ROC attribute*), 409
_dtype (*pytorch_lightning.metrics.Recall attribute*), 410
_dtype (*pytorch_lightning.metrics.SklearnMetric attribute*), 411
_dtype (*pytorch_lightning.metrics.classification.AUROC attribute*), 424
_dtype (*pytorch_lightning.metrics.classification.Accuracy attribute*), 424
_dtype (*pytorch_lightning.metrics.classification.AveragePrecision attribute*), 425
_dtype (*pytorch_lightning.metrics.classification.ConfusionMatrix attribute*), 426
_dtype (*pytorch_lightning.metrics.classification.DiceCoefficient attribute*), 427
_dtype (*pytorch_lightning.metrics.classification.F1 attribute*), 428
_dtype (*pytorch_lightning.metrics.classification.FBeta attribute*), 428
_dtype (*pytorch_lightning.metrics.classification.IoU attribute*), 429
_dtype (*pytorch_lightning.metrics.classification.MulticlassPrecisionReca attribute*), 430
_dtype (*pytorch_lightning.metrics.classification.MulticlassROC attribute*), 431
_dtype (*pytorch_lightning.metrics.classification.Precision attribute*), 432
_dtype (*pytorch_lightning.metrics.classification.PrecisionRecall attribute*), 433

_dtype (*pytorch_lightning.metrics.classification.ROC attribute*), 434
_dtype (*pytorch_lightning.metrics.classification.Recall attribute*), 435
_dtype (*pytorch_lightning.metrics.metric.Metric attribute*), 438
_dtype (*pytorch_lightning.metrics.metric.NumpyMetric attribute*), 438
_dtype (*pytorch_lightning.metrics.metric.TensorCollectionMetric attribute*), 439
_dtype (*pytorch_lightning.metrics.metric.TensorMetric attribute*), 439
_dtype (*pytorch_lightning.metrics.regression.MAE attribute*), 441
_dtype (*pytorch_lightning.metrics.regression.MSE attribute*), 440
_dtype (*pytorch_lightning.metrics.regression.RMSE attribute*), 441
_dtype (*pytorch_lightning.metrics.regression.RMSLE attribute*), 442
_dtype (*pytorch_lightning.metrics.sklearns.AUC attribute*), 443
_dtype (*pytorch_lightning.metrics.sklearns.AUROC attribute*), 443
_dtype (*pytorch_lightning.metrics.sklearns.Accuracy attribute*), 444
_dtype (*pytorch_lightning.metrics.sklearns.AveragePrecision attribute*), 445
_dtype (*pytorch_lightning.metrics.sklearns.ConfusionMatrix attribute*), 446
_dtype (*pytorch_lightning.metrics.sklearns.F1 attribute*), 447
_dtype (*pytorch_lightning.metrics.sklearns.FBeta attribute*), 448
_dtype (*pytorch_lightning.metrics.sklearns.Precision attribute*), 450
_dtype (*pytorch_lightning.metrics.sklearns.PrecisionRecallCurve attribute*), 450
_dtype (*pytorch_lightning.metrics.sklearns.ROC attribute*), 452
_dtype (*pytorch_lightning.metrics.sklearns.Recall attribute*), 453
_dtype (*pytorch_lightning.metrics.sklearns.SklearnMetric attribute*), 453
_dtype (*pytorch_lightning.utilities.device_dtype_mixin.DeviceDtypeModuleMixin attribute*), 531
_evaluate() (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin method*), 502
_extract_lr() (*pytorch_lightning.callbacks.LearningRateLogger method*), 339
_extract_lr() (*pytorch_lightning.callbacks.lr_logger.LearningRateLogger method*), 346
_finalize_agg_metrics() (*pytorch_lightning.loggers.LightningLoggerBase method*), 353
_finalize_agg_metrics() (*pytorch_lightning.loggers.base.LightningLoggerBase method*), 374
_find_names() (*pytorch_lightning.callbacks.LearningRateLogger method*), 339
_find_names() (*pytorch_lightning.callbacks.lr_logger.LearningRateLogger method*), 346
_find_tensors() (*in module pytorch_lightning.overrides.data_parallel*), 454
_flatten_dict() (*pytorch_lightning.loggers.LightningLoggerBase static method*), 353
_flatten_dict() (*pytorch_lightning.loggers.base.LightningLoggerBase static method*), 374
_format_summary_table() (*in module pytorch_lightning.core.memory*), 330
_forward_example_input() (*pytorch_lightning.core.memory.ModelSummary method*), 329
_get_distributed_sampler() (*pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin method*), 491
_get_new_optimizer() (*pytorch_lightning.trainer.lr_finder._LRFinder method*), 507
_get_next_version() (*pytorch_lightning.loggers.TensorBoardLogger method*), 356
_get_next_version() (*pytorch_lightning.loggers.tensorboard.TensorBoardLogger method*), 387
_get_optimizers_iterable() (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method*), 524
_has_len() (*in module pytorch_lightning.trainer.data_loading*), 492
_info() (*in module pytorch_lightning.utilities.distributed*), 532
_init_slurm_connection() (*pytorch_lightning.core.LightningModule method*), 274
_init_slurm_connection() (*pytorch_lightning.core.lightning.LightningModule method*), 303
_load_model_state() (*pytorch_lightning.core.saving.ModelIO class method*), 331

_nested_hasattr() (in module pytorch_lightning.trainer.lr_finder), 508
_nested_setattr() (in module pytorch_lightning.trainer.lr_finder), 508
_numpy_metric_conversion() (in module pytorch_lightning.metrics.converters), 435
_numpy_metric_input_conversion() (in module pytorch_lightning.metrics.converters), 436
_reduce_agg_metrics() (pytorch_lightning.loggers.LightningLoggerBase method), 353
_reduce_agg_metrics() (pytorch_lightning.loggers.base.LightningLoggerBase method), 374
_register_hook() (pytorch_lightning.core.memory.LayerSummary method), 328
_reset_eval_dataloader() (pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin method), 491
_run_binsearch_scaling() (in module pytorch_lightning.trainer.training_tricks), 529
_run_early_stopping_check() (pytorch_lightning.callbacks.EarlyStopping method), 336
_run_early_stopping_check() (pytorch_lightning.callbacks.early_stopping.EarlyStopping method), 344
_run_lr_finder_internally() (pytorch_lightning.trainer.lr_finder.TrainerLRFinderMixin method), 505
_run_power_scaling() (in module pytorch_lightning.trainer.training_tricks), 529
_sanitize_params() (pytorch_lightning.loggers.LightningLoggerBase static method), 353
_sanitize_params() (pytorch_lightning.loggers.base.LightningLoggerBase static method), 374
_save_model() (pytorch_lightning.callbacks.ModelCheckpoint method), 337
_save_model() (pytorch_lightning.callbacks.model_checkpoint.ModelCheckpoint method), 348
_select_seed_randomly() (in module pytorch_lightning.utilities.seed), 533
_set_horovod_backend() (pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin method), 495
_set_hparams() (pytorch_lightning.core.LightningModule method), 274
_set_hparams() (pytorch_lightning.core.lightning.LightningModule method), 303
_sync_ddp_if_available() (in module pytorch_lightning.metrics.converters), 436
_tensor_collection_metric_conversion() (in module pytorch_lightning.metrics.converters), 436
_tensor_collection_metric_output_conversion() (in module pytorch_lightning.metrics.converters), 436
_tensor_metric_conversion() (in module pytorch_lightning.metrics.converters), 436
_tensor_metric_input_conversion() (in module pytorch_lightning.metrics.converters), 436
_tensor_metric_output_conversion() (in module pytorch_lightning.metrics.converters), 437
_validate_condition_metric() (pytorch_lightning.callbacks.EarlyStopping method), 336
_validate_condition_metric() (pytorch_lightning.callbacks.early_stopping.EarlyStopping method), 345
_warn() (in module pytorch_lightning.utilities.distributed), 532
_with_is_last() (in module pytorch_lightning.trainer.training_loop), 527
_worker_check() (pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin method), 491


## A


accumulate_grad_batches (pytorch_lightning.trainer.Trainer attribute), 487
accumulate_grad_batches (pytorch_lightning.trainer.trainer.Trainer attribute), 518
accumulate_grad_batches (pytorch_lightning.trainer.training_io.TrainerIOMixin attribute), 522
accumulate_grad_batches (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 525
accumulation_scheduler (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 525
Accuracy (class in pytorch_lightning.metrics), 399
Accuracy (class in pytorch_lightning.metrics.classification), 424
Accuracy (class in pytorch_lightning.metrics.sklearns), 443

accuracy() (in module pytorch_lightning.metrics.functional.classification), 411
add_argparse_args() (pytorch_lightning.trainer.Trainer class method), 483
add_argparse_args() (pytorch_lightning.trainer.trainer.Trainer class method), 515
add_param_group() (pytorch_lightning.trainer.optimizers._MockOptimizer method), 509
add_progress_bar_metrics() (pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin method), 502
add_progress_bar_metrics() (pytorch_lightning.trainer.logging.TrainerLoggingMixin method), 504
add_progress_bar_metrics() (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method), 524
AdvancedProfiler (class in pytorch_lightning.profiler), 457
AdvancedProfiler (class in pytorch_lightning.profiler.profilers), 459
agg_and_log_metrics() (pytorch_lightning.loggers.base.LightningLoggerBase method), 375
agg_and_log_metrics() (pytorch_lightning.loggers.LightningLoggerBase method), 354
amp_level (pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin attribute), 496
amp_level (pytorch_lightning.trainer.distrib_parts.TrainerDPMixin attribute), 498
append() (pytorch_lightning.trainer.supporters.TensorRunningAccum method), 509
append_tags() (pytorch_lightning.loggers.neptune.NeptuneLogger method), 384
append_tags() (pytorch_lightning.loggers.NeptuneLogger method), 364
apply_to_collection() (in module pytorch_lightning.utilities.apply_func), 529
AttributeDict (class in pytorch_lightning.utilities.parsing), 532
AUC (class in pytorch_lightning.metrics), 398
AUC (class in pytorch_lightning.metrics.sklearns), 442
auc() (in module pytorch_lightning.metrics.functional.classification), 412
auc_decorator() (in module pytorch_lightning.metrics.functional.classification), 412
AUROC (class in pytorch_lightning.metrics), 399
AUROC (class in pytorch_lightning.metrics.classification), 423
AUROC (class in pytorch_lightning.metrics.sklearns), 443
auroc() (in module pytorch_lightning.metrics.functional.classification), 412
auto_add_sampler() (pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin method), 491
auto_move_data() (in module pytorch_lightning.core.decorators), 298
auto_squeeze_dim_zeros() (in module pytorch_lightning.overrides.data_parallel), 454
average_precision() (in module pytorch_lightning.metrics.functional.classification), 413
AveragePrecision (class in pytorch_lightning.metrics), 400
AveragePrecision (class in pytorch_lightning.metrics.classification), 424
AveragePrecision (class in pytorch_lightning.metrics.sklearns), 444

## B

backward() (pytorch_lightning.core.hooks.ModelHooks method), 299
barrier() (pytorch_lightning.trainer.Trainer method), 484
barrier() (pytorch_lightning.trainer.trainer.Trainer method), 515
base_lrs (pytorch_lightning.trainer.lr_finder._ExponentialLR attribute), 506
base_lrs (pytorch_lightning.trainer.lr_finder._LinearLR attribute), 508
BaseProfiler (class in pytorch_lightning.profiler), 456
BaseProfiler (class in pytorch_lightning.profiler.profilers), 459
batch_idx (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 525
best() (pytorch_lightning.callbacks.model_checkpoint.ModelCheckpoint property), 348
best() (pytorch_lightning.callbacks.ModelCheckpoint property), 338
bypass_mode() (pytorch_lightning.loggers.trains.TrainsLogger class method), 391
bypass_mode() (pytorch_lightning.loggers.TrainsLogger class method), 370

