PyTorch Lightning Documentation

Release 1.2.1

William Falcon et al.

Feb 24, 2021

GETTING STARTED

CHAPTER

ONE

LIGHTNING IN 2 STEPS

In this guide we'll show you how to organize your PyTorch code into Lightning in 2 steps.

Organizing your code with PyTorch Lightning makes your code:

- Keep all the flexibility (this is all pure PyTorch), but removes a ton of boilerplate
- More readable by decoupling the research code from the engineering
- Easier to reproduce
- Less error-prone by automating most of the training loop and tricky engineering
- Scalable to any hardware without changing your model

Here's a 3 minute conversion guide for PyTorch projects:

1.1 Step 0: Install PyTorch Lightning

You can install using pip

```
pip install pytorch-lightning
```

Or with conda (see how to install conda here):

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

You could also use conda environments

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

Import the following:

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

1.2 Step 1: Define LightningModule

```
class LitAutoEncoder(LightningModule):

    def __init__(self):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Linear(28*28, 64),
            nn.ReLU(),
            nn.Linear(64, 3)
        )
        self.decoder = nn.Sequential(
            nn.Linear(3, 64),
            nn.ReLU(),
            nn.Linear(64, 28*28)
        )

    def forward(self, x):
        # in lightning, forward defines the prediction/inference actions
        embedding = self.encoder(x)
        return embedding

    def training_step(self, batch, batch_idx):
        # training_step defined the train loop.
        # It is independent of forward
        x, y = batch
        x = x.view(x.size(0), -1)
        z = self.encoder(x)
        x_hat = self.decoder(z)
        loss = F.mse_loss(x_hat, x)
        # Logging to TensorBoard by default
        self.log('train_loss', loss)
        return loss

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

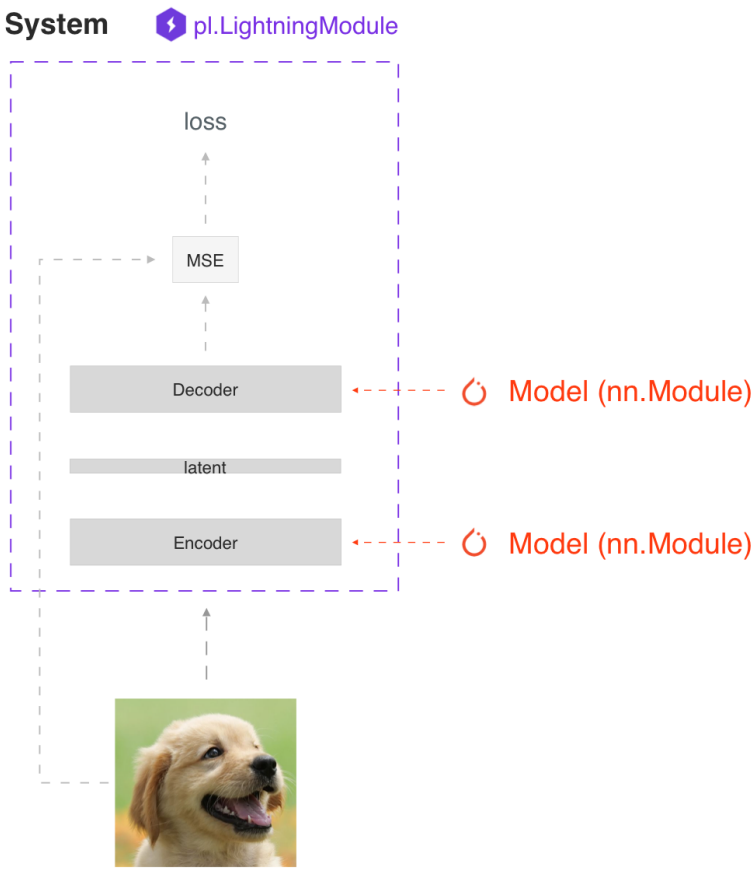
SYSTEM VS MODEL

A *lightning module* defines a *system* not a model.

Examples of systems are:

- Autoencoder
- BERT
- DQN
- GAN
- Image classifier
- Seq2seq
- SimCLR
- VAE

Under the hood a LightningModule is still just a `torch.nn.Module` that groups all research code into a single file to make it self-contained:


System
pl.LightningModule
loss
MSE
Decoder
Model (nn.Module)
latent
Encoder
Model (nn.Module)

- The Train loop
- The Validation loop
- The Test loop
- The Model or system of Models
- The Optimizer

You can customize any part of training (such as the backward pass) by overriding any of the 20+ hooks found in *Available Callback hooks*

```
class LitAutoEncoder(LightningModule):

    def backward(self, loss, optimizer, optimizer_idx):
        loss.backward()
```

FORWARD vs TRAINING_STEP

In Lightning we separate training from inference. The training_step defines the full training loop. We encourage users to use the forward to define inference actions.

For example, in this case we could define the autoencoder to act as an embedding extractor:

```
def forward(self, x):
    embeddings = self.encoder(x)
    return embeddings
```

Of course, nothing is stopping you from using forward from within the training_step.

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

It really comes down to your application. We do, however, recommend that you keep both intents separate.

- Use forward for inference (predicting).
- Use training_step for training.

More details in *lightning module* docs.

1.3 Step 2: Fit with Lightning Trainer

First, define the data however you want. Lightning just needs a `DataLoader` for the train/val/test splits.

```
dataset = MNIST(os.getcwd(), download=True, transform=transforms.ToTensor())
train_loader = DataLoader(dataset)
```

Next, init the *lightning module* and the PyTorch Lightning `Trainer`, then call fit with both the data and model.

```
# init model
autoencoder = LitAutoEncoder()

# most basic trainer, uses good defaults (auto-tensorboard, checkpoints, logs, and
↪more)
```

(continues on next page)

(continued from previous page)

```
# trainer = pl.Trainer(gpus=8) (if you have GPUs)
trainer = pl.Trainer()
trainer.fit(autoencoder, train_loader)
```

The `Trainer` automates:

- Epoch and batch iteration
- Calling of optimizer.step(), backward, zero_grad()
- Calling of .eval(), enabling/disabling grads
- *weights loading*
- Tensorboard (see *loggers* options)
- *Multi-GPU* support
- *TPU*
- *AMP* support

Tip: If you prefer to manually manage optimizers you can use the *Manual optimization* mode (ie: RL, GANs, etc...).

That's it!

These are the main 2 concepts you need to know in Lightning. All the other features of lightning are either features of the Trainer or LightningModule.

1.4 Basic features

1.4.1 Manual vs automatic optimization

Automatic optimization

With Lightning, you don't need to worry about when to enable/disable grads, do a backward pass, or update optimizers as long as you return a loss with an attached graph from the *training_step*, Lightning will automate the optimization.

```
def training_step(self, batch, batch_idx):
    loss = self.encoder(batch[0])
    return loss
```

Manual optimization

However, for certain research like GANs, reinforcement learning, or something with multiple optimizers or an inner loop, you can turn off automatic optimization and fully control the training loop yourself.

Turn off automatic optimization and you control the train loop!

```
def __init__(self):
    self.automatic_optimization = False

def training_step(self, batch, batch_idx, optimizer_idx):
    # access your optimizers with use_pl_optimizer=False. Default is True
    (opt_a, opt_b, opt_c) = self.optimizers(use_pl_optimizer=True)

    loss_a = self.generator(batch[0])

    # use this instead of loss.backward so we can automate half precision, etc...
    self.manual_backward(loss_a, opt_a, retain_graph=True)
    self.manual_backward(loss_a, opt_a)
    opt_a.step()
    opt_a.zero_grad()

    loss_b = self.discriminator(batch[0])
    self.manual_backward(loss_b, opt_b)
    ...
```

1.4.2 Predict or Deploy

When you're done training, you have 3 options to use your LightningModule for predictions.

Option 1: Sub-models

Pull out any model inside your system for predictions.

```
# ----------------------------------
# to use as embedding extractor
# ----------------------------------
autoencoder = LitAutoEncoder.load_from_checkpoint('path/to/checkpoint_file.ckpt')
encoder_model = autoencoder.encoder
encoder_model.eval()

# ----------------------------------
# to use as image generator
# ----------------------------------
decoder_model = autoencoder.decoder
decoder_model.eval()
```

Option 2: Forward

You can also add a forward method to do predictions however you want.

```
# ----------------------------------
# using the AE to extract embeddings
# ----------------------------------
class LitAutoEncoder(LightningModule):
    def __init__(self):
        super().__init__()
        self.encoder = nn.Sequential()

    def forward(self, x):
        embedding = self.encoder(x)
        return embedding

autoencoder = LitAutoEncoder()
autoencoder = autoencoder(torch.rand(1, 28 * 28))
```

```
# ----------------------------------
# or using the AE to generate images
# ----------------------------------
class LitAutoEncoder(LightningModule):
    def __init__(self):
        super().__init__()
        self.decoder = nn.Sequential()

    def forward(self):
        z = torch.rand(1, 3)
        image = self.decoder(z)
        image = image.view(1, 1, 28, 28)
        return image

autoencoder = LitAutoEncoder()
image_sample = autoencoder()
```

Option 3: Production

For production systems, onnx or torchscript are much faster. Make sure you have added a forward method or trace only the sub-models you need.

```
# ----------------------------------
# torchscript
# ----------------------------------
autoencoder = LitAutoEncoder()
torch.jit.save(autoencoder.to_torchscript(), "model.pt")
os.path.isfile("model.pt")
```

```
# ----------------------------------
# onnx
# ----------------------------------
with tempfile.NamedTemporaryFile(suffix='.onnx', delete=False) as tmpfile:
     autoencoder = LitAutoEncoder()
     input_sample = torch.randn((1, 28 * 28))
     autoencoder.to_onnx(tmpfile.name, input_sample, export_params=True)
     os.path.isfile(tmpfile.name)
```

1.4.3 Using CPUs/GPUs/TPUs

It's trivial to use CPUs, GPUs or TPUs in Lightning. There's **NO NEED** to change your code, simply change the `Trainer` options.

```
# train on CPU
trainer = Trainer()
```

```
# train on 8 CPUs
trainer = Trainer(num_processes=8)
```

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

```
# train on 1 GPU
trainer = pl.Trainer(gpus=1)
```

```
# train on multiple GPUs across nodes (32 gpus here)
trainer = pl.Trainer(
    gpus=4,
    num_nodes=8
)
```

```
# train on gpu 1, 3, 5 (3 gpus total)
trainer = pl.Trainer(gpus=[1, 3, 5])
```

```
# Multi GPU with mixed precision
trainer = pl.Trainer(gpus=2, precision=16)
```

```
# Train on TPUs
trainer = pl.Trainer(tpu_cores=8)
```

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
# using only half the training data and checking validation every quarter of a␣
↪training epoch
trainer = pl.Trainer(
    tpu_cores=8,
    precision=16,
    limit_train_batches=0.5,
    val_check_interval=0.25
)
```

1.4.4 Checkpoints

Lightning automatically saves your model. Once you've trained, you can load the checkpoints as follows:

```
model = LitModel.load_from_checkpoint(path)
```

The above checkpoint contains all the arguments needed to init the model and set the state dict. If you prefer to do it manually, here's the equivalent

```
# load the ckpt
ckpt = torch.load('path/to/checkpoint.ckpt')

# equivalent to the above
model = LitModel()
model.load_state_dict(ckpt['state_dict'])
```

1.4.5 Data flow

Each loop (training, validation, test) has three hooks you can implement:

- x_step
- x_step_end
- x_epoch_end

To illustrate how data flows, we'll use the training loop (ie: x=training)

```
outs = []
for batch in data:
    out = training_step(batch)
    outs.append(out)
training_epoch_end(outs)
```

The equivalent in Lightning is:

```
def training_step(self, batch, batch_idx):
    prediction = ...
    return prediction

def training_epoch_end(self, training_step_outputs):
    for prediction in predictions:
        # do something with these
```

In the event that you use DP or DDP2 distributed modes (ie: split a batch across GPUs), use the x_step_end to manually aggregate (or don't implement it to let lightning auto-aggregate for you).

```
for batch in data:
    model_copies = copy_model_per_gpu(model, num_gpus)
    batch_split = split_batch_per_gpu(batch, num_gpus)

    gpu_outs = []
    for model, batch_part in zip(model_copies, batch_split):
        # LightningModule hook
        gpu_out = model.training_step(batch_part)
```

(continues on next page)

(continued from previous page)

```
        gpu_outs.append(gpu_out)

    # LightningModule hook
    out = training_step_end(gpu_outs)
```

The lightning equivalent is:

```
def training_step(self, batch, batch_idx):
    loss = ...
    return loss

def training_step_end(self, losses):
    gpu_0_loss = losses[0]
    gpu_1_loss = losses[1]
    return (gpu_0_loss + gpu_1_loss) * 1/2
```

Tip: The validation and test loops have the same structure.

1.4.6 Logging

To log to Tensorboard, your favorite logger, and/or the progress bar, use the *log()* method which can be called from any method in the LightningModule.

```
def training_step(self, batch, batch_idx):
    self.log('my_metric', x)
```

The *log()* method has a few options:

- on_step (logs the metric at that step in training)
- on_epoch (automatically accumulates and logs at the end of the epoch)
- prog_bar (logs to the progress bar)
- logger (logs to the logger like Tensorboard)

Depending on where the log is called from, Lightning auto-determines the correct mode for you. But of course you can override the default behavior by manually setting the flags

Note: Setting on_epoch=True will accumulate your logged values over the full training epoch.

```
def training_step(self, batch, batch_idx):
    self.log('my_loss', loss, on_step=True, on_epoch=True, prog_bar=True, logger=True)
```

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

You can also use any method of your logger directly:

```
def training_step(self, batch, batch_idx):
    tensorboard = self.logger.experiment
    tensorboard.any_summary_writer_method_you_want())
```

Once your training starts, you can view the logs by using your favorite logger or booting up the Tensorboard logs:

```
tensorboard --logdir ./lightning_logs
```

Note: Lightning automatically shows the loss value returned from `training_step` in the progress bar. So, no need to explicitly log like this `self.log('loss', loss, prog_bar=True)`.

Read more about *loggers*.

1.4.7 Optional extensions

Callbacks

A callback is an arbitrary self-contained program that can be executed at arbitrary parts of the training loop.

Here's an example adding a not-so-fancy learning rate decay rule:

```
from pytorch_lightning.callbacks import Callback

class DecayLearningRate(Callback):

    def __init__(self):
        self.old_lrs = []

    def on_train_start(self, trainer, pl_module):
        # track the initial learning rates
        for opt_idx, optimizer in enumerate(trainer.optimizers):
            group = [param_group['lr'] for param_group in optimizer.param_groups]
            self.old_lrs.append(group)

    def on_train_epoch_end(self, trainer, pl_module, outputs):
        for opt_idx, optimizer in enumerate(trainer.optimizers):
            old_lr_group = self.old_lrs[opt_idx]
            new_lr_group = []
            for p_idx, param_group in enumerate(optimizer.param_groups):
                old_lr = old_lr_group[p_idx]
                new_lr = old_lr * 0.98
                new_lr_group.append(new_lr)
                param_group['lr'] = new_lr
            self.old_lrs[opt_idx] = new_lr_group

# And pass the callback to the Trainer
decay_callback = DecayLearningRate()
trainer = Trainer(callbacks=[decay_callback])
```

Things you can do with a callback:

- Send emails at some point in training
- Grow the model

- Update learning rates
- Visualize gradients
- ...
- You are only limited by your imagination

Learn more about custom callbacks.

LightningDataModules

DataLoaders and data processing code tends to end up scattered around. Make your data code reusable by organizing it into a *LightningDataModule*.

```
class MNISTDataModule(LightningDataModule):

      def __init__(self, batch_size=32):
          super().__init__()
          self.batch_size = batch_size

      # When doing distributed training, Datamodules have two optional arguments for
      # granular control over download/prepare/splitting data:

      # OPTIONAL, called only on 1 GPU/machine
      def prepare_data(self):
          MNIST(os.getcwd(), train=True, download=True)
          MNIST(os.getcwd(), train=False, download=True)

      # OPTIONAL, called for every GPU/machine (assigning state is OK)
      def setup(self, stage):
          # transforms
          transform=transforms.Compose([
              transforms.ToTensor(),
              transforms.Normalize((0.1307,), (0.3081,))
          ])
          # split dataset
          if stage == 'fit':
              mnist_train = MNIST(os.getcwd(), train=True, transform=transform)
              self.mnist_train, self.mnist_val = random_split(mnist_train, [55000,␣
↪5000])
          if stage == 'test':
              self.mnist_test = MNIST(os.getcwd(), train=False, transform=transform)

      # return the dataloader for each split
      def train_dataloader(self):
          mnist_train = DataLoader(self.mnist_train, batch_size=self.batch_size)
          return mnist_train

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

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

LightningDataModule is designed to enable sharing and reusing data splits and transforms across different projects. It encapsulates all the steps needed to process data: downloading, tokenizing, processing etc.

Now you can simply pass your *LightningDataModule* to the `Trainer`:

```
# init model
model = LitModel()

# init data
dm = MNISTDataModule()

# train
trainer = pl.Trainer()
trainer.fit(model, dm)

# test
trainer.test(datamodule=dm)
```

DataModules are specifically useful for building models based on data. Read more on *datamodules*.

1.4.8 Debugging

Lightning has many tools for debugging. Here is an example of just a few of them:

```
# use only 10 train batches and 3 val batches
trainer = Trainer(limit_train_batches=10, limit_val_batches=3)
```

```
# Automatically overfit the sane batch of your model for a sanity test
trainer = Trainer(overfit_batches=1)
```

```
# unit test all the code- hits every line of your code once to see if you have bugs,
# instead of waiting hours to crash on validation
trainer = Trainer(fast_dev_run=True)
```

```
# train only 20% of an epoch
trainer = Trainer(limit_train_batches=0.2)
```

```
# run validation every 25% of a training epoch
trainer = Trainer(val_check_interval=0.25)
```

```
# Profile your code to find speed/memory bottlenecks
Trainer(profiler=True)
```

1.5 Other coool features

Once you define and train your first Lightning model, you might want to try other cool features like

- *Automatic early stopping*
- *Automatic truncated-back-propagation-through-time*
- *Automatically scale your batch size*
- *Automatically find a good learning rate*

- *Load checkpoints directly from S3*
- *Scale to massive compute clusters*
- *Use multiple dataloaders per train/val/test loop*
- *Use multiple optimizers to do reinforcement learning or even GANs*

Or read our *Guide* to learn more!

1.5.1 Grid AI

Grid AI is our native solution for large scale training and tuning on the cloud provider of your choice.

Click here to request early-access.

1.6 Community

Our community of core maintainers and thousands of expert researchers is active on our Slack and Forum. Drop by to hang out, ask Lightning questions or even discuss research!

1.6.1 Masterclass

We also offer a Masterclass to teach you the advanced uses of Lightning.

CHAPTER

TWO

HOW TO ORGANIZE PYTORCH INTO LIGHTNING

To enable your code to work with Lightning, here's how to organize PyTorch into Lightning

---

2.1 1. Move your computational code

Move the model architecture and forward pass to your *lightning module*.

```
class LitModel(LightningModule):

    def __init__(self):
        super().__init__()
        self.layer_1 = nn.Linear(28 * 28, 128)
        self.layer_2 = 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
```

2.2 2. Move the optimizer(s) and schedulers

Move your optimizers to the `configure_optimizers()` hook.

```
class LitModel(LightningModule):

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

2.3 3. Find the train loop "meat"

Lightning automates most of the training for you, the epoch and batch iterations, all you need to keep is the training step logic. This should go into the `training_step()` hook (make sure to use the hook parameters, `batch` and `batch_idx` in this case):

```
class LitModel(LightningModule):

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

2.4 4. Find the val loop "meat"

To add an (optional) validation loop add logic to the `validation_step()` hook (make sure to use the hook parameters, `batch` and `batch_idx` in this case).

```
class LitModel(LightningModule):

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

Note: `model.eval()` and `torch.no_grad()` are called automatically for validation

2.5 5. Find the test loop "meat"

To add an (optional) test loop add logic to the `test_step()` hook (make sure to use the hook parameters, `batch` and `batch_idx` in this case).

```
class LitModel(LightningModule):

    def test_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self(x)
        loss = F.cross_entropy(y_hat, y)
        return loss
```

Note: `model.eval()` and `torch.no_grad()` are called automatically for testing.

The test loop will not be used until you call.

```
trainer.test()
```

Tip: `.test()` loads the best checkpoint automatically

2.6 6. Remove any .cuda() or to.device() calls

Your *lightning module* can automatically run on any hardware!

CHAPTER

THREE

RAPID PROTOTYPING TEMPLATES

Use these templates for rapid prototyping

3.1 General Use

Use case	Description	link
Scratch model	To prototype quickly / debug with random data	
Scratch model with manual optimization	To prototype quickly / debug with random data	

CHAPTER

FOUR

STYLE GUIDE

A main goal of Lightning is to improve readability and reproducibility. Imagine looking into any GitHub repo, finding a lightning module and knowing exactly where to look to find the things you care about.

The goal of this style guide is to encourage Lightning code to be structured similarly.

4.1 LightningModule

These are best practices about structuring your LightningModule

4.1.1 Systems vs models

The main principle behind a LightningModule is that a full system should be self-contained. In Lightning we differentiate between a system and a model.

A model is something like a resnet18, RNN, etc.

A system defines how a collection of models interact with each other. Examples of this are:

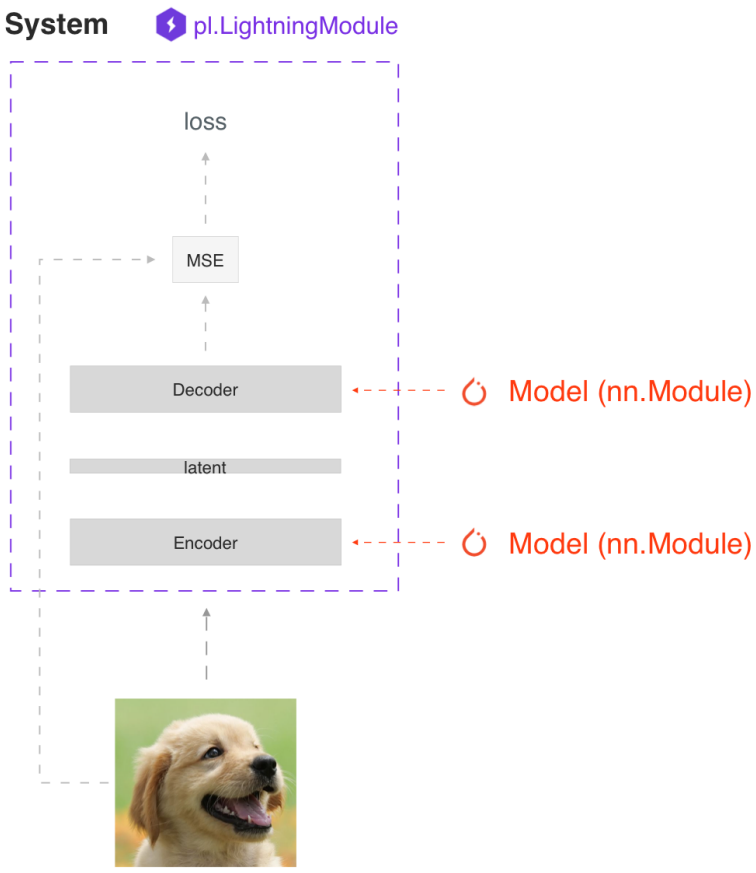
- GANs
- Seq2Seq
- BERT
- etc

A LightningModule can define both a system and a model.

Here's a LightningModule that defines a model:

```
class LitModel(LightningModule):
    def __init__(self, num_layers: int = 3):
        super().__init__()
        self.layer_1 = nn.Linear()
        self.layer_2 = nn.Linear()
        self.layer_3 = nn.Linear()
```

Here's a LightningModule that defines a system:


System
pl.LightningModule
loss
MSE
Decoder
Model (nn.Module)
latent
Encoder
Model (nn.Module)

```
class LitModel(LightningModule):
    def __init__(self, encoder: nn.Module = None, decoder: nn.Module = None):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
```

For fast prototyping it's often useful to define all the computations in a LightningModule. For reusability and scalability it might be better to pass in the relevant backbones.

4.1.2 Self-contained

A Lightning module should be self-contained. A good test to see how self-contained your model is, is to ask yourself this question:

"Can someone drop this file into a Trainer without knowing anything about the internals?"

For example, we couple the optimizer with a model because the majority of models require a specific optimizer with a specific learning rate scheduler to work well.

4.1.3 Init

The first place where LightningModules tend to stop being self-contained is in the init. Try to define all the relevant sensible defaults in the init so that the user doesn't have to guess.

Here's an example where a user will have to go hunt through files to figure out how to init this LightningModule.

```
class LitModel(LightningModule):
    def __init__(self, params):
        self.lr = params.lr
        self.coef_x = params.coef_x
```

Models defined as such leave you with many questions; what is coef_x? is it a string? a float? what is the range? etc...

Instead, be explicit in your init

```
class LitModel(LightningModule):
    def __init__(self, encoder: nn.Module, coeff_x: float = 0.2, lr: float = 1e-3):
        ...
```

Now the user doesn't have to guess. Instead they know the value type and the model has a sensible default where the user can see the value immediately.

4.1.4 Method order

The only required methods in the LightningModule are:

- init
- training_step
- configure_optimizers

However, if you decide to implement the rest of the optional methods, the recommended order is:

- model/system definition (init)

- if doing inference, define forward
- training hooks
- validation hooks
- test hooks
- configure_optimizers
- any other hooks

In practice, this code looks like:

```
class LitModel(pl.LightningModule):

    def __init__(...):

    def forward(...):

    def training_step(...)

    def training_step_end(...)

    def training_epoch_end(...)

    def validation_step(...)

    def validation_step_end(...)

    def validation_epoch_end(...)

    def test_step(...)

    def test_step_end(...)

    def test_epoch_end(...)

    def configure_optimizers(...)

    def any_extra_hook(...)
```

4.1.5 Forward vs training_step

We recommend using forward for inference/predictions and keeping training_step independent

```
def forward(...):
    embeddings = self.encoder(x)

def training_step(...):
    x, y = ...
    z = self.encoder(x)
    pred = self.decoder(z)
    ...
```

However, when using DataParallel, you will need to call forward manually

```
def training_step(...):
    x, y = ...
    z = self(x)  # < ---------- instead of self.encoder(x)
    pred = self.decoder(z)
    ...
```

4.2 Data

These are best practices for handling data.

4.2.1 Dataloaders

Lightning uses dataloaders to handle all the data flow through the system. Whenever you structure dataloaders, make sure to tune the number of workers for maximum efficiency.

Warning: Make sure not to use ddp_spawn with num_workers > 0 or you will bottleneck your code.

4.2.2 DataModules

Lightning introduced datamodules. The problem with dataloaders is that sharing full datasets is often still challenging because all these questions need to be answered:

- What splits were used?
- How many samples does this dataset have?
- What transforms were used?
- etc...

It's for this reason that we recommend you use datamodules. This is specially important when collaborating because it will save your team a lot of time as well.

All they need to do is drop a datamodule into a lightning trainer and not worry about what was done to the data.

This is true for both academic and corporate settings where data cleaning and ad-hoc instructions slow down the progress of iterating through ideas.

CHAPTER

FIVE

FAST PERFORMANCE TIPS

Lightning builds in all the micro-optimizations we can find to increase your performance. But we can only automate so much.

Here are some additional things you can do to increase your performance.

5.1 Dataloaders

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

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

5.1.1 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 `num_workers` slowly and stop once you see no more improvement in your training speed.

5.1.2 Spawn

When using `accelerator=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 `accelerator=ddp` so you can increase the `num_workers`, however your script has to be callable like so:

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

5.2 .item(), .numpy(), .cpu()

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

5.3 empty_cache()

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

5.4 Construct tensors directly on the device

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

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

# good (self is LightningModule)
t = torch.rand(2, 2, device=self.device)
```

For tensors that need to be model attributes, it is best practice to register them as buffers in the modules's `__init__` method:

```
# bad
self.t = torch.rand(2, 2, device=self.device)

# good
self.register_buffer("t", torch.rand(2, 2))
```

5.5 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.

5.6 16-bit precision

Use 16-bit to decrease the memory consumption (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
```

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

5.7 Use Sharded DDP for GPU memory and scaling optimization

Sharded DDP is a lightning integration of DeepSpeed ZeRO and ZeRO-2 provided by Fairscale.

When training on multiple GPUs sharded DDP can assist to increase memory efficiency substantially, and in some cases performance on multi-node is better than traditional DDP. This is due to efficient communication and parallelization under the hood.

To use Optimizer Sharded Training, refer to *Model Parallelism [BETA]*.

Sharded DDP can work across all DDP variants by adding the additional `--plugins ddp_sharded` flag.

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

5.8 Sequential Model Parallelism with Checkpointing

PyTorch Lightning integration for Sequential Model Parallelism using FairScale. Sequential Model Parallelism splits a sequential module onto multiple GPUs, reducing peak GPU memory requirements substantially.

For more information, refer to *Sequential Model Parallelism with Checkpointing*.

5.9 Preload Data Into RAM

When your training or preprocessing requires many operations to be performed on entire dataset(s) it can sometimes be beneficial to store all data in RAM given there is enough space. However, loading all data at the beginning of the training script has the disadvantage that it can take a long time and hence it slows down the development process. Another downside is that in multiprocessing (e.g. DDP) the data would get copied in each process. One can overcome these problems by copying the data into RAM in advance. Most UNIX-based operating systems provide direct access to tmpfs through a mount point typically named `/dev/shm`.

0. Increase shared memory if necessary. Refer to the documentation of your OS how to do this.
1. Copy training data to shared memory:

```
cp -r /path/to/data/on/disk /dev/shm/
```

2. Refer to the new data root in your script or command line arguments:

```
datamodule = MyDataModule(data_root="/dev/shm/my_data")
```

CHAPTER

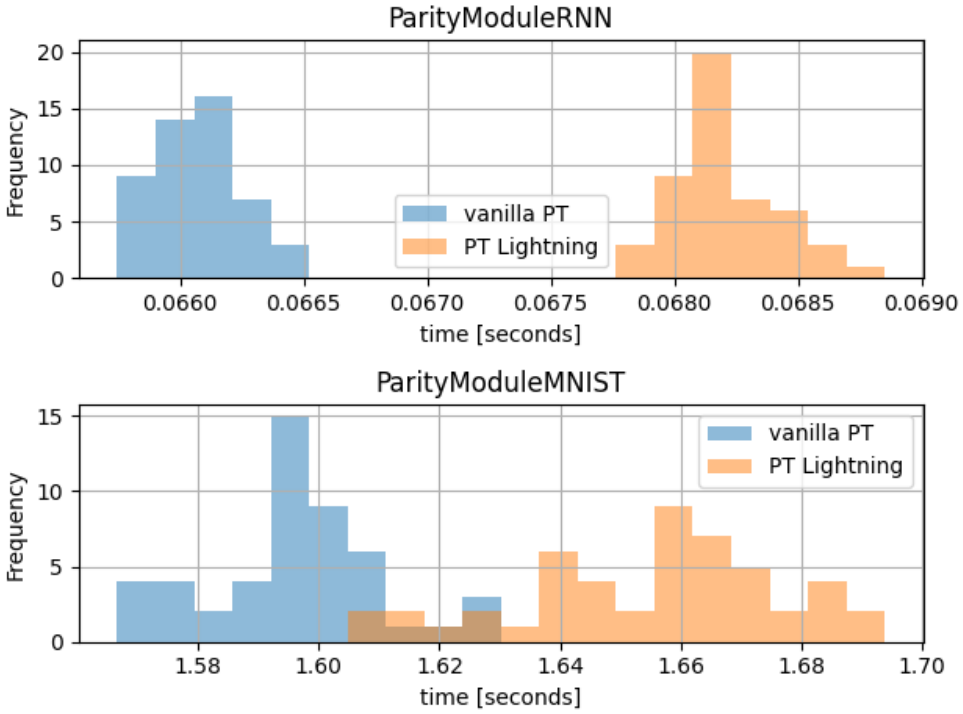
SIX

BENCHMARK WITH VANILLA PYTORCH

In this section we set grounds for comparison between vanilla PyTorch and PT Lightning for most common scenarios.

6.1 Time comparison

We have set regular benchmarking against PyTorch vanilla training loop on with RNN and simple MNIST classifier as per of out CI. In average for simple MNIST CNN classifier we are only about 0.06s slower per epoch, see detail chart bellow.



CHAPTER

SEVEN

LIGHTNINGMODULE

A `LightningModule` organizes your PyTorch code into 5 sections

- Computations (init).
- Train loop (training_step)
- Validation loop (validation_step)
- Test loop (test_step)
- Optimizers (configure_optimizers)

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)
```

(continues on next page)

(continued from previous page)

```
# 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)
```

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).

7.1 Minimal Example

Here are the only required methods.

```
>>> import pytorch_lightning as pl
>>> class LitModel(pl.LightningModule):
...
...     def __init__(self):
...         super().__init__()
...         self.l1 = 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
```

(continues on next page)

(continued from previous page)

```
...         y_hat = self(x)
...         loss = F.cross_entropy(y_hat, y)
...         return loss
...
...     def configure_optimizers(self):
...         return torch.optim.Adam(self.parameters(), lr=0.02)
```

Which you can train by doing:

```
train_loader = DataLoader(MNIST(os.getcwd(), download=True, transform=transforms.
↪ToTensor()))
trainer = pl.Trainer()
model = LitModel()

trainer.fit(model, train_loader)
```

The LightningModule has many convenience methods, but the core ones you need to know about are:

Name	Description
init	Define computations here
forward	Use for inference only (separate from training_step)
training_step	the full training loop
validation_step	the full validation loop
test_step	the full test loop
configure_optimizers	define optimizers and LR schedulers

7.2 Training

7.2.1 Training loop

To add a training loop use the *training_step* method

```
class LitClassifier(pl.LightningModule):

    def __init__(self, model):
        super().__init__()
        self.model = model

    def training_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self.model(x)
        loss = F.cross_entropy(y_hat, y)
        return loss
```

Under the hood, Lightning does the following (pseudocode):

```
# put model in train mode
model.train()
torch.set_grad_enabled(True)
```

(continues on next page)

(continued from previous page)

```
losses = []
for batch in train_dataloader:
    # forward
    loss = training_step(batch)
    losses.append(loss.detach())

    # backward
    loss.backward()

    # apply and clear grads
    optimizer.step()
    optimizer.zero_grad()
```

Training epoch-level metrics

If you want to calculate epoch-level metrics and log them, use the *.log* method

```
def training_step(self, batch, batch_idx):
    x, y = batch
    y_hat = self.model(x)
    loss = F.cross_entropy(y_hat, y)

    # logs metrics for each training_step,
    # and the average across the epoch, to the progress bar and logger
    self.log('train_loss', loss, on_step=True, on_epoch=True, prog_bar=True,␣
↪logger=True)
    return loss
```

The *.log* object automatically reduces the requested metrics across the full epoch. Here's the pseudocode of what it does under the hood:

```
outs = []
for batch in train_dataloader:
    # forward
    out = training_step(val_batch)

    # backward
    loss.backward()

    # apply and clear grads
    optimizer.step()
    optimizer.zero_grad()

epoch_metric = torch.mean(torch.stack([x['train_loss'] for x in outs]))
```

Train epoch-level operations

If you need to do something with all the outputs of each *training_step*, override *training_epoch_end* yourself.

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

def training_epoch_end(self, training_step_outputs):
   for pred in training_step_outputs:
       # do something
```

The matching pseudocode is:

```
outs = []
for batch in train_dataloader:
    # forward
    out = training_step(val_batch)

    # backward
    loss.backward()

    # apply and clear grads
    optimizer.step()
    optimizer.zero_grad()

training_epoch_end(outs)
```

Training with DataParallel

When training using a *accelerator* that splits data from each batch across GPUs, sometimes you might need to aggregate them on the master GPU for processing (dp, or ddp2).

