PyTorch-Lightning Documentation

Release 0.6.0

William Falcon et al.

Jan 21, 2020

Start Here

1 Quick Start 1

2 Callbacks 3

3 LightningModule 5

4 Logging 23

5 Trainer 35

6 GAN 45

7 MNIST 47

8 Multi-node (ddp) MNIST 49

9 Multi-node (ddp2) MNIST 51

10 Imagenet 53

11 Refactoring PyTorch into Lightning 55

12 Start a research project 57

13 Basic Lightning use 59

14 9 key Lightning tricks 61

15 Multi-node training on SLURM 63

16 Multi-gpu (same node) training 65

17 Multi-node training 67

18 16-bit precision 69

19 gradient clipping 71

20 modifying training via hooks 73

21 Contributor Covenant Code of Conduct 75

22 Contributing 77

23 How to become a core contributor 81

24 Pytorch Lightning Governance | Persons of interest 85

25 Indices and tables 87

Python Module Index 89

Index 91

CHAPTER 1

Quick Start

To start a new project define two files, a LightningModule and a Trainer file.
To illustrate the power of Lightning and its simplicity, here's an example of a typical research flow.

1.1 Case 1: BERT

Let's say you're working on something like BERT but want to try different ways of training or even different networks.
You would define a single LightningModule and use flags to switch between your different ideas.

```
class BERT(pl.LightningModule):
    def __init__(self, model_name, task):
        self.task = task

        if model_name == 'transformer':
            self.net = Transformer()
        elif model_name == 'my_cool_version':
            self.net = MyCoolVersion()

    def training_step(self, batch, batch_idx):
        if self.task == 'standard_bert':
            # do standard bert training with self.net...
            # return loss

        if self.task == 'my_cool_task':
            # do my own version with self.net
            # return loss
```

1.2 Case 2: COOLER NOT BERT

But if you wanted to try something **completely** different, you'd define a new module for that.

```
class CoolerNotBERT(pl.LightningModule):
    def __init__(self):
        self.net = ...

    def training_step(self, batch, batch_idx):
        # do some other cool task
        # return loss
```

1.3 Rapid research flow

Then you could do rapid research by switching between these two and using the same trainer.

```
if use_bert:
    model = BERT()
else:
    model = CoolerNotBERT()

trainer = Trainer(gpus=4, use_amp=True)
trainer.fit(model)
```

Notice a few things about this flow:

1. You're writing pure PyTorch... no unnecessary abstractions or new libraries to learn.
2. You get free GPU and 16-bit support without writing any of that code in your model.
3. You also get early stopping, multi-gpu training, 16-bit and MUCH more without coding anything!

CHAPTER 2

Callbacks

class pytorch_lightning.callbacks.**EarlyStopping**(*monitor='val_loss'*, *min_delta=0.0*, *patience=0*, *verbose=0*, *mode='auto'*)

Bases: pytorch_lightning.callbacks.pt_callbacks.Callback

Stop training when a monitored quantity has stopped improving.

Parameters

- **monitor** (*str*) – quantity to be monitored.
- **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.
- **patience** (*int*) – number of epochs with no improvement after which training will be stopped.
- **verbose** (*bool*) – verbosity mode.
- **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.

Example:

```
from pytorch_lightning import Trainer
from pytorch_lightning.callbacks import EarlyStopping

early_stopping = EarlyStopping('val_loss')
Trainer(early_stop_callback=early_stopping)
```

class pytorch_lightning.callbacks.**ModelCheckpoint**(*filepath*, *monitor='val_loss'*, *verbose=0*, *save_top_k=1*, *save_weights_only=False*, *mode='auto'*, *period=1*, *prefix=''*)

Bases: pytorch_lightning.callbacks.pt_callbacks.Callback

Save the model after every epoch.

Parameters

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

  Example:

```
# save epoch and val_loss in name
ModelCheckpoint(filepath='{epoch:02d}-{val_loss:.2f}.hdf5')
# saves file like: /path/epoch_2-val_loss_0.2.hdf5
```

- **monitor** (*str*) – quantity to monitor.
- **verbose** (*bool*) – verbosity mode, 0 or 1.
- **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

checkpoint_callback = ModelCheckpoint(filepath='my_path')
Trainer(checkpoint_callback=checkpoint_callback)

# saves checkpoints to my_path whenever 'val_loss' has a new min
```

class pytorch_lightning.callbacks.**GradientAccumulationScheduler**(*scheduling: dict*)

Bases: pytorch_lightning.callbacks.pt_callbacks.Callback

Change gradient accumulation factor according to scheduling.

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

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(accumulate_grad_batches=accumulator)
```

CHAPTER 3

LightningModule

A LightningModule is a strict superclass of torch.nn.Module but provides an interface to standardize the "ingredients" for a research or production system.

- The model/system definition (__init__)
- The model/system computations (forward)
- What happens in the training loop (training_step, training_end)
- What happens in the validation loop (validation_step, validation_end)
- What happens in the test loop (test_step, test_end)
- What optimizers to use (configure_optimizers)
- What data to use (train_dataloader, val_dataloader, test_dataloader)

Most methods are optional. Here's a minimal example.

```
import os
import torch
from torch.nn import functional as F
from torch.utils.data import DataLoader
from torchvision.datasets import MNIST
import torchvision.transforms as transforms

import pytorch_lightning as pl

class CoolModel(pl.LightningModule):

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

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

(continues on next page)

(continued from previous page)

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

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

    def validation_end(self, outputs):
        # OPTIONAL
        val_loss_mean = torch.stack([x['val_loss'] for x in outputs]).mean()
        return {'val_loss': val_loss_mean}

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

    def test_end(self, outputs):
        # OPTIONAL
        test_loss_mean = torch.stack([x['test_loss'] for x in outputs]).mean()
        return {'test_loss': test_loss_mean}

    def configure_optimizers(self):
        # REQUIRED
        return torch.optim.Adam(self.parameters(), lr=0.02)

    @pl.data_loader
    def train_dataloader(self):
        return DataLoader(MNIST(os.getcwd(), train=True, download=True,
                          transform=transforms.ToTensor()), batch_size=32)

    @pl.data_loader
    def val_dataloader(self):
        # OPTIONAL
        # can also return a list of val dataloaders
        return DataLoader(MNIST(os.getcwd(), train=True, download=True,
                          transform=transforms.ToTensor()), batch_size=32)

    @pl.data_loader
    def test_dataloader(self):
        # OPTIONAL
        # can also return a list of test dataloaders
        return DataLoader(MNIST(os.getcwd(), train=False, download=True,
                          transform=transforms.ToTensor()), batch_size=32)
```

Once you've defined the LightningModule, fit it using a trainer.

Check out this COLAB for a live demo.