## C

call_checkpoint_callback() (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method), 524
Callback (class in pytorch_lightning.callbacks), 334
Callback (class in pytorch_lightning.callbacks.base), 343
callback_metrics (pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin attribute), 502
callback_metrics (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
callbacks (pytorch_lightning.trainer.callback_config.TrainerCallbackConfigMixin attribute), 489
callbacks (pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin attribute), 490
callbacks (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
can_prepare_data() (pytorch_lightning.trainer.Trainer method), 484
can_prepare_data() (pytorch_lightning.trainer.trainer.Trainer method), 515
check_gpus_data_type() (in module pytorch_lightning.trainer.distrib_parts), 499
check_horovod() (pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin method), 495
check_model_configuration() (pytorch_lightning.trainer.Trainer method), 484
check_model_configuration() (pytorch_lightning.trainer.trainer.Trainer method), 515
check_monitor_top_k() (pytorch_lightning.callbacks.model_checkpoint.ModelCheckpoint method), 348
check_monitor_top_k() (pytorch_lightning.callbacks.ModelCheckpoint method), 337
check_val_every_n_epoch (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
checkpoint_callback (pytorch_lightning.trainer.callback_config.TrainerCallbackConfigMixin attribute), 489
checkpoint_callback (pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin attribute), 496
checkpoint_callback (pytorch_lightning.trainer.Trainer attribute), 487
checkpoint_callback (pytorch_lightning.trainer.trainer.Trainer attribute), 518
checkpoint_callback (pytorch_lightning.trainer.training_io.TrainerIOMixin attribute), 522
checkpoint_callback (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
CHECKPOINT_HYPER_PARAMS_KEY (pytorch_lightning.core.saving.ModelIO attribute), 333
CHECKPOINT_HYPER_PARAMS_NAME (pytorch_lightning.core.saving.ModelIO attribute), 333
CHECKPOINT_HYPER_PARAMS_TYPE (pytorch_lightning.core.saving.ModelIO attribute), 333
ckpt_path (pytorch_lightning.trainer.callback_config.TrainerCallbackCo attribute), 489
clean_namespace() (in module pytorch_lightning.utilities.parsing), 532
clip_gradients() (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method), 525
clip_gradients() (pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixi method), 527
close() (pytorch_lightning.loggers.base.LightningLoggerBase method), 375
close() (pytorch_lightning.loggers.base.LoggerCollection method), 376
close() (pytorch_lightning.loggers.LightningLoggerBase method), 354
close() (pytorch_lightning.loggers.LoggerCollection method), 355
close() (pytorch_lightning.loggers.test_tube.TestTubeLogger method), 389
close() (pytorch_lightning.loggers.TestTubeLogger method), 366
collect_init_args() (in module pytorch_lightning.utilities.parsing), 533
CometLogger (class in pytorch_lightning.loggers), 357
CometLogger (class in pytorch_lightning.loggers.comet), 378
configure_accumulated_gradients() (pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixi method), 527
configure_apex() (pytorch_lightning.core.lightning.LightningModule method), 303
configure_apex() (pytorch_lightning.core.LightningModule method), 274

configure_checkpoint_callback() (*pytorch_lightning.trainer.callback_config.TrainerCallbackConfigMixin method*), 489
configure_ddp() (*pytorch_lightning.core.lightning.LightningModule method*), 304
configure_ddp() (*pytorch_lightning.core.LightningModule method*), 274
configure_early_stopping() (*pytorch_lightning.trainer.callback_config.TrainerCallbackConfigMixin method*), 489
configure_logger() (*pytorch_lightning.trainer.logging.TrainerLoggingMixin method*), 504
configure_optimizers() (*pytorch_lightning.core.lightning.LightningModule method*), 304
configure_optimizers() (*pytorch_lightning.core.LightningModule method*), 275
configure_progress_bar() (*pytorch_lightning.trainer.callback_config.TrainerCallbackConfigMixin method*), 489
configure_schedulers() (*pytorch_lightning.trainer.optimizers.TrainerOptimizersMixin method*), 508
configure_slurm_ddp() (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin method*), 495
confusion_matrix() (*in module pytorch_lightning.metrics.functional.classification*), 413
ConfusionMatrix (*class in pytorch_lightning.metrics*), 401
ConfusionMatrix (*class in pytorch_lightning.metrics.classification*), 425
ConfusionMatrix (*class in pytorch_lightning.metrics.sklearns*), 445
convert() (*in module pytorch_lightning.core.saving*), 333
convert_inf() (*in module pytorch_lightning.callbacks.progress*), 352
copy_trainer_model_properties() (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin method*), 495
copy_trainer_model_properties() (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin method*), 497
copy_trainer_model_properties() (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin method*), 502
cpu() (*pytorch_lightning.utilities.device_dtype_mixin.DeviceDtypeModuleMixin method*), 530
cuda() (*pytorch_lightning.utilities.device_dtype_mixin.DeviceDtypeModuleMixin method*), 530
current_epoch (*pytorch_lightning.core.lightning.LightningModule attribute*), 327
current_epoch (*pytorch_lightning.core.LightningModule attribute*), 297
current_epoch (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin attribute*), 503
current_epoch (*pytorch_lightning.trainer.logging.TrainerLoggingMixin attribute*), 504


## D


data_loader() (*in module pytorch_lightning.core*), 298
data_loader() (*in module pytorch_lightning.core.decorators*), 299
data_parallel() (*pytorch_lightning.trainer.Trainer property*), 487
data_parallel() (*pytorch_lightning.trainer.trainer.Trainer property*), 518
data_parallel_device_ids (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin attribute*), 496
data_parallel_device_ids (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin attribute*), 498
data_parallel_device_ids (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin attribute*), 503
data_parallel_device_ids (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 526
ddp_train() (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin method*), 495
default_attributes() (*pytorch_lightning.trainer.Trainer class method*), 484
default_attributes() (*pytorch_lightning.trainer.trainer.Trainer class method*), 515
default_root_dir (*pytorch_lightning.trainer.callback_config.TrainerCallbackConfigMixin attribute*), 489
default_root_dir (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin attribute*), 496
default_root_dir (*pytorch_lightning.trainer.logging.TrainerLoggingMixin attribute*), 504

default_root_dir (pytorch_lightning.trainer.lr_finder.TrainerLRFinderMixin attribute), 506
default_root_dir (pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixin attribute), 528
DEPRECATED_IN_0_9 (pytorch_lightning.trainer.Trainer attribute), 487
DEPRECATED_IN_0_9 (pytorch_lightning.trainer.trainer.Trainer attribute), 518
describe() (pytorch_lightning.profiler.AdvancedProfiler method), 458
describe() (pytorch_lightning.profiler.BaseProfiler method), 456
describe() (pytorch_lightning.profiler.profilers.AdvancedProfiler method), 459
describe() (pytorch_lightning.profiler.profilers.BaseProfiler method), 459
describe() (pytorch_lightning.profiler.profilers.SimpleProfiler method), 460
describe() (pytorch_lightning.profiler.SimpleProfiler method), 457
detect_nan_tensors() (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method), 525
detect_nan_tensors() (pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixin method), 527
determine_data_use_amount() (pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin method), 491
determine_ddp_node_rank() (pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin method), 495
determine_local_rank() (pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin method), 495
determine_root_gpu_device() (in module pytorch_lightning.trainer.distrib_parts), 499
device() (pytorch_lightning.utilities.device_dtype_mixin.DeviceDtypeModuleMixin property), 531
DeviceDtypeModuleMixin (class in pytorch_lightning.utilities.device_dtype_mixin), 530
dice_score() (in module pytorch_lightning.metrics.functional.classification), 414
DiceCoefficient (class in pytorch_lightning.metrics), 401
DiceCoefficient (class in pytorch_lightning.metrics.classification), 426
disable() (pytorch_lightning.callbacks.progress.ProgressBar method), 349
disable() (pytorch_lightning.callbacks.progress.ProgressBarBase method), 351
disable() (pytorch_lightning.callbacks.ProgressBar method), 341
disable() (pytorch_lightning.callbacks.ProgressBarBase method), 339
disable_validation (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
distributed_backend (pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin attribute), 492
distributed_backend (pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin attribute), 496
double() (pytorch_lightning.utilities.device_dtype_mixin.DeviceDtypeModuleMixin method), 530
dp_train() (pytorch_lightning.trainer.distrib_parts.TrainerDPMixin method), 497
dtype() (pytorch_lightning.utilities.device_dtype_mixin.DeviceDtypeModuleMixin property), 531
DummyExperiment (class in pytorch_lightning.loggers.base), 373
DummyLogger (class in pytorch_lightning.loggers.base), 373
dump_checkpoint() (pytorch_lightning.trainer.training_io.TrainerIOMixin method), 521


## E


early_stop_callback (pytorch_lightning.trainer.Trainer attribute), 487
early_stop_callback (pytorch_lightning.trainer.trainer.Trainer attribute), 518
early_stop_callback (pytorch_lightning.trainer.training_io.TrainerIOMixin attribute), 522
early_stop_callback (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
EarlyStopping (class in pytorch_lightning.callbacks), 335
EarlyStopping (class in pytorch_lightning.callbacks.early_stopping), 344
enable() (pytorch_lightning.callbacks.progress.ProgressBar method), 349
enable() (pytorch_lightning.callbacks.progress.ProgressBarBase method), 351
enable() (pytorch_lightning.callbacks.ProgressBar method), 341

enable() (*pytorch_lightning.callbacks.ProgressBarBase method*), 339
enable_early_stop (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 526
evaluation_forward() (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin method*), 502
example_input_array() (*pytorch_lightning.core.lightning.LightningModule property*), 327
example_input_array() (*pytorch_lightning.core.LightningModule property*), 297
experiment() (*pytorch_lightning.loggers.base.DummyLogger property*), 373
experiment() (*pytorch_lightning.loggers.base.LightningLoggerBase property*), 376
experiment() (*pytorch_lightning.loggers.base.LoggerCollection property*), 377
experiment() (*pytorch_lightning.loggers.comet.CometLogger property*), 379
experiment() (*pytorch_lightning.loggers.CometLogger property*), 359
experiment() (*pytorch_lightning.loggers.LightningLoggerBase property*), 355
experiment() (*pytorch_lightning.loggers.LoggerCollection property*), 356
experiment() (*pytorch_lightning.loggers.mlflow.MLFlowLogger property*), 381
experiment() (*pytorch_lightning.loggers.MLFlowLogger property*), 361
experiment() (*pytorch_lightning.loggers.neptune.NeptuneLogger property*), 386
experiment() (*pytorch_lightning.loggers.NeptuneLogger property*), 365
experiment() (*pytorch_lightning.loggers.tensorboard.TensorBoardLogger property*), 387
experiment() (*pytorch_lightning.loggers.TensorBoardLogger property*), 357
experiment() (*pytorch_lightning.loggers.test_tube.TestTubeLogger property*), 389
experiment() (*pytorch_lightning.loggers.TestTubeLogger property*), 367
experiment() (*pytorch_lightning.loggers.trains.TrainsLogger property*), 393
experiment() (*pytorch_lightning.loggers.TrainsLogger property*), 372
experiment() (*pytorch_lightning.loggers.wandb.WandbLogger property*), 395
experiment() (*pytorch_lightning.loggers.WandbLogger property*), 368

# F

F1 (*class in pytorch_lightning.metrics*), 402
F1 (*class in pytorch_lightning.metrics.classification*), 427
F1 (*class in pytorch_lightning.metrics.sklearns*), 446
f1_score() (*in module pytorch_lightning.metrics.functional.classification*), 415
fast_dev_run (*pytorch_lightning.trainer.evaluation_loop.TrainerEvalu attribute*), 503
fast_dev_run (*pytorch_lightning.trainer.training_loop.TrainerTrainLo attribute*), 526
FBeta (*class in pytorch_lightning.metrics*), 403
FBeta (*class in pytorch_lightning.metrics.classification*), 428
FBeta (*class in pytorch_lightning.metrics.sklearns*), 447
fbeta_score() (*in module pytorch_lightning.metrics.functional.classification*), 415
finalize() (*pytorch_lightning.loggers.base.LightningLoggerBase method*), 375
finalize() (*pytorch_lightning.loggers.base.LoggerCollection method*), 376
finalize() (*pytorch_lightning.loggers.comet.CometLogger method*), 379
finalize() (*pytorch_lightning.loggers.CometLogger method*), 359
finalize() (*pytorch_lightning.loggers.LightningLoggerBase method*), 354
finalize() (*pytorch_lightning.loggers.LoggerCollection method*), 355
finalize() (*pytorch_lightning.loggers.mlflow.MLFlowLogger method*), 380
finalize() (*pytorch_lightning.loggers.MLFlowLogger method*), 360
finalize() (*pytorch_lightning.loggers.neptune.NeptuneLogger method*), 384
finalize() (*pytorch_lightning.loggers.NeptuneLogger method*), 364
finalize() (*pytorch_lightning.loggers.tensorboard.TensorBoardLogger method*), 387
finalize() (*pytorch_lightning.loggers.TensorBoardLogger method*), 356
finalize() (*pytorch_lightning.loggers.test_tube.TestTubeLogger method*), 389
finalize() (*pytorch_lightning.loggers.TestTubeLogger method*), 366
finalize() (*pytorch_lightning.loggers.trains.TrainsLogger method*), 391
finalize() (*pytorch_lightning.loggers.TrainsLogger method*), 370
fit() (*pytorch_lightning.trainer.lr_finder.TrainerLRFinderMixin method*), 505
fit() (*pytorch_lightning.trainer.Trainer method*), 484