In this case, implement the *training_step_end* method

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

def training_step_end(self, batch_parts):
    gpu_0_prediction = batch_parts[0]['pred']
    gpu_1_prediction = batch_parts[1]['pred']

    # do something with both outputs
    return (batch_parts[0]['loss'] + batch_parts[1]['loss']) / 2

def training_epoch_end(self, training_step_outputs):
   for out in training_step_outputs:
       # do something with preds
```

The full pseudocode that lighting does under the hood is:

```
outs = []
for train_batch in train_dataloader:
    batches = split_batch(train_batch)
    dp_outs = []
    for sub_batch in batches:
        # 1
        dp_out = training_step(sub_batch)
        dp_outs.append(dp_out)

    # 2
    out = training_step_end(dp_outs)
    outs.append(out)

# do something with the outputs for all batches
# 3
training_epoch_end(outs)
```

7.2.2 Validation loop

To add a validation loop, override the *validation_step* method of the `LightningModule`:

```
class LitModel(pl.LightningModule):
    def validation_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self.model(x)
        loss = F.cross_entropy(y_hat, y)
        self.log('val_loss', loss)
```

Under the hood, Lightning does the following:

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

    if validate_at_some_point:
        # disable grads + batchnorm + dropout
        torch.set_grad_enabled(False)
        model.eval()

        # ---------------- VAL LOOP ---------------
        for val_batch in model.val_dataloader:
            val_out = model.validation_step(val_batch)
        # ---------------- VAL LOOP ---------------

        # enable grads + batchnorm + dropout
        torch.set_grad_enabled(True)
        model.train()
```

Validation epoch-level metrics

If you need to do something with all the outputs of each *validation_step*, override *validation_epoch_end*.

```
def validation_step(self, batch, batch_idx):
    x, y = batch
    y_hat = self.model(x)
    loss = F.cross_entropy(y_hat, y)
    pred =  ...
    return pred

def validation_epoch_end(self, validation_step_outputs):
   for pred in validation_step_outputs:
       # do something with a pred
```

Validating with DataParallel

When training using a *accelerator* that splits data from each batch across GPUs, sometimes you might need to aggregate them on the master GPU for processing (dp, or ddp2).

In this case, implement the *validation_step_end* method

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

def validation_step_end(self, batch_parts):
    gpu_0_prediction = batch_parts.pred[0]['pred']
    gpu_1_prediction = batch_parts.pred[1]['pred']

    # do something with both outputs
    return (batch_parts[0]['loss'] + batch_parts[1]['loss']) / 2

def validation_epoch_end(self, validation_step_outputs):
   for out in validation_step_outputs:
       # do something with preds
```

The full pseudocode that lighting does under the hood is:

```
outs = []
for batch in dataloader:
    batches = split_batch(batch)
    dp_outs = []
    for sub_batch in batches:
        # 1
        dp_out = validation_step(sub_batch)
        dp_outs.append(dp_out)

    # 2
    out = validation_step_end(dp_outs)
    outs.append(out)

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

(continues on next page)

(continued from previous page)

```
# 3
validation_epoch_end(outs)
```

7.2.3 Test loop

The process for adding a test loop is the same as the process for adding a validation loop. Please refer to the section above for details.

The only difference is that the test loop is only called when *.test()* is used:

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

# automatically loads the best weights for you
trainer.test(model)
```

There are two ways to call *test()*:

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

# automatically auto-loads the best weights
trainer.test(test_dataloaders=test_dataloader)

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

7.3 Inference

For research, LightningModules are best structured as systems.

```
import pytorch_lightning as pl
import torch
from torch import nn

class Autoencoder(pl.LightningModule):

    def __init__(self, latent_dim=2):
        super().__init__()
        self.encoder = nn.Sequential(nn.Linear(28 * 28, 256), nn.ReLU(), nn.
↪Linear(256, latent_dim))
        self.decoder = nn.Sequential(nn.Linear(latent_dim, 256), nn.ReLU(), nn.
↪Linear(256, 28 * 28))

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

(continues on next page)

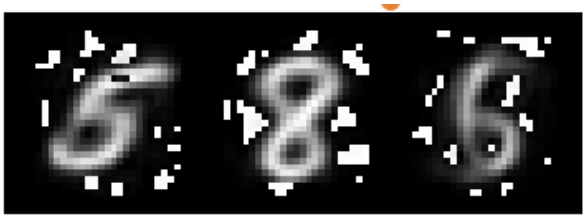
(continued from previous page)

```
        x, _ = batch

        # encode
        x = x.view(x.size(0), -1)
        z = self.encoder(x)

        # decode
        recons = self.decoder(z)

        # reconstruction
        reconstruction_loss = nn.functional.mse_loss(recons, x)
        return reconstruction_loss

    def validation_step(self, batch, batch_idx):
        x, _ = batch
        x = x.view(x.size(0), -1)
        z = self.encoder(x)
        recons = self.decoder(z)
        reconstruction_loss = nn.functional.mse_loss(recons, x)
        self.log('val_reconstruction', reconstruction_loss)

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(), lr=0.0002)
```

Which can be trained like this:

```
autoencoder = Autoencoder()
trainer = pl.Trainer(gpus=1)
trainer.fit(autoencoder, train_dataloader, val_dataloader)
```

This simple model generates examples that look like this (the encoders and decoders are too weak)



The methods above are part of the lightning interface:

- training_step
- validation_step
- test_step
- configure_optimizers

Note that in this case, the train loop and val loop are exactly the same. We can of course reuse this code.

```
class Autoencoder(pl.LightningModule):

    def __init__(self, latent_dim=2):
        super().__init__()
        self.encoder = nn.Sequential(nn.Linear(28 * 28, 256), nn.ReLU(), nn.
↪Linear(256, latent_dim))
```

(continues on next page)

(continued from previous page)

```
        self.decoder = nn.Sequential(nn.Linear(latent_dim, 256), nn.ReLU(), nn.
↪Linear(256, 28 * 28))

    def training_step(self, batch, batch_idx):
        loss = self.shared_step(batch)

        return loss

    def validation_step(self, batch, batch_idx):
        loss = self.shared_step(batch)
        self.log('val_loss', loss)

    def shared_step(self, batch):
        x, _ = batch

        # encode
        x = x.view(x.size(0), -1)
        z = self.encoder(x)

        # decode
        recons = self.decoder(z)

        # loss
        return nn.functional.mse_loss(recons, x)

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(), lr=0.0002)
```

We create a new method called *shared_step* that all loops can use. This method name is arbitrary and NOT reserved.

7.3.1 Inference in research

In the case where we want to perform inference with the system we can add a *forward* method to the LightningModule.

```
class Autoencoder(pl.LightningModule):
    def forward(self, x):
        return self.decoder(x)
```

The advantage of adding a forward is that in complex systems, you can do a much more involved inference procedure, such as text generation:

```
class Seq2Seq(pl.LightningModule):

    def forward(self, x):
        embeddings = self(x)
        hidden_states = self.encoder(embeddings)
        for h in hidden_states:
            # decode
            ...
        return decoded
```

7.3.2 Inference in production

For cases like production, you might want to iterate different models inside a LightningModule.

```
import pytorch_lightning as pl
from pytorch_lightning.metrics import functional as FM

class ClassificationTask(pl.LightningModule):

    def __init__(self, model):
        super().__init__()
        self.model = model

    def training_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self.model(x)
        loss = F.cross_entropy(y_hat, y)
        return loss

    def validation_step(self, batch, batch_idx):
       x, y = batch
       y_hat = self.model(x)
       loss = F.cross_entropy(y_hat, y)
       acc = FM.accuracy(y_hat, y)

       # loss is tensor. The Checkpoint Callback is monitoring 'checkpoint_on'
       metrics = {'val_acc': acc, 'val_loss': loss}
       self.log_dict(metrics)
       return metrics

    def test_step(self, batch, batch_idx):
       metrics = self.validation_step(batch, batch_idx)
       metrics = {'test_acc': metrics['val_acc'], 'test_loss': metrics['val_loss']}
       self.log_dict(metrics)

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

Then pass in any arbitrary model to be fit with this task

```
for model in [resnet50(), vgg16(), BidirectionalRNN()]:
    task = ClassificationTask(model)

    trainer = Trainer(gpus=2)
    trainer.fit(task, train_dataloader, val_dataloader)
```

Tasks can be arbitrarily complex such as implementing GAN training, self-supervised or even RL.

```
class GANTask(pl.LightningModule):

    def __init__(self, generator, discriminator):
        super().__init__()
        self.generator = generator
        self.discriminator = discriminator
    ...
```

When used like this, the model can be separated from the Task and thus used in production without needing to keep it in a *LightningModule*.

- You can export to onnx.
- Or trace using Jit.
- or run in the python runtime.

```
task = ClassificationTask(model)

trainer = Trainer(gpus=2)
trainer.fit(task, train_dataloader, val_dataloader)

# use model after training or load weights and drop into the production system
model.eval()
y_hat = model(x)
```

7.4 LightningModule API

7.4.1 Methods

configure_callbacks

LightningModule.**configure_callbacks**()

Configure model-specific callbacks. When the model gets attached, e.g., when `.fit()` or `.test()` gets called, the list returned here will be merged with the list of callbacks passed to the Trainer's `callbacks` argument. If a callback returned here has the same type as one or several callbacks already present in the Trainer's callbacks list, it will take priority and replace them. In addition, Lightning will make sure *ModelCheckpoint* callbacks run last.

Returns A list of callbacks which will extend the list of callbacks in the Trainer.

Example:

```
def configure_callbacks(self):
    early_stop = EarlyStopping(monitor"val_acc", mode="max")
    checkpoint = ModelCheckpoint(monitor="val_loss")
    return [early_stop, checkpoint]
```

Note: Certain callback methods like *on_init_start()* will never be invoked on the new callbacks returned here.

configure_optimizers

LightningModule.**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.

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 whose 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 the scheduler and its associated configuration. The default configuration is shown below.

```
{
    'scheduler': lr_scheduler, # The LR scheduler instance (required)
    '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 for ReduceLROnPlateau to monitor
    'strict': True, # Whether to crash the training if `monitor` is not found
    'name': None, # Custom name for LearningRateMonitor to use
}
```

Only the `scheduler` key is required, the rest will be set to the defaults 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,
}
```

forward

`LightningModule.`**`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

LightningModule.**freeze**()

Freeze all params for inference.

Example:

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

Return type None

log

LightningModule.**log**(*name*, *value*, *prog_bar=False*, *logger=True*, *on_step=None*, *on_epoch=None*, *reduce_fx=torch.mean*, *tbptt_reduce_fx=torch.mean*, *tbptt_pad_token=0*, *enable_graph=False*, *sync_dist=False*, *sync_dist_op='mean'*, *sync_dist_group=None*)

Log a key, value

Example:

```
self.log('train_loss', loss)
```

The default behavior per hook is as follows

Table 1: * also applies to the test loop

LightningMoule Hook	on_step	on_epoch	prog_bar	logger
training_step	T	F	F	T
training_step_end	T	F	F	T
training_epoch_end	F	T	F	T
validation_step*	F	T	F	T
validation_step_end*	F	T	F	T
validation_epoch_end*	F	T	F	T

Parameters

- **name**¶ (str) – key name
- **value**¶ (Any) – value name
- **prog_bar**¶ (bool) – if True logs to the progress bar
- **logger**¶ (bool) – if True logs to the logger
- **on_step**¶ (Optional[bool]) – if True logs at this step. None auto-logs at the training_step but not validation/test_step
- **on_epoch**¶ (Optional[bool]) – if True logs epoch accumulated metrics. None auto-logs at the val/test step but not training_step
- **reduce_fx**¶ (Callable) – reduction function over step values for end of epoch. Torch.mean by default
- **tbptt_reduce_fx**¶ (Callable) – function to reduce on truncated back prop
- **tbptt_pad_token**¶ (int) – token to use for padding
- **enable_graph**¶ (bool) – if True, will not auto detach the graph
- **sync_dist**¶ (bool) – if True, reduces the metric across GPUs/TPUs
- **sync_dist_op**¶ (Union[Any, str]) – the op to sync across GPUs/TPUs
- **sync_dist_group**¶ (Optional[Any]) – the ddp group

log_dict

LightningModule.**log_dict**(*dictionary*, *prog_bar=False*, *logger=True*, *on_step=None*, *on_epoch=None*, *reduce_fx=torch.mean*, *tbptt_reduce_fx=torch.mean*, *tbptt_pad_token=0*, *enable_graph=False*, *sync_dist=False*, *sync_dist_op='mean'*, *sync_dist_group=None*)

Log a dictonary of values at once

Example:

```
values = {'loss': loss, 'acc': acc, ..., 'metric_n': metric_n}
self.log_dict(values)
```

Parameters

- **dictionary** (dict) – key value pairs (str, tensors)
- **prog_bar** (bool) – if True logs to the progress base
- **logger** (bool) – if True logs to the logger
- **on_step** (Optional[bool]) – if True logs at this step. None auto-logs for training_step but not validation/test_step
- **on_epoch** (Optional[bool]) – if True logs epoch accumulated metrics. None auto-logs for val/test step but not training_step
- **reduce_fx** (Callable) – reduction function over step values for end of epoch. Torch.mean by default
- **tbptt_reduce_fx** (Callable) – function to reduce on truncated back prop
- **tbptt_pad_token** (int) – token to use for padding
- **enable_graph** (bool) – if True, will not auto detach the graph
- **sync_dist** (bool) – if True, reduces the metric across GPUs/TPUs
- **sync_dist_op** (Union[Any, str]) – the op to sync across GPUs/TPUs
- **sync_dist_group** (Optional[Any]) – the ddp group:

print

LightningModule.**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

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

Save all model arguments.

Parameters **args**[¶] – single object of *dict*, *NameSpace* or *OmegaConf* or string names or arguments from class *__init__*

```
>>> class ManuallyArgsModel(LightningModule):
...     def __init__(self, arg1, arg2, arg3):
...         super().__init__()
...         # manually assign 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

test_step

LightningModule.**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 dataloaders used).

Returns

Any of.

- Any object or value
- None - Testing will skip to the next batch

```
# 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)
    test_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'test_loss': loss, 'test_acc': test_acc})
```

If you pass in multiple test dataloaders, *test_step()* will have an additional argument.

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

Note: If you don't need to test 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

LightningModule.**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.

Returns None or anything

```
# 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)
    self.log('test_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.encoder(x)
    return out

def test_step_end(self, output_results):
```

(continues on next page)

(continued from previous page)

```
    # this out is now the full size of the batch
    all_test_step_outs = output_results.out
    loss = nce_loss(all_test_step_outs)
    self.log('test_loss', loss)
```

See also:

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

test_epoch_end

LightningModule.**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**¶ (List[Any]) – 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

Return type None

Returns None

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

Examples

With a single dataloader:

```
def test_epoch_end(self, outputs):
    # do something with the outputs of all test batches
    all_test_preds = test_step_outputs.predictions

    some_result = calc_all_results(all_test_preds)
    self.log(some_result)
```

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):
    final_value = 0
    for dataloader_outputs in outputs:
        for test_step_out in dataloader_outputs:
            # do something
            final_value += test_step_out

    self.log('final_metric', final_value)
```

to_onnx

LightningModule.**to_onnx**(*file_path*, *input_sample=None*, ***kwargs*)

Saves the model in ONNX format

Parameters

- **file_path** (Union[str, Path]) – The path of the file the onnx model should be saved to.
- **input_sample** (Optional[Any]) – An input for tracing. Default: None (Use self.example_input_array)
- ****kwargs** – Will be passed to torch.onnx.export function.

Example

```
>>> class SimpleModel(LightningModule):
...     def __init__(self):
...         super().__init__()
...         self.l1 = torch.nn.Linear(in_features=64, out_features=4)
...
...     def forward(self, x):
...         return torch.relu(self.l1(x.view(x.size(0), -1)))
```

```
>>> with tempfile.NamedTemporaryFile(suffix='.onnx', delete=False) as tmpfile:
...     model = SimpleModel()
...     input_sample = torch.randn((1, 64))
...     model.to_onnx(tmpfile.name, input_sample, export_params=True)
...     os.path.isfile(tmpfile.name)
True
```

to_torchscript

LightningModule.**to_torchscript**(*file_path=None*, *method='script'*, *example_inputs=None*, ***kwargs*)

By default compiles the whole model to a ScriptModule. If you want to use tracing, please provided the argument *method='trace'* and make sure that either the example_inputs argument is provided, or the model has self.example_input_array set. If you would like to customize the modules that are scripted you should override this method. In case you want to return multiple modules, we recommend using a dictionary.

Parameters

- **file_path** (Union[str, Path, None]) – Path where to save the torchscript. Default: None (no file saved).
- **method** (Optional[str]) – Whether to use TorchScript's script or trace method. Default: 'script'
- **example_inputs** (Optional[Any]) – An input to be used to do tracing when method is set to 'trace'. Default: None (Use self.example_input_array)
- ****kwargs** – Additional arguments that will be passed to the torch.jit.script() or torch.jit.trace() function.

Note:

- Requires the implementation of the *forward()* method.
- The exported script will be set to evaluation mode.
- It is recommended that you install the latest supported version of PyTorch to use this feature without limitations. See also the torch.jit documentation for supported features.

Example

```
>>> class SimpleModel(LightningModule):
...     def __init__(self):
...         super().__init__()
...         self.l1 = torch.nn.Linear(in_features=64, out_features=4)
...
...     def forward(self, x):
...         return torch.relu(self.l1(x.view(x.size(0), -1)))
...
>>> model = SimpleModel()
>>> torch.jit.save(model.to_torchscript(), "model.pt")
>>> os.path.isfile("model.pt")
>>> torch.jit.save(model.to_torchscript(file_path="model_trace.pt", method='trace
↪',
...                                     example_inputs=torch.randn(1, 64)))
>>> os.path.isfile("model_trace.pt")
True
```

Return type Union[ScriptModule, Dict[str, ScriptModule]]

Returns This LightningModule as a torchscript, regardless of whether file_path is defined or not.

training_step

LightningModule.**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.

Returns

Any of.

- Tensor - The loss tensor
- dict - A dictionary. Can include any keys, but must include the key 'loss'
- None - Training will skip to the next batch

Note: Returning `None` is currently not supported for multi-GPU or TPU, or with 16-bit precision enabled.

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.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

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': loss, 'hiddens': hiddens}
```

Note: 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

`LightningModule.`**`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.

Returns Anything

When using dp/ddp2 distributed backends, only a portion of the batch is inside the training_step:

```
def training_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self(x)

    # softmax uses only a portion of the batch in the denomintaor
    loss = self.softmax(out)
    loss = nce_loss(loss)
    return loss
```

If you wish to do something with all the parts of the batch, then use this method to do it:

```
def training_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self.encoder(x)
    return {'pred': out}

def training_step_end(self, training_step_outputs):
    gpu_0_pred = training_step_outputs[0]['pred']
    gpu_1_pred = training_step_outputs[1]['pred']
    gpu_n_pred = training_step_outputs[n]['pred']

    # this softmax now uses the full batch
    loss = nce_loss([gpu_0_pred, gpu_1_pred, gpu_n_pred])
    return loss
```

See also:

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

training_epoch_end

LightningModule.**training_epoch_end**(*outputs*)

Called at the end of the training epoch with the outputs of all training steps. Use this in case you need to do something with all the outputs for every training_step.

```
# 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**¶ (List[Any]) – 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 None

Returns None

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

Example:

```
def training_epoch_end(self, training_step_outputs):
    # do something with all training_step outputs
    return result
```

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, training_step_outputs):
    for out in training_step_outputs:
        # do something here
```

unfreeze

LightningModule.**unfreeze**()

Unfreeze all parameters for training.

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

Return type None

validation_step

LightningModule.**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(val_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 dataloaders used)

Returns

Any of.

- Any object or value
- None - Validation will skip to the next batch

```
# 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)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, *validation_step()* will have an additional argument.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx):
    # dataloader_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

LightningModule.**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.

Returns None or anything

```
# 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.encoder(x)
    loss = self.softmax(out)
    loss = nce_loss(loss)
    self.log('val_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

def validation_step_end(self, val_step_outputs):
    for out in val_step_outputs:
        # do something with these
```

See also:

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

validation_epoch_end

LightningModule.**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**¶ (List[Any]) – 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 None

Returns None

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

Examples

With a single dataloader:

```
def validation_epoch_end(self, val_step_outputs):
    for out in val_step_outputs:
        # do something
```

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):
    for dataloader_output_result in outputs:
        dataloader_outs = dataloader_output_result.dataloader_i_outputs

    self.log('final_metric', final_value)
```

write_prediction

LightningModule.**write_prediction**(*name*, *value*, *filename='predictions.pt'*)

Write predictions to disk using `torch.save`

Example:

```
self.write_prediction('pred', torch.tensor(...), filename='my_predictions.pt')
```

Parameters

- **name**¶ (str) – a string indicating the name to save the predictions under
- **value**¶ (Union[Tensor, List[Tensor]]) – the predictions, either a single Tensor or a list of them

- **filename** (str) – name of the file to save the predictions to

Note: when running in distributed mode, calling `write_prediction` will create a file for each device with respective names: `filename_rank_0.pt`, `filename_rank_1.pt`, ...

write_prediction_dict

LightningModule.**write_prediction_dict**(*predictions_dict*, *filename='predictions.pt'*)

Write a dictonary of predictions to disk at once using `torch.save`

Example:

```
pred_dict = {'pred1': torch.tensor(...), 'pred2': torch.tensor(...)}
self.write_prediction_dict(pred_dict)
```

Parameters **predictions_dict** (Dict[str, Any]) – dict containing predictions, where each prediction should either be single `Tensor` or a list of them

Note: when running in distributed mode, calling `write_prediction_dict` will create a file for each device with respective names: `filename_rank_0.pt`, `filename_rank_1.pt`, ...

7.4.2 Properties

These are properties available in a LightningModule.

current_epoch

The current epoch

```
def training_step(...):
    if self.current_epoch == 0:
```

device

The device the module is on. Use it to keep your code device agnostic

```
def training_step(...):
    z = torch.rand(2, 3, device=self.device)
```

global_rank

The global_rank of this LightningModule. Lightning saves logs, weights etc only from global_rank = 0. You normally do not need to use this property

Global rank refers to the index of that GPU across ALL GPUs. For example, if using 10 machines, each with 4 GPUs, the 4th GPU on the 10th machine has global_rank = 39

---

global_step

The current step (does not reset each epoch)

```
def training_step(...):
    self.logger.experiment.log_image(..., step=self.global_step)
```

---

hparams

The arguments saved by calling `save_hyperparameters` passed through `__init__()` could be accessed by the `hparams` attribute.

```
def __init__(self, learning_rate):
    self.save_hyperparameters()

def configure_optimizers(self):
    return Adam(self.parameters(), lr=self.hparams.learning_rate)
```

---

logger

The current logger being used (tensorboard or other supported logger)

```
def training_step(...):
    # the generic logger (same no matter if tensorboard or other supported logger)
    self.logger

    # the particular logger
    tensorboard_logger = self.logger.experiment
```

---

local_rank

The local_rank of this LightningModule. Lightning saves logs, weights etc only from global_rank = 0. You normally do not need to use this property

Local rank refers to the rank on that machine. For example, if using 10 machines, the GPU at index 0 on each machine has local_rank = 0.

---

precision

The type of precision used:

```
def training_step(...):
    if self.precision == 16:
```

---

trainer

Pointer to the trainer

```
def training_step(...):
    max_steps = self.trainer.max_steps
    any_flag = self.trainer.any_flag
```

---

use_amp

True if using Automatic Mixed Precision (AMP)

---

use_ddp

True if using ddp

---

use_ddp2

True if using ddp2

---

use_dp

True if using dp

---

use_tpu

True if using TPUs

---

automatic_optimization

When set to `False`, Lightning does not automate the optimization process. This means you are responsible for handling your optimizers. However, we do take care of precision and any accelerators used.

```
def __init__(self):
    self.automatic_optimization = False

def training_step(self, batch, batch_idx):
    opt = self.optimizers(use_pl_optimizer=True)

    loss = ...
    self.manual_backward(loss, opt)
    opt.step()
    opt.zero_grad()
```

This is recommended only if using 2+ optimizers AND if you know how to perform the optimization procedure properly. Note that automatic optimization can still be used with multiple optimizers by relying on the `optimizer_idx` parameter. Manual optimization is most useful for research topics like reinforcement learning, sparse coding, and GAN research.

In the multi-optimizer case, ignore the `optimizer_idx` argument and use the optimizers directly

```
def __init__(self):
    self.automatic_optimization = False

def training_step(self, batch, batch_idx, optimizer_idx):
    # access your optimizers with use_pl_optimizer=False. Default is True
    (opt_a, opt_b) = self.optimizers(use_pl_optimizer=True)

    gen_loss = ...
    opt_a.zero_grad()
    self.manual_backward(gen_loss, opt_a)
    opt_a.step()

    disc_loss = ...
    opt_b.zero_grad()
    self.manual_backward(disc_loss, opt_b)
    opt_b.step()
```

---

example_input_array

Set and access example_input_array which is basically a single batch.

```
def __init__(self):
    self.example_input_array = ...
    self.generator = ...

def on_train_epoch_end(...):
    # generate some images using the example_input_array
    gen_images = self.generator(self.example_input_array)
```

---

datamodule

Set or access your datamodule.

```
def configure_optimizers(self):
    num_training_samples = len(self.datamodule.train_dataloader())
    ...
```

---

model_size

Get the model file size (in megabytes) using `self.model_size` inside LightningModule.

7.4.3 Hooks

This is the pseudocode to describe how all the hooks are called during a call to `.fit()`.

```
def fit(...):
    on_fit_start()

    if global_rank == 0:
        # prepare data is called on GLOBAL_ZERO only
        prepare_data()

    for gpu/tpu in gpu/tpus:
        train_on_device(model.copy())

    on_fit_end()

def train_on_device(model):
    # setup is called PER DEVICE
    setup()
    configure_optimizers()
    on_pretrain_routine_start()

    for epoch in epochs:
        train_loop()

    teardown()

def train_loop():
    on_train_epoch_start()
    train_outs = []
    for train_batch in train_dataloader():
        on_train_batch_start()

        # ----- train_step methods -------
        out = training_step(batch)
        train_outs.append(out)

        loss = out.loss
```

(continues on next page)

(continued from previous page)

```
        backward()
        on_after_backward()
        optimizer_step()
        on_before_zero_grad()
        optimizer_zero_grad()

        on_train_batch_end(out)

        if should_check_val:
            val_loop()

    # end training epoch
    logs = training_epoch_end(outs)

def val_loop():
    model.eval()
    torch.set_grad_enabled(False)

    on_validation_epoch_start()
    val_outs = []
    for val_batch in val_dataloader():
        on_validation_batch_start()

        # -------- val step methods -------
        out = validation_step(val_batch)
        val_outs.append(out)

        on_validation_batch_end(out)

    validation_epoch_end(val_outs)
    on_validation_epoch_end()

    # set up for train
    model.train()
    torch.set_grad_enabled(True)
```

backward

LightningModule.**backward**(*loss*, *optimizer*, *optimizer_idx*, **args*, ***kwargs*)

Override backward with your own implementation if you need to.

Parameters

- **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, loss, optimizer, optimizer_idx):
    loss.backward()
```

Return type None

get_progress_bar_dict

LightningModule.**get_progress_bar_dict**()

Implement this to override the default items displayed in the progress bar. By default it includes the average loss value, split index of BPTT (if used) and the version of the experiment when using a logger.

```
Epoch 1:   4%|         | 40/1095 [00:03<01:37, 10.84it/s, loss=4.501, v_num=10]
```

Here is an example how to override the defaults:

```
def get_progress_bar_dict(self):
    # don't show the version number
    items = super().get_progress_bar_dict()
    items.pop("v_num", None)
    return items
```

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

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

manual_backward

LightningModule.**manual_backward**(*loss*, *optimizer=None*, **args*, ***kwargs*)

Call this directly from your training_step when doing optimizations manually. By using this we can ensure that all the proper scaling when using 16-bit etc has been done for you

This function forwards all args to the .backward() call as well.

Tip: In manual mode we still automatically clip grads if Trainer(gradient_clip_val=x) is set

---

Tip: In manual mode we still automatically accumulate grad over batches if Trainer(accumulate_grad_batches=x) is set and you use *optimizer.step()*

Example:

```
def training_step(...):
    (opt_a, opt_b) = self.optimizers()
    loss = ...
    # automatically applies scaling, etc...
    self.manual_backward(loss, opt_a)
    opt_a.step()
```

Return type None

on_after_backward

LightningModule.**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
        for k, v in self.named_parameters():
            self.logger.experiment.add_histogram(
                tag=k, values=v.grad, global_step=self.trainer.global_step
            )
```

Return type None

on_before_zero_grad

LightningModule.**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_fit_start

ModelHooks.**on_fit_start**()
Called at the very beginning of fit. If on DDP it is called on every process

Return type None

on_fit_end

ModelHooks.**on_fit_end**()
Called at the very end of fit. If on DDP it is called on every process

Return type None

on_load_checkpoint

LightningModule.**on_load_checkpoint**(*checkpoint*)

Do something with the checkpoint. Gives model a chance to load something before state_dict is restored.

Parameters **checkpoint** – A dictionary with variables from the checkpoint.

Return type None

on_save_checkpoint

LightningModule.**on_save_checkpoint**(*checkpoint*)

Give the model a chance to add something to the checkpoint. state_dict is already there.

Parameters **checkpoint** – A dictionary in which you can save variables to save in a checkpoint. Contents need to be pickleable.

Return type None

on_pretrain_routine_start

ModelHooks.**on_pretrain_routine_start**()

Called at the beginning of the pretrain routine (between fit and train start).

- fit
- pretrain_routine start
- pretrain_routine end
- training_start

Return type None

on_pretrain_routine_end

ModelHooks.**on_pretrain_routine_end**()

Called at the end of the pretrain routine (between fit and train start).

- fit
- pretrain_routine start
- pretrain_routine end
- training_start

Return type None

on_test_batch_start

ModelHooks.**on_test_batch_start**(*batch*, *batch_idx*, *dataloader_idx*)
Called in the test loop before anything happens for that batch.

Parameters

- **batch** (Any) – The batched data as it is returned by the test DataLoader.
- **batch_idx** (int) – the index of the batch
- **dataloader_idx** (int) – the index of the dataloader

Return type None

on_test_batch_end

ModelHooks.**on_test_batch_end**(*outputs*, *batch*, *batch_idx*, *dataloader_idx*)
Called in the test loop after the batch.

Parameters

- **outputs** (Any) – The outputs of test_step_end(test_step(x))
- **batch** (Any) – The batched data as it is returned by the test DataLoader.
- **batch_idx** (int) – the index of the batch
- **dataloader_idx** (int) – the index of the dataloader

Return type None

on_test_epoch_start

ModelHooks.**on_test_epoch_start**()
Called in the test loop at the very beginning of the epoch.

Return type None

on_test_epoch_end

ModelHooks.**on_test_epoch_end**()
Called in the test loop at the very end of the epoch.

Return type None

on_train_batch_start

ModelHooks.**on_train_batch_start**(*batch*, *batch_idx*, *dataloader_idx*)
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.
- **batch_idx** (int) – the index of the batch
- **dataloader_idx** (int) – the index of the dataloader

Return type None

on_train_batch_end

ModelHooks.**on_train_batch_end**(*outputs*, *batch*, *batch_idx*, *dataloader_idx*)
Called in the training loop after the batch.

Parameters

- **outputs** (Any) – The outputs of training_step_end(training_step(x))
- **batch** (Any) – The batched data as it is returned by the training DataLoader.
- **batch_idx** (int) – the index of the batch
- **dataloader_idx** (int) – the index of the dataloader

Return type None

on_train_epoch_start

ModelHooks.**on_train_epoch_start**()
Called in the training loop at the very beginning of the epoch.

Return type None

on_train_epoch_end

ModelHooks.**on_train_epoch_end**(*outputs*)
Called in the training loop at the very end of the epoch.

Return type None

on_validation_batch_start

ModelHooks.**on_validation_batch_start**(*batch*, *batch_idx*, *dataloader_idx*)
Called in the validation loop before anything happens for that batch.

Parameters

- **batch** (Any) – The batched data as it is returned by the validation DataLoader.
- **batch_idx** (int) – the index of the batch
- **dataloader_idx** (int) – the index of the dataloader

Return type None

on_validation_batch_end

ModelHooks.**on_validation_batch_end**(*outputs*, *batch*, *batch_idx*, *dataloader_idx*)
Called in the validation loop after the batch.

Parameters

- **outputs** (Any) – The outputs of validation_step_end(validation_step(x))
- **batch** (Any) – The batched data as it is returned by the validation DataLoader.
- **batch_idx** (int) – the index of the batch
- **dataloader_idx** (int) – the index of the dataloader

Return type None

on_validation_epoch_start

ModelHooks.**on_validation_epoch_start**()
Called in the validation loop at the very beginning of the epoch.

Return type None

on_validation_epoch_end

ModelHooks.**on_validation_epoch_end**()
Called in the validation loop at the very end of the epoch.

Return type None

optimizer_step

LightningModule.**optimizer_step**(*epoch=None*, *batch_idx=None*, *optimizer=None*, *optimizer_idx=None*, *optimizer_closure=None*, *on_tpu=None*, *using_native_amp=None*, *using_lbfgs=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.

Tip: With `Trainer(enable_pl_optimizer=True)`, you can use `optimizer.step()` directly and it will handle zero_grad, accumulated gradients, AMP, TPU and more automatically for you.

Warning: If you are overriding this method, make sure that you pass the `optimizer_closure` parameter to `optimizer.step()` function as shown in the examples. This ensures that `train_step_and_backward_closure` is called within `run_training_batch()`.

Parameters

- **epoch** (Optional[int]) – Current epoch
- **batch_idx** (Optional[int]) – Index of current batch
- **optimizer** (Optional[Optimizer]) – A PyTorch optimizer

- **optimizer_idx¶** (Optional[int]) – If you used multiple optimizers this indexes into that list.
- **optimizer_closure¶** (Optional[Callable]) – closure for all optimizers
- **on_tpu¶** (Optional[bool]) – true if TPU backward is required
- **using_native_amp¶** (Optional[bool]) – True if using native amp
- **using_lbfgs¶** (Optional[bool]) – True if the matching optimizer is lbfgs

Examples:

```
# DEFAULT
def optimizer_step(self, epoch, batch_idx, optimizer, optimizer_idx,
                   optimizer_closure, on_tpu, using_native_amp, using_lbfgs):
    optimizer.step(closure=optimizer_closure)

# Alternating schedule for optimizer steps (i.e.: GANs)
def optimizer_step(self, epoch, batch_idx, optimizer, optimizer_idx,
                   optimizer_closure, on_tpu, using_native_amp, using_lbfgs):
    # update generator opt every 2 steps
    if optimizer_idx == 0:
        if batch_idx % 2 == 0 :
            optimizer.step(closure=optimizer_closure)
            optimizer.zero_grad()

    # update discriminator opt every 4 steps
    if optimizer_idx == 1:
        if batch_idx % 4 == 0 :
            optimizer.step(closure=optimizer_closure)
            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, epoch, batch_idx, optimizer, optimizer_idx,
                   optimizer_closure, on_tpu, using_native_amp, using_lbfgs):
    # 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(closure=optimizer_closure)
    optimizer.zero_grad()
```

Return type None

optimizer_zero_grad

LightningModule.**optimizer_zero_grad**(*epoch*, *batch_idx*, *optimizer*, *optimizer_idx*)

prepare_data

LightningModule.**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(stage)
model.train_dataloader()
model.val_dataloader()
model.test_dataloader()
```

Return type None

setup

ModelHooks.**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(stage):
        data = Load_data(...)
        self.l1 = nn.Linear(28, data.num_classes)
```

Return type None

tbptt_split_batch

LightningModule.**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]
```

(continues on next page)

(continued from previous page)

```
            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()*.

teardown

`ModelHooks.`**`teardown`**(*stage*)

Called at the end of fit and test.

Parameters **stage**¶ (`str`) – either 'fit' or 'test'

Return type `None`

train_dataloader

`LightningModule.`**`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`.

For data processing use the following pattern:

- download in `prepare_data()`
- process and split in `setup()`

However, the above are only necessary for distributed processing.

Warning: do not assign state in prepare_data

- `fit()`
- ...
- `prepare_data()`
- `setup()`
- `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
```

val_dataloader

LightningModule.**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
```

(continues on next page)

(continued from previous page)

```
# 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 `dataloader_idx` which matches the order here.

test_dataloader

`LightningModule.`**`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`.