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

Bases: abc.ABC, pytorch_lightning.core.grads.GradInformation, pytorch_lightning.core.saving.ModelIO, pytorch_lightning.core.hooks.ModelHooks

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*) – list of optimizers passed in configure_optimizers()
- **amp_level** (*str*) – AMP mode chosen ('O1', 'O2', etc...)

Returns Apex wrapped model and optimizers

Example

```
# 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*) – the list of GPU ids

Returns DDP wrapped model

Example

```
# 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()
This is where you choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or something more esoteric you might have multiple.

Return: any of these 3 options:

- Single optimizer
- List or Tuple - List of optimizers
- Two lists - The first list has multiple optimizers, the second a list of learning-rate schedulers

Example

```
# most cases
def configure_optimizers(self):
    opt = Adam(self.parameters(), lr=0.01)
    return opt

# multiple optimizer case (eg: 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]
```

Note: Lightning calls .backward() and .step() on each optimizer and learning rate scheduler as needed.

Note: If you use 16-bit precision (use_amp=True), Lightning will automatically handle the optimizers for you.

Note: If you use multiple optimizers, training_step will have an additional *optimizer_idx* parameter.

Note: If you use LBFGS lightning handles the closure function automatically for you

Note: If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.

Note: If you need to control how often those optimizers step or override the default .step() schedule, override the *optimizer_step* hook.

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 (ie: on a server or as a feature extractor).

Normally you'd call self.forward() 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.

Parameters **x** (*tensor*) – Whatever you decide to define in the forward method

Returns Predicted output

Example

```
# 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.forward(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()
```

init_ddp_connection(*proc_rank*, *world_size*)

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.

Parameters

- **proc_rank** (*int*) – The current process rank within the node.
- **world_size** (*int*) – Number of GPUs being use across all nodes. (num_nodes*nb_gpu_nodes).

Example

```
def init_ddp_connection(self):
    # use slurm job id for the port number
    # guarantees unique ports across jobs from same grid search
    try:
        # use the last 4 numbers in the job id as the id
        default_port = os.environ['SLURM_JOB_ID']
        default_port = default_port[-4:]

        # all ports should be in the 10k+ range
        default_port = int(default_port) + 15000

    except Exception as e:
        default_port = 12910

    # if user gave a port number, use that one instead
    try:
        default_port = os.environ['MASTER_PORT']
    except Exception:
        os.environ['MASTER_PORT'] = str(default_port)

    # figure out the root node addr
    try:
        root_node = os.environ['SLURM_NODELIST'].split(' ')[0]
    except Exception:
        root_node = '127.0.0.2'

    root_node = self.trainer.resolve_root_node_address(root_node)
    os.environ['MASTER_ADDR'] = root_node
    dist.init_process_group(
        'nccl',
        rank=self.proc_rank,
        world_size=self.world_size
    )
```

classmethod load_from_checkpoint(*checkpoint_path*, *map_location=None*)

Primary way of loading model from a checkpoint. When Lightning saves a checkpoint it stores the hyperparameters in the checkpoint if you initialized your LightningModule with an argument called *hparams* which is a Namespace or dictionary of hyperparameters

```
# --------------
# Case 1
# when using Namespace (output of using Argparse to parse command line␣
↪arguments)
from argparse import Namespace
hparams = Namespace(**{'learning_rate': 0.1})

model = MyModel(hparams)

class MyModel(pl.LightningModule):
    def __init__(self, hparams):
        self.learning_rate = hparams.learning_rate

# --------------
# Case 2
# when using a dict
```

(continues on next page)

(continued from previous page)

```
model = MyModel({'learning_rate': 0.1})

class MyModel(pl.LightningModule):
    def __init__(self, hparams):
        self.learning_rate = hparams['learning_rate']
```

Parameters

- **checkpoint_path** (*str*) – Path to checkpoint.
- **map_location** (*dic*) – If your checkpoint saved from a GPU model and you now load on CPUs or a different number of GPUs, use this to map to the new setup.

Returns LightningModule with loaded weights.

Example

```
# load weights without mapping
MyLightningModule.load_from_checkpoint('path/to/checkpoint.ckpt')

# 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)
```

classmethod load_from_metrics(*weights_path*, *tags_csv*, *map_location=None*)

You should use *load_from_checkpoint* instead! However, if your .ckpt weights don't have the hyperparameters saved, use this method to pass in a .csv with the hparams you'd like to use. These will be converted into a argparse.Namespace and passed into your LightningModule for use.

Parameters

- **weights_path** (*str*) – Path to a PyTorch checkpoint
- **tags_csv** (*str*) – Path to a .csv with two columns (key, value) as in this Example:

```
key,value
drop_prob,0.2
batch_size,32
```

- **map_location** (*dict*) – A dictionary mapping saved weight GPU devices to new GPU devices (example: {'cuda:1':'cuda:0'})

Returns LightningModule with loaded weights

Example

```
pretrained_model = MyLightningModule.load_from_metrics(
    weights_path='/path/to/pytorch_checkpoint.ckpt',
    tags_csv='/path/to/hparams_file.csv',
    on_gpu=True,
    map_location=None
)

# predict
```

(continues on next page)

(continued from previous page)

```
pretrained_model.eval()
pretrained_model.freeze()
y_hat = pretrained_model(x)
```

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*) – 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: Lighting auto-restores global step, epoch, and all training state including amp scaling. No need for you to restore anything regarding training.

---

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** (*dic*) – 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: Lighting saves all aspects of training (epoch, global step, etc...) including amp scaling. No need for you to store anything about training.

---

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** (*torch.nn.Optimizer*) – A PyTorch optimizer
- **optimizer_idx** (*int*) – If you used multiple optimizers this indexes into that list
- **second_order_closure** (*int*) – closure for second order methods

Example

```
# 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 (ie: 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.hparams.learning_rate

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

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** (*torch.nn.Tensor*) – Current batch
- **split_size** (*int*) – How big the split is

Returns list of batch splits. Each split will be passed to forward_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.

Example

```
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()

test_end(*outputs*)

Outputs has the appended output after each test step.

Parameters **outputs** – List of outputs you defined in test_step, or if there are multiple dataloaders, a list containing a list of outputs for each dataloader

Return dict Dict of OrderedDict with metrics to display in progress bar

If you didn't define a test_step, this won't be called. Called at the end of the test step with the output of each test_step. The outputs here are strictly for the progress bar. If you don't need to display anything, don't return anything.