fit() (pytorch_lightning.trainer.trainer.Trainer method), 515
fit() (pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixin method), 527
float() (pytorch_lightning.utilities.device_dtype_mixin.DeviceDtypeModuleMixin method), 530
format_checkpoint_name() (pytorch_lightning.callbacks.model_checkpoint.ModelCheckpoint method), 348
format_checkpoint_name() (pytorch_lightning.callbacks.ModelCheckpoint method), 337
forward() (pytorch_lightning.core.lightning.LightningModule method), 306
forward() (pytorch_lightning.core.LightningModule method), 277
forward() (pytorch_lightning.metrics.Accuracy method), 400
forward() (pytorch_lightning.metrics.AUC method), 398
forward() (pytorch_lightning.metrics.AUROC method), 399
forward() (pytorch_lightning.metrics.AveragePrecision method), 400
forward() (pytorch_lightning.metrics.classification.Accuracy method), 424
forward() (pytorch_lightning.metrics.classification.AUROC method), 423
forward() (pytorch_lightning.metrics.classification.AveragePrecision method), 425
forward() (pytorch_lightning.metrics.classification.ConfusionMatrix method), 426
forward() (pytorch_lightning.metrics.classification.DiceCoefficient method), 427
forward() (pytorch_lightning.metrics.classification.F1 method), 428
forward() (pytorch_lightning.metrics.classification.FBeta method), 428
forward() (pytorch_lightning.metrics.classification.IoU method), 429
forward() (pytorch_lightning.metrics.classification.MulticlassPrecisionRecall method), 430
forward() (pytorch_lightning.metrics.classification.MulticlassROC method), 431
forward() (pytorch_lightning.metrics.classification.Precision method), 432
forward() (pytorch_lightning.metrics.classification.PrecisionRecall method), 433
forward() (pytorch_lightning.metrics.classification.Recall method), 434
forward() (pytorch_lightning.metrics.classification.ROC method), 433
forward() (pytorch_lightning.metrics.ConfusionMatrix method), 401
forward() (pytorch_lightning.metrics.DiceCoefficient method), 402
forward() (pytorch_lightning.metrics.F1 method), 403
forward() (pytorch_lightning.metrics.FBeta method), 404
forward() (pytorch_lightning.metrics.IoU method), 410
forward() (pytorch_lightning.metrics.MAE method), 397
forward() (pytorch_lightning.metrics.metric.Metric method), 438
forward() (pytorch_lightning.metrics.MSE method), 396
forward() (pytorch_lightning.metrics.MulticlassPrecisionRecall method), 405
forward() (pytorch_lightning.metrics.MulticlassROC method), 405
forward() (pytorch_lightning.metrics.Precision method), 406
forward() (pytorch_lightning.metrics.PrecisionRecall method), 407
forward() (pytorch_lightning.metrics.PrecisionRecallCurve method), 408
forward() (pytorch_lightning.metrics.Recall method), 409
forward() (pytorch_lightning.metrics.regression.MAE method), 441
forward() (pytorch_lightning.metrics.regression.MSE method), 440
forward() (pytorch_lightning.metrics.regression.RMSE method), 440
forward() (pytorch_lightning.metrics.regression.RMSLE method), 442
forward() (pytorch_lightning.metrics.RMSE method), 396
forward() (pytorch_lightning.metrics.RMSLE method), 398
forward() (pytorch_lightning.metrics.ROC method), 409
forward() (pytorch_lightning.metrics.SklearnMetric method), 411
forward() (pytorch_lightning.metrics.sklearns.Accuracy method), 444
forward() (pytorch_lightning.metrics.sklearns.AUC method), 442
forward() (pytorch_lightning.metrics.sklearns.AUROC method), 443
forward() (pytorch_lightning.metrics.sklearns.AveragePrecision method), 445
forward() (pytorch_lightning.metrics.sklearns.ConfusionMatrix method), 446
forward() (pytorch_lightning.metrics.sklearns.F1 method), 447

forward() (*pytorch_lightning.metrics.sklearns.FBeta method*), 448
forward() (*pytorch_lightning.metrics.sklearns.Precision method*), 449
forward() (*pytorch_lightning.metrics.sklearns.PrecisionRecallCurve method*), 450
forward() (*pytorch_lightning.metrics.sklearns.Recall method*), 452
forward() (*pytorch_lightning.metrics.sklearns.ROC method*), 451
forward() (*pytorch_lightning.metrics.sklearns.SklearnMetric method*), 453
forward() (*pytorch_lightning.overrides.data_parallel.LightningDataParallel method*), 454
forward() (*pytorch_lightning.overrides.data_parallel.LightningDistributedDataParallel method*), 454
freeze() (*pytorch_lightning.core.lightning.LightningModule method*), 307
freeze() (*pytorch_lightning.core.LightningModule method*), 278
from_argparse_args() (*pytorch_lightning.trainer.Trainer class method*), 485
from_argparse_args() (*pytorch_lightning.trainer.trainer.Trainer class method*), 516

## G

garbage_collection_cuda() (*in module pytorch_lightning.utilities.memory*), 532
get_a_var() (*in module pytorch_lightning.overrides.data_parallel*), 454
get_all_available_gpus() (*in module pytorch_lightning.trainer.distrib_parts*), 499
get_deprecated_arg_names() (*pytorch_lightning.trainer.Trainer class method*), 485
get_deprecated_arg_names() (*pytorch_lightning.trainer.trainer.Trainer class method*), 516
get_gpu_memory_map() (*in module pytorch_lightning.core.memory*), 330
get_human_readable_count() (*in module pytorch_lightning.core.memory*), 330
get_init_args() (*in module pytorch_lightning.utilities.parsing*), 533
get_init_arguments_and_types() (*pytorch_lightning.trainer.Trainer class method*), 485
get_init_arguments_and_types() (*pytorch_lightning.trainer.trainer.Trainer class method*), 516
get_lr() (*pytorch_lightning.trainer.lr_finder._ExponentialLR method*), 506
get_lr() (*pytorch_lightning.trainer.lr_finder._LinearLR method*), 508
get_memory_profile() (*in module pytorch_lightning.core.memory*), 330
get_model (*pytorch_lightning.trainer.callback_hook.TrainerCallbackHoo attribute*), 490
get_model() (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin method*), 497
get_model() (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluat method*), 502
get_model() (*pytorch_lightning.trainer.model_hooks.TrainerModelHoo method*), 508
get_model() (*pytorch_lightning.trainer.training_io.TrainerIOMixin method*), 521
get_model() (*pytorch_lightning.trainer.training_loop.TrainerTrainLoop method*), 525
get_model() (*pytorch_lightning.trainer.training_tricks.TrainerTraining method*), 527
get_num_classes() (*in module pytorch_lightning.metrics.functional.classification*), 416
get_progress_bar_dict() (*pytorch_lightning.core.lightning.LightningModule method*), 307
get_progress_bar_dict() (*pytorch_lightning.core.LightningModule method*), 278
get_tqdm_dict() (*pytorch_lightning.core.lightning.LightningModule method*), 307
get_tqdm_dict() (*pytorch_lightning.core.LightningModule method*), 278
global_rank (*pytorch_lightning.trainer.data_loading.TrainerDataLoadi attribute*), 492
global_rank (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin attribute*), 498
global_rank (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluat attribute*), 503
global_rank (*pytorch_lightning.trainer.logging.TrainerLoggingMixin attribute*), 504
global_rank (*pytorch_lightning.trainer.Trainer attribute*), 487
global_rank (*pytorch_lightning.trainer.trainer.Trainer attribute*), 518
global_rank (*pytorch_lightning.trainer.training_io.TrainerIOMixin attribute*), 522
global_rank (*pytorch_lightning.trainer.training_loop.TrainerTrainLoop attribute*), 526
global_step (*pytorch_lightning.core.lightning.LightningModule attribute*), 327
global_step (*pytorch_lightning.core.LightningModule*

*attribute*), 297
global_step (*pytorch_lightning.trainer.logging.TrainerLoggingMixin attribute*), 504
global_step (*pytorch_lightning.trainer.lr_finder.TrainerLRFinderMixin attribute*), 506
global_step (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 526
gpus (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin attribute*), 496
grad_norm() (*pytorch_lightning.core.grads.GradInformation method*), 299
gradient_clip_val (*pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixin attribute*), 528
GradientAccumulationScheduler (*class in pytorch_lightning.callbacks*), 338
GradientAccumulationScheduler (*class in pytorch_lightning.callbacks.gradient_accumulation_scheduler*), 345
GradInformation (*class in pytorch_lightning.core.grads*), 299

## H

half() (*pytorch_lightning.utilities.device_dtype_mixin.DeviceDtypeModuleMixin method*), 530
has_arg() (*pytorch_lightning.trainer.model_hooks.TrainerModelHooksMixin method*), 508
has_arg() (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method*), 525
has_horovodrun() (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin static method*), 495
horovod_train() (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin method*), 497
hparams() (*pytorch_lightning.core.lightning.LightningModule property*), 327
hparams() (*pytorch_lightning.core.LightningModule property*), 298
hpc_load() (*pytorch_lightning.trainer.training_io.TrainerIOMixin method*), 521
hpc_save() (*pytorch_lightning.trainer.training_io.TrainerIOMixin method*), 521

## I

id() (*pytorch_lightning.loggers.trains.TrainsLogger property*), 393
id() (*pytorch_lightning.loggers.TrainsLogger property*), 372
ignore_scalar_return_in_dp() (*in module pytorch_lightning.trainer.ignored_warnings*), 504
in_size() (*pytorch_lightning.core.memory.LayerSummary property*), 328
in_sizes() (*pytorch_lightning.core.memory.ModelSummary property*), 329
init_amp() (*pytorch_lightning.trainer.auto_mix_precision.TrainerAMPMixin method*), 488
init_ddp_connection() (*pytorch_lightning.core.lightning.LightningModule method*), 308
init_ddp_connection() (*pytorch_lightning.core.LightningModule method*), 278
init_optimizers() (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin method*), 496
init_optimizers() (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin method*), 497
init_optimizers() (*pytorch_lightning.trainer.lr_finder.TrainerLRFinderMixin method*), 505
init_optimizers() (*pytorch_lightning.trainer.optimizers.TrainerOptimizersMixin method*), 508
init_sanity_tqdm() (*pytorch_lightning.callbacks.progress.ProgressBar method*), 349
init_sanity_tqdm() (*pytorch_lightning.callbacks.ProgressBar method*), 341
init_test_tqdm() (*pytorch_lightning.callbacks.progress.ProgressBar method*), 350
init_test_tqdm() (*pytorch_lightning.callbacks.ProgressBar method*), 341
init_tpu() (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin method*), 496
init_train_tqdm() (*pytorch_lightning.callbacks.progress.ProgressBar method*), 350
init_train_tqdm() (*pytorch_lightning.callbacks.ProgressBar method*), 342
init_validation_tqdm() (*pytorch_lightning.callbacks.progress.ProgressBar method*), 350
init_validation_tqdm() (*pytorch_lightning.callbacks.ProgressBar method*), 342
interactive_ddp_procs (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 526
interrupted (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 526
IoU (*class in pytorch_lightning.metrics*), 410

IoU (*class in pytorch_lightning.metrics.classification*), 429
iou() (*in module pytorch_lightning.metrics.functional.classification*), 416
is_cuda_out_of_memory() (*in module pytorch_lightning.utilities.memory*), 532
is_cudnn_snafu() (*in module pytorch_lightning.utilities.memory*), 532
is_disabled() (*pytorch_lightning.callbacks.progress.ProgressBar property*), 350
is_disabled() (*pytorch_lightning.callbacks.ProgressBar property*), 342
is_enabled() (*pytorch_lightning.callbacks.progress.ProgressBar property*), 350
is_enabled() (*pytorch_lightning.callbacks.ProgressBar property*), 342
is_function_implemented() (*pytorch_lightning.trainer.model_hooks.TrainerModelHooksMixin method*), 508
is_function_implemented() (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method*), 525
is_global_zero() (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin property*), 496
is_global_zero() (*pytorch_lightning.trainer.Trainer property*), 487
is_global_zero() (*pytorch_lightning.trainer.trainer.Trainer property*), 518
is_oom_error() (*in module pytorch_lightning.utilities.memory*), 532
is_out_of_cpu_memory() (*in module pytorch_lightning.utilities.memory*), 532
is_overridden() (*pytorch_lightning.trainer.callback_config.TrainerCallbackConfigMixin method*), 489
is_overridden() (*pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin method*), 491
is_overridden() (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin method*), 502
is_overridden() (*pytorch_lightning.trainer.model_hooks.TrainerModelHooksMixin method*), 508
is_overridden() (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method*), 525


## K


kth_best_model() (*pytorch_lightning.callbacks.model_checkpoint.ModelCheckpoint property*), 348
kth_best_model() (*pytorch_lightning.callbacks.ModelCheckpoint property*), 338