For data processing use the following pattern:

- download in `prepare_data()`
- process and split in `setup()`

However, the above are only necessary for distributed processing.

Warning: do not assign state in prepare_data

- `fit()`
- ...
- `prepare_data()`
- `setup()`
- `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

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

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

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

transfer_batch_to_device

`DataHooks.`**`transfer_batch_to_device`**(*batch*, *device=None*)

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` or anything that implements *.to(...)*
- `list`
- `dict`
- `tuple`
- `torchtext.data.batch.Batch`

For anything else, you need to define how the data is moved to the target device (CPU, GPU, TPU, ...).

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).

Note: This hook only runs on single GPU training and DDP (no data-parallel). If you need multi-GPU support for your custom batch objects, you need to define your custom `DistributedDataParallel` or `LightningDistributedDataParallel` and override `configure_ddp()`.

Parameters

- **`batch`** (`Any`) – A batch of data that needs to be transferred to a new device.

- **device** (Optional[device]) – The target device as defined in PyTorch.

Return type Any

Returns A reference to the data on the new device.

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
```

See also:

- move_data_to_device()
- apply_to_collection()

on_before_batch_transfer

DataHooks.**on_before_batch_transfer**(*batch*, *dataloader_idx*)

Override to alter or apply batch augmentations to your batch before it is transferred to the device.

Warning: dataloader_idx always returns 0, and will be updated to support the true idx in the future.

Note: This hook only runs on single GPU training and DDP (no data-parallel).

Parameters

- **batch** – A batch of data that needs to be altered or augmented.
- **dataloader_idx** – DataLoader idx for batch

Returns A batch of data

Example:

```
def on_before_batch_transfer(self, batch, dataloader_idx):
    batch['x'] = transforms(batch['x'])
    return batch
```

See also:

- *on_after_batch_transfer()*
- *transfer_batch_to_device()*

on_after_batch_transfer

DataHooks.**on_after_batch_transfer**(*batch*, *dataloader_idx*)

Override to alter or apply batch augmentations to your batch after it is transferred to the device.

> **Warning:** `dataloader_idx` always returns 0, and will be updated to support the true `idx` in the future.

Note: This hook only runs on single GPU training and DDP (no data-parallel).

Parameters

- **batch** – A batch of data that needs to be altered or augmented.
- **dataloader_idx** – DataLoader idx for batch (Default: 0)

Returns A batch of data

Example:

```
def on_after_batch_transfer(self, batch, dataloader_idx):
    batch['x'] = gpu_transforms(batch['x'])
    return batch
```

See also:

- *on_before_batch_transfer()*
- *transfer_batch_to_device()*

CHAPTER

EIGHT

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.

8.1 Basic use

This is the basic use of the trainer:

```
model = MyLightningModule()

trainer = Trainer()
trainer.fit(model, train_dataloader, val_dataloader)
```

8.2 Under the hood

Under the hood, the Lightning Trainer handles the training loop details for you, some examples include:

- Automatically enabling/disabling grads
- Running the training, validation and test dataloaders
- Calling the Callbacks at the appropriate times
- Putting batches and computations on the correct devices

Here's the pseudocode for what the trainer does under the hood (showing the train loop only)

```
# put model in train mode
model.train()
torch.set_grad_enabled(True)

losses = []
for batch in train_dataloader:
    # calls hooks like this one
    on_train_batch_start()

    # train step
    loss = training_step(batch)

    # backward
    loss.backward()

    # apply and clear grads
    optimizer.step()
    optimizer.zero_grad()

    losses.append(loss)
```

8.3 Trainer in Python scripts

In Python scripts, it's recommended you use a main function to call the Trainer.

```
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: Pro-tip: You don't need to define all flags manually. Lightning can add them automatically

```
from argparse import ArgumentParser

def main(args):
    model = LightningModule()
    trainer = Trainer.from_argparse_args(args)
    trainer.fit(model)

if __name__ == '__main__':
    parser = ArgumentParser()
    parser = Trainer.add_argparse_args(
        # group the Trainer arguments together
        parser.add_argument_group(title="pl.Trainer args")
    )
    args = parser.parse_args()

    main(args)
```

So you can run it like so:

```
python main.py --gpus 2 --max_steps 10 --limit_train_batches 10 --any_trainer_arg x
```

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.

8.4 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(test_dataloaders=test_dataloader)
```

8.5 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.

8.6 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)
```

8.7 Trainer flags

8.7.1 accelerator

The accelerator backend to use (previously known as distributed_backend).

- (`'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 efficient 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(accelerator=None)
```

Example:

```
# dp = DataParallel
trainer = Trainer(gpus=2, accelerator='dp')

# ddp = DistributedDataParallel
trainer = Trainer(gpus=2, num_nodes=2, accelerator='ddp')

# ddp2 = DistributedDataParallel + dp
trainer = Trainer(gpus=2, num_nodes=2, accelerator='ddp2')
```

Note: This option does not apply to TPU. TPUs use `'ddp'` by default (over each core)

You can also modify hardware behavior by subclassing an existing accelerator to adjust for your needs.

Example:

```
class MyOwnDDP(DDPAccelerator):
    ...

Trainer(accelerator=MyOwnDDP())
```

Warning: Passing in custom accelerators is experimental but work is in progress to enable full compatibility.

8.7.2 accumulate_grad_batches

Accumulates grads every k batches or as set up in the dict. Trainer also calls `optimizer.step()` for the last indivisible step number.

```
# 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})
```

8.7.3 amp_backend

Use PyTorch AMP ('native') (available PyTorch 1.6+), or NVIDIA apex ('apex').

```
# using PyTorch built-in AMP, default used by the Trainer
trainer = Trainer(amp_backend='native')

# using NVIDIA Apex
trainer = Trainer(amp_backend='apex')
```

8.7.4 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='O2')
```

8.7.5 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')

# call tune to find the batch size
trainer.tune(model)
```

8.7.6 auto_select_gpus

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.

Example:

```
# no auto selection (picks first 2 gpus on system, may fail if other process is␣
↪occupying)
trainer = Trainer(gpus=2, auto_select_gpus=False)

# enable auto selection (will find two available gpus on system)
trainer = Trainer(gpus=2, auto_select_gpus=True)

# specifies all GPUs regardless of its availability
Trainer(gpus=-1, auto_select_gpus=False)

# specifies all available GPUs (if only one GPU is not occupied, uses one gpu)
Trainer(gpus=-1, auto_select_gpus=True)
```

8.7.7 auto_lr_find

Runs a learning rate finder algorithm (see this paper) when calling trainer.tune(), 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)

# call tune to find the lr
trainer.tune(model)
```

Example:

```
# run learning rate finder, results override hparams.my_lr_arg
trainer = Trainer(auto_lr_find='my_lr_arg')

# call tune to find the lr
trainer.tune(model)
```

Note: See the *learning rate finder guide*.

8.7.8 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)
```

8.7.9 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)
```

8.7.10 callbacks

Add a list of *Callback*. Callbacks run sequentially in the order defined here with the exception of *ModelCheckpoint* callbacks which run after all others to ensure all states are saved to the checkpoints.

```
# 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.")
```

Model-specific callbacks can also be added inside the `LightningModule` through *configure_callbacks()*. Callbacks returned in this hook will extend the list initially given to the `Trainer` argument, and replace the trainer callbacks should there be two or more of the same type. *ModelCheckpoint* callbacks always run last.

8.7.11 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)
```

8.7.12 checkpoint_callback

By default Lightning saves a checkpoint for you in your current working directory, with the state of your last training epoch, Checkpoints capture the exact value of all parameters used by a model. To disable automatic checkpointing, set this to *False*.

```
# default used by Trainer
trainer = Trainer(checkpoint_callback=True)

# turn off automatic checkpointing
trainer = Trainer(checkpoint_callback=False)
```

You can override the default behavior by initializing the *ModelCheckpoint* callback, and adding it to the *callbacks* list. See *Saving and Loading Weights* for how to customize checkpointing.

```
from pytorch_lightning.callbacks import ModelCheckpoint
# Init ModelCheckpoint callback, monitoring 'val_loss'
checkpoint_callback = ModelCheckpoint(monitor='val_loss')

# Add your callback to the callbacks list
trainer = Trainer(callbacks=[checkpoint_callback])
```

Warning: Passing a ModelCheckpoint instance to this argument is deprecated since v1.1 and will be unsupported from v1.3. Use *callbacks* argument instead.

8.7.13 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 argument for convenience. Paths can be local paths or remote paths such as *s3://bucket/path* or 'hdfs://path/'. Credentials will need to be set up to use remote filepaths.

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

8.7.14 distributed_backend

Deprecated: This has been renamed `accelerator`.

8.7.15 fast_dev_run

Runs n if set to `n` (int) else 1 if set to `True` batch(es) of train, val and test to find any bugs (ie: a sort of unit test).

Under the hood the pseudocode looks like this when running *fast_dev_run* with a single batch:

```
# 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)

# runs 7 train, val, test batches and program ends
trainer = Trainer(fast_dev_run=7)
```

Note: This argument is a bit different from `limit_train/val/test_batches`. Setting this argument will disable tuner, checkpoint callbacks, early stopping callbacks, loggers and logger callbacks like `LearningRateLogger` and runs for only 1 epoch. This must be used only for debugging purposes. `limit_train/val/test_batches` only limits the number of batches and won't disable anything.

8.7.16 flush_logs_every_n_steps

Writes logs to disk this often.

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

See Also:

- *logging*

8.7.17 gpus

- Number of GPUs to train on (int)
- or which GPUs to train on (list)
- can handle strings

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

# equivalent
trainer = Trainer(gpus=0)
```

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)

# train only on GPUs 1 and 4 across nodes
trainer = Trainer(gpus=[1, 4], num_nodes=4)
```

See Also:

- *Multi-GPU training guide*.

8.7.18 gradient_clip_val

Gradient clipping value

- 0 means don't clip.

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

8.7.19 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)
```

8.7.20 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)
```

In the case of multiple test dataloaders, the limit applies to each dataloader individually.

8.7.21 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)
```

In the case of multiple validation dataloaders, the limit applies to each dataloader individually.

8.7.22 log_every_n_steps

How often to add logging rows (does not write to disk)

```
# default used by the Trainer
trainer = Trainer(log_every_n_steps=50)
```

See Also:

- *logging*

8.7.23 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`.

8.7.24 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)
```

8.7.25 max_epochs

Stop training once this number of epochs is reached

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

8.7.26 min_epochs

Force training for at least these many epochs

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

8.7.27 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)
```

8.7.28 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)
```

8.7.29 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)
```

8.7.30 num_processes

Number of processes to train with. Automatically set to the number of GPUs when using `accelerator="ddp"`. Set to a number greater than 1 when using `accelerator="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 efficient use of multiple CPUs.

```
# Simulate DDP for debugging on your GPU-less laptop
trainer = Trainer(accelerator="ddp_cpu", num_processes=2)
```

8.7.31 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 2 steps by default. Turn it off or modify it here.

```
# default used by the Trainer
trainer = Trainer(num_sanity_val_steps=2)

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

# check all validation data
trainer = Trainer(num_sanity_val_steps=-1)
```

This option will reset the validation dataloader unless `num_sanity_val_steps=0`.

8.7.32 overfit_batches

Uses this much data of the training set. If nonzero, will use the same training set for validation and testing. If the training dataloaders have *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)
```

8.7.33 plugins

Plugins allow you to connect arbitrary backends, precision libraries, SLURM, etc... For example:

- DDP
- SLURM
- TorchElastic
- Apex

To define your own behavior, subclass the relevant class and pass it in. Here's an example linking up your own cluster.

```
from pytorch_lightning.plugins.environments import cluster_environment

class MyCluster(ClusterEnvironment):

    def master_address(self):
        return your_master_address

    def master_port(self):
        return your_master_port

    def world_size(self):
        return the_world_size

trainer = Trainer(cluster_environment=cluster_environment())
```

8.7.34 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)
```

8.7.35 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, gpus=1)
```

Example:

```
# one day
trainer = Trainer(precision=8|4|2)
```

8.7.36 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`.

8.7.37 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, equivalent to `profiler=SimpleProfiler()`
trainer = Trainer(profiler="simple")

# advanced profiler for function-level stats, equivalent to␣
↪`profiler=AdvancedProfiler()`
trainer = Trainer(profiler="advanced")
```

8.7.38 progress_bar_refresh_rate

How often to refresh progress bar (in steps).

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

# disable progress bar
trainer = Trainer(progress_bar_refresh_rate=0)
```

Note:

- In Google Colab notebooks, faster refresh rates (lower number) is known to crash them because of their screen refresh rates. Lightning will set it to 20 in these environments if the user does not provide a value.
- This argument is ignored if a custom callback is passed to `callbacks`.

8.7.39 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:
```

8.7.40 replace_sampler_ddp

Enables auto adding of distributed sampler. By default it will add `shuffle=True` for train sampler and `shuffle=False` for val/test sampler. If you want to customize it, you can set `replace_sampler_ddp=False` and add your own distributed sampler. If `replace_sampler_ddp=True` and a distributed sampler was already added, Lightning will not replace the existing one.

```
# 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)
```

8.7.41 resume_from_checkpoint

To resume training from a specific checkpoint pass in the path here. If resuming from a mid-epoch checkpoint, training will start from the beginning of the next epoch.

```
# 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')
```

8.7.42 sync_batchnorm

Enable synchronization between batchnorm layers across all GPUs.

```
trainer = Trainer(sync_batchnorm=True)
```

8.7.43 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)
```

8.7.44 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 `DistributedSampler`, Lightning automatically does it for you.

This parameter can be either 1 or 8.

Example:

```
# 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])
```

(continues on next page)

(continued from previous page)

```
# 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
```

8.7.45 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)

    # remember to detach() hiddens.
    # If you don't, you will get a RuntimeError: Trying to backward through
    # the graph a second time...
```

(continues on next page)

(continued from previous page)

```
    # Using hiddens.detach() allows each split to be disconnected.

    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
```

8.7.46 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)
```

8.7.47 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(dirpath='some/path')
trainer = Trainer(
    callbacks=[checkpoint],
    weights_save_path='my/path'
)
```

8.7.48 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)
```

8.8 Trainer class API

8.8.1 Methods

init

Trainer.**__init__**(*logger=True*, *checkpoint_callback=True*, *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=None*, *overfit_batches=0.0*, *track_grad_norm=-1*, *check_val_every_n_epoch=1*, *fast_dev_run=False*, *accumulate_grad_batches=1*, *max_epochs=None*, *min_epochs=None*, *max_steps=None*, *min_steps=None*, *limit_train_batches=1.0*, *limit_val_batches=1.0*, *limit_test_batches=1.0*, *limit_predict_batches=1.0*, *val_check_interval=1.0*, *flush_logs_every_n_steps=100*, *log_every_n_steps=50*, *accelerator=None*, *sync_batchnorm=False*, *precision=32*, *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*, *plugins=None*, *amp_backend='native'*, *amp_level='O2'*, *distributed_backend=None*, *automatic_optimization=None*, *move_metrics_to_cpu=False*, *enable_pl_optimizer=None*, *multiple_trainloader_mode='max_size_cycle'*, *stochastic_weight_avg=False*)

Customize every aspect of training via flags

Parameters

- **accelerator**¶ (Union[str, Accelerator, None]) – Previously known as distributed_backend (dp, ddp, ddp2, etc...). Can also take in an accelerator object for custom hardware.
- **accumulate_grad_batches**¶ (Union[int, Dict[int, int], List[list]]) – Accumulates grads every k batches or as set up in the dict.
- **amp_backend**¶ (str) – The mixed precision backend to use ("native" or "apex")
- **amp_level**¶ (str) – The optimization level to use (O1, O2, etc...).
- **auto_lr_find**¶ (Union[bool, str]) – If set to True, will make trainer.tune() run a learning rate finder, trying to optimize initial learning for faster convergence. trainer.tune() method will set the suggested 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.
- **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.
- **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.
- **benchmark**¶ (bool) – If true enables cudnn.benchmark.
- **callbacks**¶ (Union[List[*Callback*], *Callback*, None]) – Add a callback or list of callbacks.
- **checkpoint_callback**¶ (bool) – If `True`, enable checkpointing. It will configure a default ModelCheckpoint callback if there is no user-defined ModelCheckpoint in *callbacks*. Default: `True`.

  Warning: Passing a ModelCheckpoint instance to this argument is deprecated since v1.1 and will be unsupported from v1.3. Use *callbacks* argument instead.

- **check_val_every_n_epoch**¶ (int) – Check val every n train epochs.
- **default_root_dir**¶ (Optional[str]) – Default path for logs and weights when no logger/ckpt_callback passed. Default: `os.getcwd()`. Can be remote file paths such as *s3://mybucket/path* or 'hdfs://path/'
- **deterministic**¶ (bool) – If true enables cudnn.deterministic.
- **distributed_backend**¶ (Optional[str]) – deprecated. Please use 'accelerator'
- **fast_dev_run**¶ (Union[int, bool]) – runs n if set to `n` (int) else 1 if set to `True` batch(es) of train, val and test to find any bugs (ie: a sort of unit test).
- **flush_logs_every_n_steps**¶ (int) – How often to flush logs to disk (defaults to every 100 steps).
- **gpus**¶ (Union[int, str, List[int], None]) – number of gpus to train on (int) or which GPUs to train on (list or str) applied per node
- **gradient_clip_val**¶ (float) – 0 means don't clip.
- **limit_train_batches**¶ (Union[int, float]) – How much of training dataset to check (floats = percent, int = num_batches)

- **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)
- **logger**¶ (Union[*LightningLoggerBase*, Iterable[*LightningLoggerBase*], bool]) – Logger (or iterable collection of loggers) for experiment tracking.
- **log_gpu_memory**¶ (Optional[str]) – None, 'min_max', 'all'. Might slow performance
- **log_every_n_steps**¶ (int) – How often to log within steps (defaults to every 50 steps).
- **automatic_optimization**¶ (Optional[bool]) – If False you are responsible for calling .backward, .step, zero_grad in LightningModule. This argument has been moved to LightningModule. It is deprecated here in v1.1 and will be removed in v1.3.
- **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
- **process_position**¶ (int) – orders the progress bar when running multiple models on same machine.
- **progress_bar_refresh_rate**¶ (Optional[int]) – How often to refresh progress bar (in steps). Value `0` disables progress bar. Ignored when a custom progress bar is passed to `callbacks`. Default: None, means a suitable value will be chosen based on the environment (terminal, Google COLAB, etc.).
- **profiler**¶ (Union[*BaseProfiler*, bool, str, None]) – To profile individual steps during training and assist in identifying bottlenecks. Passing bool value is deprecated in v1.1 and will be removed in v1.3.
- **overfit_batches**¶ (Union[int, float]) – Overfit a percent of training data (float) or a set number of batches (int). Default: 0.0
- **plugins**¶ (Union[Plugin, str, list, None]) – Plugins allow modification of core behavior like ddp and amp, and enable custom lightning plugins.
- **precision**¶ (int) – Full precision (32), half precision (16). Can be used on CPU, GPU or TPUs.
- **max_epochs**¶ (Optional[int]) – Stop training once this number of epochs is reached. Disabled by default (None). If both max_epochs and max_steps are not specified, defaults to `max_epochs` = 1000.
- **min_epochs**¶ (Optional[int]) – Force training for at least these many epochs. Disabled by default (None). If both min_epochs and min_steps are not specified, defaults to `min_epochs` = 1.
- **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).
- **num_nodes**¶ (int) – number of GPU nodes for distributed training.
- **num_processes**¶ (int) – number of processes for distributed training with distributed_backend="ddp_cpu"

- **num_sanity_val_steps**¶ (int) – Sanity check runs n validation batches before starting the training routine. Set it to *-1* to run all batches in all validation dataloaders. Default: 2
- **reload_dataloaders_every_epoch**¶ (bool) – Set to True to reload dataloaders every epoch.
- **replace_sampler_ddp**¶ (bool) – Explicitly enables or disables sampler replacement. If not specified this will toggled automatically when DDP is used. By default it will add `shuffle=True` for train sampler and `shuffle=False` for val/test sampler. If you want to customize it, you can set `replace_sampler_ddp=False` and add your own distributed sampler.
- **resume_from_checkpoint**¶ (Union[str, Path, None]) – Path/URL of the checkpoint from which training is resumed. If there is no checkpoint file at the path, start from scratch. If resuming from mid-epoch checkpoint, training will start from the beginning of the next epoch.
- **sync_batchnorm**¶ (bool) – Synchronize batch norm layers between process groups/whole world.
- **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.
- **tpu_cores**¶ (Union[int, str, List[int], None]) – How many TPU cores to train on (1 or 8) / Single TPU to train on [1]
- **track_grad_norm**¶ (Union[int, float, str]) – -1 no tracking. Otherwise tracks that p-norm. May be set to 'inf' infinity-norm.
- **truncated_bptt_steps**¶ (Optional[int]) – Truncated back prop breaks performs backprop every k steps of much longer sequence.
- **val_check_interval**¶ (Union[int, float]) – How often to check the validation set. Use float to check within a training epoch, use int to check every n steps (batches).
- **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*. Can be remote file paths such as *s3://mybucket/path* or 'hdfs://path/' Defaults to *default_root_dir*.
- **move_metrics_to_cpu**¶ (bool) – Whether to force internal logged metrics to be moved to cpu. This can save some gpu memory, but can make training slower. Use with attention.
- **enable_pl_optimizer**¶ (Optional[bool]) – If True, each optimizer will be wrapped by *pytorch_lightning.core.optimizer.LightningOptimizer*. It allows Lightning to handle AMP, TPU, accumulated_gradients, etc. .. warning:: Currently deprecated and it will be removed in v1.3
- **multiple_trainloader_mode**¶ (str) – How to loop over the datasets when there are multiple train loaders. In 'max_size_cycle' mode, the trainer ends one epoch when the largest dataset is traversed, and smaller datasets reload when running out of their data. In 'min_size' mode, all the datasets reload when reaching the minimum length of datasets.

- **stochastic_weight_avg**¶ (bool) – Whether to use *Stochastic Weight Averaging (SWA) <https://pytorch.org/blog/pytorch-1.6-now-includes-stochastic-weight-averaging/>_*

fit

Trainer.**fit**(*model*, *train_dataloader=None*, *val_dataloaders=None*, *datamodule=None*)

Runs the full optimization routine.

Parameters

- **datamodule**¶ (Optional[*LightningDataModule*]) – A instance of LightningDataModule.
- **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

test

Trainer.**test**(*model=None*, *test_dataloaders=None*, *ckpt_path='best'*, *verbose=True*, *datamodule=None*)

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

Parameters

- **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`.
- **datamodule**¶ (Optional[*LightningDataModule*]) – A instance of LightningDataModule.
- **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.
- **verbose**¶ (bool) – If True, prints the test results

Returns Returns a list of dictionaries, one for each test dataloader containing their respective metrics.

tune

Trainer.**tune**(*model*, *train_dataloader=None*, *val_dataloaders=None*, *datamodule=None*)

Runs routines to tune hyperparameters before training.

Parameters

- **datamodule**¶ (Optional[*LightningDataModule*]) – A instance of LightningDataModule.
- **model**¶ (*LightningModule*) – Model to tune.

- **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

8.8.2 Properties

callback_metrics

The metrics available to callbacks. These are automatically set when you log via *self.log*

```
def training_step(self, batch, batch_idx):
    self.log('a_val', 2)


callback_metrics = trainer.callback_metrics
assert callback_metrics['a_val'] == 2
```

current_epoch

The current epoch

```
def training_step(self, batch, batch_idx):
    current_epoch = self.trainer.current_epoch
    if current_epoch > 100:
        # do something
        pass
```

logger (p)

The current logger being used. Here's an example using tensorboard

```
def training_step(self, batch, batch_idx):
    logger = self.trainer.logger
    tensorboard = logger.experiment
```

logged_metrics

The metrics sent to the logger (visualizer).

```
def training_step(self, batch, batch_idx):
    self.log('a_val', 2, log=True)


logged_metrics = trainer.logged_metrics
assert logged_metrics['a_val'] == 2
```

log_dir

The directory for the current experiment. Use this to save images to, etc...

```
def training_step(self, batch, batch_idx):
    img = ...
    save_img(img, self.trainer.log_dir)
```

is_global_zero

Whether this process is the global zero in multi-node training

```
def training_step(self, batch, batch_idx):
    if self.trainer.is_global_zero:
        print('in node 0, accelerator 0')
```

progress_bar_metrics

The metrics sent to the progress bar.

```
def training_step(self, batch, batch_idx):
    self.log('a_val', 2, prog_bar=True)


progress_bar_metrics = trainer.progress_bar_metrics
assert progress_bar_metrics['a_val'] == 2
```

CHAPTER

NINE

ACCELERATORS

Accelerators connect a Lightning Trainer to arbitrary accelerators (CPUs, GPUs, TPUs, etc). Accelerators also manage distributed accelerators (like DP, DDP, HPC cluster).

Accelerators can also be configured to run on arbitrary clusters using Plugins or to link up to arbitrary computational strategies like 16-bit precision via AMP and Apex.

For help setting up custom plugin/accelerator please reach out to us at support@pytorchlightning.ai

CHAPTER

TEN

CALLBACK

A callback is a self-contained program that can be reused across projects.

Lightning has a callback system to execute callbacks when needed. Callbacks should capture NON-ESSENTIAL logic that is NOT required for your *lightning module* to run.

Here's the flow of how the callback hooks are executed:

An overall Lightning system should have:

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

Example:

```
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')

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

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

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

10.1 Examples

You can do pretty much anything with callbacks.

- Add a MLP to fine-tune self-supervised networks.
- Find how to modify an image input to trick the classification result.
- Interpolate the latent space of any variational model.
- Log images to Tensorboard for any model.

10.2 Built-in Callbacks

Lightning has a few built-in callbacks.

Note: For a richer collection of callbacks, check out our bolts library.

BackboneFinetuning	Finetune a backbone model based on a learning rate user-defined scheduling.
BaseFinetuning	This class implements the base logic for writing your own Finetuning Callback.
Callback	Abstract base class used to build new callbacks.
EarlyStopping	Monitor a metric and stop training when it stops improving.
GPUStatsMonitor	Automatically monitors and logs GPU stats during training stage.
GradientAccumulationScheduler	Change gradient accumulation factor according to scheduling.
LambdaCallback	Create a simple callback on the fly using lambda functions.
LearningRateMonitor	Automatically monitor and logs learning rate for learning rate schedulers during training.
ModelCheckpoint	Save the model after every epoch by monitoring a quantity.
ModelPruning	Model pruning Callback, using PyTorch's prune utilities.
ProgressBar	This is the default progress bar used by Lightning.
ProgressBarBase	The base class for progress bars in Lightning.
QuantizationAwareTraining	Quantization allows speeding up inference and decreasing memory requirements by performing computations and storing tensors at lower bitwidths (such as INT8 or FLOAT16) than floating point precision.
StochasticWeightAveraging	Implements the Stochastic Weight Averaging (SWA) Callback to average a model.

10.2.1 BackboneFinetuning

class pytorch_lightning.callbacks.**BackboneFinetuning**(*unfreeze_backbone_at_epoch=10*, *lambda_func=<function multiplicative>*, *backbone_initial_ratio_lr=0.1*, *backbone_initial_lr=None*, *should_align=True*, *initial_denom_lr=10.0*, *train_bn=True*, *verbose=False*, *round=12*)

Bases: pytorch_lightning.callbacks.finetuning.BaseFinetuning

Finetune a backbone model based on a learning rate user-defined scheduling. When the backbone learning rate reaches the current model learning rate and `should_align` is set to True, it will align with it for the rest of the training.

Parameters

- **unfreeze_backbone_at_epoch** (int) – Epoch at which the backbone will be unfreezed.
- **lambda_func** (Callable) – Scheduling function for increasing backbone learning rate.
- **backbone_initial_ratio_lr** (float) – Used to scale down the backbone learning rate compared to rest of model
- **backbone_initial_lr** (Optional[float]) – Optional, Inital learning rate for the backbone. By default, we will use current_learning / backbone_initial_ratio_lr
- **should_align** (bool) – Wheter to align with current learning rate when backbone learning reaches it.
- **initial_denom_lr** (float) – When unfreezing the backbone, the intial learning rate will current_learning_rate / initial_denom_lr.
- **train_bn** (bool) – Wheter to make Batch Normalization trainable.
- **verbose** (bool) – Display current learning rate for model and backbone
- **round** (int) – Precision for displaying learning rate

Example:

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.callbacks import BackboneFinetuning
>>> multiplicative = lambda epoch: 1.5
>>> backbone_finetuning = BackboneFinetuning(200, multiplicative)
>>> trainer = Trainer(callbacks=[backbone_finetuning])
```

finetune_function(*pl_module*, *epoch*, *optimizer*, *opt_idx*)
Called when the epoch begins.

freeze_before_training(*pl_module*)
Override to add your freeze logic

on_fit_start(*trainer*, *pl_module*)

Raises **MisconfigurationException** – If LightningModule has no nn.Module *backbone* attribute.

10.2.2 BaseFinetuning

class pytorch_lightning.callbacks.**BaseFinetuning**

Bases: *pytorch_lightning.callbacks.base.Callback*

This class implements the base logic for writing your own Finetuning Callback.

Override `freeze_before_training` and `finetune_function` methods with your own logic.

`freeze_before_training`: This method is called before `configure_optimizers` and should be used to freeze any modules parameters.

`finetune_function`: This method is called on every train epoch start and should be used to `unfreeze` any parameters. Those parameters needs to be added in a new `param_group` within the optimizer.

Note: Make sure to filter the parameters based on `requires_grad`.

Example:

```
class MyModel(LightningModule)

    ...

    def configure_optimizer(self):
        # Make sure to filter the parameters based on `requires_grad`
        return Adam(filter(lambda p: p.requires_grad, self.parameters))

class FeatureExtractorFreezeUnfreeze(BaseFinetuning):

    def __init__(self, unfreeze_at_epoch=10)
        self._unfreeze_at_epoch = unfreeze_at_epoch

    def freeze_before_training(self, pl_module):
        # freeze any module you want
        # Here, we are freezing ``feature_extractor``
        self.freeze(pl_module.feature_extractor)

    def finetune_function(self, pl_module, current_epoch, optimizer, optimizer_
↪idx):
        # When `current_epoch` is 10, feature_extractor will start training.
        if current_epoch == self._unfreeze_at_epoch:
            self.unfreeze_and_add_param_group(
                module=pl_module.feature_extractor,
                optimizer=optimizer,
                train_bn=True,
            )
```

static filter_on_optimizer(*optimizer*, *params*)

This function is used to exclude any parameter which already exists in this optimizer

Parameters

- **optimizer** (`Optimizer`) – Optimizer used for parameter exclusion
- **params** (`Iterable`) – Iterable of parameters used to check against the provided optimizer

Return type `List`

Returns List of parameters not contained in this optimizer param groups

static filter_params(*modules*, *train_bn=True*, *requires_grad=True*)
Yields the *requires_grad* parameters of a given module or list of modules.

Parameters

- **modules** (Union[Module, Iterable[Union[Module, Iterable]]]) – A given module or an iterable of modules
- **train_bn** (bool) – Whether to train BatchNorm module
- **requires_grad** (bool) – Whether to create a generator for trainable or non-trainable parameters.

Return type Generator

Returns Generator

finetune_function(*pl_module*, *epoch*, *optimizer*, *opt_idx*)
Override to add your unfreeze logic

static flatten_modules(*modules*)
This function is used to flatten a module or an iterable of modules into a list of its modules.

Parameters **modules** (Union[Module, Iterable[Union[Module, Iterable]]]) – A given module or an iterable of modules

Return type List[Module]

Returns List of modules

static freeze(*modules*, *train_bn=True*)
Freezes the parameters of the provided modules

Parameters

- **modules** (Union[Module, Iterable[Union[Module, Iterable]]]) – A given module or an iterable of modules
- **train_bn** (bool) – If True, leave the BatchNorm layers in training mode

Return type None

Returns None

freeze_before_training(*pl_module*)
Override to add your freeze logic

static make_trainable(*modules*)
Unfreezes the parameters of the provided modules

Parameters **modules** (Union[Module, Iterable[Union[Module, Iterable]]]) – A given module or an iterable of modules

Return type None

on_before_accelerator_backend_setup(*trainer*, *pl_module*)
Called before accelerator is being setup

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

static unfreeze_and_add_param_group(*modules*, *optimizer*, *lr=None*, *initial_denom_lr=10.0*, *train_bn=True*)
Unfreezes a module and adds its parameters to an optimizer.

Parameters

- **modules** (Union[Module, Iterable[Union[Module, Iterable]]]) – A module or iterable of modules to unfreeze. Their parameters will be added to an optimizer as a new param group.
- **optimizer** (Optimizer) – The provided optimizer will receive new parameters and will add them to *add_param_group*
- **lr** (Optional[float]) – Learning rate for the new param group.
- **initial_denom_lr** (float) – If no lr is provided, the learning from the first param group will be used and divided by initial_denom_lr.
- **train_bn** (bool) – Whether to train the BatchNormalization layers.

Return type None

Returns None

10.2.3 Callback

class pytorch_lightning.callbacks.**Callback**

Bases: abc.ABC

Abstract base class used to build new callbacks.

Subclass this class and override any of the relevant hooks

on_after_backward(*trainer*, *pl_module*)

Called after `loss.backward()` and before optimizers do anything.

Return type None

on_batch_end(*trainer*, *pl_module*)

Called when the training batch ends.

Return type None

on_batch_start(*trainer*, *pl_module*)

Called when the training batch begins.

Return type None

on_before_accelerator_backend_setup(*trainer*, *pl_module*)

Called before accelerator is being setup

Return type None

on_before_zero_grad(*trainer*, *pl_module*, *optimizer*)

Called after `optimizer.step()` and before `optimizer.zero_grad()`.

Return type None

on_epoch_end(*trainer*, *pl_module*)

Called when the epoch ends.

Return type None

on_epoch_start(*trainer*, *pl_module*)

Called when the epoch begins.

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

on_load_checkpoint(*checkpointed_state*)
Called when loading a model checkpoint, use to reload state.

Return type None

on_pretrain_routine_end(*trainer*, *pl_module*)
Called when the pretrain routine ends.

Return type None

on_pretrain_routine_start(*trainer*, *pl_module*)
Called when the pretrain routine begins.

Return type None

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

Return type None

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

Return type None

on_save_checkpoint(*trainer*, *pl_module*)
Called when saving a model checkpoint, use to persist state.

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

on_train_batch_end(*trainer*, *pl_module*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)
Called when the train batch ends.

Return type None

on_train_batch_start(*trainer*, *pl_module*, *batch*, *batch_idx*, *dataloader_idx*)
Called when the train batch begins.

Return type None

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

Return type None

on_train_epoch_end(*trainer*, *pl_module*, *outputs*)
Called when the train epoch ends.

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

10.2.4 EarlyStopping

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

Bases: *pytorch_lightning.callbacks.base.Callback*

Monitor a metric and stop training when it stops improving.

Parameters

- **monitor** (str) – quantity to be monitored. Default: 'early_stop_on'.
- **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: 0.0.
- **patience** (int) – number of validation epochs with no improvement after which training will be stopped. Default: 3.
- **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.

  Warning: Setting mode='auto' has been deprecated in v1.1 and will be removed in v1.3.

- **strict** (bool) – whether to crash the training if *monitor* is not found in the validation metrics. Default: True.

Raises

- **MisconfigurationException** – If mode is none of "min", "max", and "auto".
- **RuntimeError** – If the metric monitor is not available.

Example:

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

on_load_checkpoint(*checkpointed_state*)
Called when loading a model checkpoint, use to reload state.

on_save_checkpoint(*trainer*, *pl_module*)
Called when saving a model checkpoint, use to persist state.

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

10.2.5 GPUStatsMonitor

class pytorch_lightning.callbacks.**GPUStatsMonitor**(*memory_utilization=True*, *gpu_utilization=True*, *intra_step_time=False*, *inter_step_time=False*, *fan_speed=False*, *temperature=False*)

Bases: *pytorch_lightning.callbacks.base.Callback*

Automatically monitors and logs GPU stats during training stage. `GPUStatsMonitor` is a callback and in order to use it you need to assign a logger in the `Trainer`.

Parameters

- **memory_utilization** (`bool`) – Set to `True` to monitor used, free and percentage of memory utilization at the start and end of each step. Default: `True`.
- **gpu_utilization** (`bool`) – Set to `True` to monitor percentage of GPU utilization at the start and end of each step. Default: `True`.
- **intra_step_time** (`bool`) – Set to `True` to monitor the time of each step. Default: `False`.
- **inter_step_time** (`bool`) – Set to `True` to monitor the time between the end of one step and the start of the next step. Default: `False`.
- **fan_speed** (`bool`) – Set to `True` to monitor percentage of fan speed. Default: `False`.
- **temperature** (`bool`) – Set to `True` to monitor the memory and gpu temperature in degree Celsius. Default: `False`.

Raises **MisconfigurationException** – If NVIDIA driver is not installed, not running on GPUs, or `Trainer` has no logger.

Example:

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.callbacks import GPUStatsMonitor
>>> gpu_stats = GPUStatsMonitor()
>>> trainer = Trainer(callbacks=[gpu_stats])
```

GPU stats are mainly based on *nvidia-smi –query-gpu* command. The description of the queries is as follows:

- **fan.speed** – The fan speed value is the percent of maximum speed that the device's fan is currently intended to run at. It ranges from 0 to 100 %. Note: The reported speed is the intended fan speed. If the fan is

physically blocked and unable to spin, this output will not match the actual fan speed. Many parts do not report fan speeds because they rely on cooling via fans in the surrounding enclosure.