Example

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

    test_loss_mean /= len(outputs)
    test_acc_mean /= len(outputs)
    tqdm_dict = {'test_loss': test_loss_mean.item(), '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_loss': val_loss_mean.item()}
```

(continues on next page)

(continued from previous page)

```
    }
    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 test_end(self, outputs):
    test_loss_mean = 0
    test_acc_mean = 0
    i = 0
    for dataloader_outputs in outputs:
        for output in dataloader_outputs:
            test_loss_mean += output['test_loss']
            test_acc_mean += output['test_acc']
            i += 1

    test_loss_mean /= i
    test_acc_mean /= i
    tqdm_dict = {'test_loss': test_loss_mean.item(), '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_loss': val_loss_mean.item()}
    }
    return results
```

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

return whatever outputs will need to be aggregated in test_end

Parameters

- **batch** – The output of your dataloader. A tensor, tuple or list
- **batch_idx** (*int*) – Integer displaying which batch this is
- **dataloader_idx** (*int*) – Integer displaying which dataloader this is (only if multiple test datasets used)

Return dict Dict or OrderedDict with metrics to display in progress bar. All keys must be tensors.

```
# 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_idxdx)
```

OPTIONAL If you don't need to test you don't need to implement this method. In this step you'd normally generate examples or calculate anything of interest such as accuracy.

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, model goes back to training mode and gradients are enabled.

The dict you return here will be available in the *test_end* method.

This function is used when you execute *trainer.test()*.

Example

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

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

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

    # all optional...
    # return whatever you need for the collation function test_end
    output = OrderedDict({
        'test_loss': loss_test,
        'test_acc': torch.tensor(test_acc), # everything must be a tensor
    })

    # return an optional dict
    return output
```

If you pass in multiple test 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.
```

The *dataset_idx* corresponds to the order of datasets returned in *test_dataloader*.

tng_dataloader()

train_dataloader()

training_end(**args*, ***kwargs*)
return loss, dict with metrics for tqdm

Parameters **outputs** – What you return in *training_step*.

Return dict dictionary with loss key and optional log, progress 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)

In certain cases (dp, ddp2), you might want to use all outputs of every process to do something. For instance, if using negative samples, you could run a batch via dp and use ALL the outputs for a single softmax across the full batch (ie: the denominator would use the full batch).

In this case you should define training_end to perform those calculations.

Example

```
# WITHOUT training_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.forward(x)
    loss = self.softmax(out)
    loss = nce_loss(loss)
    return {'loss': loss}

# --------------
# with training_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.forward(x)
    return {'out': out}

def training_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 = self.softmax(out)
    loss = nce_loss(loss)
    return {'loss': loss}
```

If you define multiple optimizers, this step will also be called with an additional *optimizer_idx* param.

```
# Multiple optimizers (ie: 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 hiddens from the previous truncated backprop step
```

You can also return a -1 instead of a dict to stop the current loop. This is useful if you want to break out of the current training epoch early.

training_step(**args*, ***kwargs*)
return loss, dict with metrics for tqdm

Parameters

- **batch** – The output of your dataloader. A tensor, tuple or list
- **batch_idx** (*int*) – Integer displaying which batch this is

Returns

dict with loss key and optional log, progress keys if 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 specific to your model.

Example

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

    # implement your own
    out = self.forward(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 also be called with an additional *optimizer_idx* param.

```
# Multiple optimizers (ie: 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 hiddens from the previous truncated backprop step
```

You can also return a -1 instead of a dict to stop the current loop. This is useful if you want to break out of the current training epoch early.

unfreeze()
Unfreeze all params for inference.

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

val_dataloader()

validation_end(*outputs*)
Outputs has the appended output after each validation step.

Parameters **outputs** – 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 dict Dictionary or OrderedDict with optional: 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)

If you didn't define a validation_step, this won't be called. Called at the end of the validation loop with the outputs of validation_step.

The outputs here are strictly for the progress bar. If you don't need to display anything, don't return anything. Any keys present in 'log', 'progress_bar' or the rest of the dictionary are available for callbacks to access.

Example

With a single dataloader

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

    val_loss_mean /= len(outputs)
    val_acc_mean /= len(outputs)
    tqdm_dict = {'val_loss': val_loss_mean.item(), '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_loss': val_loss_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_end(self, outputs):
    val_loss_mean = 0
    val_acc_mean = 0
    i = 0
    for dataloader_outputs in outputs:
        for output in dataloader_outputs:
```

(continues on next page)

(continued from previous page)

```
            val_loss_mean += output['val_loss']
            val_acc_mean += output['val_acc']
            i += 1

    val_loss_mean /= i
    val_acc_mean /= i
    tqdm_dict = {'val_loss': val_loss_mean.item(), '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_loss': val_loss_mean.item()}
    }
    return results
```

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

This is the validation loop. It is called for each batch of the validation set. Whatever is returned from here will be passed in as a list on validation_end. In this step you'd normally generate examples or calculate anything of interest such as accuracy.

Parameters

- **batch** (*torch.nn.Tensor | (Tensor, 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)

Returns Dict or OrderedDict - passed to the validation_end step

```
# 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_idxdx)
```

Example

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

    # implement your own
    out = self.forward(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
```

(continues on next page)

(continued from previous page)

```
    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_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, 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, model goes back to training mode and gradients are enabled.

current_epoch = None
The current epoch

dtype = None
Current dtype

global_step = None
Total training batches seen across all epochs

logger = None
Pointer to the logger object

on_gpu = None
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

CHAPTER 4

Logging

Lightning supports most popular logging frameworks (Tensorboard, comet, weights and biases, etc...). To use a logger, simply pass it into the trainer.

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

4.1 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.logging import LightningLoggerBase, rank_zero_only

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
        # your code to record metrics goes here
        pass

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

(continues on next page)

(continued from previous page)

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

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

4.2 Using loggers

Call the logger anywhere from your LightningModule by doing:

```
def train_step(...):
    # example
    self.logger.experiment.whatever_method_summary_writer_supports(...)

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

4.3 Supported Loggers

class pytorch_lightning.logging.**CometLogger**(*api_key=None*, *save_dir=None*, *workspace=None*, *rest_api_key=None*, *project_name=None*, *experiment_name=None*, ***kwargs*)

Bases: pytorch_lightning.logging.base.LightningLoggerBase

Log using comet.

Requires either an API Key (online mode) or a local directory path (offline mode)

```
# ONLINE MODE
from pytorch_lightning.logging import CometLogger

# arguments made to CometLogger are passed on to the comet_ml.Experiment class
comet_logger = CometLogger(
    api_key=os.environ["COMET_KEY"],
    workspace=os.environ["COMET_WORKSPACE"], # Optional
    project_name="default_project", # Optional
    rest_api_key=os.environ["COMET_REST_KEY"], # Optional
    experiment_name="default" # Optional
)
trainer = Trainer(logger=comet_logger)
```

```
# OFFLINE MODE
from pytorch_lightning.logging import CometLogger

# arguments made to CometLogger are passed on to the comet_ml.Experiment class
comet_logger = CometLogger(
    save_dir=".",
    workspace=os.environ["COMET_WORKSPACE"], # Optional
    project_name="default_project", # Optional
```

(continues on next page)

(continued from previous page)