## L


last() (*pytorch_lightning.trainer.supporters.TensorRunningAccum method*), 509
last_epoch (*pytorch_lightning.trainer.lr_finder._ExponentialLR attribute*), 506
last_epoch (*pytorch_lightning.trainer.lr_finder._LinearLR attribute*), 508
layer_names() (*pytorch_lightning.core.memory.ModelSummary property*), 329
layer_type() (*pytorch_lightning.core.memory.LayerSummary property*), 328
layer_types() (*pytorch_lightning.core.memory.ModelSummary property*), 329
LayerSummary (*class in pytorch_lightning.core.memory*), 328
LearningRateLogger (*class in pytorch_lightning.callbacks*), 338
LearningRateLogger (*class in pytorch_lightning.callbacks.lr_logger*), 346
LightningDataParallel (*class in pytorch_lightning.overrides.data_parallel*), 454
LightningDistributedDataParallel (*class in pytorch_lightning.overrides.data_parallel*), 454
LightningLoggerBase (*class in pytorch_lightning.loggers*), 352
LightningLoggerBase (*class in pytorch_lightning.loggers.base*), 373
LightningModule (*class in pytorch_lightning.core*), 273
LightningModule (*class in pytorch_lightning.core.lightning*), 303
limit_test_batches (*pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin attribute*), 492
limit_test_batches (*pytorch_lightning.trainer.deprecated_api.TrainerDeprecatedAPITill attribute*), 492
limit_train_batches (*pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin attribute*), 492
limit_train_batches (*pytorch_lightning.trainer.deprecated_api.TrainerDeprecatedAPITill

attribute), 492
limit_val_batches (pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin attribute), 492
limit_val_batches (pytorch_lightning.trainer.deprecated_api.TrainerDeprecatedAPITillVer0_10 attribute), 492
load() (in module pytorch_lightning.utilities.cloud_io), 530
load_from_checkpoint() (pytorch_lightning.core.saving.ModelIO class method), 331
load_from_metrics() (pytorch_lightning.core.saving.ModelIO class method), 332
load_hparams_from_tags_csv() (in module pytorch_lightning.core.saving), 333
load_hparams_from_yaml() (in module pytorch_lightning.core.saving), 333
load_spawn_weights() (pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin method), 496
load_state_dict() (pytorch_lightning.trainer.optimizers._MockOptimizer method), 509
log_artifact() (pytorch_lightning.loggers.neptune.NeptuneLogger method), 384
log_artifact() (pytorch_lightning.loggers.NeptuneLogger method), 364
log_artifact() (pytorch_lightning.loggers.trains.TrainsLogger method), 391
log_artifact() (pytorch_lightning.loggers.TrainsLogger method), 370
log_dir() (pytorch_lightning.loggers.tensorboard.TensorBoardLogger property), 387
log_dir() (pytorch_lightning.loggers.TensorBoardLogger property), 357
log_gpu_memory (pytorch_lightning.trainer.logging.TrainerLoggingMixin attribute), 504
log_hyperparams() (pytorch_lightning.loggers.base.DummyLogger method), 373
log_hyperparams() (pytorch_lightning.loggers.base.LightningLoggerBase method), 375
log_hyperparams() (pytorch_lightning.loggers.base.LoggerCollection method), 376
log_hyperparams() (pytorch_lightning.loggers.comet.CometLogger method), 379
log_hyperparams() (pytorch_lightning.loggers.CometLogger method), 359
log_hyperparams() (pytorch_lightning.loggers.LightningLoggerBase method), 354
log_hyperparams() (pytorch_lightning.loggers.LoggerCollection method), 355
log_hyperparams() (pytorch_lightning.loggers.mlflow.MLFlowLogger method), 380
log_hyperparams() (pytorch_lightning.loggers.MLFlowLogger method), 360
log_hyperparams() (pytorch_lightning.loggers.neptune.NeptuneLogger method), 384
log_hyperparams() (pytorch_lightning.loggers.NeptuneLogger method), 364
log_hyperparams() (pytorch_lightning.loggers.tensorboard.TensorBoardLogger method), 387
log_hyperparams() (pytorch_lightning.loggers.TensorBoardLogger method), 356
log_hyperparams() (pytorch_lightning.loggers.test_tube.TestTubeLogger method), 389
log_hyperparams() (pytorch_lightning.loggers.TestTubeLogger method), 366
log_hyperparams() (pytorch_lightning.loggers.trains.TrainsLogger method), 392
log_hyperparams() (pytorch_lightning.loggers.TrainsLogger method), 371
log_hyperparams() (pytorch_lightning.loggers.wandb.WandbLogger method), 395
log_hyperparams() (pytorch_lightning.loggers.WandbLogger method), 368
log_image() (pytorch_lightning.loggers.neptune.NeptuneLogger method), 385
log_image() (pytorch_lightning.loggers.NeptuneLogger method), 364
log_image() (pytorch_lightning.loggers.trains.TrainsLogger method), 392
log_image() (pytorch_lightning.loggers.TrainsLogger

*method*), 371
log_metric() (*pytorch_lightning.loggers.neptune.NeptuneLogger method*), 385
log_metric() (*pytorch_lightning.loggers.NeptuneLogger method*), 364
log_metric() (*pytorch_lightning.loggers.trains.TrainsLogger method*), 392
log_metric() (*pytorch_lightning.loggers.TrainsLogger method*), 371
log_metrics() (*pytorch_lightning.loggers.base.DummyLogger method*), 373
log_metrics() (*pytorch_lightning.loggers.base.LightningLoggerBase method*), 375
log_metrics() (*pytorch_lightning.loggers.base.LoggerCollection method*), 376
log_metrics() (*pytorch_lightning.loggers.comet.CometLogger method*), 379
log_metrics() (*pytorch_lightning.loggers.CometLogger method*), 359
log_metrics() (*pytorch_lightning.loggers.LightningLoggerBase method*), 354
log_metrics() (*pytorch_lightning.loggers.LoggerCollection method*), 355
log_metrics() (*pytorch_lightning.loggers.mlflow.MLFlowLogger method*), 381
log_metrics() (*pytorch_lightning.loggers.MLFlowLogger method*), 360
log_metrics() (*pytorch_lightning.loggers.neptune.NeptuneLogger method*), 385
log_metrics() (*pytorch_lightning.loggers.NeptuneLogger method*), 365
log_metrics() (*pytorch_lightning.loggers.tensorboard.TensorBoardLogger method*), 387
log_metrics() (*pytorch_lightning.loggers.TensorBoardLogger method*), 357
log_metrics() (*pytorch_lightning.loggers.test_tube.TestTubeLogger method*), 389
log_metrics() (*pytorch_lightning.loggers.TestTubeLogger method*), 367
log_metrics() (*pytorch_lightning.loggers.trains.TrainsLogger method*), 392
log_metrics() (*pytorch_lightning.loggers.TrainsLogger method*), 371
log_metrics() (*pytorch_lightning.loggers.wandb.WandbLogger method*), 395
log_metrics() (*pytorch_lightning.loggers.WandbLogger method*), 368
log_metrics() (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopM method*), 502
log_metrics() (*pytorch_lightning.trainer.logging.TrainerLoggingMixin method*), 504
log_metrics() (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method*), 525
log_save_interval (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 526
log_text() (*pytorch_lightning.loggers.neptune.NeptuneLogger method*), 385
log_text() (*pytorch_lightning.loggers.NeptuneLogger method*), 365
log_text() (*pytorch_lightning.loggers.trains.TrainsLogger method*), 393
log_text() (*pytorch_lightning.loggers.TrainsLogger method*), 372
logger (*pytorch_lightning.core.lightning.LightningModule attribute*), 327
logger (*pytorch_lightning.core.LightningModule attribute*), 298
logger (*pytorch_lightning.trainer.callback_config.TrainerCallbackConfig attribute*), 489
logger (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin attribute*), 496
logger (*pytorch_lightning.trainer.logging.TrainerLoggingMixin attribute*), 504
logger (*pytorch_lightning.trainer.Trainer attribute*), 487
logger (*pytorch_lightning.trainer.trainer.Trainer attribute*), 518
logger (*pytorch_lightning.trainer.training_io.TrainerIOMixin attribute*), 522
logger (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 526
LoggerCollection (*class in pytorch_lightning.loggers*), 355
LoggerCollection (*class in pytorch_lightning.loggers.base*), 376

lr() (pytorch_lightning.trainer.lr_finder._ExponentialLR property), 506
lr() (pytorch_lightning.trainer.lr_finder._LinearLR property), 508
lr_find() (pytorch_lightning.trainer.lr_finder.TrainerLRFinderMixin method), 505
lr_schedulers (pytorch_lightning.trainer.Trainer attribute), 487
lr_schedulers (pytorch_lightning.trainer.trainer.Trainer attribute), 518
lr_schedulers (pytorch_lightning.trainer.training_io.TrainerIOMixin attribute), 522
lr_schedulers (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526


## M


MAE (class in pytorch_lightning.metrics), 397
MAE (class in pytorch_lightning.metrics.regression), 441
max() (pytorch_lightning.trainer.supporters.TensorRunningAccum method), 509
max_ckpt_in_folder() (pytorch_lightning.trainer.training_io.TrainerIOMixin method), 521
max_epochs (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
max_steps (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
mean() (pytorch_lightning.trainer.supporters.TensorRunningAccum method), 509
merge_dicts() (in module pytorch_lightning.loggers.base), 377
Metric (class in pytorch_lightning.metrics.metric), 438
metric_fn() (pytorch_lightning.metrics.SklearnMetric property), 411
metric_fn() (pytorch_lightning.metrics.sklearns.SklearnMetric property), 453
metrics_to_scalars() (pytorch_lightning.trainer.logging.TrainerLoggingMixin method), 504
min() (pytorch_lightning.trainer.supporters.TensorRunningAccum method), 509
min_epochs (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
min_steps (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
MisconfigurationException, 532
MLFlowLogger (class in pytorch_lightning.loggers), 359
MLFlowLogger (class in pytorch_lightning.loggers.mlflow), 380
MODE_DEFAULT (pytorch_lightning.core.memory.ModelSummary attribute), 329
mode_dict (pytorch_lightning.callbacks.early_stopping.EarlyStopping attribute), 345
mode_dict (pytorch_lightning.callbacks.EarlyStopping attribute), 336
MODE_FULL (pytorch_lightning.core.memory.ModelSummary attribute), 329
MODE_TOP (pytorch_lightning.core.memory.ModelSummary attribute), 329
model (pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin attribute), 503
model (pytorch_lightning.trainer.Trainer attribute), 487
model (pytorch_lightning.trainer.trainer.Trainer attribute), 518
model (pytorch_lightning.trainer.training_io.TrainerIOMixin attribute), 522
model (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
ModelCheckpoint (class in pytorch_lightning.callbacks), 336
ModelCheckpoint (class in pytorch_lightning.callbacks.model_checkpoint), 346
ModelHooks (class in pytorch_lightning.core.hooks), 299
ModelIO (class in pytorch_lightning.core.saving), 331
ModelSummary (class in pytorch_lightning.core.memory), 328
MODES (pytorch_lightning.core.memory.ModelSummary attribute), 329
monitor_op() (pytorch_lightning.callbacks.early_stopping.EarlyStopping property), 345
monitor_op() (pytorch_lightning.callbacks.EarlyStopping property), 336
move_data_to_device() (in module pytorch_lightning.utilities.apply_func), 529
MSE (class in pytorch_lightning.metrics), 396
MSE (class in pytorch_lightning.metrics.regression), 440
multiclass_auc_decorator() (in module pytorch_lightning.metrics.functional.classification), 417
multiclass_precision_recall_curve() (in module pytorch_lightning.metrics.functional.classification), 417
multiclass_roc() (in module pytorch_lightning.metrics.functional.classification), 417
MulticlassPrecisionRecall (class in pytorch_lightning.metrics), 404
MulticlassPrecisionRecall (class in pytorch_lightning.metrics.classification), 429
MulticlassROC (class in pytorch_lightning.metrics),