- **memory.used** – Total memory allocated by active contexts.
- **memory.free** – Total free memory.
- **utilization.gpu** – Percent of time over the past sample period during which one or more kernels was executing on the GPU. The sample period may be between 1 second and 1/6 second depending on the product.
- **utilization.memory** – Percent of time over the past sample period during which global (device) memory was being read or written. The sample period may be between 1 second and 1/6 second depending on the product.
- **temperature.gpu** – Core GPU temperature, in degrees C.
- **temperature.memory** – HBM memory temperature, in degrees C.

on_train_batch_end(*trainer*, **args*, ***kwargs*)
Called when the train batch ends.

on_train_batch_start(*trainer*, **args*, ***kwargs*)
Called when the train batch begins.

on_train_epoch_start(**args*, ***kwargs*)
Called when the train epoch begins.

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

10.2.6 GradientAccumulationScheduler

class pytorch_lightning.callbacks.**GradientAccumulationScheduler**(*scheduling*)
Bases: *pytorch_lightning.callbacks.base.Callback*

Change gradient accumulation factor according to scheduling.

Parameters **scheduling** (Dict[int, int]) – scheduling in format {epoch: accumulation_factor}

Raises

- **TypeError** – If `scheduling` is an empty `dict`, or not all keys and values of `scheduling` are integers.
- **IndexError** – If `minimal_epoch` is less than 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.

10.2.7 LambdaCallback

class pytorch_lightning.callbacks.**LambdaCallback**(*on_before_accelerator_backend_setup=None*, *setup=None*, *teardown=None*, *on_init_start=None*, *on_init_end=None*, *on_fit_start=None*, *on_fit_end=None*, *on_sanity_check_start=None*, *on_sanity_check_end=None*, *on_train_batch_start=None*, *on_train_batch_end=None*, *on_train_epoch_start=None*, *on_train_epoch_end=None*, *on_validation_epoch_start=None*, *on_validation_epoch_end=None*, *on_test_epoch_start=None*, *on_test_epoch_end=None*, *on_epoch_start=None*, *on_epoch_end=None*, *on_batch_start=None*, *on_validation_batch_start=None*, *on_validation_batch_end=None*, *on_test_batch_start=None*, *on_test_batch_end=None*, *on_batch_end=None*, *on_train_start=None*, *on_train_end=None*, *on_pretrain_routine_start=None*, *on_pretrain_routine_end=None*, *on_validation_start=None*, *on_validation_end=None*, *on_test_start=None*, *on_test_end=None*, *on_keyboard_interrupt=None*, *on_save_checkpoint=None*, *on_load_checkpoint=None*, *on_after_backward=None*, *on_before_zero_grad=None*)

Bases: *pytorch_lightning.callbacks.base.Callback*

Create a simple callback on the fly using lambda functions.

Parameters ****kwargs**¶ – hooks supported by *Callback*

Example:

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.callbacks import LambdaCallback
>>> trainer = Trainer(callbacks=[LambdaCallback(setup=lambda *args: print('setup
↪'))])
```

10.2.8 LearningRateMonitor

class pytorch_lightning.callbacks.**LearningRateMonitor**(*logging_interval=None*, *log_momentum=False*)

Bases: *pytorch_lightning.callbacks.base.Callback*

Automatically monitor and logs learning rate for learning rate schedulers during training.

Parameters

- **logging_interval** (Optional[str]) – set to 'epoch' or 'step' to log lr of all optimizers at the same interval, set to None to log at individual interval according to the interval key of each scheduler. Defaults to None.
- **log_momentum** (bool) – option to also log the momentum values of the optimizer, if the optimizer has the momentum or betas attribute. Defaults to False.

Raises **MisconfigurationException** – If logging_interval is none of "step", "epoch", or None.

Example:

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.callbacks import LearningRateMonitor
>>> lr_monitor = LearningRateMonitor(logging_interval='step')
>>> trainer = Trainer(callbacks=[lr_monitor])
```

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_scheduler.LambdaLR(optimizer, ...)
        'name': 'my_logging_name'
    }
    return [optimizer], [lr_scheduler]
```

on_train_batch_start(*trainer*, **args*, ***kwargs*)
Called when the train batch begins.

on_train_epoch_start(*trainer*, **args*, ***kwargs*)
Called when the train epoch begins.

on_train_start(*trainer*, **args*, ***kwargs*)
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

Raises **MisconfigurationException** – If Trainer has no logger.

10.2.9 ModelCheckpoint

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

Bases: *pytorch_lightning.callbacks.base.Callback*

Save the model after every epoch by monitoring a quantity.

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

- **dirpath**¶ (`Union[str, Path, None]`) – directory to save the model file.

  Example:

  ```
  # custom path
  # saves a file like: my/path/epoch=0-step=10.ckpt
  >>> checkpoint_callback = ModelCheckpoint(dirpath='my/path/')
  ```

  By default, dirpath is `None` and will be set at runtime to the location specified by *Trainer*'s *default_root_dir* or *weights_save_path* arguments, and if the Trainer uses a logger, the path will also contain logger name and version.

- **filename**¶ (`Optional[str]`) – checkpoint filename. Can contain named formatting options to be auto-filled.

  Example:

  ```
  # save any arbitrary metrics like `val_loss`, etc. in name
  # saves a file like: my/path/epoch=2-val_loss=0.02-other_metric=0.
  ↪03.ckpt
  >>> checkpoint_callback = ModelCheckpoint(
  ...     dirpath='my/path',
  ...     filename='{epoch}-{val_loss:.2f}-{other_metric:.2f}'
  ... )
  ```

  By default, filename is `None` and will be set to `'{epoch}-{step}'`.

- **monitor**¶ (`Optional[str]`) – quantity to monitor. By default it is `None` which saves a checkpoint only for the last epoch.

- **verbose**¶ (`bool`) – verbosity mode. Default: `False`.

- **save_last**¶ (`Optional[bool]`) – When `True`, always saves the model at the end of the epoch to a file *last.ckpt*. Default: `None`.

- **save_top_k**¶ (`Optional[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 `v1`.

- **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.

> **Warning:** Setting `mode='auto'` has been deprecated in v1.1 and will be removed in v1.3.

- **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.
- **prefix** (str) – A string to put at the beginning of checkpoint filename.

> **Warning:** This argument has been deprecated in v1.1 and will be removed in v1.3

Note: For extra customization, ModelCheckpoint includes the following attributes:

- `CHECKPOINT_JOIN_CHAR = "-"`
- `CHECKPOINT_NAME_LAST = "last"`
- `FILE_EXTENSION = ".ckpt"`
- `STARTING_VERSION = 1`

For example, you can change the default last checkpoint name by doing `checkpoint_callback.CHECKPOINT_NAME_LAST = "{epoch}-last"`

Raises

- **MisconfigurationException** – If `save_top_k` is neither `None` nor more than or equal to `-1`, if `monitor` is `None` and `save_top_k` is none of `None`, `-1`, and `0`, or if `mode` is none of `"min"`, `"max"`, and `"auto"`.
- **ValueError** – If `trainer.save_checkpoint` is `None`.

Example:

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

# saves checkpoints to 'my/path/' at every epoch
>>> checkpoint_callback = ModelCheckpoint(dirpath='my/path/')
>>> trainer = Trainer(callbacks=[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(
...     monitor='val_loss',
...     dirpath='my/path/',
...     filename='sample-mnist-{epoch:02d}-{val_loss:.2f}'
... )
```

(continues on next page)

(continued from previous page)

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

file_exists (*filepath*, *trainer*)
Checks if a file exists on rank 0 and broadcasts the result to all other ranks, preventing the internal state to diverge between ranks.

Return type bool

format_checkpoint_name (*epoch*, *step*, *metrics*, *ver=None*)
Generate a filename according to the defined template.

Example:

```
>>> tmpdir = os.path.dirname(__file__)
>>> ckpt = ModelCheckpoint(dirpath=tmpdir, filename='{epoch}')
>>> os.path.basename(ckpt.format_checkpoint_name(0, 1, metrics={}))
'epoch=0.ckpt'
>>> ckpt = ModelCheckpoint(dirpath=tmpdir, filename='{epoch:03d}')
>>> os.path.basename(ckpt.format_checkpoint_name(5, 2, metrics={}))
'epoch=005.ckpt'
>>> ckpt = ModelCheckpoint(dirpath=tmpdir, filename='{epoch}-{val_loss:.2f}')
>>> os.path.basename(ckpt.format_checkpoint_name(2, 3, metrics=dict(val_
↪loss=0.123456)))
'epoch=2-val_loss=0.12.ckpt'
>>> ckpt = ModelCheckpoint(dirpath=tmpdir, filename='{missing:d}')
>>> os.path.basename(ckpt.format_checkpoint_name(0, 4, metrics={}))
'missing=0.ckpt'
>>> ckpt = ModelCheckpoint(filename='{step}')
>>> os.path.basename(ckpt.format_checkpoint_name(0, 0, {}))
'step=0.ckpt'
```

Return type str

on_load_checkpoint (*checkpointed_state*)
Called when loading a model checkpoint, use to reload state.

on_pretrain_routine_start (*trainer*, *pl_module*)
When pretrain routine starts we build the ckpt dir on the fly

on_save_checkpoint (*trainer*, *pl_module*)
Called when saving a model checkpoint, use to persist state.

Return type Dict[str, Any]

on_validation_end (*trainer*, *pl_module*)
checkpoints can be saved at the end of the val loop

save_checkpoint (*trainer*, *pl_module*)
Performs the main logic around saving a checkpoint. This method runs on all ranks, it is the responsibility of *self.save_function* to handle correct behaviour in distributed training, i.e., saving only on rank 0.

to_yaml (*filepath=None*)
Saves the *best_k_models* dict containing the checkpoint paths with the corresponding scores to a YAML file.

10.2.10 ModelPruning

class pytorch_lightning.callbacks.**ModelPruning**(*pruning_fn*, *parameters_to_prune=None*, *parameter_names=None*, *use_global_unstructured=True*, *amount=0.5*, *apply_pruning=True*, *make_pruning_permanent=True*, *use_lottery_ticket_hypothesis=True*, *resample_parameters=False*, *pruning_dim=None*, *pruning_norm=None*, *verbose=0*)

Bases: *pytorch_lightning.callbacks.base.Callback*

Model pruning Callback, using PyTorch's prune utilities. This callback is responsible of pruning networks parameters during training.

To learn more about pruning with PyTorch, please take a look at this tutorial.

> **Warning:** `ModelPruning` is in beta and subject to change.

```
parameters_to_prune = [
    (model.mlp_1, "weight"),
    (model.mlp_2, "weight")
]

trainer = Trainer(callbacks=[
    ModelPruning(
        pruning_fn='l1_unstructured',
        parameters_to_prune=parameters_to_prune,
        amount=0.01,
        use_global_unstructured=True,
    )
])
```

When `parameters_to_prune` is `None`, `parameters_to_prune` will contain all parameters from the model. The user can override `filter_parameters_to_prune` to filter any `nn.Module` to be pruned.

Parameters

- **pruning_fn**¶ (`Union[Callable, str]`) – Function from torch.nn.utils.prune module or your own PyTorch `BasePruningMethod` subclass. Can also be string e.g. *"l1_unstructured"*. See pytorch docs for more details.
- **parameters_to_prune**¶ (`Union[List[Tuple[Module, str]], Tuple[Tuple[Module, str]], None]`) – List of tuples `(nn.Module, "parameter_name_string")`.
- **parameter_names**¶ (`Optional[List[str]]`) – List of parameter names to be pruned from the nn.Module. Can either be `"weight"` or `"bias"`.

- **use_global_unstructured** (bool) – Whether to apply pruning globally on the model. If parameters_to_prune is provided, global unstructured will be restricted on them.
- **amount** (Union[int, float, Callable[[int], Union[int, float]]]) – Quantity of parameters to prune:
  - float. Between 0.0 and 1.0. Represents the fraction of parameters to prune.
  - int. Represents the absolute number of parameters to prune.
  - Callable. For dynamic values. Will be called every epoch. Should return a value.
- **apply_pruning** (Union[bool, Callable[[int], bool]]) – Whether to apply pruning.
  - bool. Always apply it or not.
  - Callable[[epoch], bool]. For dynamic values. Will be called every epoch.
- **make_pruning_permanent** (bool) – Whether to remove all reparametrization pre-hooks and apply masks when training ends or the model is saved.
- **use_lottery_ticket_hypothesis** (Union[bool, Callable[[int], bool]]) – See The lottery ticket hypothesis:
  - bool. Whether to apply it or not.
  - Callable[[epoch], bool]. For dynamic values. Will be called every epoch.
- **resample_parameters** (bool) – Used with use_lottery_ticket_hypothesis. If True, the model parameters will be resampled, otherwise, the exact original parameters will be used.
- **pruning_dim** (Optional[int]) – If you are using a structured pruning method you need to specify the dimension.
- **pruning_norm** (Optional[int]) – If you are using ln_structured you need to specify the norm.
- **verbose** (int) – Verbosity level. 0 to disable, 1 to log overall sparsity, 2 to log per-layer sparsity

Raises **MisconfigurationException** – If parameter_names is neither "weight" nor "bias", if the provided pruning_fn is not supported, if pruning_dim is not provided when "unstructured", if pruning_norm is not provided when "ln_structured", if pruning_fn is neither str nor torch.nn.utils.prune.BasePruningMethod, or if amount is none of int, float and Callable.

apply_lottery_ticket_hypothesis()

Lottery ticket hypothesis algorithm (see page 2 of the paper):

1. Randomly initialize a neural network $f(x; \theta_0)$ (where $\theta_0 \sim \mathcal{D}_\theta$).
2. Train the network for j iterations, arriving at parameters θ_j.
3. Prune $p\%$ of the parameters in θ_j, creating a mask m.
4. Reset the remaining parameters to their values in θ_0, creating the winning ticket $f(x; m \odot \theta_0)$.

This function implements the step 4.

The resample_parameters argument can be used to reset the parameters with a new $\theta_z \sim \mathcal{D}_\theta$

apply_pruning(*amount*)

Applies pruning to parameters_to_prune.

filter_parameters_to_prune(*parameters_to_prune=None*)
This function can be overridden to control which module to prune.

Return type Union[List[Tuple[Module, str]], Tuple[Tuple[Module, str]], None]

make_pruning_permanent()
Makes parameters_to_prune current pruning permanent.

on_before_accelerator_backend_setup(*trainer*, *pl_module*)
Called before accelerator is being setup

on_save_checkpoint(**args*)
Called when saving a model checkpoint, use to persist state.

on_train_end(**args*)
Called when the train ends.

on_train_epoch_end(*trainer*, *pl_module*, **args*)
Called when the train epoch ends.

static sanitize_parameters_to_prune(*pl_module*, *parameters_to_prune=None*, *parameter_names=None*)
This function is responsible of sanitizing parameters_to_prune and parameter_names. If parameters_to_prune is None, it will be generated with all parameters of the model.

Raises **MisconfigurationException** – If parameters_to_prune doesn't exist in the model, or if parameters_to_prune is neither a list of tuple nor None.

Return type Union[List[Tuple[Module, str]], Tuple[Tuple[Module, str]]]

10.2.11 ProgressBar

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
```

(continues on next page)

(continued from previous page)

```
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_predict_tqdm()
Override this to customize the tqdm bar for predicting.

Return type *tqdm*

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_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*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)
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_batch_end(*trainer*, *pl_module*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)
Called when the train batch ends.

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*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)
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.

10.2.12 ProgressBarBase

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_train_batch_end(self, trainer, pl_module, outputs):
        super().on_train_batch_end(trainer, pl_module, outputs)  # 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_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*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)
Called when the test batch ends.

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

on_train_batch_end(*trainer*, *pl_module*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)
Called when the train batch ends.

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

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

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

property predict_batch_idx
The current batch index being processed during predicting. Use this to update your progress bar.

Return type int

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

Return type int

property total_predict_batches
The total number of predicting 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 predict dataloader is of infinite size.

Return type int

property total_test_batches
The total number of testing 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 validation 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

10.2.13 QuantizationAwareTraining

class pytorch_lightning.callbacks.**QuantizationAwareTraining**(*qconfig='fbgemm'*, *observer_type='average'*, *collect_quantization=None*, *modules_to_fuse=None*, *input_compatible=True*)

Bases: *pytorch_lightning.callbacks.base.Callback*

Quantization allows speeding up inference and decreasing memory requirements by performing computations and storing tensors at lower bitwidths (such as INT8 or FLOAT16) than floating point precision. We use native PyTorch API so for more information see Quantization.

Warning: QuantizationAwareTraining is in beta and subject to change.

Parameters

- **qconfig** (Union[str, QConfig]) – quantization configuration:
  - 'fbgemm' for server inference.
  - 'qnnpack' for mobile inference.
  - a custom torch.quantization.QConfig.
- **observer_type** (str) – allows switching between MovingAverageMinMaxObserver as "average" (default) and HistogramObserver as "histogram" which is more computationally expensive.
- **collect_quantization** (Union[Callable, int, None]) – count or custom function to collect quantization statistics:
  - **None (deafult). The quantization observer is called in each module forward** (useful for collecting extended statistic when useing image/data augmentation).
  - int. Use to set a fixed number of calls, starting from the beginning.
  - **Callable. Custom function with single trainer argument.** See this example to trigger only the last epoch:

```
def custom_trigger_last(trainer):
    return trainer.current_epoch == (trainer.max_epochs - 1)

QuantizationAwareTraining(collect_quantization=custom_trigger_
↪last)
```

(continues on next page)

(continued from previous page)

- **modules_to_fuse** (Optional[Sequence]) – allows you fuse a few layers together as shown in diagram to find which layer types can be fused, check https://github.com/pytorch/pytorch/pull/43286.
- **input_compatible** (bool) – preserve quant/dequant layers. This allows to feat any input as to the original model, but break compatibility to torchscript.

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

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

10.2.14 StochasticWeightAveraging

class pytorch_lightning.callbacks.**StochasticWeightAveraging**(*swa_epoch_start=0.8*, *swa_lrs=None*, *annealing_epochs=10*, *annealing_strategy='cos'*, *avg_fn=None*, *device=torch.device*)

Bases: *pytorch_lightning.callbacks.base.Callback*

Implements the Stochastic Weight Averaging (SWA) Callback to average a model.

Stochastic Weight Averaging was proposed in `Averaging Weights Leads to Wider Optima and Better Generalization` by Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov and Andrew Gordon Wilson (UAI 2018).

This documentation is highly inspired by PyTorch's work on SWA. The callback arguments follow the scheme defined in PyTorch's `swa_utils` package.

For a SWA explanation, please take a look here.

Warning: `StochasticWeightAveraging` is in beta and subject to change.

Warning: `StochasticWeightAveraging` is currently not supported for multiple optimizers/schedulers.

SWA can easily be activated directly from the Trainer as follow:

```
Trainer(stochastic_weight_avg=True)
```

Parameters

- **swa_epoch_start** (Union[int, float]) – If provided as int, the procedure will start from the `swa_epoch_start`-th epoch. If provided as float between 0 and 1, the procedure will start from `int(swa_epoch_start * max_epochs)` epoch

- **swa_lrs** (Union[float, list, None]) – the learning rate value for all param groups together or separately for each group.
- **annealing_epochs** (int) – number of epochs in the annealing phase (default: 10)
- **annealing_strategy** (str) – Specifies the annealing strategy (default: "cos"):
  - "cos". For cosine annealing.
  - "linear" For linear annealing
- **avg_fn** (Optional[Callable[[Tensor, Tensor, LongTensor], FloatTensor]]) – the averaging function used to update the parameters; the function must take in the current value of the AveragedModel parameter, the current value of model parameter and the number of models already averaged; if None, equally weighted average is used (default: None)
- **device** (Union[device, str, None]) – if provided, the averaged model will be stored on the device. When None is provided, it will infer the *device* from pl_module. (default: "cpu")

static avg_fn(*averaged_model_parameter*, *model_parameter*, *num_averaged*)
Adapted from https://github.com/pytorch/pytorch/blob/v1.7.1/torch/optim/swa_utils.py#L95-L97

Return type FloatTensor

on_before_accelerator_backend_setup(*trainer*, *pl_module*)
Called before accelerator is being setup

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

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

on_train_epoch_end(*trainer*, **args*)
Called when the train epoch ends.

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

reset_batch_norm_and_save_state(*pl_module*)
Adapted from https://github.com/pytorch/pytorch/blob/v1.7.1/torch/optim/swa_utils.py#L140-L154

reset_momenta()
Adapted from https://github.com/pytorch/pytorch/blob/v1.7.1/torch/optim/swa_utils.py#L164-L165

static update_parameters(*average_model*, *model*, *n_averaged*, *avg_fn*)
Adapted from https://github.com/pytorch/pytorch/blob/v1.7.1/torch/optim/swa_utils.py#L104-L112

10.3 Persisting State

Some callbacks require internal state in order to function properly. You can optionally choose to persist your callback's state as part of model checkpoint files using the callback hooks *on_save_checkpoint()* and *on_load_checkpoint()*. However, you must follow two constraints:

1. Your returned state must be able to be pickled.
2. You can only use one instance of that class in the Trainer callbacks list. We don't support persisting state for multiple callbacks of the same class.

10.4 Best Practices

The following are best practices when using/designing callbacks.

1. Callbacks should be isolated in their functionality.
2. Your callback should not rely on the behavior of other callbacks in order to work properly.
3. Do not manually call methods from the callback.
4. Directly calling methods (eg. *on_validation_end*) is strongly discouraged.
5. Whenever possible, your callbacks should not depend on the order in which they are executed.

10.5 Available Callback hooks

10.5.1 setup

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

Return type None

10.5.2 teardown

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

Return type None

10.5.3 on_init_start

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

Return type None

10.5.4 on_init_end

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

Return type None

10.5.5 on_fit_start

Callback.**on_save_checkpoint**(*trainer*, *pl_module*)
Called when saving a model checkpoint, use to persist state.

Return type None

10.5.6 on_fit_end

Callback.**on_fit_end**(*trainer*, *pl_module*)
Called when fit ends

Return type None

10.5.7 on_sanity_check_start

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

Return type None

10.5.8 on_sanity_check_end

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

Return type None

10.5.9 on_train_batch_start

Callback.**on_train_batch_start**(*trainer*, *pl_module*, *batch*, *batch_idx*, *dataloader_idx*)
Called when the train batch begins.

Return type None

10.5.10 on_train_batch_end

Callback.**on_train_batch_end**(*trainer*, *pl_module*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)
Called when the train batch ends.

Return type None

10.5.11 on_train_epoch_start

Callback.**on_train_epoch_start**(*trainer*, *pl_module*)
Called when the train epoch begins.

Return type None

10.5.12 on_train_epoch_end

Callback.**on_train_epoch_end**(*trainer*, *pl_module*, *outputs*)
Called when the train epoch ends.

Return type None

10.5.13 on_validation_epoch_start

Callback.**on_validation_epoch_start**(*trainer*, *pl_module*)
Called when the val epoch begins.

Return type None

10.5.14 on_validation_epoch_end

Callback.**on_validation_epoch_end**(*trainer*, *pl_module*)
Called when the val epoch ends.

Return type None

10.5.15 on_test_epoch_start

Callback.**on_test_epoch_start**(*trainer*, *pl_module*)
Called when the test epoch begins.

Return type None

10.5.16 on_test_epoch_end

Callback.**on_test_epoch_end**(*trainer*, *pl_module*)
Called when the test epoch ends.

Return type None

10.5.17 on_epoch_start

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

Return type None

10.5.18 on_epoch_end

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

Return type None

10.5.19 on_batch_start

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

Return type None

10.5.20 on_validation_batch_start

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

Return type None

10.5.21 on_validation_batch_end

Callback.**on_validation_batch_end**(*trainer*, *pl_module*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)
Called when the validation batch ends.

Return type None

10.5.22 on_test_batch_start

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

Return type None

10.5.23 on_test_batch_end

Callback.**on_test_batch_end**(*trainer*, *pl_module*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)
Called when the test batch ends.

Return type None

10.5.24 on_batch_end

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

Return type None

10.5.25 on_train_start

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

Return type None

10.5.26 on_train_end

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

Return type None

10.5.27 on_pretrain_routine_start

Callback.**on_pretrain_routine_start**(*trainer*, *pl_module*)
Called when the pretrain routine begins.

Return type None

10.5.28 on_pretrain_routine_end

Callback.**on_pretrain_routine_end**(*trainer*, *pl_module*)
Called when the pretrain routine ends.

Return type None

10.5.29 on_validation_start

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

Return type None

10.5.30 on_validation_end

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

Return type None

10.5.31 on_test_start

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

Return type None

10.5.32 on_test_end

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

Return type None

10.5.33 on_keyboard_interrupt

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

Return type None

10.5.34 on_save_checkpoint

Callback.**on_save_checkpoint**(*trainer*, *pl_module*)
Called when saving a model checkpoint, use to persist state.

Return type None

10.5.35 on_load_checkpoint

Callback.**on_load_checkpoint**(*checkpointed_state*)
Called when loading a model checkpoint, use to reload state.

Return type None

CHAPTER

ELEVEN

LIGHTNINGDATAMODULE

A datamodule is a shareable, reusable class that encapsulates all the steps needed to process data:

A datamodule encapsulates the five steps involved in data processing in PyTorch:

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

This class can then be shared and used anywhere:

```
from pl_bolts.datamodules import CIFAR10DataModule, ImagenetDataModule

model = LitClassifier()
trainer = Trainer()

imagenet = ImagenetDataModule()
trainer.fit(model, imagenet)

cifar10 = CIFAR10DataModule()
trainer.fit(model, cifar10)
```

11.1 Why do I need a DataModule?

In normal PyTorch code, the data cleaning/preparation is usually scattered across many files. This makes sharing and reusing the exact splits and transforms across projects impossible.

Datamodules are for you if you ever asked the questions:

- what splits did you use?
- what transforms did you use?
- what normalization did you use?
- how did you prepare/tokenize the data?

11.2 What is a DataModule

A DataModule is simply a collection of a train_dataloader, val_dataloader(s), test_dataloader(s) along with the matching transforms and data processing/downloads steps required.

Here's a simple PyTorch example:

```
# regular PyTorch
test_data = MNIST(my_path, train=False, download=True)
train_data = MNIST(my_path, train=True, download=True)
train_data, val_data = random_split(train_data, [55000, 5000])

train_loader = DataLoader(train_data, batch_size=32)
val_loader = DataLoader(val_data, batch_size=32)
test_loader = DataLoader(test_data, batch_size=32)
```

The equivalent DataModule just organizes the same exact code, but makes it reusable across projects.

```
class MNISTDataModule(pl.LightningDataModule):

    def __init__(self, data_dir: str = "path/to/dir", batch_size: int = 32):
        super().__init__()
        self.data_dir = data_dir
        self.batch_size = batch_size

    def setup(self, stage=None):
        self.mnist_test = MNIST(self.data_dir, train=False)
        mnist_full = MNIST(self.data_dir, train=True)
        self.mnist_train, self.mnist_val = random_split(mnist_full, [55000, 5000])

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

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

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

But now, as the complexity of your processing grows (transforms, multiple-GPU training), you can let Lightning handle those details for you while making this dataset reusable so you can share with colleagues or use in different projects.

```
mnist = MNISTDataModule(my_path)
model = LitClassifier()

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

Here's a more realistic, complex DataModule that shows how much more reusable the datamodule is.

```
import pytorch_lightning as pl
from torch.utils.data import random_split, DataLoader

# Note - you must have torchvision installed for this example
from torchvision.datasets import MNIST
from torchvision import transforms


class MNISTDataModule(pl.LightningDataModule):

    def __init__(self, data_dir: str = './'):
        super().__init__()
        self.data_dir = data_dir
        self.transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.1307,), (0.3081,))
        ])

        # self.dims is returned when you call dm.size()
        # Setting default dims here because we know them.
        # Could optionally be assigned dynamically in dm.setup()
        self.dims = (1, 28, 28)

    def prepare_data(self):
        # download
        MNIST(self.data_dir, train=True, download=True)
        MNIST(self.data_dir, train=False, download=True)

    def setup(self, stage=None):

        # Assign train/val datasets for use in dataloaders
        if stage == 'fit' or stage is None:
            mnist_full = MNIST(self.data_dir, train=True, transform=self.transform)
            self.mnist_train, self.mnist_val = random_split(mnist_full, [55000, 5000])

            # Optionally...
            # self.dims = tuple(self.mnist_train[0][0].shape)

        # Assign test dataset for use in dataloader(s)
        if stage == 'test' or stage is None:
            self.mnist_test = MNIST(self.data_dir, train=False, transform=self.
↪transform)

            # Optionally...
            # self.dims = tuple(self.mnist_test[0][0].shape)
```

(continues on next page)

(continued from previous page)

```
    def train_dataloader(self):
        return DataLoader(self.mnist_train, batch_size=32)

    def val_dataloader(self):
        return DataLoader(self.mnist_val, batch_size=32)

    def test_dataloader(self):
        return DataLoader(self.mnist_test, batch_size=32)
```

Note: `setup` expects a string arg `stage`. It is used to separate setup logic for `trainer.fit` and `trainer.test`.

11.3 LightningDataModule API

To define a DataModule define 5 methods:

- prepare_data (how to download(), tokenize, etc...)
- setup (how to split, etc...)
- train_dataloader
- val_dataloader(s)
- test_dataloader(s)

11.3.1 prepare_data

Use this method to do things that might write to disk or that need to be done only from a single process in distributed settings.

- download
- tokenize
- etc...

```
class MNISTDataModule(pl.LightningDataModule):
    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())
```

Warning: `prepare_data` is called from a single process (e.g. GPU 0). Do not use it to assign state (*self.x = y*).

11.3.2 setup

There are also data operations you might want to perform on every GPU. Use setup to do things like:

- count number of classes
- build vocabulary
- perform train/val/test splits
- apply transforms (defined explicitly in your datamodule or assigned in init)
- etc...

```
import pytorch_lightning as pl


class MNISTDataModule(pl.LightningDataModule):

    def setup(self, stage: Optional[str] = None):

        # Assign Train/val split(s) for use in Dataloaders
        if stage == 'fit' or stage is None:
            mnist_full = MNIST(
                self.data_dir,
                train=True,
                download=True,
                transform=self.transform
            )
            self.mnist_train, self.mnist_val = random_split(mnist_full, [55000, 5000])
            self.dims = self.mnist_train[0][0].shape

        # Assign Test split(s) for use in Dataloaders
        if stage == 'test' or stage is None:
            self.mnist_test = MNIST(
                self.data_dir,
                train=False,
                download=True,
                transform=self.transform
            )
            self.dims = getattr(self, 'dims', self.mnist_test[0][0].shape)
```

Warning: *setup* is called from every process. Setting state here is okay.

11.3.3 train_dataloader

Use this method to generate the train dataloader. Usually you just wrap the dataset you defined in `setup`.

```
import pytorch_lightning as pl


class MNISTDataModule(pl.LightningDataModule):
    def train_dataloader(self):
        return DataLoader(self.mnist_train, batch_size=64)
```

11.3.4 val_dataloader

Use this method to generate the val dataloader. Usually you just wrap the dataset you defined in `setup`.

```
import pytorch_lightning as pl


class MNISTDataModule(pl.LightningDataModule):
    def val_dataloader(self):
        return DataLoader(self.mnist_val, batch_size=64)
```

11.3.5 test_dataloader

Use this method to generate the test dataloader. Usually you just wrap the dataset you defined in `setup`.

```
import pytorch_lightning as pl


class MNISTDataModule(pl.LightningDataModule):
    def test_dataloader(self):
        return DataLoader(self.mnist_test, batch_size=64)
```

11.3.6 transfer_batch_to_device

Override to define how you want to move an arbitrary batch to a device.

```
class MNISTDataModule(LightningDataModule):
    def transfer_batch_to_device(self, batch, device):
        x = batch['x']
        x = CustomDataWrapper(x)
        batch['x'] = x.to(device)
        return batch
```

Note: This hook only runs on single GPU training and DDP (no data-parallel).

11.3.7 on_before_batch_transfer

Override to alter or apply augmentations to your batch before it is transferred to the device.

```
class MNISTDataModule(LightningDataModule):
    def on_before_batch_transfer(self, batch, dataloader_idx):
        batch['x'] = transforms(batch['x'])
        return batch
```

Warning: Currently dataloader_idx always returns 0 and will be updated to support the true idx in the future.

Note: This hook only runs on single GPU training and DDP (no data-parallel).

11.3.8 on_after_batch_transfer

Override to alter or apply augmentations to your batch after it is transferred to the device.

```
class MNISTDataModule(LightningDataModule):
    def on_after_batch_transfer(self, batch, dataloader_idx):
        batch['x'] = gpu_transforms(batch['x'])
        return batch
```

Warning: Currently `dataloader_idx` always returns 0 and will be updated to support the true `idx` in the future.

Note: This hook only runs on single GPU training and DDP (no data-parallel). This hook will also be called when using CPU device, so adding augmentations here or in `on_before_batch_transfer` means the same thing.

Note: To decouple your data from transforms you can parametrize them via `__init__`.

```
class MNISTDataModule(pl.LightningDataModule):
    def __init__(self, train_transforms, val_transforms, test_transforms):
        super().__init__()
        self.train_transforms = train_transforms
        self.val_transforms = val_transforms
        self.test_transforms = test_transforms
```

11.4 Using a DataModule

The recommended way to use a DataModule is simply:

```
dm = MNISTDataModule()
model = Model()
trainer.fit(model, dm)

trainer.test(datamodule=dm)
```

If you need information from the dataset to build your model, then run *prepare_data* and *setup* manually (Lightning still ensures the method runs on the correct devices)

```
dm = MNISTDataModule()
dm.prepare_data()
dm.setup('fit')

model = Model(num_classes=dm.num_classes, width=dm.width, vocab=dm.vocab)
trainer.fit(model, dm)

dm.setup('test')
trainer.test(datamodule=dm)
```

11.5 Datamodules without Lightning

You can of course use DataModules in plain PyTorch code as well.

```
# download, etc...
dm = MNISTDataModule()
dm.prepare_data()

# splits/transforms
dm.setup('fit')

# use data
for batch in dm.train_dataloader():
    ...
for batch in dm.val_dataloader():
    ...

# lazy load test data
dm.setup('test')
for batch in dm.test_dataloader():
    ...
```

But overall, DataModules encourage reproducibility by allowing all details of a dataset to be specified in a unified structure.

CHAPTER

TWELVE

LOGGING

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

```
from pytorch_lightning import loggers as pl_loggers

tb_logger = pl_loggers.TensorBoardLogger('logs/')
trainer = Trainer(logger=tb_logger)
```

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

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

Note: By default, lightning logs every 50 steps. Use Trainer flags to *Control logging frequency*.

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')*

12.1 Logging from a LightningModule

Lightning offers automatic log functionalities for logging scalars, or manual logging for anything else.