```
    rest_api_key=os.environ["COMET_REST_KEY"], # Optional
    experiment_name="default" # Optional
)
trainer = Trainer(logger=comet_logger)
```

Parameters

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

finalize(*status*)

Do any processing that is necessary to finalize an experiment.

Parameters **status** – Status that the experiment finished with (e.g. success, failed, aborted)

log_hyperparams(*params*)

Record hyperparameters.

Parameters **params** – argparse.Namespace containing the hyperparameters

log_metrics(*metrics*, *step=None*)

Record metrics.

Parameters

- **metric** (*float*) – Dictionary with metric names as keys and measured quanties as values
- **step** (*int|None*) – Step number at which the metrics should be recorded

experiment

Actual comet object. To use comet features do the following.

Example:

```
self.logger.experiment.some_comet_function()
```

name

Return the experiment name.

version

Return the experiment version.

class pytorch_lightning.logging.**MLFlowLogger**(*experiment_name*, *tracking_uri=None*, *tags=None*)

Bases: pytorch_lightning.logging.base.LightningLoggerBase

Logs using MLFlow

Parameters

- **experiment_name** (*str*) – The name of the experiment
- **tracking_uri** (*str*) – where this should track
- **tags** (*dict*) – todo this param

finalize(*status='FINISHED'*)

Do any processing that is necessary to finalize an experiment.

Parameters **status** – Status that the experiment finished with (e.g. success, failed, aborted)

log_hyperparams(*params*)

Record hyperparameters.

Parameters **params** – argparse.Namespace containing the hyperparameters

log_metrics(*metrics*, *step=None*)

Record metrics.

Parameters

- **metric** (*float*) – Dictionary with metric names as keys and measured quanties as values
- **step** (*int|None*) – Step number at which the metrics should be recorded

save()

Save log data.

experiment

Actual mlflow object. To use mlflow features do the following.

Example:

```
self.logger.experiment.some_mlflow_function()
```

name

Return the experiment name.

run_id

version

Return the experiment version.

class pytorch_lightning.logging.**NeptuneLogger**(*api_key=None*, *project_name=None*, *offline_mode=False*, *experiment_name=None*, *upload_source_files=None*, *params=None*, *properties=None*, *tags=None*, ***kwargs*)

Bases: pytorch_lightning.logging.base.LightningLoggerBase

Initialize a neptune.ml logger.

Note: Requires either an API Key (online mode) or a local directory path (offline mode)

```
# ONLINE MODE
from pytorch_lightning.logging import NeptuneLogger
# arguments made to NeptuneLogger are passed on to the neptune.experiments.
↪Experiment class

neptune_logger = NeptuneLogger(
    api_key=os.environ["NEPTUNE_API_TOKEN"],
    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)
```

```
# OFFLINE MODE
from pytorch_lightning.logging import NeptuneLogger
# arguments made to NeptuneLogger are passed on to the neptune.experiments.
↪Experiment class

neptune_logger = NeptuneLogger(
    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)
```

Parameters

- **api_key** (*str | None*) – Required in online mode. Neputne API token, found on https://neptune.ml. Read how to get your API key https://docs.neptune.ml/python-api/tutorials/get-started.html#copy-api-token.
- **project_name** (*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.ml first.
- **offline_mode** (*bool*) – Optional default False. If offline_mode=True no logs will be send to neptune. Usually used for debug purposes.
- **experiment_name** (*str|None*) – 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** (*list|None*) – 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, Python file from which experiment was created will be uploaded. Pass 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** (*dict|None*) – 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 experiments view as a column.
- **properties** (*dict|None*) – Optional default is {}. Properties of the experiment. They

are editable after experiment is created. Properties are displayed in the experiment's Details and each key-value pair can be viewed in experiments view as a column.

- **tags** (*list|None*) – Optional default []. Must be list of str. Tags of the experiment. They are editable after experiment is created (see: append_tag() and remove_tag()). Tags are displayed in the experiment's Details and can be viewed in experiments view as a column.

append_tags(*tags*)
appends tags to neptune experiment

Parameters **tags** (*str|tuple|list(str)*) – Tags to add to the current experiment. If str is passed, singe 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.

finalize(*status*)
Do any processing that is necessary to finalize an experiment.

Parameters **status** – Status that the experiment finished with (e.g. success, failed, aborted)

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** (*str|None*) – Optional default None. A destination path. If None is passed, an artifact file name will be used.

log_hyperparams(*params*)
Record hyperparameters.

Parameters **params** – argparse.Namespace containing the hyperparameters

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** (*str|PIL.Image|matplotlib.figure.Figure*) – 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** (*int|None*) – Step number at which the metrics should be recorded, must be strictly increasing

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** (*str*) – The value of the log (data-point).
- **step** (*int|None*) – Step number at which the metrics should be recorded, must be strictly increasing

log_metrics(*metrics*, *step=None*)
Log metrics (numeric values) in Neptune experiments

Parameters

- **metric** (*float*) – Dictionary with metric names as keys and measured quanties as values
- **step** (*int|None*) – Step number at which the metrics should be recorded, must be strictly increasing

log_text(*log_name*, *text*, *step=None*)
Log text data in Neptune experiment

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** (*int|None*) – Step number at which the metrics should be recorded, must be strictly increasing

set_property(*key*, *value*)
Set key-value pair as Neptune experiment property.

Parameters

- **key** (*str*) – Property key.
- **value** (*obj*) – New value of a property.

experiment
Actual neptune object. To use neptune features do the following.

Example:

```
self.logger.experiment.some_neptune_function()
```

name
Return the experiment name.

version
Return the experiment version.

class pytorch_lightning.logging.**TensorBoardLogger**(*save_dir*, *name='default'*, *version=None*, ***kwargs*)
Bases: pytorch_lightning.logging.base.LightningLoggerBase

Log to local file system in TensorBoard format

Implemented using torch.utils.tensorboard.SummaryWriter. Logs are saved to *os.path.join(save_dir, name, version)*

Example

```
logger = TensorBoardLogger("tb_logs", name="my_model")
trainer = Trainer(logger=logger)
trainer.train(model)
```

Parameters

- **save_dir** (*str*) – Save directory
- **name** (*str*) – Experiment name. Defaults to "default".
- **version** (*int*) – Experiment version. If version is not specified the logger inspects the save

- **for existing versions, then automatically assigns the next available version.** (*directory*) –
- ****kwargs** (*dict*) – 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** – Status that the experiment finished with (e.g. success, failed, aborted)

log_hyperparams(*params*)
Record hyperparameters.

Parameters **params** – argparse.Namespace containing the hyperparameters

log_metrics(*metrics*, *step=None*)
Record metrics.

Parameters

- **metric** (*float*) – Dictionary with metric names as keys and measured quanties as values
- **step** (*int | None*) – Step number at which the metrics should be recorded

save()
Save log data.

NAME_CSV_TAGS = 'meta_tags.csv'

experiment
Actual tensorboard object. To use tensorboard features do the following.

Example:

```
self.logger.experiment.some_tensorboard_function()
```

name
Return the experiment name.

version
Return the experiment version.

class pytorch_lightning.logging.**TestTubeLogger**(*save_dir*, *name='default'*, *description=None*, *debug=False*, *version=None*, *create_git_tag=False*)
Bases: `pytorch_lightning.logging.base.LightningLoggerBase`

Log to local file system in TensorBoard format but using a nicer folder structure.

Implemented using `torch.utils.tensorboard.SummaryWriter`. Logs are saved to *os.path.join(save_dir, name, version)*

Example

```
logger = TestTubeLogger("tt_logs", name="my_exp_name")
trainer = Trainer(logger=logger)
trainer.train(model)
```

Parameters

- **save_dir** (*str*) – Save directory
- **name** (*str*) – Experiment name. Defaults to "default".
- **description** (*str*) – A short snippet about this experiment
- **debug** (*bool*) – If True, it doesn't log anything
- **version** (*int*) – Experiment version. If version is not specified the logger inspects the save
- **for existing versions, then automatically assigns the next available version.** (*directory*) –
- **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.

finalize(*status*)
Do any processing that is necessary to finalize an experiment.

Parameters **status** – Status that the experiment finished with (e.g. success, failed, aborted)

log_hyperparams(*params*)
Record hyperparameters.

Parameters **params** – argparse.Namespace containing the hyperparameters

log_metrics(*metrics*, *step=None*)
Record metrics.

Parameters

- **metric** (*float*) – Dictionary with metric names as keys and measured quanties as values
- **step** (*int|None*) – Step number at which the metrics should be recorded

save()
Save log data.

experiment
Actual test-tube object. To use test-tube features do the following.

Example:

```
self.logger.experiment.some_test_tube_function()
```

name
Return the experiment name.

rank
Process rank. In general, metrics should only be logged by the process with rank 0.

version
Return the experiment version.

class pytorch_lightning.logging.**WandbLogger**(*name=None*, *save_dir=None*, *offline=False*, *id=None*, *anonymous=False*, *version=None*, *project=None*, *tags=None*, *experiment=None*)

Bases: pytorch_lightning.logging.base.LightningLoggerBase

Logger for W&B.

Parameters

- **name** (*str*) – display name for the run.
- **save_dir** (*str*) – path where data is saved.
- **offline** (*bool*) – run offline (data can be streamed later to wandb servers).
- **or version** (*id*) – sets the version, mainly used to resume a previous run.
- **anonymous** (*bool*) – enables or explicitly disables anonymous logging.
- **project** (*str*) – the name of the project to which this run will belong.
- **tags** (*list of str*) – tags associated with this run.

Example

```
from pytorch_lightning.logging import WandbLogger
from pytorch_lightning import Trainer

wandb_logger = WandbLogger()
trainer = Trainer(logger=wandb_logger)
```

finalize(*status='success'*)

Do any processing that is necessary to finalize an experiment.

Parameters **status** – Status that the experiment finished with (e.g. success, failed, aborted)

log_hyperparams(*params*)

Record hyperparameters.

Parameters **params** – argparse.Namespace containing the hyperparameters

log_metrics(*metrics*, *step=None*)

Record metrics.

Parameters

- **metric** (*float*) – Dictionary with metric names as keys and measured quanties as values
- **step** (*int | None*) – Step number at which the metrics should be recorded

save()

Save log data.

watch(*model*, *log='gradients'*, *log_freq=100*)

experiment

Actual wandb object. To use wandb features do the following.

Example:

```
self.logger.experiment.some_wandb_function()
```

name
Return the experiment name.

version
Return the experiment version.

CHAPTER 5

Trainer

The trainer de-couples the engineering code (16-bit, early stopping, GPU distribution, etc...) from the science code (GAN, BERT, your project, etc...). It uses many assumptions which are best practices in AI research today.

The trainer automates all parts of training except:

- what happens in training , test, val loop
- where the data come from
- which optimizers to use
- how to do the computations

The Trainer delegates those calls to your LightningModule which defines how to do those parts.

This is the basic use of the trainer:

```
from pytorch_lightning import Trainer

model = MyLightningModule()

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

class pytorch_lightning.trainer.**Trainer**(*logger=True*, *checkpoint_callback=True*, *early_stop_callback=True*, *default_save_path=None*, *gradient_clip_val=0*, *gradient_clip=None*, *process_position=0*, *nb_gpu_nodes=None*, *num_nodes=1*, *gpus=None*, *log_gpu_memory=None*, *show_progress_bar=True*, *overfit_pct=0.0*, *track_grad_norm=-1*, *check_val_every_n_epoch=1*, *fast_dev_run=False*, *accumulate_grad_batches=1*, *max_nb_epochs=None*, *min_nb_epochs=None*, *max_epochs=1000*, *min_epochs=1*, *train_percent_check=1.0*, *val_percent_check=1.0*, *test_percent_check=1.0*, *val_check_interval=1.0*, *log_save_interval=100*, *row_log_interval=10*, *add_row_log_interval=None*, *distributed_backend=None*, *use_amp=False*, *print_nan_grads=False*, *weights_summary='full'*, *weights_save_path=None*, *amp_level='O1'*, *nb_sanity_val_steps=None*, *num_sanity_val_steps=5*, *truncated_bptt_steps=None*, *resume_from_checkpoint=None*)

Bases: pytorch_lightning.trainer.training_io.TrainerIOMixin, 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.auto_mix_precision.TrainerAMPMixin, pytorch_lightning.trainer.evaluation_loop.TrainerEvaluationLoopMixin, pytorch_lightning.trainer.training_loop.TrainerTrainLoopMixin, pytorch_lightning.trainer.callback_config.TrainerCallbackConfigMixin

Customize every aspect of training via flags

Parameters

- **logger** (Logger) – Logger for experiment tracking. Example:

```
from pytorch_lightning.logging import TensorBoardLogger

# default logger used by trainer
logger = TensorBoardLogger(
    save_dir=os.getcwd(),
    version=self.slurm_job_id,
    name='lightning_logs'
)

Trainer(logger=logger)
```

- **checkpoint_callback** (CheckpointCallback) – Callback for checkpointing. Example:

```
from pytorch_lightning.callbacks import ModelCheckpoint
```

(continues on next page)

(continued from previous page)

```
# default used by the Trainer
checkpoint_callback = ModelCheckpoint(
    filepath=os.getcwd(),
    save_best_only=True,
    verbose=True,
    monitor='val_loss',
    mode='min',
    prefix=''
)

trainer = Trainer(checkpoint_callback=checkpoint_callback)
```

- **early_stop_callback** (EarlyStopping) – Callback for early stopping Example:

```
from pytorch_lightning.callbacks import EarlyStopping

# default used by the Trainer
early_stop_callback = EarlyStopping(
    monitor='val_loss',
    patience=3,
    verbose=True,
    mode='min'
)

trainer = Trainer(early_stop_callback=early_stop_callback)
```