405
MulticlassROC (*class in pytorch_lightning.metrics.classification*), 430

## N

name() (*pytorch_lightning.loggers.base.DummyLogger property*), 373
name() (*pytorch_lightning.loggers.base.LightningLoggerBase property*), 376
name() (*pytorch_lightning.loggers.base.LoggerCollection property*), 377
name() (*pytorch_lightning.loggers.comet.CometLogger property*), 379
name() (*pytorch_lightning.loggers.CometLogger property*), 359
name() (*pytorch_lightning.loggers.LightningLoggerBase property*), 355
name() (*pytorch_lightning.loggers.LoggerCollection property*), 356
name() (*pytorch_lightning.loggers.mlflow.MLFlowLogger property*), 381
name() (*pytorch_lightning.loggers.MLFlowLogger property*), 361
name() (*pytorch_lightning.loggers.neptune.NeptuneLogger property*), 386
name() (*pytorch_lightning.loggers.NeptuneLogger property*), 365
name() (*pytorch_lightning.loggers.tensorboard.TensorBoardLogger property*), 387
name() (*pytorch_lightning.loggers.TensorBoardLogger property*), 357
name() (*pytorch_lightning.loggers.test_tube.TestTubeLogger property*), 389
name() (*pytorch_lightning.loggers.TestTubeLogger property*), 367
name() (*pytorch_lightning.loggers.trains.TrainsLogger property*), 393
name() (*pytorch_lightning.loggers.TrainsLogger property*), 372
name() (*pytorch_lightning.loggers.wandb.WandbLogger property*), 395
name() (*pytorch_lightning.loggers.WandbLogger property*), 368
NAME_HPARAMS_FILE (*pytorch_lightning.loggers.tensorboard.TensorBoardLogger attribute*), 387
NAME_HPARAMS_FILE (*pytorch_lightning.loggers.TensorBoardLogger attribute*), 357
named_modules() (*pytorch_lightning.core.memory.ModelSummary property*), 329
NeptuneLogger (*class in pytorch_lightning.loggers*), 361
NeptuneLogger (*class in pytorch_lightning.loggers.neptune*), 381
node_rank (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDP attribute*), 496
nop() (*pytorch_lightning.loggers.base.DummyExperiment method*), 373
normalize_parse_gpu_input_to_list() (*in module pytorch_lightning.trainer.distrib_parts*), 499
normalize_parse_gpu_string_input() (*in module pytorch_lightning.trainer.distrib_parts*), 499
num_gpu_nodes (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin attribute*), 496
num_gpus (*pytorch_lightning.trainer.logging.TrainerLoggingMixin attribute*), 504
num_gpus() (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDD property*), 496
num_gpus() (*pytorch_lightning.trainer.Trainer property*), 487
num_gpus() (*pytorch_lightning.trainer.trainer.Trainer property*), 518
num_nodes (*pytorch_lightning.trainer.data_loading.TrainerDataLoading attribute*), 492
num_nodes (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDP attribute*), 497
num_parameters() (*pytorch_lightning.core.memory.LayerSummary property*), 328
num_processes (*pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin attribute*), 492
num_processes (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin attribute*), 497
num_test_batches (*pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin attribute*), 492
num_test_batches (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopM attribute*), 503
num_tpu_cores() (*pytorch_lightning.trainer.deprecated_api.TrainerDeprecatedAPITill property*), 493
num_training_batches (*pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin attribute*), 492
num_training_batches (*pytorch_lightning.trainer.Trainer attribute*), 487
num_training_batches (*pytorch_lightning.trainer.trainer.Trainer attribute*), 518

num_training_batches (pytorch_lightning.trainer.training_io.TrainerIOMixin attribute), 522
num_training_batches (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
num_val_batches (pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin attribute), 492
num_val_batches (pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin attribute), 503
num_val_batches (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
numpy_metric() (in module pytorch_lightning.metrics.converters), 437
NumpyMetric (class in pytorch_lightning.metrics.metric), 438

# O

on_after_backward() (pytorch_lightning.core.hooks.ModelHooks method), 300
on_batch_end (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
on_batch_end() (pytorch_lightning.callbacks.base.Callback method), 343
on_batch_end() (pytorch_lightning.callbacks.Callback method), 334
on_batch_end() (pytorch_lightning.callbacks.progress.ProgressBar method), 350
on_batch_end() (pytorch_lightning.callbacks.progress.ProgressBarBase method), 351
on_batch_end() (pytorch_lightning.callbacks.ProgressBar method), 342
on_batch_end() (pytorch_lightning.callbacks.ProgressBarBase method), 340
on_batch_end() (pytorch_lightning.core.hooks.ModelHooks method), 300
on_batch_end() (pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method), 489
on_batch_end() (pytorch_lightning.trainer.lr_finder._LRCallback method), 507
on_batch_start (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
on_batch_start() (pytorch_lightning.callbacks.base.Callback method), 343
on_batch_start() (pytorch_lightning.callbacks.Callback method), 334
on_batch_start() (pytorch_lightning.callbacks.LearningRateLogger method), 339
on_batch_start() (pytorch_lightning.callbacks.lr_logger.LearningRateLogger method), 346
on_batch_start() (pytorch_lightning.core.hooks.ModelHooks method), 300
on_batch_start() (pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method), 489
on_batch_start() (pytorch_lightning.trainer.lr_finder._LRCallback method), 507
on_before_zero_grad() (pytorch_lightning.core.hooks.ModelHooks method), 300
on_colab_kaggle (pytorch_lightning.trainer.distrib_parts.TrainerDPMixin attribute), 498
on_epoch_end (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
on_epoch_end() (pytorch_lightning.callbacks.base.Callback method), 343
on_epoch_end() (pytorch_lightning.callbacks.Callback method), 334
on_epoch_end() (pytorch_lightning.core.hooks.ModelHooks method), 301
on_epoch_end() (pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method), 489
on_epoch_start (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
on_epoch_start() (pytorch_lightning.callbacks.base.Callback method), 343
on_epoch_start() (pytorch_lightning.callbacks.Callback method), 334
on_epoch_start() (py-

torch_lightning.callbacks.gradient_accumulation_scheduler.GradientAccumulationScheduler method), 345
on_epoch_start() (pytorch_lightning.callbacks.GradientAccumulationScheduler method), 338
on_epoch_start() (pytorch_lightning.callbacks.LearningRateLogger method), 339
on_epoch_start() (pytorch_lightning.callbacks.lr_logger.LearningRateLogger method), 346
on_epoch_start() (pytorch_lightning.callbacks.progress.ProgressBar method), 350
on_epoch_start() (pytorch_lightning.callbacks.progress.ProgressBarBase method), 351
on_epoch_start() (pytorch_lightning.callbacks.ProgressBar method), 342
on_epoch_start() (pytorch_lightning.callbacks.ProgressBarBase method), 340
on_epoch_start() (pytorch_lightning.core.hooks.ModelHooks method), 301
on_epoch_start() (pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method), 489
on_fit_end() (pytorch_lightning.callbacks.base.Callback method), 343
on_fit_end() (pytorch_lightning.callbacks.Callback method), 334
on_fit_end() (pytorch_lightning.core.hooks.ModelHooks method), 301
on_fit_end() (pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method), 489
on_fit_start() (pytorch_lightning.callbacks.base.Callback method), 343
on_fit_start() (pytorch_lightning.callbacks.Callback method), 334
on_fit_start() (pytorch_lightning.core.hooks.ModelHooks method), 301
on_fit_start() (pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method), 489
on_gpu (pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin attribute), 497
on_gpu (pytorch_lightning.trainer.distrib_parts.TrainerDPMixin attribute), 498
on_gpu (pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin attribute), 503
on_gpu (pytorch_lightning.trainer.logging.TrainerLoggingMixin attribute), 504
on_gpu (pytorch_lightning.trainer.lr_finder.TrainerLRFinderMixin attribute), 506
on_gpu (pytorch_lightning.trainer.Trainer attribute), 487
on_gpu (pytorch_lightning.trainer.trainer.Trainer attribute), 518
on_gpu (pytorch_lightning.trainer.training_io.TrainerIOMixin attribute), 522
on_gpu (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
on_gpu (pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixin attribute), 528
on_gpu() (pytorch_lightning.core.lightning.LightningModule property), 327
on_gpu() (pytorch_lightning.core.LightningModule property), 298
on_hpc_load() (pytorch_lightning.core.saving.ModelIO method), 332
on_hpc_save() (pytorch_lightning.core.saving.ModelIO method), 332
on_init_end() (pytorch_lightning.callbacks.base.Callback method), 343
on_init_end() (pytorch_lightning.callbacks.Callback method), 334
on_init_end() (pytorch_lightning.callbacks.progress.ProgressBarBase method), 351
on_init_end() (pytorch_lightning.callbacks.ProgressBarBase method), 340
on_init_end() (pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method), 489
on_init_start() (pytorch_lightning.callbacks.base.Callback method), 343
on_init_start() (pytorch_lightning.callbacks.Callback method), 334
on_init_start() (pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method), 489
on_keyboard_interrupt (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
on_keyboard_interrupt() (pytorch_lightning.callbacks.base.Callback

method), 343
on_keyboard_interrupt() (pytorch_lightning.callbacks.Callback method), 335
on_keyboard_interrupt() (pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method), 490
on_load_checkpoint() (pytorch_lightning.core.lightning.LightningModule method), 308
on_load_checkpoint() (pytorch_lightning.core.LightningModule method), 278
on_load_checkpoint() (pytorch_lightning.core.saving.ModelIO method), 333
on_post_performance_check() (pytorch_lightning.core.hooks.ModelHooks method), 301
on_pre_performance_check() (pytorch_lightning.core.hooks.ModelHooks method), 301
on_sanity_check_end() (pytorch_lightning.callbacks.base.Callback method), 343
on_sanity_check_end() (pytorch_lightning.callbacks.Callback method), 335
on_sanity_check_end() (pytorch_lightning.callbacks.progress.ProgressBar method), 350
on_sanity_check_end() (pytorch_lightning.callbacks.ProgressBar method), 342
on_sanity_check_end() (pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method), 490
on_sanity_check_start() (pytorch_lightning.callbacks.base.Callback method), 343
on_sanity_check_start() (pytorch_lightning.callbacks.Callback method), 335
on_sanity_check_start() (pytorch_lightning.callbacks.progress.ProgressBar method), 350
on_sanity_check_start() (pytorch_lightning.callbacks.ProgressBar method), 342
on_sanity_check_start() (pytorch_lightning.core.hooks.ModelHooks method), 301
on_sanity_check_start() (pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method), 490
on_save_checkpoint() (pytorch_lightning.core.lightning.LightningModule method), 308
on_save_checkpoint() (pytorch_lightning.core.LightningModule method), 279
on_save_checkpoint() (pytorch_lightning.core.saving.ModelIO method), 333
on_test_batch_end (pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopM attribute), 503
on_test_batch_end() (pytorch_lightning.callbacks.base.Callback method), 343
on_test_batch_end() (pytorch_lightning.callbacks.Callback method), 335
on_test_batch_end() (pytorch_lightning.callbacks.progress.ProgressBar method), 350
on_test_batch_end() (pytorch_lightning.callbacks.progress.ProgressBarBase method), 351
on_test_batch_end() (pytorch_lightning.callbacks.ProgressBar method), 342
on_test_batch_end() (pytorch_lightning.callbacks.ProgressBarBase method), 340
on_test_batch_end() (pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method), 490
on_test_batch_start (pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopM attribute), 503
on_test_batch_start() (pytorch_lightning.callbacks.base.Callback method), 343
on_test_batch_start() (pytorch_lightning.callbacks.Callback method), 335
on_test_batch_start() (pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method), 490
on_test_end (pytorch_lightning.trainer.evaluation_loop.TrainerEvaluat attribute), 503
on_test_end() (pytorch_lightning.callbacks.base.Callback method), 343
on_test_end() (pytorch_lightning.callbacks.Callback method), 335

on_test_end() (pytorch_lightning.callbacks.progress.ProgressBar method), 350
on_test_end() (pytorch_lightning.callbacks.ProgressBar method), 342
on_test_end() (pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method), 490
on_test_start (pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin attribute), 503
on_test_start() (pytorch_lightning.callbacks.base.Callback method), 343
on_test_start() (pytorch_lightning.callbacks.Callback method), 335
on_test_start() (pytorch_lightning.callbacks.progress.ProgressBar method), 350
on_test_start() (pytorch_lightning.callbacks.progress.ProgressBarBase method), 351
on_test_start() (pytorch_lightning.callbacks.ProgressBar method), 342
on_test_start() (pytorch_lightning.callbacks.ProgressBarBase method), 340
on_test_start() (pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method), 490
on_tpu (pytorch_lightning.trainer.Trainer attribute), 487
on_tpu (pytorch_lightning.trainer.trainer.Trainer attribute), 518
on_tpu (pytorch_lightning.trainer.training_io.TrainerIOMixin attribute), 522
on_train_end (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
on_train_end() (pytorch_lightning.callbacks.base.Callback method), 343
on_train_end() (pytorch_lightning.callbacks.Callback method), 335
on_train_end() (pytorch_lightning.callbacks.early_stopping.EarlyStopping method), 345
on_train_end() (pytorch_lightning.callbacks.EarlyStopping method), 336
on_train_end() (pytorch_lightning.callbacks.progress.ProgressBar method), 350
on_train_end() (pytorch_lightning.callbacks.ProgressBar method), 342
on_train_end() (pytorch_lightning.core.hooks.ModelHooks method), 301
on_train_end() (pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method), 490
on_train_start (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 526
on_train_start() (pytorch_lightning.callbacks.base.Callback method), 343
on_train_start() (pytorch_lightning.callbacks.Callback method), 335
on_train_start() (pytorch_lightning.callbacks.early_stopping.EarlyStopping method), 345
on_train_start() (pytorch_lightning.callbacks.EarlyStopping method), 336
on_train_start() (pytorch_lightning.callbacks.LearningRateLogger method), 339
on_train_start() (pytorch_lightning.callbacks.lr_logger.LearningRateLogger method), 346
on_train_start() (pytorch_lightning.callbacks.progress.ProgressBar method), 350
on_train_start() (pytorch_lightning.callbacks.progress.ProgressBarBase method), 351
on_train_start() (pytorch_lightning.callbacks.ProgressBar method), 342
on_train_start() (pytorch_lightning.callbacks.ProgressBarBase method), 340
on_train_start() (pytorch_lightning.core.hooks.ModelHooks method), 301
on_train_start() (pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method), 490
on_validation_batch_end (pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin attribute), 503
on_validation_batch_end() (py-