12.1.1 Automatic Logging

Use the *log()* method to log from anywhere in a *lightning module* and *callbacks* except functions with *batch_start* in their names.

```
def training_step(self, batch, batch_idx):
    self.log('my_metric', x)
```

Depending on where log is called from, Lightning auto-determines the correct logging mode for you. But of course you can override the default behavior by manually setting the *log()* parameters.

```
def training_step(self, batch, batch_idx):
    self.log('my_loss', loss, on_step=True, on_epoch=True, prog_bar=True, logger=True)
```

The *log()* method has a few options:

- *on_step*: Logs the metric at the current step. Defaults to *True* in *training_step()*, and *training_step_end()*.
- *on_epoch*: Automatically accumulates and logs at the end of the epoch. Defaults to True anywhere in validation or test loops, and in *training_epoch_end()*.
- *prog_bar*: Logs to the progress bar.
- *logger*: Logs to the logger like Tensorboard, or any other custom logger passed to the *Trainer*.

Note:

- Setting `on_epoch=True` will cache all your logged values during the full training epoch and perform a reduction *on_epoch_end*. We recommend using the *metrics* API when working with custom reduction.
- Setting both `on_step=True` and `on_epoch=True` will create two keys per metric you log with suffix `_step` and `_epoch`, respectively. You can refer to these keys e.g. in the *monitor* argument of *ModelCheckpoint* or in the graphs plotted to the logger of your choice.

If your work requires to log in an unsupported function, please open an issue with a clear description of why it is blocking you.

12.1.2 Manual logging

If you want to log anything that is not a scalar, like histograms, text, images, etc... you may need to use the logger object directly.

```
def training_step(...):
    ...
    # the logger you used (in this case tensorboard)
    tensorboard = self.logger.experiment
    tensorboard.add_image()
    tensorboard.add_histogram(...)
    tensorboard.add_figure(...)
```

12.1.3 Access your logs

Once your training starts, you can view the logs by using your favorite logger or booting up the Tensorboard logs:

```
tensorboard --logdir ./lightning_logs
```

12.2 Make a custom logger

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

```
from pytorch_lightning.utilities import rank_zero_only
from pytorch_lightning.loggers import LightningLoggerBase
from pytorch_lightning.loggers.base import rank_zero_experiment

class MyLogger(LightningLoggerBase):

    @property
    def name(self):
        return 'MyLogger'

    @property
    @rank_zero_experiment
    def experiment(self):
        # Return the experiment object associated with this logger.
        pass

    @property
    def version(self):
        # Return the experiment version, int or str.
        return '0.1'

    @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

    @rank_zero_only
    def save(self):
        # Optional. Any code necessary to save logger data goes here
        # If you implement this, remember to call `super().save()`
        # at the start of the method (important for aggregation of metrics)
        super().save()

    @rank_zero_only
```

(continues on next page)

(continued from previous page)

```
    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 Lightning!

12.3 Control logging frequency

12.3.1 Logging frequency

It may slow training down to log every single batch. By default, Lightning logs every 50 rows, or 50 training steps. To change this behaviour, set the *log_every_n_steps* `Trainer` flag.

```
k = 10
trainer = Trainer(log_every_n_steps=k)
```

12.3.2 Log writing frequency

Writing to a logger can be expensive, so by default Lightning write logs to disc or to the given logger every 100 training steps. To change this behaviour, set the interval at which you wish to flush logs to the filesystem using *log_every_n_steps* `Trainer` flag.

```
k = 100
trainer = Trainer(flush_logs_every_n_steps=k)
```

Unlike the *log_every_n_steps*, this argument does not apply to all loggers. The example shown here works with `TensorBoardLogger`, which is the default logger in Lightning.

12.4 Progress Bar

You can add any metric to the progress bar using `log()` method, setting *prog_bar=True*.

```
def training_step(self, batch, batch_idx):
    self.log('my_loss', loss, prog_bar=True)
```

12.4.1 Modifying the progress bar

The progress bar by default already includes the training loss and version number of the experiment if you are using a logger. These defaults can be customized by overriding the `get_progress_bar_dict()` hook in your module.

```
def get_progress_bar_dict(self):
    # don't show the version number
    items = super().get_progress_bar_dict()
    items.pop("v_num", None)
    return items
```

12.5 Configure console logging

Lightning logs useful information about the training process and user warnings to the console. You can retrieve the Lightning logger and change it to your liking. For example, increase the logging level to see fewer messages like so:

```
import logging
logging.getLogger("lightning").setLevel(logging.ERROR)
```

Read more about custom Python logging here.

12.6 Logging 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 "hyper_parameters" with the hyperparams.

```
lightning_checkpoint = torch.load(filepath, map_location=lambda storage, loc: storage)
hyperparams = lightning_checkpoint['hyper_parameters']
```

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.

12.7 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)
```

12.8 Supported Loggers

The following are loggers we support

Note: The following loggers will normally plot an additional chart (**global_step VS epoch**).

Note: postfix `_step` and `_epoch` will be appended to the name you logged if `on_step` and `on_epoch` are set to `True` in `self.log()`.

Note: Depending on the loggers you use, there might be some additional charts.

CometLogger	Log using Comet.ml.
CSVLogger	Log to local file system in yaml and CSV format.
MLFlowLogger	Log using MLflow.
NeptuneLogger	Log using Neptune.
TensorBoardLogger	Log to local file system in TensorBoard format.
TestTubeLogger	Log to local file system in TensorBoard format but using a nicer folder structure (see full docs).
WandbLogger	Log using Weights and Biases.

12.8.1 CometLogger

class pytorch_lightning.loggers.**CometLogger**(*api_key=None*, *save_dir=None*, *project_name=None*, *rest_api_key=None*, *experiment_name=None*, *experiment_key=None*, *offline=False*, *prefix=''*, ***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

```
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_key=os.environ.get('COMET_EXPERIMENT_KEY'),  # Optional
    experiment_name='default'  # Optional
)
trainer = Trainer(logger=comet_logger)
```

OFFLINE MODE

```
from pytorch_lightning.loggers import CometLogger
# arguments made to CometLogger are passed on to the comet_ml.Experiment class
comet_logger = CometLogger(
    save_dir='.',
```

(continues on next page)

(continued from previous page)

```
    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. If not given, this will be loaded from the environment variable COMET_API_KEY or ~/.comet.config if either exists.
- **save_dir** (Optional[str]) – Required in offline mode. The path for the directory to save local comet logs. If given, this also sets the directory for saving checkpoints.
- **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.
- **offline** (bool) – If api_key and save_dir are both given, this determines whether the experiment will be in online or offline mode. This is useful if you use save_dir to control the checkpoints directory and have a ~/.comet.config file but still want to run offline experiments.
- **prefix** (str) – A string to put at the beginning of metric keys.
- ****kwargs** – Additional arguments like *workspace*, *log_code*, etc. used by CometExperiment can be passed as keyword arguments in this logger.

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_graph(*model*, *input_array=None*)

Record model graph

Parameters

- **model** (*LightningModule*) – lightning model
- **input_array** – input passes to *model.forward*

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()
```

property name
Return the experiment name.

Return type str

property save_dir
Return the root directory where experiment logs get saved, or *None* if the logger does not save data locally.

Return type Optional[str]

property version
Return the experiment version.

Return type str

12.8.2 CSVLogger

class pytorch_lightning.loggers.**CSVLogger**(*save_dir*, *name='default'*, *version=None*, *prefix=''*)
Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Log to local file system in yaml and CSV format.

Logs are saved to `os.path.join(save_dir, name, version)`.

Example

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.loggers import CSVLogger
>>> logger = CSVLogger("logs", name="my_exp_name")
>>> trainer = Trainer(logger=logger)
```

Parameters

- **save_dir** (str) – Save directory
- **name** (Optional[str]) – Experiment name. Defaults to 'default'.
- **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.
- **prefix** (str) – A string to put at the beginning of metric keys.

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 ExperimentWriter object. To use ExperimentWriter features in your *LightningModule* do the following.

Example:

```
self.logger.experiment.some_experiment_writer_function()
```

Return type *ExperimentWriter*

property log_dir
The log directory for this run. 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 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 save_dir
Return the root directory where experiment logs get saved, or *None* if the logger does not save data locally.

Return type `Optional[str]`

property version
Return the experiment version.

Return type `int`

12.8.3 MLFlowLogger

class pytorch_lightning.loggers.**MLFlowLogger**(*experiment_name='default'*, *tracking_uri=None*, *tags=None*, *save_dir='./mlruns'*, *prefix=''*)

Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Log using MLflow.

Install it with pip:

```
pip install mlflow
```

```
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 your *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 *file:<save_dir>*.
- **tags** (Optional[Dict[str, Any]]) – A dictionary tags for the experiment.
- **save_dir** (Optional[str]) – A path to a local directory where the MLflow runs get saved. Defaults to *./mlflow* if *tracking_uri* is not provided. Has no effect if *tracking_uri* is provided.
- **prefix** (str) – A string to put at the beginning of metric keys.

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 save_dir

The root file directory in which MLflow experiments are saved.

Return type Optional[str]

Returns Local path to the root experiment directory if the tracking uri is local. Otherwhise returns *None*.

property version
Return the experiment version.

Return type str

12.8.4 NeptuneLogger

class pytorch_lightning.loggers.**NeptuneLogger**(*api_key=None*, *project_name=None*, *close_after_fit=True*, *offline_mode=False*, *experiment_name=None*, *experiment_id=None*, *prefix=''*, ***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

```
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

```
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:

```
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))
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.
- **experiment_id** (`Optional[str]`) – Optional. Default is `None`. The ID of the existing experiment. If specified, connect to experiment with experiment_id in project_name. Input arguments "experiment_name", "params", "properties" and "tags" will be overriden based on fetched experiment data.
- **prefix** (`str`) – A string to put at the beginning of metric keys.
- ****kwargs** – Additional arguments like *params*, *tags*, *properties*, etc. used by `neptune.Session.create_experiment()` can be passed as keyword arguments in this logger.

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, 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, currently ignored

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 save_dir

Return the root directory where experiment logs get saved, or *None* if the logger does not save data locally.

Return type Optional[str]

property version

Return the experiment version.

Return type str

12.8.5 TensorBoardLogger

class pytorch_lightning.loggers.**TensorBoardLogger**(*save_dir*, *name='default'*, *version=None*, *log_graph=False*, *default_hp_metric=True*, *prefix=''*, ***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.

- **log_graph**¶ (bool) – Adds the computational graph to tensorboard. This requires that the user has defined the *self.example_input_array* attribute in their model.
- **default_hp_metric**¶ (bool) – Enables a placeholder metric with key *hp_metric* when *log_hyperparams* is called without a metric (otherwise calls to log_hyperparams without a metric are ignored).
- **prefix**¶ (str) – A string to put at the beginning of metric keys.
- ****kwargs**¶ – Additional arguments like *comment*, *filename_suffix*, etc. used by SummaryWriter can be passed as keyword arguments in this logger.

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_graph(*model*, *input_array=None*)
Record model graph

Parameters

- **model**¶ (*LightningModule*) – lightning model
- **input_array**¶ – input passes to *model.forward*

log_hyperparams(*params*, *metrics=None*)
Record hyperparameters. TensorBoard logs with and without saved hyperparameters are incompatible, the hyperparameters are then not displayed in the TensorBoard. Please delete or move the previously saved logs to display the new ones with hyperparameters.

Parameters

- **params**¶ (Union[Dict[str, Any], Namespace]) – a dictionary-like container with the hyperparameters
- **metrics**¶ (Optional[Dict[str, Any]]) – Dictionary with metric names as keys and measured quantities as values

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 save_dir
Return the root directory where experiment logs get saved, or *None* if the logger does not save data locally.

Return type Optional[str]

property version
Return the experiment version.

Return type int

12.8.6 TestTubeLogger

class pytorch_lightning.loggers.**TestTubeLogger**(*save_dir*, *name='default'*, *description=None*, *debug=False*, *version=None*, *create_git_tag=False*, *log_graph=False*, *prefix=''*)

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
```

```
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.
- **log_graph** (bool) – Adds the computational graph to tensorboard. This requires that the user has defined the *self.example_input_array* attribute in their model.
- **prefix** (str) – A string to put at the beginning of metric keys.

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_graph(*model*, *input_array=None*)
Record model graph

Parameters

- **model** (*LightningModule*) – lightning model
- **input_array** – input passes to *model.forward*

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 save_dir
Return the root directory where experiment logs get saved, or *None* if the logger does not save data locally.

Return type Optional[str]

property version
Return the experiment version.

Return type int

12.8.7 WandbLogger

class pytorch_lightning.loggers.**WandbLogger**(*name=None*, *save_dir=None*, *offline=False*, *id=None*, *anonymous=False*, *version=None*, *project=None*, *log_model=False*, *experiment=None*, *prefix=''*, *sync_step=True*, ***kwargs*)
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 (wandb dir by default).
- **offline** (Optional[bool]) – Run offline (data can be streamed later to wandb servers).
- **id** (Optional[str]) – Sets the version, mainly used to resume a previous run.
- **version** (Optional[str]) – Same as id.
- **anonymous** (Optional[bool]) – Enables or explicitly disables anonymous logging.
- **project** (Optional[str]) – The name of the project to which this run will belong.
- **log_model** (Optional[bool]) – Save checkpoints in wandb dir to upload on W&B servers.
- **prefix** (Optional[str]) – A string to put at the beginning of metric keys.
- **sync_step** (Optional[bool]) – Sync Trainer step with wandb step.
- **experiment** – WandB experiment object. Automatically set when creating a run.
- ****kwargs** – Additional arguments like *entity*, *group*, *tags*, etc. used by `wandb.init()` can be passed as keyword arguments in this logger.

Example:

```
from pytorch_lightning.loggers import WandbLogger
from pytorch_lightning import Trainer
wandb_logger = WandbLogger()
trainer = Trainer(logger=wandb_logger)
```

Note: When logging manually through *wandb.log* or *trainer.logger.experiment.log*, make sure to use *commit=False* so the logging step does not increase.

See also:

- Tutorial on how to use W&B with PyTorch Lightning
- W&B Documentation

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

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 Optional[str]

property save_dir

Return the root directory where experiment logs get saved, or *None* if the logger does not save data locally.

Return type Optional[str]

property version

Return the experiment version.

Return type Optional[str]

CHAPTER

THIRTEEN

METRICS

`pytorch_lightning.metrics` is a Metrics API created for easy metric development and usage in PyTorch and PyTorch Lightning. It is rigorously tested for all edge cases and includes a growing list of common metric implementations.

The metrics API provides `update()`, `compute()`, `reset()` functions to the user. The metric base class inherits `nn.Module` which allows us to call `metric(...)` directly. The `forward()` method of the base `Metric` class serves the dual purpose of calling `update()` on its input and simultaneously returning the value of the metric over the provided input.

Warning: From v1.2 onward `compute()` will no longer automatically call `reset()`, and it is up to the user to reset metrics between epochs, except in the case where the metric is directly passed to `LightningModule``s ``self.log`.

These metrics work with DDP in PyTorch and PyTorch Lightning by default. When `.compute()` is called in distributed mode, the internal state of each metric is synced and reduced across each process, so that the logic present in `.compute()` is applied to state information from all processes.

The example below shows how to use a metric in your `LightningModule`:

```
def __init__(self):
    ...
    self.accuracy = pl.metrics.Accuracy()

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

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

`Metric` objects can also be directly logged, in which case Lightning will log the metric based on `on_step` and `on_epoch` flags present in `self.log(...)`. If `on_epoch` is True, the logger automatically logs the end of epoch metric value by calling `.compute()`.

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

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

```
def __init__(self):
    ...
    self.train_acc = pl.metrics.Accuracy()
    self.valid_acc = pl.metrics.Accuracy()

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

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

Note: If using metrics in data parallel mode (dp), the metric update/logging should be done in the `<mode>_step_end` method (where `<mode>` is either `training`, `validation` or `test`). This is due to metric states else being destroyed after each forward pass, leading to wrong accumulation. In practice do the following:

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

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

This metrics API is independent of PyTorch Lightning. Metrics can directly be used in PyTorch as shown in the example:

```
from pytorch_lightning import metrics

train_accuracy = metrics.Accuracy()
valid_accuracy = metrics.Accuracy(compute_on_step=False)

for epoch in range(epochs):
    for x, y in train_data:
        y_hat = model(x)

        # training step accuracy
        batch_acc = train_accuracy(y_hat, y)

    for x, y in valid_data:
        y_hat = model(x)
        valid_accuracy(y_hat, y)
```

(continues on next page)

(continued from previous page)

```
# total accuracy over all training batches
total_train_accuracy = train_accuracy.compute()

# total accuracy over all validation batches
total_valid_accuracy = valid_accuracy.compute()
```

Note: Metrics contain internal states that keep track of the data seen so far. Do not mix metric states across training, validation and testing. It is highly recommended to re-initialize the metric per mode as shown in the examples above. For easy initializing the same metric multiple times, the `.clone()` method can be used:

```
from pytorch_lightning.metrics import Accuracy

def __init__(self):
    ...
    metric = Accuracy()
    self.train_acc = metric.clone()
    self.val_acc = metric.clone()
    self.test_acc = metric.clone()
```

Note: Metric states are **not** added to the models `state_dict` by default. To change this, after initializing the metric, the method `.persistent(mode)` can be used to enable (`mode=True`) or disable (`mode=False`) this behaviour.

13.1 Metrics and devices

Metrics are simple subclasses of `Module` and their metric states behave similar to buffers and parameters of modules. This means that metrics states should be moved to the same device as the input of the metric:

```
from pytorch_lightning.metrics import Accuracy

target = torch.tensor([1, 1, 0, 0], device=torch.device("cuda", 0))
preds = torch.tensor([0, 1, 0, 0], device=torch.device("cuda", 0))

# Metric states are always initialized on cpu, and needs to be moved to
# the correct device
confmat = Accuracy(num_classes=2).to(torch.device("cuda", 0))
out = confmat(preds, target)
print(out.device) # cuda:0
```

However, when **properly defined** inside a *LightningModule*, Lightning will automatically move the metrics to the same device as the data. Being **properly defined** means that the metric is correctly identified as a child module of the model (check `.children()` attribute of the model). Therefore, metrics cannot be placed in native python `list` and `dict`, as they will not be correctly identified as child modules. Instead of `list` use `ModuleList` and instead of `dict` use `ModuleDict`.

```
from pytorch_lightning.metrics import Accuracy

class MyModule(LightningModule):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        ...
        # valid ways metrics will be identified as child modules
        self.metric1 = Accuracy()
        self.metric2 = nn.ModuleList(Accuracy())
        self.metric3 = nn.ModuleDict({'accuracy': Accuracy()})

    def training_step(self, batch, batch_idx):
        # all metrics will be on the same device as the input batch
        data, target = batch
        preds = self(data)
        ...
        val1 = self.metric1(preds, target)
        val2 = self.metric2[0](preds, target)
        val3 = self.metric3['accuracy'](preds, target)
```

13.2 Implementing a Metric

To implement your custom metric, subclass the base `Metric` class and implement the following methods:

- `__init__()`: Each state variable should be called using `self.add_state(...)`.
- `update()`: Any code needed to update the state given any inputs to the metric.
- `compute()`: Computes a final value from the state of the metric.

All you need to do is call `add_state` correctly to implement a custom metric with DDP. `reset()` is called on metric state variables added using `add_state()`.

To see how metric states are synchronized across distributed processes, refer to `add_state()` docs from the base `Metric` class.

Example implementation:

```
from pytorch_lightning.metrics import Metric

class MyAccuracy(Metric):
    def __init__(self, dist_sync_on_step=False):
        super().__init__(dist_sync_on_step=dist_sync_on_step)

        self.add_state("correct", default=torch.tensor(0), dist_reduce_fx="sum")
        self.add_state("total", default=torch.tensor(0), dist_reduce_fx="sum")

    def update(self, preds: torch.Tensor, target: torch.Tensor):
        preds, target = self._input_format(preds, target)
        assert preds.shape == target.shape

        self.correct += torch.sum(preds == target)
        self.total += target.numel()

    def compute(self):
        return self.correct.float() / self.total
```

Metrics support backpropagation, if all computations involved in the metric calculation are differentiable. However, note that the cached state is detached from the computational graph and cannot be backpropagated. Not doing this would mean storing the computational graph for each update call, which can lead to out-of-memory errors. In practise this means that:

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

13.2.1 Metric API

class pytorch_lightning.metrics.**Metric**(*compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Bases: torch.nn., abc.ABC

Base class for all metrics present in the Metrics API.

Implements add_state(), forward(), reset() and a few other things to handle distributed synchronization and per-step metric computation.

Override update() and compute() functions to implement your own metric. Use add_state() to register metric state variables which keep track of state on each call of update() and are synchronized across processes when compute() is called.

Note: Metric state variables can either be torch.Tensors or an empty list which can we used to store *torch.Tensors`*.

Note: Different metrics only override update() and not forward(). A call to update() is valid, but it won't return the metric value at the current step. A call to forward() automatically calls update() and also returns the metric value at the current step.

Parameters

- **compute_on_step**¶ (bool) – Forward only calls update() and returns None if this is set to False. default: True
- **dist_sync_on_step**¶ (bool) – Synchronize metric state across processes at each forward() before returning the value at the step.
- **process_group**¶ (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)
- **dist_sync_fn**¶ (Optional[Callable]) – Callback that performs the allgather operation on the metric state. When *None*, DDP will be used to perform the allgather. default: None

add_state(*name*, *default*, *dist_reduce_fx=None*, *persistent=False*)

Adds metric state variable. Only used by subclasses.

Parameters

- **name**¶ (str) – The name of the state variable. The variable will then be accessible at self.name.
- **default**¶ – Default value of the state; can either be a torch.Tensor or an empty list. The state will be reset to this value when self.reset() is called.
- **dist_reduce_fx**¶ (*Optional*) – Function to reduce state accross mutliple processes in distributed mode. If value is "sum", "mean", or "cat", we will use torch.sum, torch.mean, and torch.cat respectively, each with argument dim=0. Note that the

"cat" reduction only makes sense if the state is a list, and not a tensor. The user can also pass a custom function in this parameter.

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

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

The metric states would be synced as follows

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

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

---

clone()
Make a copy of the metric

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

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

reset()
This method automatically resets the metric state variables to their default value.

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

Return type None

13.2.2 Internal implementation details

This section briefly describe how metrics work internally. We encourage looking at the source code for more info. Internally, Lightning wraps the user defined update() and compute() method. We do this to automatically synchronize and reduce metric states across multiple devices. More precisely, calling update() does the following internally:

1. Clears computed cache
2. Calls user-defined update()

Simiarly, calling compute() does the following internally

1. Syncs metric states between processes
2. Reduce gathered metric states
3. Calls the user defined compute() method on the gathered metric states

4. Cache computed result

From a user's standpoint this has one important side-effect: computed results are cached. This means that no matter how many times `compute` is called after one and another, it will continue to return the same result. The cache is first emptied on the next call to `update`.

`forward` serves the dual purpose of both returning the metric on the current data and updating the internal metric state for accumulating over multiple batches. The `forward()` method achives this by combining calls to `update` and `compute` in the following way (assuming metric is initialized with `compute_on_step=True`):

1. Calls `update()` to update the global metric states (for accumulation over multiple batches)
2. Caches the global state
3. Calls `reset()` to clear global metric state
4. Calls `update()` to update local metric state
5. Calls `compute()` to calculate metric for current batch
6. Restores the global state

This procedure has the consequence of calling the user defined `update` **twice** during a single forward call (one to update global statistics and one for getting the batch statistics).

13.3 Metric Arithmetics

Metrics support most of python built-in operators for arithmetic, logic and bitwise operations.

For example for a metric that should return the sum of two different metrics, implementing a new metric is an overhead that is not necessary. It can now be done with:

```
first_metric = MyFirstMetric()
second_metric = MySecondMetric()

new_metric = first_metric + second_metric
```

`new_metric.update(*args, **kwargs)` now calls update of `first_metric` and `second_metric`. It forwards all positional arguments but forwards only the keyword arguments that are available in respective metric's update declaration.

Similarly `new_metric.compute()` now calls compute of `first_metric` and `second_metric` and adds the results up.

This pattern is implemented for the following operators (with `a` being metrics and `b` being metrics, tensors, integer or floats):

- Addition (`a + b`)
- Bitwise AND (`a & b`)
- Equality (`a == b`)
- Floordivision (`a // b`)
- Greater Equal (`a >= b`)
- Greater (`a > b`)
- Less Equal (`a <= b`)
- Less (`a < b`)

- Matrix Multiplication (`a @ b`)
- Modulo (`a % b`)
- Multiplication (`a * b`)
- Inequality (`a != b`)
- Bitwise OR (`a | b`)
- Power (`a ** b`)
- Substraction (`a - b`)
- True Division (`a / b`)
- Bitwise XOR (`a ^ b`)
- Absolute Value (`abs(a)`)
- Inversion (`~a`)
- Negative Value (`neg(a)`)
- Positive Value (`pos(a)`)

13.4 MetricCollection

In many cases it is beneficial to evaluate the model output by multiple metrics. In this case the *MetricCollection* class may come in handy. It accepts a sequence of metrics and wraps theses into a single callable metric class, with the same interface as any other metric.

Example:

```
from pytorch_lightning.metrics import MetricCollection, Accuracy, Precision, Recall
target = torch.tensor([0, 2, 0, 2, 0, 1, 0, 2])
preds = torch.tensor([2, 1, 2, 0, 1, 2, 2, 2])
metric_collection = MetricCollection([
    Accuracy(),
    Precision(num_classes=3, average='macro'),
    Recall(num_classes=3, average='macro')
])
print(metric_collection(preds, target))
```

```
{'Accuracy': tensor(0.1250),
 'Precision': tensor(0.0667),
 'Recall': tensor(0.1111)}
```

Similarly it can also reduce the amount of code required to log multiple metrics inside your LightningModule

```
def __init__(self):
    ...
    metrics = pl.metrics.MetricCollection(...)
    self.train_metrics = metrics.clone()
    self.valid_metrics = metrics.clone()

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

(continues on next page)

(continued from previous page)

```
    # use log_dict instead of log
    self.log_dict(self.train_metrics, on_step=True, on_epoch=False, prefix='train')

def validation_step(self, batch, batch_idx):
    logits = self(x)
    ...
    self.valid_metrics(logits, y)
    # use log_dict instead of log
    self.log_dict(self.valid_metrics, on_step=True, on_epoch=True, prefix='val')
```

Note: *MetricCollection* as default assumes that all the metrics in the collection have the same call signature. If this is not the case, input that should be given to different metrics can given as keyword arguments to the collection.

class pytorch_lightning.metrics.**MetricCollection**(*metrics*)

Bases: torch.nn.

MetricCollection class can be used to chain metrics that have the same call pattern into one single class.

Parameters **metrics**¶ (Union[List[Metric], Tuple[Metric], Dict[str, Metric]]) – One of the following

- list or tuple: if metrics are passed in as a list, will use the metrics class name as key for output dict. Therefore, two metrics of the same class cannot be chained this way.
- dict: if metrics are passed in as a dict, will use each key in the dict as key for output dict. Use this format if you want to chain together multiple of the same metric with different parameters.

Example (input as list):

```
>>> from pytorch_lightning.metrics import MetricCollection, Accuracy, Precision,␣
↪Recall
>>> target = torch.tensor([0, 2, 0, 2, 0, 1, 0, 2])
>>> preds = torch.tensor([2, 1, 2, 0, 1, 2, 2, 2])
>>> metrics = MetricCollection([Accuracy(),
...                             Precision(num_classes=3, average='macro'),
...                             Recall(num_classes=3, average='macro')])
>>> metrics(preds, target)
{'Accuracy': tensor(0.1250), 'Precision': tensor(0.0667), 'Recall': tensor(0.
↪1111)}
```

Example (input as dict):

```
>>> metrics = MetricCollection({'micro_recall': Recall(num_classes=3, average=
↪'micro'),
...                             'macro_recall': Recall(num_classes=3, average=
↪'macro')})
>>> metrics(preds, target)
{'micro_recall': tensor(0.1250), 'macro_recall': tensor(0.1111)}
```

clone()

Make a copy of the metric collection

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

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

Return type Dict[str, Any]

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

reset()
Iteratively call reset for each metric

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

13.5 Class vs Functional Metrics

The functional metrics follow the simple paradigm input in, output out. This means, they don't provide any advanced mechanisms for syncing across DDP nodes or aggregation over batches. They simply compute the metric value based on the given inputs.

Also, the integration within other parts of PyTorch Lightning will never be as tight as with the class-based interface. If you look for just computing the values, the functional metrics are the way to go. However, if you are looking for the best integration and user experience, please consider also using the class interface.

13.6 Classification Metrics

13.6.1 Input types

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

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

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

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

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

```
# Binary inputs
binary_preds  = torch.tensor([0.6, 0.1, 0.9])
binary_target = torch.tensor([1, 0, 2])
```

(continues on next page)

(continued from previous page)

```
# Multi-class inputs
mc_preds  = torch.tensor([0, 2, 1])
mc_target = torch.tensor([0, 1, 2])

# Multi-class inputs with probabilities
mc_preds_probs  = torch.tensor([[0.8, 0.2, 0], [0.1, 0.2, 0.7], [0.3, 0.6, 0.1]])
mc_target_probs = torch.tensor([0, 1, 2])

# Multi-label inputs
ml_preds  = torch.tensor([[0.2, 0.8, 0.9], [0.5, 0.6, 0.1], [0.3, 0.1, 0.1]])
ml_target = torch.tensor([[0, 1, 1], [1, 0, 0], [0, 0, 0]])
```

Using the is_multiclass parameter

In some cases, you might have inputs which appear to be (multi-dimensional) multi-class but are actually binary/multi-label - for example, if both predictions and targets are integer (binary) tensors. Or it could be the other way around, you want to treat binary/multi-label inputs as 2-class (multi-dimensional) multi-class inputs.

For these cases, the metrics where this distinction would make a difference, expose the `is_multiclass` argument. Let's see how this is used on the example of `StatScores` metric.

First, let's consider the case with label predictions with 2 classes, which we want to treat as binary.

```
from pytorch_lightning.metrics.functional import stat_scores

# These inputs are supposed to be binary, but appear as multi-class
preds  = torch.tensor([0, 1, 0])
target = torch.tensor([1, 1, 0])
```

As you can see below, by default the inputs are treated as multi-class. We can set `is_multiclass=False` to treat the inputs as binary - which is the same as converting the predictions to float beforehand.

```
>>> stat_scores(preds, target, reduce='macro', num_classes=2)
tensor([[1, 1, 1, 0, 1],
        [1, 0, 1, 1, 2]])
>>> stat_scores(preds, target, reduce='macro', num_classes=1, is_multiclass=False)
tensor([[1, 0, 1, 1, 2]])
>>> stat_scores(preds.float(), target, reduce='macro', num_classes=1)
tensor([[1, 0, 1, 1, 2]])
```

Next, consider the opposite example: inputs are binary (as predictions are probabilities), but we would like to treat them as 2-class multi-class, to obtain the metric for both classes.

```
preds  = torch.tensor([0.2, 0.7, 0.3])
target = torch.tensor([1, 1, 0])
```

In this case we can set `is_multiclass=True`, to treat the inputs as multi-class.

```
>>> stat_scores(preds, target, reduce='macro', num_classes=1)
tensor([[1, 0, 1, 1, 2]])
>>> stat_scores(preds, target, reduce='macro', num_classes=2, is_multiclass=True)
tensor([[1, 1, 1, 0, 1],
        [1, 0, 1, 1, 2]])
```

13.6.2 Class Metrics (Classification)

Accuracy

class pytorch_lightning.metrics.classification.**Accuracy**(*threshold=0.5*, *top_k=None*, *subset_accuracy=False*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Bases: `torch.nn.`, `abc.ABC`

Computes Accuracy:

$$\text{Accuracy} = \frac{1}{N}\sum_{i}^{N} 1(y_i = \hat{y}_i)$$

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

For multi-class and multi-dimensional multi-class data with probability predictions, the parameter `top_k` generalizes this metric to a Top-K accuracy metric: for each sample the top-K highest probability items are considered to find the correct label.

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

Accepts all input types listed in *Input types*.

Parameters

- **threshold** (`float`) – Threshold probability value for transforming probability predictions to binary (0,1) predictions, in the case of binary or multi-label inputs.
- **top_k** (`Optional[int]`) – Number of highest probability predictions considered to find the correct label, relevant only for (multi-dimensional) multi-class inputs with probability predictions. The default value (`None`) will be interpreted as 1 for these inputs.

  Should be left at default (`None`) for all other types of inputs.
- **subset_accuracy** (`bool`) – Whether to compute subset accuracy for multi-label and multi-dimensional multi-class inputs (has no effect for other input types).
  - For multi-label inputs, if the parameter is set to `True`, then all labels for each sample must be correctly predicted for the sample to count as correct. If it is set to `False`, then all labels are counted separately - this is equivalent to flattening inputs beforehand (i.e. `preds = preds.flatten()` and same for `target`).
  - For multi-dimensional multi-class inputs, if the parameter is set to `True`, then all sub-sample (on the extra axis) must be correct for the sample to be counted as correct. If it is set to `False`, then all sub-samples are counter separately - this is equivalent, in the case of label predictions, to flattening the inputs beforehand (i.e. `preds = preds.flatten()` and same for `target`). Note that the `top_k` parameter still applies in both cases, if set.
- **compute_on_step** (`bool`) – Forward only calls `update()` and return `None` if this is set to `False`.
- **dist_sync_on_step** (`bool`) – Synchronize metric state across processes at each `forward()` before returning the value at the step

- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)
- **dist_sync_fn** (Optional[Callable]) – Callback that performs the allgather operation on the metric state. When None, DDP will be used to perform the allgather

Example

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

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

compute()
Computes accuracy based on inputs passed in to update previously.

Return type Tensor

update(*preds*, *target*)
Update state with predictions and targets. See *Input types* for more information on input types.

Parameters

- **preds** (Tensor) – Predictions from model (probabilities, or labels)
- **target** (Tensor) – Ground truth labels

AveragePrecision

class pytorch_lightning.metrics.classification.**AveragePrecision**(*num_classes=None*, *pos_label=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)

Bases: torch.nn., abc.ABC

Computes the average precision score, which summarises the precision recall curve into one number. Works for both binary and multiclass problems. In the case of multiclass, the values will be calculated based on a one-vs-the-rest approach.