- **default_save_path** (*str*) – Default path for logs and weights when no logger/ckpt_callback passed Example:

```
# default used by the Trainer
trainer = Trainer(default_save_path=os.getcwd())
```

- **gradient_clip_val** (*float*) – 0 means don't clip. Example:

```
# default used by the Trainer
trainer = Trainer(gradient_clip_val=0.0)
```

- **gradient_clip** (*int*) – Deprecated since version 0.5.0: Use *gradient_clip_val* instead. Will remove 0.8.0.
- **process_position** (*int*) – orders the tqdm bar when running multiple models on same machine. Example:

```
# default used by the Trainer
trainer = Trainer(process_position=0)
```

- **num_nodes** (*int*) – number of GPU nodes for distributed training. Example:

```
# default used by the Trainer
trainer = Trainer(num_nodes=1)

# to train on 8 nodes
trainer = Trainer(num_nodes=8)
```

- **nb_gpu_nodes** (*int*) – Deprecated since version 0.5.0: Use *num_nodes* instead. Will remove 0.8.0.

- **gpus** (*list|str|int*) – Which GPUs to train on. Example:

```
# default used by the Trainer (ie: train on CPU)
trainer = Trainer(gpus=None)

# 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
trainer = Trainer(gpus=2, num_nodes=4) # uses 8 gpus in total
```

- **log_gpu_memory** (*str*) – None, 'min_max', 'all'. Might slow performance because it uses the output of nvidia-smi. Example:

```
# 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')
```

- **show_progress_bar** (*bool*) – If true shows tqdm progress bar Example:

```
# default used by the Trainer
trainer = Trainer(show_progress_bar=True)
```

- **overfit_pct** (*float*) – uses this much data of all datasets. Example:

```
# default used by the Trainer
trainer = Trainer(overfit_pct=0.0)

# use only 1% of the train, test, val datasets
trainer = Trainer(overfit_pct=0.01)
```

- **track_grad_norm** (*int*) – -1 no tracking. Otherwise tracks that norm Example:

```
# default used by the Trainer
trainer = Trainer(track_grad_norm=-1)

# track the 2-norm
trainer = Trainer(track_grad_norm=2)
```

- **check_val_every_n_epoch** (*int*) – check val every n train epochs Example:

```
# default used by the Trainer
trainer = Trainer(check_val_every_n_epoch=1)
```

(continues on next page)

(continued from previous page)

```
# run val loop every 10 training epochs
trainer = Trainer(check_val_every_n_epoch=10)
```

- **fast_dev_run** (*bool*) – runs 1 batch of train, test and val to find any bugs (ie: a sort of unit test). Example:

```
# 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)
```

- **accumulate_grad_batches** (*int | dict*) – Accumulates grads every k batches or as set up in the dict. Example:

```
# default used by the Trainer (no accumulation)
trainer = Trainer(accumulate_grad_batches=1)

# 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})
```

- **max_epochs** (*int*) – Stop training once this number of epochs is reached Example:

```
# default used by the Trainer
trainer = Trainer(max_epochs=1000)
```

- **max_nb_epochs** (*int*) – Deprecated since version 0.5.0: Use *max_epochs* instead. Will remove 0.8.0.
- **min_epochs** (*int*) – Force training for at least these many epochs Example:

```
# default used by the Trainer
trainer = Trainer(min_epochs=1)
```

- **min_nb_epochs** (*int*) – Deprecated since version 0.5.0: Use *min_nb_epochs* instead. Will remove 0.8.0.
- **train_percent_check** (*int*) – How much of training dataset to check. Useful when debugging or testing something that happens at the end of an epoch. Example:

```
# default used by the Trainer
trainer = Trainer(train_percent_check=1.0)

# run through only 25% of the training set each epoch
trainer = Trainer(train_percent_check=0.25)
```

- **val_percent_check** (*int*) – How much of validation dataset to check. Useful when debugging or testing something that happens at the end of an epoch. Example:

```
# default used by the Trainer
trainer = Trainer(val_percent_check=1.0)
```

(continues on next page)

(continued from previous page)

```
# run through only 25% of the validation set each epoch
trainer = Trainer(val_percent_check=0.25)
```

- **test_percent_check** (*int*) – How much of test dataset to check. Useful when debugging or testing something that happens at the end of an epoch. Example:

```
# default used by the Trainer
trainer = Trainer(test_percent_check=1.0)

# run through only 25% of the test set each epoch
trainer = Trainer(test_percent_check=0.25)
```

- **val_check_interval** (*float | int*) – How often within one training epoch to check the validation set If float, % of tng epoch. If int, check every n batch Example:

```
# 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)
```

- **log_save_interval** (*int*) – Writes logs to disk this often Example:

```
# default used by the Trainer
trainer = Trainer(log_save_interval=100)
```

- **row_log_interval** (*int*) – How often to add logging rows (does not write to disk) Example:

```
# default used by the Trainer
trainer = Trainer(row_log_interval=10)
```

- **add_row_log_interval** (*int*) – Deprecated since version 0.5.0: Use *row_log_interval* instead. Will remove 0.8.0.

- **distributed_backend** (*str*) – The distributed backend to use. Options: ‘dp’, ‘ddp’, ‘ddp2’. Example:

```
# default used by the Trainer
trainer = Trainer(distributed_backend=None)

# dp = DataParallel (split a batch onto k gpus on same machine).
trainer = Trainer(gpus=2, distributed_backend='dp')

# ddp = DistributedDataParallel
# Each gpu trains by itself on a subset of the data.
# Gradients sync across all gpus and all machines.
trainer = Trainer(gpus=2, num_nodes=2, distributed_backend='ddp')

# ddp2 = DistributedDataParallel + dp
```

(continues on next page)

(continued from previous page)

```
# behaves like dp on every node
# syncs gradients across nodes like ddp
# useful for things like increasing the number of negative samples
trainer = Trainer(gpus=2, num_nodes=2, distributed_backend='ddp2')
```

- **use_amp** (*bool*) – If true uses apex for 16bit precision Example:

```
# default used by the Trainer
trainer = Trainer(use_amp=False)
```

- **print_nan_grads** (*bool*) – Prints gradients with nan values Example:

```
# default used by the Trainer
trainer = Trainer(print_nan_grads=False)
```

- **weights_summary** (*str*) – Prints a summary of the weights when training begins. Options: 'full', 'top', None. Example:

```
# default used by the Trainer (ie: print all weights)
trainer = Trainer(weights_summary='full')

# print only the top level modules
trainer = Trainer(weights_summary='top')

# don't print a summary
trainer = Trainer(weights_summary=None)
```