*torch_lightning.callbacks.base.Callback method*), 343
on_validation_batch_end() (*pytorch_lightning.callbacks.Callback method*), 335
on_validation_batch_end() (*pytorch_lightning.callbacks.progress.ProgressBar method*), 350
on_validation_batch_end() (*pytorch_lightning.callbacks.progress.ProgressBarBase method*), 351
on_validation_batch_end() (*pytorch_lightning.callbacks.ProgressBar method*), 342
on_validation_batch_end() (*pytorch_lightning.callbacks.ProgressBarBase method*), 340
on_validation_batch_end() (*pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method*), 490
on_validation_batch_start (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin attribute*), 503
on_validation_batch_start() (*pytorch_lightning.callbacks.base.Callback method*), 344
on_validation_batch_start() (*pytorch_lightning.callbacks.Callback method*), 335
on_validation_batch_start() (*pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method*), 490
on_validation_end (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin attribute*), 503
on_validation_end (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 526
on_validation_end() (*pytorch_lightning.callbacks.base.Callback method*), 344
on_validation_end() (*pytorch_lightning.callbacks.Callback method*), 335
on_validation_end() (*pytorch_lightning.callbacks.early_stopping.EarlyStopping method*), 345
on_validation_end() (*pytorch_lightning.callbacks.EarlyStopping method*), 336
on_validation_end() (*pytorch_lightning.callbacks.model_checkpoint.ModelCheckpoint method*), 348
on_validation_end() (*pytorch_lightning.callbacks.ModelCheckpoint method*), 338
on_validation_end() (*pytorch_lightning.callbacks.progress.ProgressBar method*), 350
on_validation_end() (*pytorch_lightning.callbacks.ProgressBar method*), 342
on_validation_end() (*pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method*), 490
on_validation_start (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin attribute*), 503
on_validation_start() (*pytorch_lightning.callbacks.base.Callback method*), 344
on_validation_start() (*pytorch_lightning.callbacks.Callback method*), 335
on_validation_start() (*pytorch_lightning.callbacks.progress.ProgressBar method*), 350
on_validation_start() (*pytorch_lightning.callbacks.progress.ProgressBarBase method*), 351
on_validation_start() (*pytorch_lightning.callbacks.ProgressBar method*), 342
on_validation_start() (*pytorch_lightning.callbacks.ProgressBarBase method*), 340
on_validation_start() (*pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method*), 490
optimizer_frequencies (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 526
optimizer_step() (*pytorch_lightning.core.lightning.LightningModule method*), 309
optimizer_step() (*pytorch_lightning.core.LightningModule method*), 279
optimizers (*pytorch_lightning.trainer.Trainer attribute*), 488
optimizers (*pytorch_lightning.trainer.trainer.Trainer attribute*), 519
optimizers (*pytorch_lightning.trainer.training_io.TrainerIOMixin attribute*), 522
optimizers (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 526
out_size() (*pytorch_lightning.core.memory.LayerSummary property*), 328

out_sizes() (*pytorch_lightning.core.memory.ModelSummary property*), 330
overfit_batches (*pytorch_lightning.trainer.deprecated_api.TrainerDeprecatedAPITillVer0_10 attribute*), 492
overfit_pct() (*pytorch_lightning.trainer.deprecated_api.TrainerDeprecatedAPITillVer0_10 property*), 492

## P

parallel_apply() (*in module pytorch_lightning.overrides.data_parallel*), 454
parallel_apply() (*pytorch_lightning.overrides.data_parallel.LightningDataParallel method*), 454
parallel_apply() (*pytorch_lightning.overrides.data_parallel.LightningDistributedDataParallel method*), 454
param_nums() (*pytorch_lightning.core.memory.ModelSummary property*), 330
parse_argparser() (*pytorch_lightning.trainer.Trainer static method*), 486
parse_argparser() (*pytorch_lightning.trainer.trainer.Trainer static method*), 517
parse_batch_shape() (*in module pytorch_lightning.core.memory*), 331
parse_gpu_ids() (*in module pytorch_lightning.trainer.distrib_parts*), 499
PassThroughProfiler (*class in pytorch_lightning.profiler*), 458
PassThroughProfiler (*class in pytorch_lightning.profiler.profilers*), 460
pick_multiple_gpus() (*in module pytorch_lightning.trainer.distrib_parts*), 499
pick_single_gpu() (*in module pytorch_lightning.trainer.distrib_parts*), 499
plot() (*pytorch_lightning.trainer.lr_finder._LRFinder method*), 507
Precision (*class in pytorch_lightning.metrics*), 406
Precision (*class in pytorch_lightning.metrics.classification*), 431
Precision (*class in pytorch_lightning.metrics.sklearns*), 448
precision (*pytorch_lightning.trainer.auto_mix_precision.TrainerAMPMixin attribute*), 488
precision (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin attribute*), 498
precision (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 526
precision (*pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixin attribute*), 528
precision() (*in module pytorch_lightning.metrics.functional.classification*), 418
precision_recall() (*in module pytorch_lightning.metrics.functional.classification*), 419
precision_recall_curve() (*in module pytorch_lightning.metrics.functional.classification*), 419
PrecisionRecall (*class in pytorch_lightning.metrics*), 406
PrecisionRecall (*class in pytorch_lightning.metrics.classification*), 432
PrecisionRecallCurve (*class in pytorch_lightning.metrics*), 407
PrecisionRecallCurve (*class in pytorch_lightning.metrics.sklearns*), 450
prepare_data() (*pytorch_lightning.core.lightning.LightningModule method*), 310
prepare_data() (*pytorch_lightning.core.LightningModule method*), 280
print() (*pytorch_lightning.core.lightning.LightningModule method*), 310
print() (*pytorch_lightning.core.LightningModule method*), 281
print_nan_gradients() (*pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixi method*), 528
process_output (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopM attribute*), 503
process_output() (*pytorch_lightning.trainer.logging.TrainerLoggingMixin method*), 504
process_output() (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method*), 525
process_position() (*pytorch_lightning.callbacks.progress.ProgressBar property*), 350
process_position() (*pytorch_lightning.callbacks.ProgressBar property*), 342
profile() (*pytorch_lightning.profiler.BaseProfiler method*), 457
profile() (*pytorch_lightning.profiler.profilers.BaseProfiler method*), 459
profile_iterable() (*pytorch_lightning.profiler.BaseProfiler method*), 457
profile_iterable() (*pytorch_lightning.profiler.profilers.BaseProfiler*

*method*), 459
profiler (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 526
progress_bar_callback (*pytorch_lightning.trainer.deprecated_api.TrainerDeprecatedAPITillVer0_9 attribute*), 493
progress_bar_callback (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin attribute*), 497
progress_bar_callback (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin attribute*), 498
progress_bar_callback (*pytorch_lightning.trainer.lr_finder.TrainerLRFinderMixin attribute*), 506
progress_bar_callback (*pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixin attribute*), 528
progress_bar_callback() (*pytorch_lightning.trainer.Trainer property*), 488
progress_bar_callback() (*pytorch_lightning.trainer.trainer.Trainer property*), 519
progress_bar_dict (*pytorch_lightning.trainer.deprecated_api.TrainerDeprecatedAPITillVer0_9 attribute*), 493
progress_bar_dict (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin attribute*), 503
progress_bar_dict (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 527
progress_bar_dict() (*pytorch_lightning.trainer.Trainer property*), 488
progress_bar_dict() (*pytorch_lightning.trainer.trainer.Trainer property*), 519
progress_bar_metrics (*pytorch_lightning.trainer.logging.TrainerLoggingMixin attribute*), 504
ProgressBar (*class in pytorch_lightning.callbacks*), 340
ProgressBar (*class in pytorch_lightning.callbacks.progress*), 349
ProgressBarBase (*class in pytorch_lightning.callbacks*), 339
ProgressBarBase (*class in pytorch_lightning.callbacks.progress*), 351
pytorch_lightning.callbacks (*module*), 334
pytorch_lightning.callbacks.base (*module*), 343
pytorch_lightning.callbacks.early_stopping (*module*), 344
pytorch_lightning.callbacks.gradient_accumulation_s (*module*), 345
pytorch_lightning.callbacks.lr_logger (*module*), 346
pytorch_lightning.callbacks.model_checkpoint (*module*), 346
pytorch_lightning.callbacks.progress (*module*), 349
pytorch_lightning.core (*module*), 267
pytorch_lightning.core.decorators (*module*), 298
pytorch_lightning.core.grads (*module*), 299
pytorch_lightning.core.hooks (*module*), 299
pytorch_lightning.core.lightning (*module*), 303
pytorch_lightning.core.memory (*module*), 328
pytorch_lightning.core.saving (*module*), 331
pytorch_lightning.loggers (*module*), 352
pytorch_lightning.loggers.base (*module*), 373
pytorch_lightning.loggers.comet (*module*), 378
pytorch_lightning.loggers.mlflow (*module*), 380
pytorch_lightning.loggers.neptune (*module*), 381
pytorch_lightning.loggers.tensorboard (*module*), 386
pytorch_lightning.loggers.test_tube (*module*), 388
pytorch_lightning.loggers.trains (*module*), 390
pytorch_lightning.loggers.wandb (*module*), 394
pytorch_lightning.metrics (*module*), 396
pytorch_lightning.metrics.classification (*module*), 423
pytorch_lightning.metrics.converters (*module*), 435
pytorch_lightning.metrics.functional (*module*), 411
pytorch_lightning.metrics.functional.classificatio (*module*), 411
pytorch_lightning.metrics.functional.reduction (*module*), 423
pytorch_lightning.metrics.metric (*module*), 438
pytorch_lightning.metrics.regression (*module*), 440
pytorch_lightning.metrics.sklearns (*module*), 442

pytorch_lightning.overrides (*module*), 454
pytorch_lightning.overrides.data_parallel (*module*), 454
pytorch_lightning.profiler (*module*), 455
pytorch_lightning.profiler.profilers (*module*), 459
pytorch_lightning.trainer (*module*), 461
pytorch_lightning.trainer.auto_mix_precision (*module*), 488
pytorch_lightning.trainer.callback_config (*module*), 489
pytorch_lightning.trainer.callback_hook (*module*), 489
pytorch_lightning.trainer.data_loading (*module*), 491
pytorch_lightning.trainer.deprecated_api (*module*), 492
pytorch_lightning.trainer.distrib_data_parallel (*module*), 493
pytorch_lightning.trainer.distrib_parts (*module*), 497
pytorch_lightning.trainer.evaluation_loop (*module*), 500
pytorch_lightning.trainer.ignored_warnings (*module*), 504
pytorch_lightning.trainer.logging (*module*), 504
pytorch_lightning.trainer.lr_finder (*module*), 505
pytorch_lightning.trainer.model_hooks (*module*), 508
pytorch_lightning.trainer.optimizers (*module*), 508
pytorch_lightning.trainer.supporters (*module*), 509
pytorch_lightning.trainer.trainer (*module*), 510
pytorch_lightning.trainer.training_io (*module*), 519
pytorch_lightning.trainer.training_loop (*module*), 522
pytorch_lightning.trainer.training_tricks (*module*), 527
pytorch_lightning.utilities (*module*), 529
pytorch_lightning.utilities.apply_func (*module*), 529
pytorch_lightning.utilities.cloud_io (*module*), 530
pytorch_lightning.utilities.device_dtype_mixin (*module*), 530
pytorch_lightning.utilities.distributed (*module*), 532
pytorch_lightning.utilities.exceptions (*module*), 532
pytorch_lightning.utilities.memory (*module*), 532
pytorch_lightning.utilities.parsing (*module*), 532
pytorch_lightning.utilities.seed (*module*), 533