Forward accepts

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

Parameters

- **num_classes**¶ (Optional[int]) – integer with number of classes. Not nessesary to provide for binary problems.
- **pos_label**¶ (Optional[int]) – integer determining the positive class. Default is None which for binary problem is translate to 1. For multiclass problems this argument should not be set as we iteratively change it in the range [0,num_classes-1]
- **compute_on_step**¶ (bool) – Forward only calls update() and return None if this is set to False. default: True
- **dist_sync_on_step**¶ (bool) – Synchronize metric state across processes at each forward() before returning the value at the step. default: False
- **process_group**¶ (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

Example (binary case):

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 1, 1])
>>> average_precision = AveragePrecision(pos_label=1)
>>> average_precision(pred, target)
tensor(1.)
```

Example (multiclass case):

```
>>> pred = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                      [0.05, 0.75, 0.05, 0.05, 0.05],
...                      [0.05, 0.05, 0.75, 0.05, 0.05],
...                      [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> average_precision = AveragePrecision(num_classes=5)
>>> average_precision(pred, target)
[tensor(1.), tensor(1.), tensor(0.2500), tensor(0.2500), tensor(nan)]
```

compute()
Compute the average precision score

Return type Union[Tensor, List[Tensor]]

Returns tensor with average precision. If multiclass will return list of such tensors, one for each class

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

Parameters

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

AUC

class pytorch_lightning.metrics.classification.**AUC**(*reorder=False*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Bases: `torch.nn.`, `abc.ABC`

Computes Area Under the Curve (AUC) using the trapezoidal rule

Forward accepts two input tensors that should be 1D and have the same number of elements

Parameters

- **reorder** (`bool`) – AUC expects its first input to be sorted. If this is not the case, setting this argument to `True` will use a stable sorting algorithm to sort the input in decending order
- **compute_on_step** (`bool`) – Forward only calls `update()` and return None if this is set to False.
- **dist_sync_on_step** (`bool`) – Synchronize metric state across processes at each `forward()` before returning the value at the step.
- **process_group** (`Optional[Any]`) – Specify the process group on which synchronization is called. default: None (which selects the entire world)
- **dist_sync_fn** (`Optional[Callable]`) – Callback that performs the allgather operation on the metric state. When `None`, DDP will be used to perform the allgather

compute()

Computes AUC based on inputs passed in to `update` previously.

Return type `Tensor`

update(*x*, *y*)

Update state with predictions and targets.

Parameters

- **x** (`Tensor`) – Predictions from model (probabilities, or labels)
- **y** (`Tensor`) – Ground truth labels

AUROC

class pytorch_lightning.metrics.classification.**AUROC**(*num_classes=None*, *pos_label=None*, *average='macro'*, *max_fpr=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Bases: `torch.nn.`, `abc.ABC`

Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) <https://en.wikipedia.org/wiki/Receiver_operating_characteristic#Further_interpretations>`_. Works for both binary, multilabel and multiclass problems. In the case of multiclass, the values will be calculated based on a one-vs-the-rest approach.

Forward accepts

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

For non-binary input, if the `preds` and `target` tensor have the same size the input will be interpretated as multilabel and if `preds` have one dimension more than the `target` tensor the input will be interpretated as multiclass.

Args:

num_classes: integer with number of classes. Not nessesary to provide for binary problems.

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

average:

- `'macro'` computes metric for each class and uniformly averages them
- `'weighted'` computes metric for each class and does a weighted-average, where each class is weighted by their support (accounts for class imbalance)
- `None` computes and returns the metric per class

max_fpr: If not `None`, calculates standardized partial AUC over the range [0, max_fpr]. Should be a float between 0 and 1.

compute_on_step: Forward only calls `update()` and return None if this is set to False. default: True

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

process_group: Specify the process group on which synchronization is called. default: None (which selects the entire world)

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

Example (binary case):

```
>>> preds = torch.tensor([0.13, 0.26, 0.08, 0.19, 0.34])
>>> target = torch.tensor([0, 0, 1, 1, 1])
>>> auroc = AUROC(pos_label=1)
>>> auroc(preds, target)
tensor(0.5000)
```

Example (multiclass case):

```
>>> preds = torch.tensor([[0.90, 0.05, 0.05],
...                       [0.05, 0.90, 0.05],
...                       [0.05, 0.05, 0.90],
...                       [0.85, 0.05, 0.10],
...                       [0.10, 0.10, 0.80]])
>>> target = torch.tensor([0, 1, 1, 2, 2])
>>> auroc = AUROC(num_classes=3)
>>> auroc(preds, target)
tensor(0.7778)
```

compute()
Computes AUROC based on inputs passed in to `update` previously.

Return type `Tensor`

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

Parameters

- **preds**¶ (`Tensor`) – Predictions from model (probabilities, or labels)
- **target**¶ (`Tensor`) – Ground truth labels

ConfusionMatrix

class pytorch_lightning.metrics.classification.**ConfusionMatrix**(*num_classes*, *normalize=None*, *threshold=0.5*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)

Bases: `torch.nn.`, `abc.ABC`

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

Note: This metric produces a multi-dimensional output, so it can not be directly logged.

Forward accepts

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

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

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

Parameters

- **num_classes**¶ (`int`) – Number of classes in the dataset.
- **normalize**¶ (`Optional[str]`) – Normalization mode for confusion matrix. Choose from
  - `None` or `'none'`: no normalization (default)
  - `'true'`: normalization over the targets (most commonly used)
  - `'pred'`: normalization over the predictions
  - `'all'`: normalization over the whole matrix
- **threshold**¶ (`float`) – Threshold value for binary or multi-label probabilites. default: 0.5

- **compute_on_step**¶ (bool) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step**¶ (bool) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False
- **process_group**¶ (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

Example

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

compute()
Computes confusion matrix

Return type Tensor

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

Parameters

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

F1

class pytorch_lightning.metrics.classification.**F1**(*num_classes*, *threshold=0.5*, *average='micro'*, *multilabel=False*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)

Bases: torch.nn., abc.ABC

Computes F1 metric. F1 metrics correspond to a harmonic mean of the precision and recall scores.

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

Forward accepts

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

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

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

Parameters

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

- **threshold**¶ (float) – Threshold value for binary or multi-label logits. default: 0.5
- **average**¶ (str) –
  - 'micro' computes metric globally
  - 'macro' computes metric for each class and uniformly averages them
  - 'weighted' computes metric for each class and does a weighted-average, where each class is weighted by their support (accounts for class imbalance)
  - 'none' or None computes and returns the metric per class
- **multilabel**¶ (bool) – If predictions are from multilabel classification.
- **compute_on_step**¶ (bool) – Forward only calls update() and returns None if this is set to False. default: True
- **dist_sync_on_step**¶ (bool) – Synchronize metric state across processes at each forward() before returning the value at the step. default: False
- **process_group**¶ (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

Example

```
>>> from pytorch_lightning.metrics import F1
>>> target = torch.tensor([0, 1, 2, 0, 1, 2])
>>> preds = torch.tensor([0, 2, 1, 0, 0, 1])
>>> f1 = F1(num_classes=3)
>>> f1(preds, target)
tensor(0.3333)
```

FBeta

class pytorch_lightning.metrics.classification.**FBeta**(*num_classes*, *beta=1.0*, *threshold=0.5*, *average='micro'*, *multilabel=False*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)

Bases: torch.nn., abc.ABC

Computes F-score, specifically:

$$F_\beta = (1 + \beta^2) * \frac{\text{precision} * \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}}$$

Where β is some positive real factor. Works with binary, multiclass, and multilabel data. Accepts probabilities from a model output or integer class values in prediction. Works with multi-dimensional preds and target.

Forward accepts

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

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

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

Parameters

- **num_classes** (int) – Number of classes in the dataset.
- **beta** (float) – Beta coefficient in the F measure.
- **threshold** (float) – Threshold value for binary or multi-label probabilities. default: 0.5
- **average** (str) –
  - 'micro' computes metric globally
  - 'macro' computes metric for each class and uniformly averages them
  - 'weighted' computes metric for each class and does a weighted-average, where each class is weighted by their support (accounts for class imbalance)
  - 'none' or None computes and returns the metric per class
- **multilabel** (bool) – If predictions are from multilabel classification.
- **compute_on_step** (bool) – Forward only calls update() and return None if this is set to False. default: True
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each forward() before returning the value at the step. default: False
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

Example

```
>>> from pytorch_lightning.metrics import FBeta
>>> target = torch.tensor([0, 1, 2, 0, 1, 2])
>>> preds = torch.tensor([0, 2, 1, 0, 0, 1])
>>> f_beta = FBeta(num_classes=3, beta=0.5)
>>> f_beta(preds, target)
tensor(0.3333)
```

compute()
Computes fbeta over state.

Return type Tensor

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

Parameters

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

IoU

class pytorch_lightning.metrics.classification.**IoU**(*num_classes*, *ignore_index=None*, *absent_score=0.0*, *threshold=0.5*, *reduction='elementwise_mean'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)

Bases: `torch.nn.`, `abc.ABC`

Computes Intersection over union, or Jaccard index calculation:

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

Where: A and B are both tensors of the same size, containing integer class values. They may be subject to conversion from input data (see description below). Note that it is different from box IoU.

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

Forward accepts

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

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

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

Parameters

- **num_classes** (`int`) – Number of classes in the dataset.
- **ignore_index** (`Optional[int]`) – optional int specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. Has no effect if given an int that is not in the range [0, num_classes-1]. By default, no index is ignored, and all classes are used.
- **absent_score** (`float`) – score to use for an individual class, if no instances of the class index were present in *pred* AND no instances of the class index were present in *target*. For example, if we have 3 classes, [0, 0] for *pred*, and [0, 2] for *target*, then class 1 would be assigned the *absent_score*.
- **threshold** (`float`) – Threshold value for binary or multi-label probabilities.
- **reduction** (`str`) – a method to reduce metric score over labels.
  - `'elementwise_mean'`: takes the mean (default)
  - `'sum'`: takes the sum
  - `'none'`: no reduction will be applied
- **compute_on_step** (`bool`) – Forward only calls `update()` and return None if this is set to False.
- **dist_sync_on_step** (`bool`) – Synchronize metric state across processes at each `forward()` before returning the value at the step.
- **process_group** (`Optional[Any]`) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

Example

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

compute()
Computes intersection over union (IoU)

Return type Tensor

Hamming Distance

class pytorch_lightning.metrics.classification.**HammingDistance**(*threshold=0.5*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Bases: torch.nn., abc.ABC

Computes the average Hamming distance (also known as Hamming loss) between targets and predictions:

$$\text{Hamming distance} = \frac{1}{N \cdot L} \sum_{i}^{N} \sum_{l}^{L} 1(y_{il} \neq \hat{y_{il}})$$

Where y is a tensor of target values, $\hat{y}$ is a tensor of predictions, and $\bullet_{il}$ refers to the l-th label of the i-th sample of that tensor.

This is the same as `1-accuracy` for binary data, while for all other types of inputs it treats each possible label separately - meaning that, for example, multi-class data is treated as if it were multi-label.

Accepts all input types listed in *Input types*.

Parameters

- **threshold** (float) – Threshold probability value for transforming probability predictions to binary (0 or 1) predictions, in the case of binary or multi-label inputs.
- **compute_on_step** (bool) – Forward only calls update() and return None if this is set to False.
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each forward() before returning the value at the step.
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)
- **dist_sync_fn** (Optional[Callable]) – Callback that performs the allgather operation on the metric state. When None, DDP will be used to perform the all gather.

Example

```
>>> from pytorch_lightning.metrics import HammingDistance
>>> target = torch.tensor([[0, 1], [1, 1]])
>>> preds = torch.tensor([[0, 1], [0, 1]])
>>> hamming_distance = HammingDistance()
>>> hamming_distance(preds, target)
tensor(0.2500)
```

compute()
Computes hamming distance based on inputs passed in to `update` previously.

Return type `Tensor`

update(*preds*, *target*)
Update state with predictions and targets. See *Input types* for more information on input types.

Parameters

- **preds** (`Tensor`) – Predictions from model (probabilities, or labels)
- **target** (`Tensor`) – Ground truth labels

Precision

class pytorch_lightning.metrics.classification.**Precision**(*num_classes=None*, *threshold=0.5*, *average='micro'*, *multilabel=False*, *mdmc_average=None*, *ignore_index=None*, *top_k=None*, *is_multiclass=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Bases: `torch.nn.`, `abc.ABC`

Computes Precision:

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

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

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

Parameters

- **num_classes** (`Optional`[`int`]) – Number of classes. Necessary for `'macro'`, `'weighted'` and `None` average methods.
- **threshold** (`float`) – Threshold probability value for transforming probability predictions to binary (0,1) predictions, in the case of binary or multi-label inputs.
- **average** (`str`) – Defines the reduction that is applied. Should be one of the following:

  - 'micro' [default]: Calculate the metric globally, accross all samples and classes.
  - 'macro': Calculate the metric for each class separately, and average the metrics accross classes (with equal weights for each class).
  - 'weighted': Calculate the metric for each class separately, and average the metrics accross classes, weighting each class by its support (tp + fn).
  - 'none' or None: Calculate the metric for each class separately, and return the metric for every class.
  - 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

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

- **multilabel** (bool) –

  Warning: This parameter is deprecated and has no effect. Will be removed in v1.4.0.

- **mdmc_average** (Optional[str]) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the average parameter). Should be one of the following:
  - None [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - 'samplewise': In this case, the statistics are computed separately for each sample on the N axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... (see *Input types*) as the N dimension within the sample, and computing the metric for the sample based on that.
  - 'global': In this case the N and ... dimensions of the inputs (see *Input types*) are flattened into a new N_X sample axis, i.e. the inputs are treated as if they were (N_X, C). From here on the average parameter applies as usual.
- **ignore_index** (Optional[int]) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and average=None or 'none', the score for the ignored class will be returned as nan.
- **top_k** (Optional[int]) – Number of highest probability entries for each sample to convert to 1s - relevant only for inputs with probability predictions. If this parameter is set for multi-label inputs, it will take precedence over threshold. For (multi-dim) multi-class inputs, this parameter defaults to 1.

  Should be left unset (None) for inputs with label predictions.
- **is_multiclass** (Optional[bool]) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.
- **compute_on_step** (bool) – Forward only calls update() and return None if this is set to False.
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each forward() before returning the value at the step
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

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

Example

```
>>> from pytorch_lightning.metrics import Precision
>>> preds  = torch.tensor([2, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> precision = Precision(average='macro', num_classes=3)
>>> precision(preds, target)
tensor(0.1667)
>>> precision = Precision(average='micro')
>>> precision(preds, target)
tensor(0.2500)
```

compute()

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

Return type Tensor

Returns

The shape of the returned tensor depends on the average parameter

- If average in ['micro', 'macro', 'weighted', 'samples'], a one-element tensor will be returned
- If average in ['none', None], the shape will be (C,), where C stands for the number of classes

PrecisionRecallCurve

class pytorch_lightning.metrics.classification.**PrecisionRecallCurve**(*num_classes=None*, *pos_label=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)

Bases: torch.nn., abc.ABC

Computes precision-recall pairs for different thresholds. Works for both binary and multiclass problems. In the case of multiclass, the values will be calculated based on a one-vs-the-rest approach.

Forward accepts

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

Parameters

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

- **compute_on_step** (bool) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

Example (binary case):

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 1, 0])
>>> pr_curve = PrecisionRecallCurve(pos_label=1)
>>> precision, recall, thresholds = pr_curve(pred, target)
>>> precision
tensor([0.6667, 0.5000, 0.0000, 1.0000])
>>> recall
tensor([1.0000, 0.5000, 0.0000, 0.0000])
>>> thresholds
tensor([1, 2, 3])
```

Example (multiclass case):

```
>>> pred = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                      [0.05, 0.75, 0.05, 0.05, 0.05],
...                      [0.05, 0.05, 0.75, 0.05, 0.05],
...                      [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> pr_curve = PrecisionRecallCurve(num_classes=5)
>>> precision, recall, thresholds = pr_curve(pred, target)
>>> precision
[tensor([1., 1.]), tensor([1., 1.]), tensor([0.2500, 0.0000, 1.0000]),
 tensor([0.2500, 0.0000, 1.0000]), tensor([0., 1.])]
>>> recall
[tensor([1., 0.]), tensor([1., 0.]), tensor([1., 0., 0.]), tensor([1., 0., 0.]),␣
↪tensor([nan, 0.])]
>>> thresholds
[tensor([0.7500]), tensor([0.7500]), tensor([0.0500, 0.7500]), tensor([0.0500, 0.
↪7500]), tensor([0.0500])]
```

compute()

Compute the precision-recall curve

Returns: 3-element tuple containing

precision: tensor where element i is the precision of predictions with score >= thresholds[i] and the last element is 1. If multiclass, this is a list of such tensors, one for each class.

recall: tensor where element i is the recall of predictions with score >= thresholds[i] and the last element is 0. If multiclass, this is a list of such tensors, one for each class.

thresholds: Thresholds used for computing precision/recall scores

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

update(*preds*, *target*)

Update state with predictions and targets.

Parameters

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

Recall

class pytorch_lightning.metrics.classification.**Recall**(*num_classes=None*, *threshold=0.5*, *average='micro'*, *multilabel=False*, *mdmc_average=None*, *ignore_index=None*, *top_k=None*, *is_multiclass=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Bases: torch.nn., abc.ABC

Computes Recall:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where TP and FN represent the number of true positives and false negatives respecitively. With the use of top_k parameter, this metric can generalize to Recall@K.

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

Parameters

- **num_classes** (Optional[int]) – Number of classes. Necessary for 'macro', 'weighted' and None average methods.
- **threshold** (float) – Threshold probability value for transforming probability predictions to binary (0,1) predictions, in the case of binary or multi-label inputs.
- **average** (str) – Defines the reduction that is applied. Should be one of the following:
  - 'micro' [default]: Calculate the metric globally, accross all samples and classes.
  - 'macro': Calculate the metric for each class separately, and average the metrics accross classes (with equal weights for each class).
  - 'weighted': Calculate the metric for each class separately, and average the metrics accross classes, weighting each class by its support (tp + fn).
  - 'none' or None: Calculate the metric for each class separately, and return the metric for every class.
  - 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

  Note that what is considered a sample in the multi-dimensional multi-class case depends on the value of mdmc_average.
- **multilabel** (bool) –

Warning: This parameter is deprecated and has no effect. Will be removed in v1.4.0.

- **mdmc_average** (Optional[str]) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
  - `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - `'samplewise'`: In this case, the statistics are computed separately for each sample on the `N` axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes `...` (see *Input types*) as the `N` dimension within the sample, and computing the metric for the sample based on that.
  - `'global'`: In this case the `N` and `...` dimensions of the inputs (see *Input types*) are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were `(N_X, C)`. From here on the `average` parameter applies as usual.
- **ignore_index** (Optional[int]) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or `'none'`, the score for the ignored class will be returned as `nan`.
- **top_k** (Optional[int]) – Number of highest probability entries for each sample to convert to 1s - relevant only for inputs with probability predictions. If this parameter is set for multi-label inputs, it will take precedence over `threshold`. For (multi-dim) multi-class inputs, this parameter defaults to 1.

  Should be left unset (`None`) for inputs with label predictions.
- **is_multiclass** (Optional[bool]) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.
- **compute_on_step** (bool) – Forward only calls `update()` and return `None` if this is set to `False`.
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each `forward()` before returning the value at the step
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: `None` (which selects the entire world)
- **dist_sync_fn** (Optional[Callable]) – Callback that performs the allgather operation on the metric state. When `None`, DDP will be used to perform the allgather.

Example

```
>>> from pytorch_lightning.metrics import Recall
>>> preds  = torch.tensor([2, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> recall = Recall(average='macro', num_classes=3)
>>> recall(preds, target)
tensor(0.3333)
>>> recall = Recall(average='micro')
>>> recall(preds, target)
tensor(0.2500)
```

compute()
Computes the recall score based on inputs passed in to update previously.

Return type Tensor

Returns

The shape of the returned tensor depends on the average parameter

- If average in ['micro', 'macro', 'weighted', 'samples'], a one-element tensor will be returned
- If average in ['none', None], the shape will be (C,), where C stands for the number of classes

ROC

class pytorch_lightning.metrics.classification.**ROC**(*num_classes=None*, *pos_label=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)

Bases: torch.nn., abc.ABC

Computes the Receiver Operating Characteristic (ROC). Works for both binary and multiclass problems. In the case of multiclass, the values will be calculated based on a one-vs-the-rest approach.

Forward accepts

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

Parameters

- **num_classes** (Optional[int]) – integer with number of classes. Not nessesary to provide for binary problems.
- **pos_label** (Optional[int]) – integer determining the positive class. Default is None which for binary problem is translate to 1. For multiclass problems this argument should not be set as we iteratively change it in the range [0,num_classes-1]
- **compute_on_step** (bool) – Forward only calls update() and return None if this is set to False. default: True
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each forward() before returning the value at the step. default: False
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

Example (binary case):

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 1, 1])
>>> roc = ROC(pos_label=1)
>>> fpr, tpr, thresholds = roc(pred, target)
>>> fpr
tensor([0., 0., 0., 0., 1.])
```

(continues on next page)

(continued from previous page)

```
>>> tpr
tensor([0.0000, 0.3333, 0.6667, 1.0000, 1.0000])
>>> thresholds
tensor([4, 3, 2, 1, 0])
```

Example (multiclass case):

```
>>> pred = torch.tensor([[0.75, 0.05, 0.05, 0.05],
...                      [0.05, 0.75, 0.05, 0.05],
...                      [0.05, 0.05, 0.75, 0.05],
...                      [0.05, 0.05, 0.05, 0.75]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> roc = ROC(num_classes=4)
>>> fpr, tpr, thresholds = roc(pred, target)
>>> fpr
[tensor([0., 0., 1.]), tensor([0., 0., 1.]), tensor([0.0000, 0.3333, 1.0000]),␣
↪tensor([0.0000, 0.3333, 1.0000])]
>>> tpr
[tensor([0., 1., 1.]), tensor([0., 1., 1.]), tensor([0., 0., 1.]), tensor([0., 0.,
↪ 1.])]
>>> thresholds
[tensor([1.7500, 0.7500, 0.0500]),
 tensor([1.7500, 0.7500, 0.0500]),
 tensor([1.7500, 0.7500, 0.0500]),
 tensor([1.7500, 0.7500, 0.0500])]
```

compute()

Compute the receiver operating characteristic

Returns: 3-element tuple containing

fpr: tensor with false positive rates. If multiclass, this is a list of such tensors, one for each class.

tpr: tensor with true positive rates. If multiclass, this is a list of such tensors, one for each class.

thresholds: thresholds used for computing false- and true postive rates

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

update(*preds*, *target*)

Update state with predictions and targets.

Parameters

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

StatScores

class pytorch_lightning.metrics.classification.**StatScores**(*threshold=0.5*, *top_k=None*, *reduce='micro'*, *num_classes=None*, *ignore_index=None*, *mdmc_reduce=None*, *is_multiclass=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Bases: `torch.nn.`, `abc.ABC`

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

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

Accepts all inputs listed in *Input types*.

Parameters

- **threshold** (`float`) – Threshold probability value for transforming probability predictions to binary (0 or 1) predictions, in the case of binary or multi-label inputs.
- **top_k** (`Optional[int]`) – Number of highest probability entries for each sample to convert to 1s - relevant only for inputs with probability predictions. If this parameter is set for multi-label inputs, it will take precedence over `threshold`. For (multi-dim) multi-class inputs, this parameter defaults to 1.

  Should be left unset (`None`) for inputs with label predictions.
- **reduce** (`str`) – Defines the reduction that is applied. Should be one of the following:
  - `'micro'` [default]: Counts the statistics by summing over all [sample, class] combinations (globally). Each statistic is represented by a single integer.
  - `'macro'`: Counts the statistics for each class separately (over all samples). Each statistic is represented by a `(C,)` tensor. Requires `num_classes` to be set.
  - `'samples'`: Counts the statistics for each sample separately (over all classes). Each statistic is represented by a `(N, )` 1d tensor.

  Note that what is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_reduce`.
- **num_classes** (`Optional[int]`) – Number of classes. Necessary for (multi-dimensional) multi-class or multi-label data.
- **ignore_index** (`Optional[int]`) – Specify a class (label) to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `reduce='macro'`, the class statistics for the ignored class will all be returned as `-1`.
- **mdmc_reduce** (`Optional[str]`) – Defines how the multi-dimensional multi-class inputs are handeled. Should be one of the following:

  - `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class (see *Input types* for the definition of input types).
  - `'samplewise'`: In this case, the statistics are computed separately for each sample on the `N` axis, and then the outputs are concatenated together. In each sample the extra axes `...` are flattened to become the sub-sample axis, and statistics for each sample are computed by treating the sub-sample axis as the `N` axis for that sample.
  - `'global'`: In this case the `N` and `...` dimensions of the inputs are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were `(N_X, C)`. From here on the `reduce` parameter applies as usual.

- **is_multiclass**¶ (`Optional`[`bool`]) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.
- **compute_on_step**¶ (`bool`) – Forward only calls `update()` and return `None` if this is set to `False`.
- **dist_sync_on_step**¶ (`bool`) – Synchronize metric state across processes at each `forward()` before returning the value at the step
- **process_group**¶ (`Optional`[`Any`]) – Specify the process group on which synchronization is called. default: `None` (which selects the entire world)
- **dist_sync_fn**¶ (`Optional`[`Callable`]) – Callback that performs the allgather operation on the metric state. When `None`, DDP will be used to perform the allgather.

Example

```
>>> from pytorch_lightning.metrics.classification import StatScores
>>> preds  = torch.tensor([1, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> stat_scores = StatScores(reduce='macro', num_classes=3)
>>> stat_scores(preds, target)
tensor([[0, 1, 2, 1, 1],
        [1, 1, 1, 1, 2],
        [1, 0, 3, 0, 1]])
>>> stat_scores = StatScores(reduce='micro')
>>> stat_scores(preds, target)
tensor([2, 2, 6, 2, 4])
```

compute()

Computes the stat scores based on inputs passed in to `update` previously.

Return type `Tensor`

Returns

The metric returns a tensor of shape `(..., 5)`, where the last dimension corresponds to `[tp, fp, tn, fn, sup]` (`sup` stands for support and equals `tp + fn`). The shape depends on the `reduce` and `mdmc_reduce` (in case of multi-dimensional multi-class data) parameters:

- If the data is not multi-dimensional multi-class, then
  - If `reduce='micro'`, the shape will be `(5, )`
  - If `reduce='macro'`, the shape will be `(C, 5)`, where `C` stands for the number of classes

  - If `reduce='samples'`, the shape will be `(N, 5)`, where `N` stands for the number of samples

- If the data is multi-dimensional multi-class and `mdmc_reduce='global'`, then
  - If `reduce='micro'`, the shape will be `(5, )`
  - If `reduce='macro'`, the shape will be `(C, 5)`
  - If `reduce='samples'`, the shape will be `(N*X, 5)`, where `X` stands for the product of sizes of all "extra" dimensions of the data (i.e. all dimensions except for `C` and `N`)
- If the data is multi-dimensional multi-class and `mdmc_reduce='samplewise'`, then
  - If `reduce='micro'`, the shape will be `(N, 5)`
  - If `reduce='macro'`, the shape will be `(N, C, 5)`
  - If `reduce='samples'`, the shape will be `(N, X, 5)`

update (*preds*, *target*)

Update state with predictions and targets. See *Input types* for more information on input types.

Parameters

- **preds** (`Tensor`) – Predictions from model (probabilities or labels)
- **target** (`Tensor`) – Ground truth values

13.6.3 Functional Metrics (Classification)

accuracy [func]

pytorch_lightning.metrics.functional.**accuracy** (*preds*, *target*, *threshold=0.5*, *top_k=None*, *subset_accuracy=False*)

Computes Accuracy:

$$\text{Accuracy} = \frac{1}{N}\sum_{i}^{N} 1(y_i = \hat{y}_i)$$

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

For multi-class and multi-dimensional multi-class data with probability predictions, the parameter `top_k` generalizes this metric to a Top-K accuracy metric: for each sample the top-K highest probability items are considered to find the correct label.

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

Accepts all input types listed in *Input types*.

Parameters

- **preds** (`Tensor`) – Predictions from model (probabilities, or labels)
- **target** (`Tensor`) – Ground truth labels
- **threshold** (`float`) – Threshold probability value for transforming probability predictions to binary (0,1) predictions, in the case of binary or multi-label inputs.

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

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

- **subset_accuracy** (bool) – Whether to compute subset accuracy for multi-label and multi-dimensional multi-class inputs (has no effect for other input types).

  – For multi-label inputs, if the parameter is set to True, then all labels for each sample must be correctly predicted for the sample to count as correct. If it is set to False, then all labels are counted separately - this is equivalent to flattening inputs beforehand (i.e. preds = preds.flatten() and same for target).

  – For multi-dimensional multi-class inputs, if the parameter is set to True, then all sub-sample (on the extra axis) must be correct for the sample to be counted as correct. If it is set to False, then all sub-samples are counter separately - this is equivalent, in the case of label predictions, to flattening the inputs beforehand (i.e. preds = preds.flatten() and same for target). Note that the top_k parameter still applies in both cases, if set.

Example

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

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

Return type Tensor

auc [func]

pytorch_lightning.metrics.functional.**auc**(*x*, *y*, *reorder=False*)

Computes Area Under the Curve (AUC) using the trapezoidal rule

Parameters

- **x** (Tensor) – x-coordinates
- **y** (Tensor) – y-coordinates
- **reorder** (bool) – if True, will reorder the arrays

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.)
```

auroc [func]

pytorch_lightning.metrics.functional.**auroc**(*preds*, *target*, *num_classes=None*, *pos_label=None*, *average='macro'*, *max_fpr=None*, *sample_weights=None*)

Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC)

Parameters

- **preds** (Tensor) – predictions from model (logits or probabilities)
- **target** (Tensor) – Ground truth labels
- **num_classes** (Optional[int]) – integer with number of classes. Not nessesary to provide for binary problems.
- **pos_label** (Optional[int]) – integer determining the positive class. Default is None which for binary problem is translate to 1. For multiclass problems this argument should not be set as we iteratively change it in the range [0,num_classes-1]
- **average** (Optional[str]) –
  - 'macro' computes metric for each class and uniformly averages them
  - 'weighted' computes metric for each class and does a weighted-average, where each class is weighted by their support (accounts for class imbalance)
  - None computes and returns the metric per class
- **max_fpr** (Optional[float]) – If not None, calculates standardized partial AUC over the range [0, max_fpr]. Should be a float between 0 and 1.
- **sample_weight** – sample weights for each data point

Example (binary case):

```
>>> preds = torch.tensor([0.13, 0.26, 0.08, 0.19, 0.34])
>>> target = torch.tensor([0, 0, 1, 1, 1])
>>> auroc(preds, target, pos_label=1)
tensor(0.5000)
```

Example (multiclass case):

```
>>> preds = torch.tensor([[0.90, 0.05, 0.05],
...                       [0.05, 0.90, 0.05],
...                       [0.05, 0.05, 0.90],
...                       [0.85, 0.05, 0.10],
...                       [0.10, 0.10, 0.80]])
>>> target = torch.tensor([0, 1, 1, 2, 2])
>>> auroc(preds, target, num_classes=3)
tensor(0.7778)
```

Return type Tensor

average_precision [func]

pytorch_lightning.metrics.functional.**average_precision**(*preds*, *target*, *num_classes=None*, *pos_label=None*, *sample_weights=None*)

Computes the average precision score.

Parameters

- **preds** (Tensor) – predictions from model (logits or probabilities)
- **target** (Tensor) – ground truth values
- **num_classes** (Optional[int]) – integer with number of classes. Not nessesary to provide for binary problems.
- **pos_label** (Optional[int]) – integer determining the positive class. Default is None which for binary problem is translate to 1. For multiclass problems this argument should not be set as we iteratively change it in the range [0,num_classes-1]
- **sample_weights** (Optional[Sequence]) – sample weights for each data point

Return type Union[List[Tensor], Tensor]

Returns tensor with average precision. If multiclass will return list of such tensors, one for each class

Example (binary case):

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 1, 1])
>>> average_precision(pred, target, pos_label=1)
tensor(1.)
```

Example (multiclass case):

```
>>> pred = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                      [0.05, 0.75, 0.05, 0.05, 0.05],
...                      [0.05, 0.05, 0.75, 0.05, 0.05],
...                      [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> average_precision(pred, target, num_classes=5)
[tensor(1.), tensor(1.), tensor(0.2500), tensor(0.2500), tensor(nan)]
```

confusion_matrix [func]

pytorch_lightning.metrics.functional.**confusion_matrix**(*preds*, *target*, *num_classes*, *normalize=None*, *threshold=0.5*)

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

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

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

Parameters

- **preds** (Tensor) – (float or long tensor), Either a `(N, ...)` tensor with labels or `(N, C, ...)` where C is the number of classes, tensor with labels/probabilities
- **target** (Tensor) – `target` (long tensor), tensor with shape `(N, ...)` with ground true labels
- **num_classes** (int) – Number of classes in the dataset.
- **normalize** (Optional[str]) – Normalization mode for confusion matrix. Choose from
  - `None` or `'none'`: no normalization (default)
  - `'true'`: normalization over the targets (most commonly used)
  - `'pred'`: normalization over the predictions
  - `'all'`: normalization over the whole matrix
- **threshold** (float) – Threshold value for binary or multi-label probabilities. default: 0.5

Example

```
>>> from pytorch_lightning.metrics.functional import confusion_matrix
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0, 1, 0, 0])
>>> confusion_matrix(preds, target, num_classes=2)
tensor([[2., 0.],
        [1., 1.]])
```

Return type Tensor

dice_score [func]

pytorch_lightning.metrics.functional.classification.**dice_score**(*pred*, *target*, *bg=False*, *nan_score=0.0*, *no_fg_score=0.0*, *reduction='elementwise_mean'*)

Compute dice score from prediction scores

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
- **no_fg_score** (float) – score to return, if no foreground pixel was found in target
- **reduction** (str) – a method to reduce metric score over labels.
  - `'elementwise_mean'`: takes the mean (default)
  - `'sum'`: takes the sum

- 'none': no reduction will be applied

Return type Tensor

Returns Tensor containing dice score

Example

```
>>> pred = torch.tensor([[0.85, 0.05, 0.05, 0.05],
...                      [0.05, 0.85, 0.05, 0.05],
...                      [0.05, 0.05, 0.85, 0.05],
...                      [0.05, 0.05, 0.05, 0.85]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> dice_score(pred, target)
tensor(0.3333)
```

f1 [func]

pytorch_lightning.metrics.functional.**f1**(*preds*, *target*, *num_classes*, *threshold=0.5*, *average='micro'*, *multilabel=False*)

Computes F1 metric. F1 metrics correspond to a equally weighted average of the precision and recall scores.

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

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

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

Parameters

- **preds** (Tensor) – predictions from model (probabilities, or labels)
- **target** (Tensor) – ground truth labels
- **num_classes** (int) – Number of classes in the dataset.
- **threshold** (float) – Threshold value for binary or multi-label probabilities. default: 0.5
- **average** (str) –
  - 'micro' computes metric globally
  - 'macro' computes metric for each class and uniformly averages them
  - 'weighted' computes metric for each class and does a weighted-average, where each class is weighted by their support (accounts for class imbalance)
  - 'none' or None computes and returns the metric per class
- **multilabel** (bool) – If predictions are from multilabel classification.

Example

```
>>> from pytorch_lightning.metrics.functional import f1
>>> target = torch.tensor([0, 1, 2, 0, 1, 2])
>>> preds = torch.tensor([0, 2, 1, 0, 0, 1])
>>> f1(preds, target, num_classes=3)
tensor(0.3333)
```

Return type Tensor

fbeta [func]

pytorch_lightning.metrics.functional.**fbeta**(*preds*, *target*, *num_classes*, *beta=1.0*, *threshold=0.5*, *average='micro'*, *multilabel=False*)

Computes f_beta metric.

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

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

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

Parameters

- **preds** (Tensor) – predictions from model (probabilities, or labels)
- **target** (Tensor) – ground truth labels
- **num_classes** (int) – Number of classes in the dataset.
- **beta** (float) – Beta coefficient in the F measure.
- **threshold** (float) – Threshold value for binary or multi-label probabilities. default: 0.5
- **average** (str) –
  - `'micro'` computes metric globally
  - `'macro'` computes metric for each class and uniformly averages them
  - `'weighted'` computes metric for each class and does a weighted-average, where each class is weighted by their support (accounts for class imbalance)
  - `'none'` or `None` computes and returns the metric per class
- **multilabel** (bool) – If predictions are from multilabel classification.

Example

```
>>> from pytorch_lightning.metrics.functional import fbeta
>>> target = torch.tensor([0, 1, 2, 0, 1, 2])
>>> preds = torch.tensor([0, 2, 1, 0, 0, 1])
>>> fbeta(preds, target, num_classes=3, beta=0.5)
tensor(0.3333)
```

Return type Tensor

hamming_distance [func]

pytorch_lightning.metrics.functional.**hamming_distance**(*preds*, *target*, *threshold=0.5*)

Computes the average Hamming distance (also known as Hamming loss) between targets and predictions:

$$\text{Hamming distance} = \frac{1}{N \cdot L} \sum_{i}^{N} \sum_{l}^{L} 1(y_{il} \neq \hat{y}_{il})$$

Where y is a tensor of target values, $\hat{y}$ is a tensor of predictions, and $\bullet_{il}$ refers to the l-th label of the i-th sample of that tensor.

This is the same as `1-accuracy` for binary data, while for all other types of inputs it treats each possible label separately - meaning that, for example, multi-class data is treated as if it were multi-label.

Accepts all input types listed in *Input types*.

Parameters

- **preds** (Tensor) – Predictions from model
- **target** (Tensor) – Ground truth
- **threshold** (float) – Threshold probability value for transforming probability predictions to binary (0 or 1) predictions, in the case of binary or multi-label inputs.

Example

```
>>> from pytorch_lightning.metrics.functional import hamming_distance
>>> target = torch.tensor([[0, 1], [1, 1]])
>>> preds = torch.tensor([[0, 1], [0, 1]])
>>> hamming_distance(preds, target)
tensor(0.2500)
```

Return type Tensor

iou [func]

pytorch_lightning.metrics.functional.**iou**(*pred*, *target*, *ignore_index=None*, *absent_score=0.0*, *threshold=0.5*, *num_classes=None*, *reduction='elementwise_mean'*)

Computes Intersection over union, or Jaccard index calculation:

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

Where: A and B are both tensors of the same size, containing integer class values. They may be subject to conversion from input data (see description below).

Note that it is different from box IoU.

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

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

Parameters

- **preds** – tensor containing predictions from model (probabilities, or labels) with shape `[N, d1, d2, ...]`
- **target** (`Tensor`) – tensor containing ground truth labels with shape `[N, d1, d2, ...]`
- **ignore_index** (`Optional[int]`) – optional int specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. Has no effect if given an int that is not in the range [0, num_classes-1], where num_classes is either given or derived from pred and target. By default, no index is ignored, and all classes are used.
- **absent_score** (`float`) – score to use for an individual class, if no instances of the class index were present in *pred* AND no instances of the class index were present in *target*. For example, if we have 3 classes, [0, 0] for *pred*, and [0, 2] for *target*, then class 1 would be assigned the *absent_score*.
- **threshold** (`float`) – Threshold value for binary or multi-label probabilities. default: 0.5
- **num_classes** (`Optional[int]`) – Optionally specify the number of classes
- **reduction** (`str`) – a method to reduce metric score over labels.
  - `'elementwise_mean'`: takes the mean (default)
  - `'sum'`: takes the sum
  - `'none'`: no reduction will be applied

Returns Tensor containing single value if reduction is 'elementwise_mean', or number of classes if reduction is 'none'

Return type IoU score

Example

```
>>> target = torch.randint(0, 2, (10, 25, 25))
>>> pred = torch.tensor(target)
>>> pred[2:5, 7:13, 9:15] = 1 - pred[2:5, 7:13, 9:15]
>>> iou(pred, target)
tensor(0.9660)
```

roc [func]

pytorch_lightning.metrics.functional.**roc**(*preds*, *target*, *num_classes=None*, *pos_label=None*, *sample_weights=None*)

Computes the Receiver Operating Characteristic (ROC).

Parameters

- **preds** (Tensor) – predictions from model (logits or probabilities)
- **target** (Tensor) – ground truth values
- **num_classes** (Optional[int]) – integer with number of classes. Not nessesary to provide for binary problems.
- **pos_label** (Optional[int]) – integer determining the positive class. Default is None which for binary problem is translate to 1. For multiclass problems this argument should not be set as we iteratively change it in the range [0,num_classes-1]
- **sample_weights** (Optional[Sequence]) – sample weights for each data point

Returns: 3-element tuple containing

fpr: tensor with false positive rates. If multiclass, this is a list of such tensors, one for each class.

tpr: tensor with true positive rates. If multiclass, this is a list of such tensors, one for each class.

thresholds: thresholds used for computing false- and true postive rates

Example (binary case):

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 1, 1])
>>> fpr, tpr, thresholds = roc(pred, target, pos_label=1)
>>> fpr
tensor([0., 0., 0., 0., 1.])
>>> tpr
tensor([0.0000, 0.3333, 0.6667, 1.0000, 1.0000])
>>> thresholds
tensor([4, 3, 2, 1, 0])
```

Example (multiclass case):

```
>>> pred = torch.tensor([[0.75, 0.05, 0.05, 0.05],
...                      [0.05, 0.75, 0.05, 0.05],
...                      [0.05, 0.05, 0.75, 0.05],
...                      [0.05, 0.05, 0.05, 0.75]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> fpr, tpr, thresholds = roc(pred, target, num_classes=4)
>>> fpr
[tensor([0., 0., 1.]), tensor([0., 0., 1.]), tensor([0.0000, 0.3333, 1.0000]),␣
↪tensor([0.0000, 0.3333, 1.0000])]
```

(continues on next page)

(continued from previous page)

```
>>> tpr
[tensor([0., 1., 1.]), tensor([0., 1., 1.]), tensor([0., 0., 1.]), tensor([0., 0.,
↪ 1.])]
>>> thresholds
[tensor([1.7500, 0.7500, 0.0500]),
 tensor([1.7500, 0.7500, 0.0500]),
 tensor([1.7500, 0.7500, 0.0500]),
 tensor([1.7500, 0.7500, 0.0500])]
```

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

precision [func]

pytorch_lightning.metrics.functional.**precision**(*preds*, *target*, *average='micro'*, *mdmc_average=None*, *ignore_index=None*, *num_classes=None*, *threshold=0.5*, *top_k=None*, *is_multiclass=None*, *class_reduction=None*)

Computes Precision:

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

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

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

Parameters

- **preds** (Tensor) – Predictions from model (probabilities or labels)
- **target** (Tensor) – Ground truth values
- **average** (str) – Defines the reduction that is applied. Should be one of the following:
  - `'micro'` [default]: Calculate the metric globally, accross all samples and classes.
  - `'macro'`: Calculate the metric for each class separately, and average the metrics accross classes (with equal weights for each class).
  - `'weighted'`: Calculate the metric for each class separately, and average the metrics accross classes, weighting each class by its support (`tp + fn`).
  - `'none'` or `None`: Calculate the metric for each class separately, and return the metric for every class.
  - `'samples'`: Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

  Note that what is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_average`.
- **class_reduction** (Optional[str]) –

> **Warning:** This parameter is deprecated, use `average`. Will be removed in v1.4.0.