- **weights_save_path** (*str*) – Where to save weights if specified. Example:

```
# default used by the Trainer
trainer = Trainer(weights_save_path=os.getcwd())

# save to your custom path
trainer = Trainer(weights_save_path='my/path')

# if checkpoint callback used, then overrides the weights path
# **NOTE: this saves weights to some/path NOT my/path
checkpoint_callback = ModelCheckpoint(filepath='some/path')
trainer = Trainer(
    checkpoint_callback=checkpoint_callback,
    weights_save_path='my/path'
)
```

- **amp_level** (*str*) – The optimization level to use (O1, O2, etc...). Check nvidia docs for level (https://nvidia.github.io/apex/amp.html#opt-levels) Example:

```
# default used by the Trainer
trainer = Trainer(amp_level='O1')
```

- **num_sanity_val_steps** (*int*) – 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. Example:

```
# default used by the Trainer
trainer = Trainer(num_sanity_val_steps=5)
```

(continues on next page)

(continued from previous page)

```
# turn it off
trainer = Trainer(num_sanity_val_steps=0)
```

- **nb_sanity_val_steps** (*int*) – Deprecated since version 0.5.0: Use *num_sanity_val_steps* instead. Will remove 0.8.0.
- **truncated_bptt_steps** (*int*) – 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. Make sure your batches have a sequence dimension. (Williams et al. "An efficient gradient-based algorithm for on-line training of recurrent network trajectories.") Example:

```
# 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)
```

- **resume_from_checkpoint** (*str*) – To resume training from a specific checkpoint pass in the path here.k Example:

```
# 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')
```

fit (*model*)

Runs the full optimization routine.

Example:

```
trainer = Trainer()
model = LightningModule()

trainer.fit()
```

test (*model=None*)

Separates from fit to make sure you never run on your test set until you want to.

Parameters **model** (*LightningModule*) – The model to test.

Example:

```
# Option 1
# run test after fitting
trainer = Trainer()
model = LightningModule()

trainer.fit()
trainer.test()

# Option 2
# run test from a loaded model
model = LightningModule.load_from_checkpoint('path/to/checkpoint.ckpt')
```

(continues on next page)

(continued from previous page)

```
trainer = Trainer()
trainer.test(model)
```

CHAPTER 6

GAN

6.1 pl_examples.domain_templates.gan module

CHAPTER 7

MNIST

7.1 pl_examples.basic_examples.lightning_module_template module

CHAPTER 8

Multi-node (ddp) MNIST

8.1 pl_examples.multi_node_examples.multi_node_ddp_demo module

CHAPTER 9

Multi-node (ddp2) MNIST

9.1 pl_examples.multi_node_examples.multi_node_ddp2_demo module

CHAPTER 10

Imagenet

CHAPTER 11

Refactoring PyTorch into Lightning

Tutorial

CHAPTER 12

Start a research project

Research seed

CHAPTER 13

Basic Lightning use

Tutorial

CHAPTER 14

9 key Lightning tricks

Tutorial

CHAPTER 15

Multi-node training on SLURM

Tutorial

CHAPTER 16

Multi-gpu (same node) training

CHAPTER 17

Multi-node training

CHAPTER 18

16-bit precision

CHAPTER 19

gradient clipping

CHAPTER 20

modifying training via hooks

20.1 pl_examples package

20.1.1 Subpackages

pl_examples.basic_examples package

Submodules

pl_examples.basic_examples.cpu_template module

pl_examples.basic_examples.gpu_template module

pl_examples.domain_templates package

Submodules

pl_examples.multi_node_examples package

Submodules

CHAPTER 21

Contributor Covenant Code of Conduct

21.1 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.

21.2 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

21.3 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.

21.4 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.

21.5 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.

21.6 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 22

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!

22.1 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 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 avoids all sorts of subtle errors the user could make.

22.2 Lightning Design Principles

We encourage all sorts of contributions you're interested in adding! When coding for lightning, please follow these principles.

22.2.1 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.

22.2.2 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 :)

22.2.3 Force User Decisions To Best Practices

There are 1,000 ways to do something. However, something eventually becomes standard practice that everyone does. Thus we pick one way of doing it and force everyone to do it this way. A good example is accumulated gradients. There are many ways to implement, we just pick one and force users to use that one. 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 likely looks like 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.

22.2.4 Simple External API

What makes sense to you may not make sense to others. Create an issue with an API change suggestion and validate that it makes sense for others. Treat code changes how you treat a startup: validate that it's a needed feature, then add if it makes sense for many people.

22.2.5 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 :)

22.2.6 Gain User Trust

As a researcher you can't have any part of your code going wrong. So, make thorough tests that ensure an implementation of a new trick or subbtle change is correct.

22.2.7 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.

22.3 Contribution Types

Currently looking for help implementing new features or adding bug fixes.

A lot of good work has already been done in project mechanics (requirements.txt, setup.py, pep8, badges, ci, etc. . .) we're in a good state there thanks to all the early contributors (even pre-beta release)!

22.4 Bug Fixes:

1. Submit a github issue.
2. Fix it.
3. Submit a PR!

22.5 New Features:

1. Submit a github issue.
2. We'll agree on the feature scope.
3. Submit a PR! (with updated docs and tests).

22.6 Coding Styleguide

1. Test the code with flake8.
2. Use f-strings.

CHAPTER 23

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.

23.1 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.

23.2 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!

23.3 Responsibilities:

The responsibilities mainly revolve around 3 things.

23.3.1 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.

23.3.2 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 :)

23.3.3 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.

23.3.4 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.

23.3.5 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 24

Pytorch Lightning Governance | Persons of interest

24.1 Maintainers

- William Falcon (williamFalcon)
- Jirka Borovek (Borda)
- Nick Eggert (neggert)
- Jeff Ling (jeffling)
- Tullie Murrell (tullie)

CHAPTER 25

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

pytorch_lightning.trainer, 35

Index

Symbols

_get_next_version() (*pytorch_lightning.logging.TensorBoardLogger method*), 30

A

append_tags() (*pytorch_lightning.logging.NeptuneLogger method*), 28

C

close() (*pytorch_lightning.logging.TestTubeLogger method*), 31
CometLogger (*class in pytorch_lightning.logging*), 24
configure_apex() (*pytorch_lightning.core.LightningModule method*), 6
configure_ddp() (*pytorch_lightning.core.LightningModule method*), 7
configure_optimizers() (*pytorch_lightning.core.LightningModule method*), 7
current_epoch (*pytorch_lightning.core.LightningModule attribute*), 21

D

dtype (*pytorch_lightning.core.LightningModule attribute*), 21

E

EarlyStopping (*class in pytorch_lightning.callbacks*), 3
experiment (*pytorch_lightning.logging.CometLogger attribute*), 25
experiment (*pytorch_lightning.logging.MLFlowLogger attribute*), 26
experiment (*pytorch_lightning.logging.NeptuneLogger attribute*), 29
experiment (*pytorch_lightning.logging.TensorBoardLogger attribute*), 30
experiment (*pytorch_lightning.logging.TestTubeLogger attribute*), 31
experiment (*pytorch_lightning.logging.WandbLogger attribute*), 32