## R

rank_zero_info() (*in module pytorch_lightning.utilities.distributed*), 532
rank_zero_only() (*in module pytorch_lightning.utilities.distributed*), 532
rank_zero_warn() (*in module pytorch_lightning.utilities.distributed*), 532
Recall (*class in pytorch_lightning.metrics*), 409
Recall (*class in pytorch_lightning.metrics.classification*), 434
Recall (*class in pytorch_lightning.metrics.sklearns*), 452
recall() (*in module pytorch_lightning.metrics.functional.classification*), 420
recursive_detach() (*in module pytorch_lightning.utilities.memory*), 532
reduce() (*in module pytorch_lightning.metrics.functional.reduction*), 423
reduce_distributed_output() (*pytorch_lightning.trainer.logging.TrainerLoggingMixin method*), 504
reduce_lr_on_plateau_scheduler (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 527
refresh_rate() (*pytorch_lightning.callbacks.progress.ProgressBar property*), 351
refresh_rate() (*pytorch_lightning.callbacks.ProgressBar property*), 342
register_slurm_signal_handlers() (*pytorch_lightning.trainer.training_io.TrainerIOMixin method*), 521
reinit_scheduler_properties() (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin method*), 496
reinit_scheduler_properties() (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin method*), 497
reinit_scheduler_properties() (*pytorch_lightning.trainer.optimizers.TrainerOptimizersMixin method*), 508
reload_dataloaders_every_epoch (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopM attribute*), 503

reload_dataloaders_every_epoch (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 527
replace_sampler() (pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin method), 491
replace_sampler_ddp (pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin attribute), 492
request_dataloader() (pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin method), 491
reset() (pytorch_lightning.trainer.supporters.TensorRunningAccum method), 509
reset_experiment() (pytorch_lightning.loggers.comet.CometLogger method), 379
reset_experiment() (pytorch_lightning.loggers.CometLogger method), 359
reset_test_dataloader() (pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin method), 491
reset_test_dataloader() (pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin method), 502
reset_train_dataloader() (pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin method), 491
reset_train_dataloader() (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method), 525
reset_val_dataloader() (pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin method), 491
reset_val_dataloader() (pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin method), 502
reset_val_dataloader() (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method), 525
resolve_root_node_address() (pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin method), 496
restore() (pytorch_lightning.trainer.lr_finder.TrainerLRFinderMixin method), 506
restore() (pytorch_lightning.trainer.training_io.TrainerIOMixin method), 521
restore() (pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixin method), 528
restore_hpc_weights_if_needed() (pytorch_lightning.trainer.training_io.TrainerIOMixin method), 521
restore_training_state() (pytorch_lightning.trainer.training_io.TrainerIOMixin method), 521
restore_weights() (pytorch_lightning.trainer.training_io.TrainerIOMixin method), 521
resume_from_checkpoint (pytorch_lightning.trainer.Trainer attribute), 488
resume_from_checkpoint (pytorch_lightning.trainer.trainer.Trainer attribute), 519
resume_from_checkpoint (pytorch_lightning.trainer.training_io.TrainerIOMixin attribute), 522
retry_jittered_backoff() (in module pytorch_lightning.trainer.distrib_parts), 499
RMSE (class in pytorch_lightning.metrics), 396
RMSE (class in pytorch_lightning.metrics.regression), 440
RMSLE (class in pytorch_lightning.metrics), 397
RMSLE (class in pytorch_lightning.metrics.regression), 441
ROC (class in pytorch_lightning.metrics), 408
ROC (class in pytorch_lightning.metrics.classification), 433
ROC (class in pytorch_lightning.metrics.sklearns), 450
roc() (in module pytorch_lightning.metrics.functional.classification), 420
root_dir() (pytorch_lightning.loggers.tensorboard.TensorBoardLogger property), 388
root_dir() (pytorch_lightning.loggers.TensorBoardLogger property), 357
root_gpu (pytorch_lightning.trainer.distrib_parts.TrainerDPMixin attribute), 498
root_gpu (pytorch_lightning.trainer.Trainer attribute), 488
root_gpu (pytorch_lightning.trainer.trainer.Trainer attribute), 519
root_gpu (pytorch_lightning.trainer.training_io.TrainerIOMixin attribute), 522
row_log_interval (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 527
run_evaluation() (pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin method), 502
run_evaluation() (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method), 525
run_id() (pytorch_lightning.loggers.mlflow.MLFlowLogger property), 381
run_id() (pytorch_lightning.loggers.MLFlowLogger property), 361

run_pretrain_routine() (pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin method), 496
run_pretrain_routine() (pytorch_lightning.trainer.distrib_parts.TrainerDPMixin method), 497
run_pretrain_routine() (pytorch_lightning.trainer.Trainer method), 486
run_pretrain_routine() (pytorch_lightning.trainer.trainer.Trainer method), 517
run_training_batch() (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method), 525
run_training_epoch() (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method), 525
run_training_teardown() (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method), 525
running_loss (pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute), 527

# S

sanitize_gpu_ids() (in module pytorch_lightning.trainer.distrib_parts), 500
save() (pytorch_lightning.loggers.base.LightningLoggerBase method), 375
save() (pytorch_lightning.loggers.base.LoggerCollection method), 376
save() (pytorch_lightning.loggers.LightningLoggerBase method), 354
save() (pytorch_lightning.loggers.LoggerCollection method), 355
save() (pytorch_lightning.loggers.tensorboard.TensorBoardLogger method), 387
save() (pytorch_lightning.loggers.TensorBoardLogger method), 357
save() (pytorch_lightning.loggers.test_tube.TestTubeLogger method), 389
save() (pytorch_lightning.loggers.TestTubeLogger method), 367
save_checkpoint() (pytorch_lightning.trainer.callback_config.TrainerCallbackConfigMixin method), 489
save_checkpoint() (pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin method), 496
save_checkpoint() (pytorch_lightning.trainer.lr_finder.TrainerLRFinderMixin method), 506
save_checkpoint() (pytorch_lightning.trainer.training_io.TrainerIOMixin method), 521
save_checkpoint() (pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixin method), 528
save_hparams_to_tags_csv() (in module pytorch_lightning.core.saving), 334
save_hparams_to_yaml() (in module pytorch_lightning.core.saving), 334
save_hyperparameters() (pytorch_lightning.core.lightning.LightningModule method), 311
save_hyperparameters() (pytorch_lightning.core.LightningModule method), 281
save_spawn_weights (pytorch_lightning.trainer.distrib_parts.TrainerDPMixin attribute), 498
save_spawn_weights() (pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin method), 496
scale_batch_size() (pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixin method), 528
scaler (pytorch_lightning.trainer.Trainer attribute), 488
scaler (pytorch_lightning.trainer.trainer.Trainer attribute), 519
scaler (pytorch_lightning.trainer.training_io.TrainerIOMixin attribute), 522
seed_everything() (in module pytorch_lightning.trainer), 488
seed_everything() (in module pytorch_lightning.utilities.seed), 533
set_bypass_mode() (pytorch_lightning.loggers.trains.TrainsLogger class method), 393
set_bypass_mode() (pytorch_lightning.loggers.TrainsLogger class method), 372
set_credentials() (pytorch_lightning.loggers.trains.TrainsLogger class method), 393
set_credentials() (pytorch_lightning.loggers.TrainsLogger class method), 372
set_distributed_mode() (pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin method), 496
set_nvidia_flags() (pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin method), 496
set_property() (pytorch_lightning.loggers.neptune.NeptuneLogger method), 385

set_property() (*pytorch_lightning.loggers.NeptuneLogger method*), 365
set_random_port() (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin method*), 496
setup() (*pytorch_lightning.callbacks.base.Callback method*), 344
setup() (*pytorch_lightning.callbacks.Callback method*), 335
setup() (*pytorch_lightning.core.hooks.ModelHooks method*), 301
setup() (*pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method*), 490
show_progress_bar() (*pytorch_lightning.trainer.deprecated_api.TrainerDeprecatedAPITillVer0_9 property*), 493
shown_warnings (*pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin attribute*), 492
sig_handler() (*pytorch_lightning.trainer.training_io.TrainerIOMixin method*), 521
SimpleProfiler (*class in pytorch_lightning.profiler*), 457
SimpleProfiler (*class in pytorch_lightning.profiler.profilers*), 460
single_gpu (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin attribute*), 498
single_gpu (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin attribute*), 503
single_gpu (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 527
single_gpu_train() (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin method*), 497
SklearnMetric (*class in pytorch_lightning.metrics*), 410
SklearnMetric (*class in pytorch_lightning.metrics.sklearns*), 453
slurm_job_id (*pytorch_lightning.trainer.logging.TrainerLoggingMixin attribute*), 504
slurm_job_id() (*pytorch_lightning.trainer.callback_config.TrainerCallbackConfigMixin property*), 489
slurm_job_id() (*pytorch_lightning.trainer.Trainer property*), 488
slurm_job_id() (*pytorch_lightning.trainer.trainer.Trainer property*), 519
spawn_ddp_children() (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin method*), 496
start() (*pytorch_lightning.profiler.AdvancedProfiler method*), 458
start() (*pytorch_lightning.profiler.BaseProfiler method*), 457
start() (*pytorch_lightning.profiler.PassThroughProfiler method*), 458
start() (*pytorch_lightning.profiler.profilers.AdvancedProfiler method*), 459
start() (*pytorch_lightning.profiler.profilers.BaseProfiler method*), 460
start() (*pytorch_lightning.profiler.profilers.PassThroughProfiler method*), 460
start() (*pytorch_lightning.profiler.profilers.SimpleProfiler method*), 460
start() (*pytorch_lightning.profiler.SimpleProfiler method*), 457
stat_scores() (*in module pytorch_lightning.metrics.functional.classification*), 421
stat_scores_multiple_classes() (*in module pytorch_lightning.metrics.functional.classification*), 421
state_dict() (*pytorch_lightning.trainer.optimizers._MockOptimizer method*), 509
step() (*pytorch_lightning.trainer.optimizers._MockOptimizer method*), 509
stop() (*pytorch_lightning.profiler.AdvancedProfiler method*), 458
stop() (*pytorch_lightning.profiler.BaseProfiler method*), 457
stop() (*pytorch_lightning.profiler.PassThroughProfiler method*), 458
stop() (*pytorch_lightning.profiler.profilers.AdvancedProfiler method*), 459
stop() (*pytorch_lightning.profiler.profilers.BaseProfiler method*), 460
stop() (*pytorch_lightning.profiler.profilers.PassThroughProfiler method*), 460
stop() (*pytorch_lightning.profiler.profilers.SimpleProfiler method*), 460
stop() (*pytorch_lightning.profiler.SimpleProfiler method*), 457
str_to_bool() (*in module pytorch_lightning.utilities.parsing*), 533
suggestion() (*pytorch_lightning.trainer.lr_finder._LRFinder method*), 507
summarize() (*pytorch_lightning.core.lightning.LightningModule method*), 311
summarize() (*pytorch_lightning.core.LightningModule method*), 282
summarize() (*pytorch_lightning.core.memory.ModelSummary method*), 329
summary() (*pytorch_lightning.profiler.AdvancedProfiler method*), 458

summary() (*pytorch_lightning.profiler.BaseProfiler method*), 457
summary() (*pytorch_lightning.profiler.PassThroughProfiler method*), 458
summary() (*pytorch_lightning.profiler.profilers.AdvancedProfiler method*), 459
summary() (*pytorch_lightning.profiler.profilers.BaseProfiler method*), 460
summary() (*pytorch_lightning.profiler.profilers.PassThroughProfiler method*), 460
summary() (*pytorch_lightning.profiler.profilers.SimpleProfiler method*), 460
summary() (*pytorch_lightning.profiler.SimpleProfiler method*), 457
sync_ddp() (*in module pytorch_lightning.metrics.converters*), 437

## T

tbptt_split_batch() (*pytorch_lightning.core.lightning.LightningModule method*), 312
tbptt_split_batch() (*pytorch_lightning.core.LightningModule method*), 282
teardown() (*pytorch_lightning.callbacks.base.Callback method*), 344
teardown() (*pytorch_lightning.callbacks.Callback method*), 335
teardown() (*pytorch_lightning.core.hooks.ModelHooks method*), 302
teardown() (*pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin method*), 490
tensor_collection_metric() (*in module pytorch_lightning.metrics.converters*), 437
tensor_metric() (*in module pytorch_lightning.metrics.converters*), 437
TensorBoardLogger (*class in pytorch_lightning.loggers*), 356
TensorBoardLogger (*class in pytorch_lightning.loggers.tensorboard*), 386
TensorCollectionMetric (*class in pytorch_lightning.metrics.metric*), 438
TensorMetric (*class in pytorch_lightning.metrics.metric*), 439
TensorRunningAccum (*class in pytorch_lightning.trainer.supporters*), 509
term_handler() (*pytorch_lightning.trainer.training_io.TrainerIOMixin method*), 521
terminate_on_nan (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 527
test() (*pytorch_lightning.trainer.Trainer method*), 486
test() (*pytorch_lightning.trainer.trainer.Trainer method*), 517
test_batch_idx() (*pytorch_lightning.callbacks.progress.ProgressBarBase property*), 352
test_batch_idx() (*pytorch_lightning.callbacks.ProgressBarBase property*), 340
test_dataloader() (*pytorch_lightning.core.lightning.LightningModule method*), 312
test_dataloader() (*pytorch_lightning.core.LightningModule method*), 283
test_dataloaders (*pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin attribute*), 492
test_dataloaders (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopM attribute*), 503
test_end() (*pytorch_lightning.core.lightning.LightningModule method*), 313
test_end() (*pytorch_lightning.core.LightningModule method*), 284
test_epoch_end() (*pytorch_lightning.core.lightning.LightningModule method*), 313
test_epoch_end() (*pytorch_lightning.core.LightningModule method*), 284
test_percent_check() (*pytorch_lightning.trainer.deprecated_api.TrainerDeprecatedAPITill property*), 493
test_step() (*pytorch_lightning.core.lightning.LightningModule method*), 314
test_step() (*pytorch_lightning.core.LightningModule method*), 285
test_step_end() (*pytorch_lightning.core.lightning.LightningModule method*), 316
test_step_end() (*pytorch_lightning.core.LightningModule method*), 287
testing (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin attribute*), 498
testing (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 527
TestTubeLogger (*class in pytorch_lightning.loggers*), 365
TestTubeLogger (*class in pytorch_lightning.loggers.test_tube*), 388
tng_dataloader() (*pytorch_lightning.core.lightning.LightningModule method*), 317