- **mdmc_average** (Optional[str]) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
  - `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - `'samplewise'`: In this case, the statistics are computed separately for each sample on the `N` axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes `...` (see *Input types*) as the `N` dimension within the sample, and computing the metric for the sample based on that.
  - `'global'`: In this case the `N` and `...` dimensions of the inputs (see *Input types*) are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were `(N_X, C)`. From here on the `average` parameter applies as usual.
- **ignore_index** (Optional[int]) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or `'none'`, the score for the ignored class will be returned as `nan`.
- **num_classes** (Optional[int]) – Number of classes. Necessary for `'macro'`, `'weighted'` and `None` average methods.
- **threshold** (float) – Threshold probability value for transforming probability predictions to binary (0,1) predictions, in the case of binary or multi-label inputs.
- **top_k** (Optional[int]) – Number of highest probability entries for each sample to convert to 1s - relevant only for inputs with probability predictions. If this parameter is set for multi-label inputs, it will take precedence over `threshold`. For (multi-dim) multi-class inputs, this parameter defaults to 1.

  Should be left unset (`None`) for inputs with label predictions.
- **is_multiclass** (Optional[bool]) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.
- **class_reduction** –

> **Warning:** This parameter is deprecated, use `average`. Will be removed in v1.4.0.

Return type Tensor

Returns

The shape of the returned tensor depends on the `average` parameter

- If `average in ['micro', 'macro', 'weighted', 'samples']`, a one-element tensor will be returned
- If `average in ['none', None]`, the shape will be `(C,)`, where `C` stands for the number of classes

Example

```
>>> from pytorch_lightning.metrics.functional import precision
>>> preds  = torch.tensor([2, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> precision(preds, target, average='macro', num_classes=3)
tensor(0.1667)
>>> precision(preds, target, average='micro')
tensor(0.2500)
```

precision_recall [func]

pytorch_lightning.metrics.functional.**precision_recall**(*preds*, *target*, *average='micro'*, *mdmc_average=None*, *ignore_index=None*, *num_classes=None*, *threshold=0.5*, *top_k=None*, *is_multiclass=None*, *class_reduction=None*)

Computes Precision and Recall:

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

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where TPtext{FN}` and FP represent the number of true positives, false negatives and false positives respecitively. With the use of `top_k` parameter, this metric can generalize to Recall@K and Precision@K.

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

Parameters

- **preds** (`Tensor`) – Predictions from model (probabilities, or labels)
- **target** (`Tensor`) – Ground truth values
- **average** (`str`) – Defines the reduction that is applied. Should be one of the following:
  - `'micro'` [default]: Calculate the metric globally, accross all samples and classes.
  - `'macro'`: Calculate the metric for each class separately, and average the metrics accross classes (with equal weights for each class).
  - `'weighted'`: Calculate the metric for each class separately, and average the metrics accross classes, weighting each class by its support (`tp + fn`).
  - `'none'` or `None`: Calculate the metric for each class separately, and return the metric for every class.
  - `'samples'`: Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

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

- **mdmc_average**¶ (Optional[str]) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
  - `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - `'samplewise'`: In this case, the statistics are computed separately for each sample on the `N` axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes `...` (see *Input types*) as the `N` dimension within the sample, and computing the metric for the sample based on that.
  - `'global'`: In this case the `N` and `...` dimensions of the inputs (see *Input types*) are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were `(N_X, C)`. From here on the `average` parameter applies as usual.
- **ignore_index**¶ (Optional[int]) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or `'none'`, the score for the ignored class will be returned as `nan`.
- **num_classes**¶ (Optional[int]) – Number of classes. Necessary for `'macro'`, `'weighted'` and `None` average methods.
- **threshold**¶ (float) – Threshold probability value for transforming probability predictions to binary (0,1) predictions, in the case of binary or multi-label inputs
- **top_k**¶ (Optional[int]) – Number of highest probability entries for each sample to convert to 1s - relevant only for inputs with probability predictions. If this parameter is set for multi-label inputs, it will take precedence over `threshold`. For (multi-dim) multi-class inputs, this parameter defaults to 1.

  Should be left unset (`None`) for inputs with label predictions.
- **is_multiclass**¶ (Optional[bool]) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.
- **class_reduction**¶ (Optional[str]) –

  Warning: This parameter is deprecated, use `average`. Will be removed in v1.4.0.

Returns

precision and recall. Their shape depends on the `average` parameter

- If `average in ['micro', 'macro', 'weighted', 'samples']`, they are a single element tensor
- If `average in ['none', None]`, they are a tensor of shape `(C, )`, where `C` stands for the number of classes

Return type The function returns a tuple with two elements

Example

```
>>> from pytorch_lightning.metrics.functional import precision_recall
>>> preds  = torch.tensor([2, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> precision_recall(preds, target, average='macro', num_classes=3)
(tensor(0.1667), tensor(0.3333))
>>> precision_recall(preds, target, average='micro')
(tensor(0.2500), tensor(0.2500))
```

precision_recall_curve [func]

pytorch_lightning.metrics.functional.**precision_recall_curve**(*preds*, *target*, *num_classes=None*, *pos_label=None*, *sample_weights=None*)

Computes precision-recall pairs for different thresholds.

Parameters

- **preds** (Tensor) – predictions from model (probabilities)
- **target** (Tensor) – ground truth labels
- **num_classes** (Optional[int]) – integer with number of classes. Not nessesary to provide for binary problems.
- **pos_label** (Optional[int]) – integer determining the positive class. Default is None which for binary problem is translate to 1. For multiclass problems this argument should not be set as we iteratively change it in the range [0,num_classes-1]
- **sample_weights** (Optional[Sequence]) – sample weights for each data point

Returns: 3-element tuple containing

precision: tensor where element i is the precision of predictions with score >= thresholds[i] and the last element is 1. If multiclass, this is a list of such tensors, one for each class.

recall: tensor where element i is the recall of predictions with score >= thresholds[i] and the last element is 0. If multiclass, this is a list of such tensors, one for each class.

thresholds: Thresholds used for computing precision/recall scores

Example (binary case):

```
>>> pred = torch.tensor([0, 1, 2, 3])
>>> target = torch.tensor([0, 1, 1, 0])
>>> precision, recall, thresholds = precision_recall_curve(pred, target, pos_
↪label=1)
>>> precision
tensor([0.6667, 0.5000, 0.0000, 1.0000])
>>> recall
tensor([1.0000, 0.5000, 0.0000, 0.0000])
>>> thresholds
tensor([1, 2, 3])
```

Example (multiclass case):

```
>>> pred = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                      [0.05, 0.75, 0.05, 0.05, 0.05],
...                      [0.05, 0.05, 0.75, 0.05, 0.05],
...                      [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> precision, recall, thresholds = precision_recall_curve(pred, target, num_
↪classes=5)
>>> precision
[tensor([1., 1.]), tensor([1., 1.]), tensor([0.2500, 0.0000, 1.0000]),
 tensor([0.2500, 0.0000, 1.0000]), tensor([0., 1.])]
>>> recall
[tensor([1., 0.]), tensor([1., 0.]), tensor([1., 0., 0.]), tensor([1., 0., 0.]),␣
↪tensor([nan, 0.])]
>>> thresholds
[tensor([0.7500]), tensor([0.7500]), tensor([0.0500, 0.7500]), tensor([0.0500, 0.
↪7500]), tensor([0.0500])]
```

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

recall [func]

pytorch_lightning.metrics.functional.**recall**(*preds*, *target*, *average='micro'*, *mdmc_average=None*, *ignore_index=None*, *num_classes=None*, *threshold=0.5*, *top_k=None*, *is_multiclass=None*, *class_reduction=None*)

Computes Recall:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where TP and FN represent the number of true positives and false negatives respecitively. With the use of `top_k` parameter, this metric can generalize to Recall@K.

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

Parameters

- **preds** (Tensor) – Predictions from model (probabilities, or labels)
- **target** (Tensor) – Ground truth values
- **average** (str) – Defines the reduction that is applied. Should be one of the following:
  - `'micro'` [default]: Calculate the metric globally, accross all samples and classes.
  - `'macro'`: Calculate the metric for each class separately, and average the metrics accross classes (with equal weights for each class).
  - `'weighted'`: Calculate the metric for each class separately, and average the metrics accross classes, weighting each class by its support (`tp + fn`).
  - `'none'` or `None`: Calculate the metric for each class separately, and return the metric for every class.
  - `'samples'`: Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

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

- **mdmc_average** (Optional[str]) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
  - `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - `'samplewise'`: In this case, the statistics are computed separately for each sample on the `N` axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes `...` (see *Input types*) as the `N` dimension within the sample, and computing the metric for the sample based on that.
  - `'global'`: In this case the `N` and `...` dimensions of the inputs (see *Input types*) are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were `(N_X, C)`. From here on the `average` parameter applies as usual.
- **ignore_index** (Optional[int]) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or `'none'`, the score for the ignored class will be returned as `nan`.
- **num_classes** (Optional[int]) – Number of classes. Necessary for `'macro'`, `'weighted'` and `None` average methods.
- **threshold** (float) – Threshold probability value for transforming probability predictions to binary (0,1) predictions, in the case of binary or multi-label inputs
- **top_k** (Optional[int]) – Number of highest probability entries for each sample to convert to 1s - relevant only for inputs with probability predictions. If this parameter is set for multi-label inputs, it will take precedence over `threshold`. For (multi-dim) multi-class inputs, this parameter defaults to 1.

  Should be left unset (`None`) for inputs with label predictions.
- **is_multiclass** (Optional[bool]) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.
- **class_reduction** (Optional[str]) –

  Warning: This parameter is deprecated, use `average`. Will be removed in v1.4.0.

Return type Tensor

Returns

The shape of the returned tensor depends on the `average` parameter

- If `average in ['micro', 'macro', 'weighted', 'samples']`, a one-element tensor will be returned
- If `average in ['none', None]`, the shape will be `(C,)`, where `C` stands for the number of classes

Example

```
>>> from pytorch_lightning.metrics.functional import recall
>>> preds  = torch.tensor([2, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> recall(preds, target, average='macro', num_classes=3)
tensor(0.3333)
>>> recall(preds, target, average='micro')
tensor(0.2500)
```

select_topk [func]

pytorch_lightning.metrics.utils.**select_topk**(*prob_tensor*, *topk=1*, *dim=1*)

Convert a probability tensor to binary by selecting top-k highest entries.

Parameters

- **prob_tensor** (Tensor) – dense tensor of shape `[..., C, ...]`, where `C` is in the position defined by the `dim` argument
- **topk** (int) – number of highest entries to turn into 1s
- **dim** (int) – dimension on which to compare entries

Output: A binary tensor of the same shape as the input tensor of type torch.int32

Example

```
>>> x = torch.tensor([[1.1, 2.0, 3.0], [2.0, 1.0, 0.5]])
>>> select_topk(x, topk=2)
tensor([[0, 1, 1],
        [1, 1, 0]], dtype=torch.int32)
```

Return type Tensor

stat_scores [func]

pytorch_lightning.metrics.functional.**stat_scores**(*preds*, *target*, *reduce='micro'*, *mdmc_reduce=None*, *num_classes=None*, *top_k=None*, *threshold=0.5*, *is_multiclass=None*, *ignore_index=None*)

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

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

Parameters

- **preds** (Tensor) – Predictions from model (probabilities or labels)
- **target** (Tensor) – Ground truth values

- **threshold**¶ (float) – Threshold probability value for transforming probability predictions to binary (0 or 1) predictions, in the case of binary or multi-label inputs.
- **top_k**¶ (Optional[int]) – Number of highest probability entries for each sample to convert to 1s - relevant only for inputs with probability predictions. If this parameter is set for multi-label inputs, it will take precedence over threshold. For (multi-dim) multi-class inputs, this parameter defaults to 1.

  Should be left unset (None) for inputs with label predictions.
- **reduce**¶ (str) – Defines the reduction that is applied. Should be one of the following:
  - 'micro' [default]: Counts the statistics by summing over all [sample, class] combinations (globally). Each statistic is represented by a single integer.
  - 'macro': Counts the statistics for each class separately (over all samples). Each statistic is represented by a (C,) tensor. Requires num_classes to be set.
  - 'samples': Counts the statistics for each sample separately (over all classes). Each statistic is represented by a (N,) 1d tensor.

  Note that what is considered a sample in the multi-dimensional multi-class case depends on the value of mdmc_reduce.
- **num_classes**¶ (Optional[int]) – Number of classes. Necessary for (multi-dimensional) multi-class or multi-label data.
- **ignore_index**¶ (Optional[int]) – Specify a class (label) to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and reduce='macro', the class statistics for the ignored class will all be returned as -1.
- **mdmc_reduce**¶ (Optional[str]) – Defines how the multi-dimensional multi-class inputs are handeled. Should be one of the following:
  - None [default]: Should be left unchanged if your data is not multi-dimensional multi-class (see *Input types* for the definition of input types).
  - 'samplewise': In this case, the statistics are computed separately for each sample on the N axis, and then the outputs are concatenated together. In each sample the extra axes ... are flattened to become the sub-sample axis, and statistics for each sample are computed by treating the sub-sample axis as the N axis for that sample.
  - 'global': In this case the N and ... dimensions of the inputs are flattened into a new N_X sample axis, i.e. the inputs are treated as if they were (N_X, C). From here on the reduce parameter applies as usual.
- **is_multiclass**¶ (Optional[bool]) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be. See the parameter's *documentation section* for a more detailed explanation and examples.

Return type Tensor

Returns

The metric returns a tensor of shape (..., 5), where the last dimension corresponds to [tp, fp, tn, fn, sup] (sup stands for support and equals tp + fn). The shape depends on the reduce and mdmc_reduce (in case of multi-dimensional multi-class data) parameters:

- If the data is not multi-dimensional multi-class, then
  - If reduce='micro', the shape will be (5,)

  - If `reduce='macro'`, the shape will be `(C, 5)`, where `C` stands for the number of classes
  - If `reduce='samples'`, the shape will be `(N, 5)`, where `N` stands for the number of samples
- If the data is multi-dimensional multi-class and `mdmc_reduce='global'`, then
  - If `reduce='micro'`, the shape will be `(5, )`
  - If `reduce='macro'`, the shape will be `(C, 5)`
  - If `reduce='samples'`, the shape will be `(N*X, 5)`, where `X` stands for the product of sizes of all "extra" dimensions of the data (i.e. all dimensions except for `C` and `N`)
- If the data is multi-dimensional multi-class and `mdmc_reduce='samplewise'`, then
  - If `reduce='micro'`, the shape will be `(N, 5)`
  - If `reduce='macro'`, the shape will be `(N, C, 5)`
  - If `reduce='samples'`, the shape will be `(N, X, 5)`

Example

```
>>> from pytorch_lightning.metrics.functional import stat_scores
>>> preds  = torch.tensor([1, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> stat_scores(preds, target, reduce='macro', num_classes=3)
tensor([[0, 1, 2, 1, 1],
        [1, 1, 1, 1, 2],
        [1, 0, 3, 0, 1]])
>>> stat_scores(preds, target, reduce='micro')
tensor([2, 2, 6, 2, 4])
```

stat_scores_multiple_classes [func]

pytorch_lightning.metrics.functional.classification.**stat_scores_multiple_classes**(*pred*, *target*, *num_classes=*, *argmax_dim=*, *reduction='none'*)

Calculates the number of true positive, false positive, true negative and false negative for each class

Warning: Deprecated in favor of `stat_scores()`

Return type `Tuple[Tensor, Tensor, Tensor, Tensor, Tensor]`

to_categorical [func]

pytorch_lightning.metrics.utils.**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

Return type Tensor

Returns A tensor with categorical labels [N, d2, ...]

Example

```
>>> x = torch.tensor([[0.2, 0.5], [0.9, 0.1]])
>>> to_categorical(x)
tensor([1, 0])
```

to_onehot [func]

pytorch_lightning.metrics.utils.**to_onehot**(*label_tensor*, *num_classes=None*)

Converts a dense label tensor to one-hot format

Parameters

- **label_tensor** (Tensor) – dense label tensor, with shape [N, d1, d2, ...]
- **num_classes** (Optional[int]) – number of classes C

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

13.7 Regression Metrics

13.7.1 Class Metrics (Regression)

ExplainedVariance

class pytorch_lightning.metrics.regression.**ExplainedVariance**(*multioutput='uniform_average'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Bases: `torch.nn.`, `abc.ABC`

Computes explained variance:

$$\text{ExplainedVariance} = 1 - \frac{\text{Var}(y - \hat{y})}{\text{Var}(y)}$$

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

Forward accepts

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

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

Parameters

- **multioutput** (`str`) – Defines aggregation in the case of multiple output scores. Can be one of the following strings (default is *'uniform_average'*.):
  - *'raw_values'* returns full set of scores
  - *'uniform_average'* scores are uniformly averaged
  - *'variance_weighted'* scores are weighted by their individual variances
- **compute_on_step** (`bool`) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step** (`bool`) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False
- **process_group** (`Optional[Any]`) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

Example

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

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

compute()
Computes explained variance over state.

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

Parameters

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

MeanAbsoluteError

class pytorch_lightning.metrics.regression.**MeanAbsoluteError**(*compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Bases: torch.nn., abc.ABC

Computes mean absolute error (MAE):

$$\text{MAE} = \frac{1}{N}\sum_{i}^{N}|y_i - \hat{y}_i|$$

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

Parameters

- **compute_on_step**¶ (bool) – Forward only calls update() and return None if this is set to False. default: True
- **dist_sync_on_step**¶ (bool) – Synchronize metric state across processes at each forward() before returning the value at the step. default: False
- **process_group**¶ (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

Example

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

compute()
Computes mean absolute error over state.

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

Parameters

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

MeanSquaredError

class pytorch_lightning.metrics.regression.**MeanSquaredError**(*compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Bases: torch.nn., abc.ABC

Computes mean squared error (MSE):

$$\text{MSE} = \frac{1}{N}\sum_{i}^{N}(y_i - \hat{y}_i)^2$$

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

Parameters

- **compute_on_step** (bool) – Forward only calls update() and return None if this is set to False. default: True
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each forward() before returning the value at the step. default: False
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

Example

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

compute()
Computes mean squared error over state.

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

Parameters

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

MeanSquaredLogError

class pytorch_lightning.metrics.regression.**MeanSquaredLogError**(*compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Bases: torch.nn., abc.ABC

Computes mean squared logarithmic error (MSLE):

$$\text{MSLE} = \frac{1}{N}\sum_{i}^{N}(\log_e(1+y_i) - \log_e(1+\hat{y}_i))^2$$

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

Parameters

- **compute_on_step** (bool) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

Example

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

compute()
Compute mean squared logarithmic error over state.

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

Parameters

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

PSNR

class pytorch_lightning.metrics.regression.**PSNR**(*data_range=None*, *base=10.0*, *reduction='elementwise_mean'*, *dim=None*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)

Bases: torch.nn., abc.ABC

Computes peak signal-to-noise ratio (PSNR):

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

Where MSE denotes the mean-squared-error function.

Parameters

- **data_range** (Optional[float]) – the range of the data. If None, it is determined from the data (max - min). The `data_range` must be given when `dim` is not None.

- **base**¶ (float) – a base of a logarithm to use (default: 10)
- **reduction**¶ (str) – a method to reduce metric score over labels.
  - 'elementwise_mean': takes the mean (default)
  - 'sum': takes the sum
  - 'none': no reduction will be applied
- **dim**¶ (Union[int, Tuple[int, ...], None]) – Dimensions to reduce PSNR scores over, provided as either an integer or a list of integers. Default is None meaning scores will be reduced across all dimensions and all batches.
- **compute_on_step**¶ (bool) – Forward only calls update() and return None if this is set to False. default: True
- **dist_sync_on_step**¶ (bool) – Synchronize metric state across processes at each forward() before returning the value at the step. default: False
- **process_group**¶ (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

Example

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

compute()
Compute peak signal-to-noise ratio over state.

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

Parameters

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

SSIM

class pytorch_lightning.metrics.regression.**SSIM**(*kernel_size=(11, 11)*, *sigma=(1.5, 1.5)*, *reduction='elementwise_mean'*, *data_range=None*, *k1=0.01*, *k2=0.03*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*)

Bases: torch.nn., abc.ABC

Computes Structual Similarity Index Measure (SSIM).

Parameters

- **kernel_size**¶ (Sequence[int]) – size of the gaussian kernel (default: (11, 11))

- **sigma** (Sequence[float]) – Standard deviation of the gaussian kernel (default: (1.5, 1.5))
- **reduction** (str) – a method to reduce metric score over labels.
  - 'elementwise_mean': takes the mean (default)
  - 'sum': takes the sum
  - 'none': no reduction will be applied
- **data_range** (Optional[float]) – Range of the image. If None, it is determined from the image (max - min)
- **k1** (float) – Parameter of SSIM. Default: 0.01
- **k2** (float) – Parameter of SSIM. Default: 0.03

Returns Tensor with SSIM score

Example

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

compute()
Computes explained variance over state.

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

Parameters

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

R2Score

class pytorch_lightning.metrics.regression.**R2Score**(*num_outputs=1*, *adjusted=0*, *multioutput='uniform_average'*, *compute_on_step=True*, *dist_sync_on_step=False*, *process_group=None*, *dist_sync_fn=None*)

Bases: torch.nn., abc.ABC

Computes r2 score also known as coefficient of determination:

$$R^2 = 1 - \frac{SS_res}{SS_tot}$$

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

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

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

Forward accepts

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

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

Parameters

- **num_outputs** (int) – Number of outputs in multioutput setting (default is 1)
- **adjusted** (int) – number of independent regressors for calculating adjusted r2 score. Default 0 (standard r2 score).
- **multioutput** (str) – Defines aggregation in the case of multiple output scores. Can be one of the following strings (default is `'uniform_average'`.):
  - `'raw_values'` returns full set of scores
  - `'uniform_average'` scores are uniformly averaged
  - `'variance_weighted'` scores are weighted by their individual variances
- **compute_on_step** (bool) – Forward only calls `update()` and return None if this is set to False. default: True
- **dist_sync_on_step** (bool) – Synchronize metric state across processes at each `forward()` before returning the value at the step. default: False
- **process_group** (Optional[Any]) – Specify the process group on which synchronization is called. default: None (which selects the entire world)

Example

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

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

compute()
Computes r2 score over the metric states.

Return type Tensor

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

Parameters

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

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

13.7.2 Functional Metrics (Regression)

explained_variance [func]

pytorch_lightning.metrics.functional.**explained_variance**(*preds*, *target*, *multioutput='uniform_average'*)

Computes explained variance.

Parameters

- **preds**¶ (Tensor) – estimated labels
- **target**¶ (Tensor) – ground truth labels
- **multioutput**¶ (str) – Defines aggregation in the case of multiple output scores. Can be one of the following strings (default is *'uniform_average'*.):
  - *'raw_values'* returns full set of scores
  - *'uniform_average'* scores are uniformly averaged
  - *'variance_weighted'* scores are weighted by their individual variances

Example

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

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

Return type Union[Tensor, Sequence[Tensor]]

image_gradients [func]

pytorch_lightning.metrics.functional.**image_gradients**(*img*)

Computes the gradients of a given image using finite difference

Parameters **img**¶ (Tensor) – An (N, C, H, W) input tensor where C is the number of image channels

Return type Tuple[Tensor, Tensor]

Returns Tuple of (dy, dx) with each gradient of shape [N, C, H, W]

Example

```
>>> image = torch.arange(0, 1*1*5*5, dtype=torch.float32)
>>> image = torch.reshape(image, (1, 1, 5, 5))
>>> dy, dx = image_gradients(image)
>>> dy[0, 0, :, :]
tensor([[5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [0., 0., 0., 0., 0.]])
```

Note: The implementation follows the 1-step finite difference method as followed by the TF implementation. The values are organized such that the gradient of [I(x+1, y)-[I(x, y)]] are at the (x, y) location

mean_absolute_error [func]

pytorch_lightning.metrics.functional.**mean_absolute_error**(*preds*, *target*)

Computes mean absolute error

Parameters

- **pred** – estimated labels
- **target** (Tensor) – ground truth labels

Return type Tensor

Returns Tensor with MAE

Example

```
>>> x = torch.tensor([0., 1, 2, 3])
>>> y = torch.tensor([0., 1, 2, 2])
>>> mean_absolute_error(x, y)
tensor(0.2500)
```

mean_squared_error [func]

pytorch_lightning.metrics.functional.**mean_squared_error**(*preds*, *target*)

Computes mean squared error

Parameters

- **preds** (Tensor) – estimated labels
- **target** (Tensor) – ground truth labels

Return type Tensor

Returns Tensor with MSE

Example

```
>>> x = torch.tensor([0., 1, 2, 3])
>>> y = torch.tensor([0., 1, 2, 2])
>>> mean_squared_error(x, y)
tensor(0.2500)
```

mean_squared_log_error [func]

pytorch_lightning.metrics.functional.**mean_squared_log_error**(*preds*, *target*)

Computes mean squared log error

Parameters

- **preds** (Tensor) – estimated labels
- **target** (Tensor) – ground truth labels

Return type Tensor

Returns Tensor with RMSLE

Example

```
>>> x = torch.tensor([0., 1, 2, 3])
>>> y = torch.tensor([0., 1, 2, 2])
>>> mean_squared_log_error(x, y)
tensor(0.0207)
```

psnr [func]

pytorch_lightning.metrics.functional.**psnr**(*preds*, *target*, *data_range=None*, *base=10.0*, *reduction='elementwise_mean'*, *dim=None*)

Computes the peak signal-to-noise ratio

Parameters

- **preds** (Tensor) – estimated signal
- **target** (Tensor) – groun truth signal
- **data_range** (Optional[float]) – the range of the data. If None, it is determined from the data (max - min). data_range must be given when dim is not None.
- **base** (float) – a base of a logarithm to use (default: 10)
- **reduction** (str) – a method to reduce metric score over labels.
  - 'elementwise_mean': takes the mean (default)
  - 'sum': takes the sum
  - 'none': no reduction will be applied
- **dim** (Union[int, Tuple[int, ...], None]) – Dimensions to reduce PSNR scores over provided as either an integer or a list of integers. Default is None meaning scores will be reduced across all dimensions.

Return type Tensor

Returns Tensor with PSNR score

Example

```
>>> pred = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
>>> target = torch.tensor([[3.0, 2.0], [1.0, 0.0]])
>>> psnr(pred, target)
tensor(2.5527)
```

ssim [func]

pytorch_lightning.metrics.functional.**ssim**(*preds*, *target*, *kernel_size=(11, 11)*, *sigma=(1.5, 1.5)*, *reduction='elementwise_mean'*, *data_range=None*, *k1=0.01*, *k2=0.03*)

Computes Structual Similarity Index Measure

Parameters

- **preds** (Tensor) – estimated image
- **target** (Tensor) – ground truth image
- **kernel_size** (Sequence[int]) – size of the gaussian kernel (default: (11, 11))
- **sigma** (Sequence[float]) – Standard deviation of the gaussian kernel (default: (1.5, 1.5))
- **reduction** (str) – a method to reduce metric score over labels.
  - 'elementwise_mean': takes the mean (default)
  - 'sum': takes the sum
  - 'none': no reduction will be applied
- **data_range** (Optional[float]) – Range of the image. If None, it is determined from the image (max - min)
- **k1** (float) – Parameter of SSIM. Default: 0.01
- **k2** (float) – Parameter of SSIM. Default: 0.03

Return type Tensor

Returns Tensor with SSIM score

Example

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

r2score [func]

pytorch_lightning.metrics.functional.**r2score**(*preds*, *target*, *adjusted=0*, *multioutput='uniform_average'*)

Computes r2 score also known as coefficient of determination:

$$R^2 = 1 - \frac{SS_res}{SS_tot}$$

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

$$R^2_a dj = 1 - \frac{(1 - R^2)(n - 1)}{n - k - 1}$$

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

Parameters

- **preds** (`Tensor`) – estimated labels
- **target** (`Tensor`) – ground truth labels
- **adjusted** (`int`) – number of independent regressors for calculating adjusted r2 score. Default 0 (standard r2 score).
- **multioutput** (`str`) – Defines aggregation in the case of multiple output scores. Can be one of the following strings (default is `'uniform_average'`.):
  - `'raw_values'` returns full set of scores
  - `'uniform_average'` scores are uniformly averaged
  - `'variance_weighted'` scores are weighted by their individual variances

Example

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

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

Return type `Tensor`

13.8 NLP

13.8.1 bleu_score [func]

pytorch_lightning.metrics.functional.nlp.**bleu_score**(*translate_corpus*, *reference_corpus*, *n_gram=4*, *smooth=False*)

Calculate BLEU score of machine translated text with one or more references

Parameters

- **translate_corpus** (Sequence[str]) – An iterable of machine translated corpus
- **reference_corpus** (Sequence[str]) – An iterable of iterables of reference corpus
- **n_gram** (int) – Gram value ranged from 1 to 4 (Default 4)
- **smooth** (bool) – Whether or not to apply smoothing – Lin et al. 2004

Return type Tensor

Returns Tensor with BLEU Score

Example

```
>>> translate_corpus = ['the cat is on the mat'.split()]
>>> reference_corpus = [['there is a cat on the mat'.split(), 'a cat is on the mat
↪'.split()]]
>>> bleu_score(translate_corpus, reference_corpus)
tensor(0.7598)
```

13.9 Pairwise

13.9.1 embedding_similarity [func]

pytorch_lightning.metrics.functional.self_supervised.**embedding_similarity**(*batch*, *similarity='cosine'*, *reduction='none'*, *zero_diagonal=True*)

Computes representation similarity

Example

```
>>> embeddings = torch.tensor([[1., 2., 3., 4.], [1., 2., 3., 4.], [4., 5., 6., 7.
↪]])
>>> embedding_similarity(embeddings)
tensor([[0.0000, 1.0000, 0.9759],
        [1.0000, 0.0000, 0.9759],
        [0.9759, 0.9759, 0.0000]])
```

Parameters

- **batch** (Tensor) – (batch, dim)
- **similarity** (str) – ‘dot’ or ‘cosine’
- **reduction** (str) – ‘none’, ‘sum’, ‘mean’ (all along dim -1)
- **zero_diagonal** (bool) – if True, the diagonals are set to zero

Return type Tensor

Returns A square matrix (batch, batch) with the similarity scores between all elements If sum or mean are used, then returns (b, 1) with the reduced value for each row

CHAPTER

FOURTEEN

PLUGINS

Plugins allow custom integrations to the internals of the Trainer such as a custom amp or ddp implementation.

For help setting up custom plugin/accelerator please reach out to us at support@pytorchlightning.ai

CHAPTER

FIFTEEN

STEP-BY-STEP WALK-THROUGH

This guide will walk you through the core pieces of PyTorch Lightning.

We'll accomplish the following:

- Implement an MNIST classifier.
- Use inheritance to implement an AutoEncoder

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

15.1 From MNIST to AutoEncoders

15.1.1 Installing Lightning

Lightning is trivial to install. We recommend using conda environments

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

Or without conda environments, use pip.

```
pip install pytorch-lightning
```

Or conda.