F

finalize() (*pytorch_lightning.logging.CometLogger method*), 25
finalize() (*pytorch_lightning.logging.MLFlowLogger method*), 26
finalize() (*pytorch_lightning.logging.NeptuneLogger method*), 28
finalize() (*pytorch_lightning.logging.TensorBoardLogger method*), 30
finalize() (*pytorch_lightning.logging.TestTubeLogger method*), 31
finalize() (*pytorch_lightning.logging.WandbLogger method*), 32
fit() (*pytorch_lightning.trainer.Trainer method*), 42
forward() (*pytorch_lightning.core.LightningModule method*), 8
freeze() (*pytorch_lightning.core.LightningModule method*), 9

G

global_step (*pytorch_lightning.core.LightningModule attribute*), 21
GradientAccumulationScheduler (*class in pytorch_lightning.callbacks*), 4

I

init_ddp_connection() (*pytorch_lightning.core.LightningModule method*), 9

L

LightningModule (*class in pytorch_lightning.core*), 6
load_from_checkpoint() (*pytorch_lightning.core.LightningModule class method*), 10
load_from_metrics() (*pytorch_lightning.core.LightningModule class method*), 11
log_artifact() (*pytorch_lightning.logging.NeptuneLogger method*), 28
log_hyperparams() (*pytorch_lightning.logging.CometLogger method*), 25
log_hyperparams() (*pytorch_lightning.logging.MLFlowLogger method*), 26
log_hyperparams() (*pytorch_lightning.logging.NeptuneLogger method*), 28
log_hyperparams() (*pytorch_lightning.logging.TensorBoardLogger method*), 30
log_hyperparams() (*pytorch_lightning.logging.TestTubeLogger method*), 31
log_hyperparams() (*pytorch_lightning.logging.WandbLogger method*), 32
log_image() (*pytorch_lightning.logging.NeptuneLogger method*), 28
log_metric() (*pytorch_lightning.logging.NeptuneLogger method*), 28
log_metrics() (*pytorch_lightning.logging.CometLogger method*), 25
log_metrics() (*pytorch_lightning.logging.MLFlowLogger method*), 26
log_metrics() (*pytorch_lightning.logging.NeptuneLogger method*), 28
log_metrics() (*pytorch_lightning.logging.TensorBoardLogger method*), 30
log_metrics() (*pytorch_lightning.logging.TestTubeLogger method*), 31
log_metrics() (*pytorch_lightning.logging.WandbLogger method*), 32
log_text() (*pytorch_lightning.logging.NeptuneLogger method*), 29
logger (*pytorch_lightning.core.LightningModule attribute*), 21

M

MLFlowLogger (*class in pytorch_lightning.logging*), 25
ModelCheckpoint (*class in pytorch_lightning.callbacks*), 3

N

name (*pytorch_lightning.logging.CometLogger attribute*), 25
name (*pytorch_lightning.logging.MLFlowLogger attribute*), 26
name (*pytorch_lightning.logging.NeptuneLogger attribute*), 29
name (*pytorch_lightning.logging.TensorBoardLogger attribute*), 30
name (*pytorch_lightning.logging.TestTubeLogger attribute*), 31
name (*pytorch_lightning.logging.WandbLogger attribute*), 32
NAME_CSV_TAGS (*pytorch_lightning.logging.TensorBoardLogger attribute*), 30
NeptuneLogger (*class in pytorch_lightning.logging*), 26

O

on_gpu (*pytorch_lightning.core.LightningModule attribute*), 21
on_load_checkpoint() (*pytorch_lightning.core.LightningModule method*), 12
on_save_checkpoint() (*pytorch_lightning.core.LightningModule method*), 12
optimizer_step() (*pytorch_lightning.core.LightningModule method*), 12

P

pytorch_lightning.callbacks (*module*), 3
pytorch_lightning.core (*module*), 5
pytorch_lightning.logging (*module*), 23
pytorch_lightning.trainer (*module*), 35

R

rank (*pytorch_lightning.logging.TestTubeLogger attribute*), 31
run_id (*pytorch_lightning.logging.MLFlowLogger attribute*), 26

S

save() (*pytorch_lightning.logging.MLFlowLogger method*), 26
save() (*pytorch_lightning.logging.TensorBoardLogger method*), 30
save() (*pytorch_lightning.logging.TestTubeLogger method*), 31
save() (*pytorch_lightning.logging.WandbLogger method*), 32
set_property() (*pytorch_lightning.logging.NeptuneLogger method*), 29

T

tbptt_split_batch() (*pytorch_lightning.core.LightningModule method*), 13
TensorBoardLogger (*class in pytorch_lightning.logging*), 29
test() (*pytorch_lightning.trainer.Trainer method*), 42
test_dataloader() (*pytorch_lightning.core.LightningModule method*), 14
test_end() (*pytorch_lightning.core.LightningModule method*), 14
test_step() (*pytorch_lightning.core.LightningModule method*), 15
TestTubeLogger (*class in pytorch_lightning.logging*), 30
tng_dataloader() (*pytorch_lightning.core.LightningModule method*), 16
train_dataloader() (*pytorch_lightning.core.LightningModule method*), 16
Trainer (*class in pytorch_lightning.trainer*), 35
trainer (*pytorch_lightning.core.LightningModule attribute*), 21
training_end() (*pytorch_lightning.core.LightningModule method*), 16
training_step() (*pytorch_lightning.core.LightningModule method*), 17

U

unfreeze() (*pytorch_lightning.core.LightningModule method*), 18
use_amp (*pytorch_lightning.core.LightningModule attribute*), 21
use_ddp (*pytorch_lightning.core.LightningModule attribute*), 21
use_ddp2 (*pytorch_lightning.core.LightningModule attribute*), 21
use_dp (*pytorch_lightning.core.LightningModule attribute*), 21

V

val_dataloader() (*pytorch_lightning.core.LightningModule method*), 19
validation_end() (*pytorch_lightning.core.LightningModule method*), 19
validation_step() (*pytorch_lightning.core.LightningModule method*), 20
version (*pytorch_lightning.logging.CometLogger attribute*), 25
version (*pytorch_lightning.logging.MLFlowLogger attribute*), 26
version (*pytorch_lightning.logging.NeptuneLogger attribute*), 29
version (*pytorch_lightning.logging.TensorBoardLogger attribute*), 30
version (*pytorch_lightning.logging.TestTubeLogger attribute*), 31
version (*pytorch_lightning.logging.WandbLogger attribute*), 33

W

WandbLogger (*class in pytorch_lightning.logging*), 31
watch() (*pytorch_lightning.logging.WandbLogger method*), 32