tng_dataloader() (*pytorch_lightning.core.LightningModule method*), 287
to() (*pytorch_lightning.utilities.device_dtype_mixin.DeviceDtypeModuleMixin method*), 530
to_categorical() (*in module pytorch_lightning.metrics.functional.classification*), 422
to_onehot() (*in module pytorch_lightning.metrics.functional.classification*), 422
total_batch_idx (*pytorch_lightning.trainer.lr_finder.TrainerLRFinderMixin attribute*), 506
total_batch_idx (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 527
total_test_batches() (*pytorch_lightning.callbacks.progress.ProgressBarBase property*), 352
total_test_batches() (*pytorch_lightning.callbacks.ProgressBarBase property*), 340
total_train_batches() (*pytorch_lightning.callbacks.progress.ProgressBarBase property*), 352
total_train_batches() (*pytorch_lightning.callbacks.ProgressBarBase property*), 340
total_val_batches() (*pytorch_lightning.callbacks.progress.ProgressBarBase property*), 352
total_val_batches() (*pytorch_lightning.callbacks.ProgressBarBase property*), 340
tpu_cores (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin attribute*), 497
tpu_global_core_rank (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin attribute*), 498
tpu_id (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin attribute*), 498
tpu_id (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin attribute*), 503
tpu_id (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 527
tpu_local_core_rank (*pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin attribute*), 492
tpu_local_core_rank (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin attribute*), 498
tpu_train() (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin method*), 497
track_grad_norm (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 527
train() (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method*), 525
train_batch_idx() (*pytorch_lightning.callbacks.progress.ProgressBarBase property*), 352
train_batch_idx() (*pytorch_lightning.callbacks.ProgressBarBase property*), 340
train_dataloader (*pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin attribute*), 492
train_dataloader (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 527
train_dataloader() (*pytorch_lightning.core.lightning.LightningModule method*), 317
train_dataloader() (*pytorch_lightning.core.LightningModule method*), 288
train_percent_check() (*pytorch_lightning.trainer.deprecated_api.TrainerDeprecatedAPITillVer0_10 property*), 493
Trainer (*class in pytorch_lightning.trainer*), 479
Trainer (*class in pytorch_lightning.trainer.trainer*), 510
trainer (*pytorch_lightning.core.lightning.LightningModule attribute*), 327
trainer (*pytorch_lightning.core.LightningModule attribute*), 298
trainer() (*pytorch_lightning.callbacks.progress.ProgressBarBase property*), 352
trainer() (*pytorch_lightning.callbacks.ProgressBarBase property*), 340
TrainerAMPMixin (*class in pytorch_lightning.trainer.auto_mix_precision*), 488
TrainerCallbackConfigMixin (*class in pytorch_lightning.trainer.callback_config*), 489
TrainerCallbackHookMixin (*class in pytorch_lightning.trainer.callback_hook*), 489
TrainerDataLoadingMixin (*class in pytorch_lightning.trainer.data_loading*), 491
TrainerDDPMixin (*class in pytorch_lightning.trainer.distrib_data_parallel*), 495
TrainerDeprecatedAPITillVer0_10 (*class in pytorch_lightning.trainer.deprecated_api*), 492
TrainerDeprecatedAPITillVer0_9 (*class in pytorch_lightning.trainer.deprecated_api*), 493
TrainerDPMixin (*class in py-

*torch_lightning.trainer.distrib_parts*), 497

`TrainerEvaluationLoopMixin` (*class in pytorch_lightning.trainer.evaluation_loop*), 502

`TrainerIOMixin` (*class in pytorch_lightning.trainer.training_io*), 521

`TrainerLoggingMixin` (*class in pytorch_lightning.trainer.logging*), 504

`TrainerLRFinderMixin` (*class in pytorch_lightning.trainer.lr_finder*), 505

`TrainerModelHooksMixin` (*class in pytorch_lightning.trainer.model_hooks*), 508

`TrainerOptimizersMixin` (*class in pytorch_lightning.trainer.optimizers*), 508

`TrainerTrainingTricksMixin` (*class in pytorch_lightning.trainer.training_tricks*), 527

`TrainerTrainLoopMixin` (*class in pytorch_lightning.trainer.training_loop*), 524

`training_end()` (*pytorch_lightning.core.lightning.LightningModule method*), 318

`training_end()` (*pytorch_lightning.core.LightningModule method*), 288

`training_epoch_end()` (*pytorch_lightning.core.lightning.LightningModule method*), 318

`training_epoch_end()` (*pytorch_lightning.core.LightningModule method*), 288

`training_forward()` (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method*), 525

`training_step()` (*pytorch_lightning.core.lightning.LightningModule method*), 319

`training_step()` (*pytorch_lightning.core.LightningModule method*), 290

`training_step_end()` (*pytorch_lightning.core.lightning.LightningModule method*), 321

`training_step_end()` (*pytorch_lightning.core.LightningModule method*), 291

`training_tqdm_dict()` (*pytorch_lightning.trainer.deprecated_api.TrainerDeprecatedAPITillVer0_9 property*), 493

`TrainsLogger` (*class in pytorch_lightning.loggers*), 369

`TrainsLogger` (*class in pytorch_lightning.loggers.trains*), 390

`transfer_batch_to_device()` (*pytorch_lightning.core.hooks.ModelHooks method*), 302

`transfer_batch_to_gpu()` (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin method*), 497

`transfer_batch_to_gpu()` (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopM method*), 502

`transfer_batch_to_gpu()` (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method*), 525

`transfer_batch_to_tpu()` (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin method*), 498

`transfer_batch_to_tpu()` (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopM method*), 502

`transfer_batch_to_tpu()` (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method*), 525

`truncated_bptt_steps` (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 527

`type()` (*pytorch_lightning.utilities.device_dtype_mixin.DeviceDtypeModu method*), 531

## U

`unfreeze()` (*pytorch_lightning.core.lightning.LightningModule method*), 322

`unfreeze()` (*pytorch_lightning.core.LightningModule method*), 292

`update_agg_funcs()` (*pytorch_lightning.loggers.base.LightningLoggerBase method*), 375

`update_agg_funcs()` (*pytorch_lightning.loggers.LightningLoggerBase method*), 354

`update_hparams()` (*in module pytorch_lightning.core.saving*), 334

`update_learning_rates()` (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin method*), 525

`use_amp` (*pytorch_lightning.core.lightning.LightningModule attribute*), 327

`use_amp` (*pytorch_lightning.core.LightningModule attribute*), 298

`use_amp` (*pytorch_lightning.trainer.training_io.TrainerIOMixin attribute*), 522

`use_amp()` (*pytorch_lightning.trainer.auto_mix_precision.TrainerAMPM property*), 488

`use_amp()` (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPI property*), 497

`use_amp()` (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin property*), 498

`use_ddp` (*pytorch_lightning.core.lightning.LightningModule attribute*), 327

use_ddp (*pytorch_lightning.core.LightningModule attribute*), 298
use_ddp (*pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin attribute*), 492
use_ddp (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin attribute*), 498
use_ddp (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin attribute*), 503
use_ddp (*pytorch_lightning.trainer.Trainer attribute*), 488
use_ddp (*pytorch_lightning.trainer.trainer.Trainer attribute*), 519
use_ddp (*pytorch_lightning.trainer.training_io.TrainerIOMixin attribute*), 522
use_ddp (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 527
use_ddp2 (*pytorch_lightning.core.lightning.LightningModule attribute*), 327
use_ddp2 (*pytorch_lightning.core.LightningModule attribute*), 298
use_ddp2 (*pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin attribute*), 492
use_ddp2 (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin attribute*), 498
use_ddp2 (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin attribute*), 503
use_ddp2 (*pytorch_lightning.trainer.logging.TrainerLoggingMixin attribute*), 504
use_ddp2 (*pytorch_lightning.trainer.Trainer attribute*), 488
use_ddp2 (*pytorch_lightning.trainer.trainer.Trainer attribute*), 519
use_ddp2 (*pytorch_lightning.trainer.training_io.TrainerIOMixin attribute*), 522
use_ddp2 (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 527
use_dp (*pytorch_lightning.core.lightning.LightningModule attribute*), 327
use_dp (*pytorch_lightning.core.LightningModule attribute*), 298
use_dp (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin attribute*), 498
use_dp (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin attribute*), 503
use_dp (*pytorch_lightning.trainer.logging.TrainerLoggingMixin attribute*), 504
use_dp (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 527
use_horovod (*pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin attribute*), 492
use_horovod (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin attribute*), 503
use_horovod (*pytorch_lightning.trainer.Trainer attribute*), 488
use_horovod (*pytorch_lightning.trainer.trainer.Trainer attribute*), 519
use_horovod (*pytorch_lightning.trainer.training_io.TrainerIOMixin attribute*), 522
use_horovod (*pytorch_lightning.trainer.training_loop.TrainerTrainLoop attribute*), 527
use_native_amp (*pytorch_lightning.trainer.auto_mix_precision.TrainerAMPMixin attribute*), 488
use_native_amp (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin attribute*), 497
use_native_amp (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin attribute*), 499
use_native_amp (*pytorch_lightning.trainer.Trainer attribute*), 488
use_native_amp (*pytorch_lightning.trainer.trainer.Trainer attribute*), 519
use_native_amp (*pytorch_lightning.trainer.training_io.TrainerIOMixin attribute*), 522
use_tpu (*pytorch_lightning.trainer.data_loading.TrainerDataLoadingMix attribute*), 492
use_tpu (*pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMix attribute*), 497
use_tpu (*pytorch_lightning.trainer.distrib_parts.TrainerDPMixin attribute*), 499
use_tpu (*pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLo attribute*), 503
use_tpu (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 527


## V


val_batch_idx() (*pytorch_lightning.callbacks.progress.ProgressBarBase property*), 352
val_batch_idx() (*pytorch_lightning.callbacks.ProgressBarBase property*), 340
val_check_batch (*pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin attribute*), 492
val_check_batch (*pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin attribute*), 527
val_check_interval (*pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin attribute*), 492
val_dataloader() (*pytorch_lightning.core.lightning.LightningModule method*), 322

val_dataloader() (pytorch_lightning.core.LightningModule method), 292
val_dataloaders (pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin attribute), 492
val_dataloaders (pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin attribute), 503
val_percent_check() (pytorch_lightning.trainer.deprecated_api.TrainerDeprecatedAPITillVer0_10 property), 493
validation_end() (pytorch_lightning.core.lightning.LightningModule method), 323
validation_end() (pytorch_lightning.core.LightningModule method), 293
validation_epoch_end() (pytorch_lightning.core.lightning.LightningModule method), 323
validation_epoch_end() (pytorch_lightning.core.LightningModule method), 293
validation_step() (pytorch_lightning.core.lightning.LightningModule method), 324
validation_step() (pytorch_lightning.core.LightningModule method), 295
validation_step_end() (pytorch_lightning.core.lightning.LightningModule method), 326
validation_step_end() (pytorch_lightning.core.LightningModule method), 296
version() (pytorch_lightning.loggers.base.DummyLogger property), 373
version() (pytorch_lightning.loggers.base.LightningLoggerBase property), 376
version() (pytorch_lightning.loggers.base.LoggerCollection property), 377
version() (pytorch_lightning.loggers.comet.CometLogger property), 380
version() (pytorch_lightning.loggers.CometLogger property), 359
version() (pytorch_lightning.loggers.LightningLoggerBase property), 355
version() (pytorch_lightning.loggers.LoggerCollection property), 356
version() (pytorch_lightning.loggers.mlflow.MLFlowLogger property), 381
version() (pytorch_lightning.loggers.MLFlowLogger property), 361
version() (pytorch_lightning.loggers.neptune.NeptuneLogger property), 386
version() (pytorch_lightning.loggers.NeptuneLogger property), 365
version() (pytorch_lightning.loggers.tensorboard.TensorBoardLogger property), 388
version() (pytorch_lightning.loggers.TensorBoardLogger property), 357
version() (pytorch_lightning.loggers.test_tube.TestTubeLogger property), 389
version() (pytorch_lightning.loggers.TestTubeLogger property), 367
version() (pytorch_lightning.loggers.trains.TrainsLogger property), 394
version() (pytorch_lightning.loggers.TrainsLogger property), 373
version() (pytorch_lightning.loggers.wandb.WandbLogger property), 395
version() (pytorch_lightning.loggers.WandbLogger property), 368


## W


WandbLogger (class in pytorch_lightning.loggers), 367
WandbLogger (class in pytorch_lightning.loggers.wandb), 394
watch() (pytorch_lightning.loggers.wandb.WandbLogger method), 395
watch() (pytorch_lightning.loggers.WandbLogger method), 368
weights_save_path (pytorch_lightning.trainer.callback_config.TrainerCallbackConfigMixin attribute), 489
weights_save_path (pytorch_lightning.trainer.Trainer attribute), 488
weights_save_path (pytorch_lightning.trainer.trainer.Trainer attribute), 519
weights_save_path (pytorch_lightning.trainer.training_io.TrainerIOMixin attribute), 522


## Z


zero_grad() (pytorch_lightning.trainer.optimizers._MockOptimizer method), 509