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

15.1.2 The research

The Model

The *lightning module* holds all the core research ingredients:

- The model
- The optimizers
- The train/ val/ test steps

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 = nn.Linear(28 * 28, 128)
    self.layer_2 = nn.Linear(128, 256)
    self.layer_3 = 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)
    x = self.layer_1(x)
    x = F.relu(x)
    x = self.layer_2(x)
    x = F.relu(x)
    x = self.layer_3(x)

    x = F.log_softmax(x, dim=1)
    return x
```

Notice this is a *lightning module* instead of a `torch.nn.Module`. A LightningModule is equivalent to a pure 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.randn(1, 1, 28, 28)
out = net(x)
```

Out:

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

Now we add the training_step which has all our training loop logic

```
class LitMNIST(LightningModule):

```

(continues on next page)

(continued from previous page)

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

Data

Lightning operates on pure dataloaders. 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, transform=transform)
mnist_train = DataLoader(mnist_train, batch_size=64)
```

You can use DataLoaders in 3 ways:

1. Pass DataLoaders to .fit()

Pass in the dataloaders to the *.fit()* function.

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

2. LightningModule DataLoaders

For fast research prototyping, it might be easier to link the model with the dataloaders.

```
class LitMNIST(pl.LightningModule):

    def train_dataloader(self):
        # 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,␣
↪transform=transform)
        return DataLoader(mnist_train, batch_size=64)

    def val_dataloader(self):
        transforms = ...
```

(continues on next page)

(continued from previous page)

```
        mnist_val = ...
        return DataLoader(mnist_val, batch_size=64)

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

DataLoaders are already in the model, no need to specify on .fit().

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

3. DataModules (recommended)

Defining free-floating dataloaders, splits, download instructions, and such can get messy. In this case, it's better to group the full definition of a dataset into a *DataModule* which includes:

- Download instructions
- Processing instructions
- Split instructions
- Train dataloader
- Val dataloader(s)
- Test dataloader(s)

```
class MyDataModule(LightningDataModule):

    def __init__(self):
        super().__init__()
        self.train_dims = None
        self.vocab_size = 0

    def prepare_data(self):
        # called only on 1 GPU
        download_dataset()
        tokenize()
        build_vocab()

    def setup(self):
        # called on every GPU
        vocab = load_vocab()
        self.vocab_size = len(vocab)

        self.train, self.val, self.test = load_datasets()
        self.train_dims = self.train.next_batch.size()

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

    def val_dataloader(self):
```

(continues on next page)

(continued from previous page)

```
        transforms = ...
        return DataLoader(self.val, batch_size=64)

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

Using DataModules allows easier sharing of full dataset definitions.

```
# use an MNIST dataset
mnist_dm = MNISTDatamodule()
model = LitModel(num_classes=mnist_dm.num_classes)
trainer.fit(model, mnist_dm)

# or other datasets with the same model
imagenet_dm = ImagenetDatamodule()
model = LitModel(num_classes=imagenet_dm.num_classes)
trainer.fit(model, imagenet_dm)
```

Note: `prepare_data()` is called on only one GPU in distributed training (automatically)

Note: `setup()` is called on every GPU (automatically)

Models defined by data

When your models need to know about the data, it's best to process the data before passing it to the model.

```
# init dm AND call the processing manually
dm = ImagenetDataModule()
dm.prepare_data()
dm.setup()

model = LitModel(out_features=dm.num_classes, img_width=dm.img_width, img_height=dm.
↪img_height)
trainer.fit(model, dm)
```

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

An alternative to using a DataModule is to defer initialization of the models modules to the `setup` method of your LightningModule as follows:

```
class LitMNIST(LightningModule):

    def __init__(self):
        self.l1 = None
```

(continues on next page)

(continued from previous page)

```
    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)
```

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)
```

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 ------
```

(continues on next page)

(continued from previous page)

```
        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 LightningModule.

```
class LitMNIST(LightningModule):

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

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...

15.1.3 The engineering

Training

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

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

(continues on next page)

(continued from previous page)

```
        x = F.relu(x)
        x = self.layer_2(x)
        x = F.relu(x)
        x = self.layer_3(x)
        x = F.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
```

Again, this is the same PyTorch code, except that it's organized by the LightningModule.

Logging

To log to Tensorboard, your favorite logger, and/or the progress bar, use the *log()* method which can be called from any method in the LightningModule.

```
def training_step(self, batch, batch_idx):
    self.log('my_metric', x)
```

The *log()* method has a few options:

- on_step (logs the metric at that step in training)
- on_epoch (automatically accumulates and logs at the end of the epoch)
- prog_bar (logs to the progress bar)
- logger (logs to the logger like Tensorboard)

Depending on where the log is called from, Lightning auto-determines the correct mode for you. But of course you can override the default behavior by manually setting the flags.

Note: Setting on_epoch=True will accumulate your logged values over the full training epoch.

```
def training_step(self, batch, batch_idx):
    self.log('my_loss', loss, on_step=True, on_epoch=True, prog_bar=True, logger=True)
```

You can also use any method of your logger directly:

```
def training_step(self, batch, batch_idx):
    tensorboard = self.logger.experiment
    tensorboard.any_summary_writer_method_you_want())
```

Once your training starts, you can view the logs by using your favorite logger or booting up the Tensorboard logs:

```
tensorboard --logdir ./lightning_logs
```

Which will generate automatic tensorboard logs (or with the logger of your choice).

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

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

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

Train on CPU

```
from pytorch_lightning import Trainer

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

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]
```

Train on GPU

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, train_loader)
```

```
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]
```

Train on Multi-GPU

Or you can also train on multiple GPUs.

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

Or multiple nodes

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

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

Train on TPUs

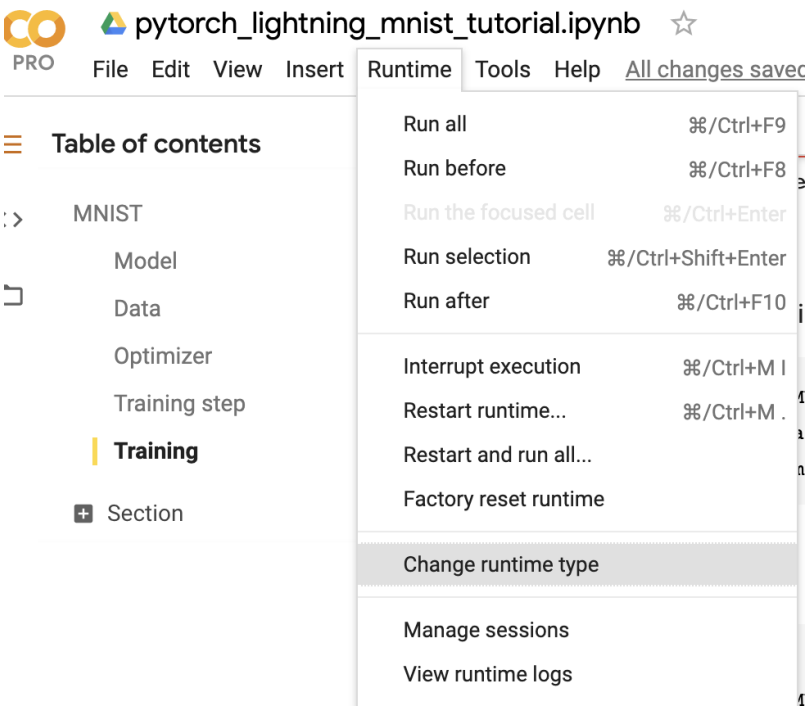
Did you know you can use PyTorch on TPUs? It's very hard to do, but we've worked with the xla team to use their awesome library to get this to work out of the box!

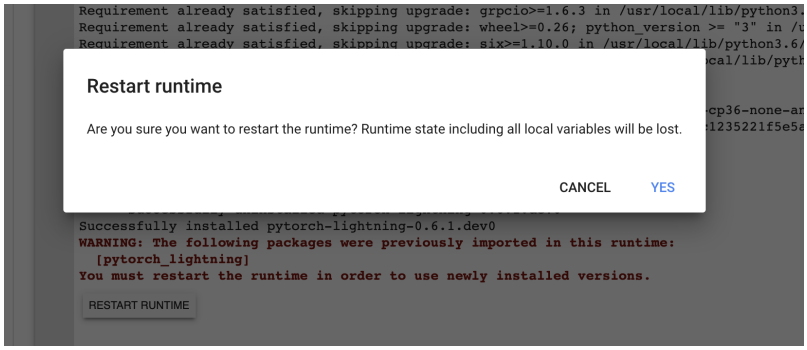
Let's train on Colab (full demo available here)

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
```


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


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

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, make sure your download code is in the `prepare_data` method in the DataModule. 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 MNISTDataModule(LightningDataModule):
    def __init__(self, batch_size=64):
        super().__init__()
        self.batch_size = batch_size

    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()])
        mnist_train = MNIST(os.getcwd(), train=True, download=False,␣
↪transform=transform)
        mnist_test = 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
        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=self.batch_size)

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

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

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!

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

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!
```

Hyperparameters

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

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
```

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 (i.e.: 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
```

(continues on next page)

(continued from previous page)

```
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)
```

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 (i.e.: 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 of 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')

        # Now possible to access layer_1_dim from hparams
        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 = nn.Linear(28 * 28, self.hparams.layer_1_dim)
        self.layer_2 = nn.Linear(self.hparams.layer_1_dim, self.hparams.layer_2_
↪dim)
        self.layer_3 = nn.Linear(self.hparams.layer_2_dim, 10)
    def train_dataloader(self):
        return DataLoader(mnist_train, batch_size=self.hparams.batch_size)
```

Warning: Deprecated since v1.1.0. This method of assigning hyperparameters to the LightningModule will no longer be supported from v1.3.0. Use the `self.save_hyperparameters()` method from above instead.

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.save_hyperparameters(conf)

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

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

# Now possible to access any stored variables from hparams
model.hparams.anything
```

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=[...
↪])
```

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 = 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)

    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':
```

(continues on next page)

(continued from previous page)

```
        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
```

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.

```
def validation_step(self, batch, batch_idx):
    loss = MSE_loss(...)
    self.log('val_loss', loss)
```

Now we can train with a validation loop as well.

```
from pytorch_lightning import Trainer

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

You may have noticed the words **Validation sanity check** logged. This is because Lightning runs 2 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 for a full epoch to find out.

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

Val loop under the hood

Under the hood, Lightning does the following:

```
model = Model()
model.train()
torch.set_grad_enabled(True)

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

(continues on next page)

(continued from previous page)

```
    # validate
    model.eval()
    torch.set_grad_enabled(False)

    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

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

Optional methods

If you still need even more fine-grain control, define the other optional methods for the loop.

```
def validation_step(self, batch, batch_idx):
    preds = ...
    return preds

def validation_epoch_end(self, val_step_outputs):
    for pred in val_step_outputs:
        # do something with all the predictions from each validation_step
```

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 a test loop

```
class LitMNIST(LightningModule):
    def test_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = F.nll_loss(logits, y)
        self.log('test_loss', loss)
```

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
result = trainer.test()
print(result)
```

Out:

```
--------------------------------------------------------------
TEST RESULTS
{'test_loss': 1.1703}
--------------------------------------------------------------
```

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.

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.randn(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 = F.relu(x)
        x = self.layer_2(x)
        x = F.relu(x)
        x = self.layer_3(x)
        x = F.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 = F.relu(x)
        x = self.layer_2(x1)
        x2 = F.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 = F.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)
```

To perform inference at scale, it is possible to use `trainer.predict` with LightningModule `predict` function By default, LightningModule `predict` calls forward, but it can be overriden to add any processing logic.

```
class LitMNISTDreamer(LightningModule):

    def forward(self, z):
```

(continues on next page)

(continued from previous page)

```
        imgs = self.decoder(z)
        return imgs

    def predict(self, batch, batch_idx: int , dataloader_idx: int = None):
        return self(batch)


model = LitMNISTDreamer()
trainer.predict(model, datamodule)
```

How you split up what goes in `forward` vs `training_step` vs `predict` depends on how you want to use this model for prediction. However, we recommend `forward` to contain only tensor operation with your model, `training_step` to encapsulate `forward` logic with logging, metrics and loss computation and `predict` to encapsulate `forward` with preprocess, postprocess functions.

15.1.4 The nonessentials

Extensibility

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

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):
    loss.backward()
```

With your own

```
class LitMNIST(LightningModule):

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

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

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()])
```

Tip: See full list of 12+ hooks in the *callbacks*.

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. We are extending our Autoencoder from the *LitMNIST*-module which already defines all the dataloading. 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()
        self.metric = MSE()

    def forward(self, x):
        return self.encoder(x)

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

        representation = self.encoder(x)
```

(continues on next page)

(continued from previous page)

```
        x_hat = self.decoder(representation)

        loss = self.metric(x, x_hat)
        return loss

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

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

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

        loss = self.metric(x, x_hat)
        self.log(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 should 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)
```

Transfer Learning

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):
```

(continues on next page)

(continued from previous page)

```
    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!

Example: Imagenet (computer Vision)

```
import torchvision.models as models

class ImagenetTransferLearning(LightningModule):
    def __init__(self):
        super().__init__()

        # init a pretrained resnet
        backbone = models.resnet50(pretrained=True)
        num_filters = backbone.fc.in_features
        layers = list(backbone.children())[:-1]
        self.feature_extractor = nn.Sequential(*layers)

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

    def forward(self, x):
        self.feature_extractor.eval()
        with torch.no_grad():
            representations = self.feature_extractor(x).flatten(1)
        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.

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
```

15.2 Why PyTorch Lightning

15.2.1 a. Less boilerplate

Research and production code starts with simple code, but quickly grows in complexity once you add GPU training, 16-bit, checkpointing, logging, etc...

PyTorch Lightning implements these features for you and tests them rigorously to make sure you can instead focus on the research idea.

Writing less engineering/bolierplate code means:

- fewer bugs
- faster iteration
- faster prototyping

15.2.2 b. More functionality

In PyTorch Lightning you leverage code written by hundreds of AI researchers, research engs and PhDs from the world's top AI labs, implementing all the latest best practices and SOTA features such as

- GPU, Multi GPU, TPU training
- Multi-node training
- Auto logging
- ...
- Gradient accumulation

15.2.3 c. Less error-prone

Why re-invent the wheel?

Use PyTorch Lightning to enjoy a deep learning structure that is rigorously tested (500+ tests) across CPUs/multi-GPUs/multi-TPUs on every pull-request.

We promise our collective team of 20+ from the top labs has thought about training more than you :)

15.2.4 d. Not a new library

PyTorch Lightning is organized PyTorch - no need to learn a new framework.

Switching your model to Lightning is straight forward - here's a 2-minute video on how to do it.

Your projects WILL grow in complexity and you WILL end up engineering more than trying out new ideas... Defer the hardest parts to Lightning!

15.3 Lightning Philosophy

Lightning structures your deep learning code in 4 parts:

- Research code
- Engineering code
- Non-essential code
- Data code

15.3.1 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 or GPT).

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

x1 = l1(x)
x2 = l2(x2)
out = decoder(features, x)

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

In Lightning, this code is organized into a *lightning module*.

15.3.2 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.

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

distributed = DistributedParallel(model)

with gpu_zero:
    download_data()

dist.barrier()
```

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

15.3.3 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.

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

In Lightning this code is organized into *callbacks*.

15.3.4 Data code

Lightning uses standard PyTorch DataLoaders or anything that gives a batch of data. This code tends to end up getting messy with transforms, normalization constants, and data splitting spread all over files.

```
# data
train = MNIST(...)
train, val = split(train, val)
test = MNIST(...)

# transforms
train_transforms = ...
val_transforms = ...
test_transforms = ...

# dataloader ...
# download with dist.barrier() for multi-gpu, etc...
```

This code gets especially complicated once you start doing multi-GPU training or needing info about the data to build your models.

In Lightning this code is organized inside a *datamodules*.

Tip: DataModules are optional but encouraged, otherwise you can use standard DataLoaders

CHAPTER

SIXTEEN

API REFERENCES

16.1 Core API

datamodule	LightningDataModule for loading DataLoaders with ease.
decorators	Decorator for LightningModule methods.
hooks	Various hooks to be used in the Lightning code.
lightning	nn.Module with additional great features.

16.1.1 datamodule

Functions

track_data_hook_calls	A decorator that checks if prepare_data/setup have been called.

Classes

LightningDataModule	A DataModule standardizes the training, val, test splits, data preparation and transforms.

LightningDataModule for loading DataLoaders with ease.

class pytorch_lightning.core.datamodule.**LightningDataModule**(**args*, ***kwargs*)

Bases: *pytorch_lightning.core.hooks.CheckpointHooks*, *pytorch_lightning.core.hooks.DataHooks*

A DataModule standardizes the training, val, test splits, data preparation and transforms. The main advantage is consistent data splits, data preparation and transforms across models.

Example:

```
class MyDataModule(LightningDataModule):
    def __init__(self):
        super().__init__()
    def prepare_data(self):
        # download, split, etc...
        # only called on 1 GPU/TPU in distributed
```

(continues on next page)

(continued from previous page)

```
    def setup(self):
        # make assignments here (val/train/test split)
        # called on every process in DDP
    def train_dataloader(self):
        train_split = Dataset(...)
        return DataLoader(train_split)
    def val_dataloader(self):
        val_split = Dataset(...)
        return DataLoader(val_split)
    def test_dataloader(self):
        test_split = Dataset(...)
        return DataLoader(test_split)
```

A DataModule implements 5 key methods:

- **prepare_data** (things to do on 1 GPU/TPU not on every GPU/TPU in distributed mode).
- **setup** (things to do on every accelerator in distributed mode).
- **train_dataloader** the training dataloader.
- **val_dataloader** the val dataloader(s).
- **test_dataloader** the test dataloader(s).

This allows you to share a full dataset without explaining how to download, split transform and process the data

classmethod add_argparse_args(*parent_parser*)
Extends existing argparse by default *LightningDataModule* attributes.

Return type ArgumentParser

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 *LightningDataModule*.
- ****kwargs¶** – Additional keyword arguments that may override ones in the parser or namespace. These must be valid DataModule arguments.

Example:

```
parser = ArgumentParser(add_help=False)
parser = LightningDataModule.add_argparse_args(parser)
module = LightningDataModule.from_argparse_args(args)
```

classmethod from_datasets(*train_dataset=None*, *val_dataset=None*, *test_dataset=None*, *batch_size=1*, *num_workers=0*)
Create an instance from torch.utils.data.Dataset.

Parameters

- **train_dataset¶** (Union[Dataset, Sequence[Dataset], Mapping[str, Dataset], None]) – (optional) Dataset to be used for train_dataloader()
- **val_dataset¶** (Union[Dataset, Sequence[Dataset], None]) – (optional) Dataset or list of Dataset to be used for val_dataloader()

- **test_dataset** (Union[Dataset, Sequence[Dataset], None]) – (optional) Dataset or list of Dataset to be used for test_dataloader()
- **batch_size** (int) – Batch size to use for each dataloader. Default is 1.
- **num_workers** (int) – Number of subprocesses to use for data loading. 0 means that the data will be loaded in the main process. Number of CPUs available.

classmethod get_init_arguments_and_types()

Scans the DataModule 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

abstract prepare_data(**args*, ***kwargs*)

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(stage)
model.train_dataloader()
model.val_dataloader()
model.test_dataloader()
```

size(*dim=None*)

Return the dimension of each input either as a tuple or list of tuples. You can index this just as you would with a torch tensor.

Return type Union[Tuple, int]

property dims
A tuple describing the shape of your data. Extra functionality exposed in size.

property has_prepared_data
Return bool letting you know if datamodule.prepare_data() has been called or not.

Returns True if datamodule.prepare_data() has been called. False by default.

Return type bool

property has_setup_fit
Return bool letting you know if datamodule.setup('fit') has been called or not.

Returns True if datamodule.setup('fit') has been called. False by default.

Return type bool

property has_setup_test
Return bool letting you know if datamodule.setup('test') has been called or not.

Returns True if datamodule.setup('test') has been called. False by default.

Return type bool

property test_transforms
Optional transforms (or collection of transforms) you can apply to test dataset

property train_transforms
Optional transforms (or collection of transforms) you can apply to train dataset

property val_transforms
Optional transforms (or collection of transforms) you can apply to validation dataset

pytorch_lightning.core.datamodule.**track_data_hook_calls**(*fn*)
A decorator that checks if prepare_data/setup have been called.

- When dm.prepare_data() is called, dm.has_prepared_data gets set to True
- When dm.setup('fit') is called, dm.has_setup_fit gets set to True
- When dm.setup('test') is called, dm.has_setup_test gets set to True
- When dm.setup() is called without stage arg, both dm.has_setup_fit and dm.has_setup_test get set to True

Parameters **fn** (*function*) – Function that will be tracked to see if it has been called.

Returns Decorated function that tracks its call status and saves it to private attrs in its obj instance.

Return type function

16.1.2 decorators

Functions

auto_move_data	Decorator for *LightningModule* methods for which input arguments should be moved automatically to the correct device.
parameter_validation	Decorator for to() method.

Decorator for LightningModule methods.

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
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.**parameter_validation**(*fn*)

Decorator for `to()` method. Validates that the module parameter lengths match after moving to the device. It is useful when tying weights on TPU's.

Parameters **fn**[] (`Callable`) – `.to` method

Note: TPU's require weights to be tied/shared after moving the module to the device. Failure to do this results in the initialization of new weights which are not tied. To overcome this issue, weights should be tied using the `on_post_move_to_device` model hook which is called after the module has been moved to the device.

See also:

- XLA Documentation

Return type `Callable`

16.1.3 hooks

Classes

CheckpointHooks	Hooks to be used with Checkpointing.
DataHooks	Hooks to be used for data related stuff.
ModelHooks	Hooks to be used in LightningModule.

Various hooks to be used in the Lightning code.

class pytorch_lightning.core.hooks.**CheckpointHooks**
Bases: object

Hooks to be used with Checkpointing.

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

class pytorch_lightning.core.hooks.**DataHooks**
Bases: object

Hooks to be used for data related stuff.

on_after_batch_transfer(*batch*, *dataloader_idx*)
Override to alter or apply batch augmentations to your batch after it is transferred to the device.

> **Warning:** `dataloader_idx` always returns 0, and will be updated to support the true `idx` in the future.

Note: This hook only runs on single GPU training and DDP (no data-parallel).

Parameters

- **batch** – A batch of data that needs to be altered or augmented.
- **dataloader_idx** – DataLoader idx for batch (Default: 0)

Returns A batch of data

Example:

```
def on_after_batch_transfer(self, batch, dataloader_idx):
    batch['x'] = gpu_transforms(batch['x'])
    return batch
```

See also:

- *on_before_batch_transfer()*
- *transfer_batch_to_device()*

on_before_batch_transfer(*batch*, *dataloader_idx*)
Override to alter or apply batch augmentations to your batch before it is transferred to the device.

> **Warning:** dataloader_idx always returns 0, and will be updated to support the true idx in the future.

Note: This hook only runs on single GPU training and DDP (no data-parallel).

Parameters

- **batch** – A batch of data that needs to be altered or augmented.
- **dataloader_idx** – DataLoader idx for batch

Returns A batch of data

Example:

```
def on_before_batch_transfer(self, batch, dataloader_idx):
    batch['x'] = transforms(batch['x'])
    return batch
```

See also:

- *on_after_batch_transfer()*

- *transfer_batch_to_device()*

predict_dataloader()
Implement one or multiple PyTorch DataLoaders for prediction.

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.

Note: In the case where you return multiple prediction dataloaders, the predict() will have an argument dataloader_idx which matches the order here.

---

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(stage)
model.train_dataloader()
model.val_dataloader()
model.test_dataloader()
```

Return type None

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`.

For data processing use the following pattern:

- download in *prepare_data()*
- process and split in `setup()`

However, the above are only necessary for distributed processing.

Warning: do not assign state in prepare_data

- `fit()`
- ...
- *prepare_data()*
- `setup()`
- *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

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

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

Note: In the case where you return multiple test dataloaders, the `test_step()` will have an argument `dataloader_idx` which matches the order here.

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`.

For data processing use the following pattern:

- download in *prepare_data()*
- process and split in `setup()`

However, the above are only necessary for distributed processing.

Warning: do not assign state in prepare_data

- `fit()`
- ...
- *prepare_data()*
- `setup()`
- *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
```

transfer_batch_to_device(*batch*, *device=None*)

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` or anything that implements *.to(...)*
- `list`
- `dict`
- `tuple`
- `torchtext.data.batch.Batch`

For anything else, you need to define how the data is moved to the target device (CPU, GPU, TPU, ...).

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).

Note: This hook only runs on single GPU training and DDP (no data-parallel). If you need multi-GPU support for your custom batch objects, you need to define your custom `DistributedDataParallel` or `LightningDistributedDataParallel` and override `configure_ddp()`.

Parameters

- **batch** (`Any`) – A batch of data that needs to be transferred to a new device.
- **device** (`Optional[device]`) – The target device as defined in PyTorch.

Return type `Any`

Returns A reference to the data on the new device.

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
```

See also:

- `move_data_to_device()`
- `apply_to_collection()`

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 `dataloader_idx` which matches the order here.

class pytorch_lightning.core.hooks.**ModelHooks**
Bases: `object`

Hooks to be used in LightningModule.

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
        for k, v in self.named_parameters():
            self.logger.experiment.add_histogram(
                tag=k, values=v.grad, global_step=self.trainer.global_step
            )
```

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

Return type `None`

on_fit_start()
Called at the very beginning of fit. If on DDP it is called on every process

Return type None

on_post_move_to_device()
Called in the parameter_validation decorator after to() is called. This is a good place to tie weights between modules after moving them to a device. Can be used when training models with weight sharing properties on TPU.

Addresses the handling of shared weights on TPU: https://github.com/pytorch/xla/blob/master/TROUBLESHOOTING.md#xla-tensor-quirks

Example:

```
def on_post_move_to_device(self):
    self.decoder.weight = self.encoder.weight
```

Return type None

on_predict_model_eval()
Sets the model to eval during the predict loop

Return type None

on_pretrain_routine_end()
Called at the end of the pretrain routine (between fit and train start).

- fit
- pretrain_routine start
- pretrain_routine end
- training_start

Return type None

on_pretrain_routine_start()
Called at the beginning of the pretrain routine (between fit and train start).

- fit
- pretrain_routine start
- pretrain_routine end
- training_start

Return type None

on_test_batch_end(*outputs*, *batch*, *batch_idx*, *dataloader_idx*)
Called in the test loop after the batch.

Parameters

- **outputs** (Any) – The outputs of test_step_end(test_step(x))
- **batch** (Any) – The batched data as it is returned by the test DataLoader.
- **batch_idx** (int) – the index of the batch
- **dataloader_idx** (int) – the index of the dataloader

Return type None

on_test_batch_start(*batch*, *batch_idx*, *dataloader_idx*)
Called in the test loop before anything happens for that batch.

Parameters

- **batch** (Any) – The batched data as it is returned by the test DataLoader.
- **batch_idx** (int) – the index of the batch
- **dataloader_idx** (int) – the index of the dataloader

Return type None

on_test_end()
Called at the end of testing.

Return type None

on_test_epoch_end()
Called in the test loop at the very end of the epoch.

Return type None

on_test_epoch_start()
Called in the test loop at the very beginning of the epoch.

Return type None

on_test_model_eval()
Sets the model to eval during the test loop

Return type None

on_test_model_train()
Sets the model to train during the test loop

Return type None

on_test_start()
Called at the beginning of testing.

Return type None

on_train_batch_end(*outputs*, *batch*, *batch_idx*, *dataloader_idx*)
Called in the training loop after the batch.

Parameters

- **outputs** (Any) – The outputs of training_step_end(training_step(x))
- **batch** (Any) – The batched data as it is returned by the training DataLoader.
- **batch_idx** (int) – the index of the batch
- **dataloader_idx** (int) – the index of the dataloader

Return type None

on_train_batch_start(*batch*, *batch_idx*, *dataloader_idx*)
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.
- **batch_idx** (int) – the index of the batch

- **dataloader_idx** (int) – the index of the dataloader

Return type None

on_train_end()
Called at the end of training before logger experiment is closed.

Return type None

on_train_epoch_end(*outputs*)
Called in the training loop at the very end of the epoch.

Return type None

on_train_epoch_start()
Called in the training loop at the very beginning of the epoch.

Return type None

on_train_start()
Called at the beginning of training after sanity check.

Return type None

on_validation_batch_end(*outputs*, *batch*, *batch_idx*, *dataloader_idx*)
Called in the validation loop after the batch.

Parameters

- **outputs** (Any) – The outputs of validation_step_end(validation_step(x))
- **batch** (Any) – The batched data as it is returned by the validation DataLoader.
- **batch_idx** (int) – the index of the batch
- **dataloader_idx** (int) – the index of the dataloader

Return type None

on_validation_batch_start(*batch*, *batch_idx*, *dataloader_idx*)
Called in the validation loop before anything happens for that batch.

Parameters

- **batch** (Any) – The batched data as it is returned by the validation DataLoader.
- **batch_idx** (int) – the index of the batch
- **dataloader_idx** (int) – the index of the dataloader

Return type None

on_validation_end()
Called at the end of validation.

Return type None

on_validation_epoch_end()
Called in the validation loop at the very end of the epoch.

Return type None

on_validation_epoch_start()
Called in the validation loop at the very beginning of the epoch.

Return type None

on_validation_model_eval()
Sets the model to eval during the val loop

Return type None

on_validation_model_train()
Sets the model to train during the val loop

Return type None

on_validation_start()
Called at the beginning of validation.

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(stage):
        data = Load_data(...)
        self.l1 = nn.Linear(28, data.num_classes)
```

Return type None

teardown(*stage*)
Called at the end of fit and test.

Parameters **stage**¶ (str) – either 'fit' or 'test'

Return type None

16.1.4 lightning

Classes

LightningModule

nn.Module with additional great features.

class pytorch_lightning.core.lightning.**LightningModule**(**args: Any, **kwargs: Any*)

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*, *pytorch_lightning.core.hooks.DataHooks*, *pytorch_lightning.core.hooks.CheckpointHooks*, torch.nn.

all_gather(*data*, *group=None*, *sync_grads=False*)

Allows users to call `self.all_gather()` from the LightningModule, thus making the `all_gather` operation accelerator agnostic.

`all_gather` is a function provided by accelerators to gather a tensor from several distributed processes

Parameters

- **tensor** – int, float, tensor of shape (batch, ...), or a (possibly nested) collection thereof.
- **group** (Optional[Any]) – the process group to gather results from. Defaults to all processes (world)
- **sync_grads** (bool) – flag that allows users to synchronize gradients for all_gather op

Returns A tensor of shape (world_size, batch, ...), or if the input was a collection the output will also be a collection with tensors of this shape.

backward(*loss*, *optimizer*, *optimizer_idx*, **args*, ***kwargs*)

Override backward with your own implementation if you need to.

Parameters

- **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, loss, optimizer, optimizer_idx):
    loss.backward()
```

Return type None

configure_callbacks()

Configure model-specific callbacks. When the model gets attached, e.g., when `.fit()` or `.test()` gets called, the list returned here will be merged with the list of callbacks passed to the Trainer's `callbacks` argument. If a callback returned here has the same type as one or several callbacks already present in the Trainer's callbacks list, it will take priority and replace them. In addition, Lightning will make sure *ModelCheckpoint* callbacks run last.

Returns A list of callbacks which will extend the list of callbacks in the Trainer.

Example:

```
def configure_callbacks(self):
    early_stop = EarlyStopping(monitor"val_acc", mode="max")
    checkpoint = ModelCheckpoint(monitor="val_loss")
    return [early_stop, checkpoint]
```

Note: Certain callback methods like *on_init_start()* will never be invoked on the new callbacks returned here.

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.

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 whose 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 the scheduler and its associated configuration. The default configuration is shown below.

```
{
    'scheduler': lr_scheduler, # The LR scheduler instance (required)
    '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 for ReduceLROnPlateau to monitor
    'strict': True, # Whether to crash the training if `monitor` is not found
    'name': None, # Custom name for LearningRateMonitor to use
}
```

Only the `scheduler` key is required, the rest will be set to the defaults 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)
```

(continues on next page)

(continued from previous page)

```
    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'
```

(continues on next page)

(continued from previous page)

```
    'monitor': 'val_f1',
    'frequency': x,
}
```

---

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()

Implement this to override the default items displayed in the progress bar. By default it includes the average loss value, split index of BPTT (if used) and the version of the experiment when using a logger.

```
Epoch 1:   4%|          | 40/1095 [00:03<01:37, 10.84it/s, loss=4.501, v_
↪num=10]
```

Here is an example how to override the defaults:

```
def get_progress_bar_dict(self):
    # don't show the version number
    items = super().get_progress_bar_dict()
    items.pop("v_num", None)
    return items
```

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

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

log(*name*, *value*, *prog_bar=False*, *logger=True*, *on_step=None*, *on_epoch=None*, *reduce_fx=torch.mean*, *tbptt_reduce_fx=torch.mean*, *tbptt_pad_token=0*, *enable_graph=False*, *sync_dist=False*, *sync_dist_op='mean'*, *sync_dist_group=None*)

Log a key, value

Example:

```
self.log('train_loss', loss)
```

The default behavior per hook is as follows

Table 7: * also applies to the test loop

LightningMoule Hook	on_step	on_epoch	prog_bar	logger
training_step	T	F	F	T
training_step_end	T	F	F	T
training_epoch_end	F	T	F	T
validation_step*	F	T	F	T
validation_step_end*	F	T	F	T
validation_epoch_end*	F	T	F	T

Parameters

- **name**¶ (str) – key name
- **value**¶ (Any) – value name
- **prog_bar**¶ (bool) – if True logs to the progress bar
- **logger**¶ (bool) – if True logs to the logger
- **on_step**¶ (Optional[bool]) – if True logs at this step. None auto-logs at the training_step but not validation/test_step

- **on_epoch** (Optional[bool]) – if True logs epoch accumulated metrics. None auto-logs at the val/test step but not training_step
- **reduce_fx** (Callable) – reduction function over step values for end of epoch. Torch.mean by default
- **tbptt_reduce_fx** (Callable) – function to reduce on truncated back prop
- **tbptt_pad_token** (int) – token to use for padding
- **enable_graph** (bool) – if True, will not auto detach the graph
- **sync_dist** (bool) – if True, reduces the metric across GPUs/TPUs
- **sync_dist_op** (Union[Any, str]) – the op to sync across GPUs/TPUs
- **sync_dist_group** (Optional[Any]) – the ddp group

log_dict(*dictionary*, *prog_bar=False*, *logger=True*, *on_step=None*, *on_epoch=None*, *reduce_fx=torch.mean*, *tbptt_reduce_fx=torch.mean*, *tbptt_pad_token=0*, *enable_graph=False*, *sync_dist=False*, *sync_dist_op='mean'*, *sync_dist_group=None*)

Log a dictonary of values at once

Example:

```
values = {'loss': loss, 'acc': acc, ..., 'metric_n': metric_n}
self.log_dict(values)
```

Parameters

- **dictionary** (dict) – key value pairs (str, tensors)
- **prog_bar** (bool) – if True logs to the progress base
- **logger** (bool) – if True logs to the logger
- **on_step** (Optional[bool]) – if True logs at this step. None auto-logs for training_step but not validation/test_step
- **on_epoch** (Optional[bool]) – if True logs epoch accumulated metrics. None auto-logs for val/test step but not training_step
- **reduce_fx** (Callable) – reduction function over step values for end of epoch. Torch.mean by default
- **tbptt_reduce_fx** (Callable) – function to reduce on truncated back prop
- **tbptt_pad_token** (int) – token to use for padding
- **enable_graph** (bool) – if True, will not auto detach the graph
- **sync_dist** (bool) – if True, reduces the metric across GPUs/TPUs
- **sync_dist_op** (Union[Any, str]) – the op to sync across GPUs/TPUs
- **sync_dist_group** (Optional[Any]) – the ddp group:

manual_backward(*loss*, *optimizer=None*, **args*, ***kwargs*)

Call this directly from your training_step when doing optimizations manually. By using this we can ensure that all the proper scaling when using 16-bit etc has been done for you

This function forwards all args to the .backward() call as well.

Tip: In manual mode we still automatically clip grads if Trainer(gradient_clip_val=x) is set

Tip: In manual mode we still automatically accumulate grad over batches if Trainer(accumulate_grad_batches=x) is set and you use *optimizer.step()*

Example:

```
def training_step(...):
    (opt_a, opt_b) = self.optimizers()
    loss = ...
    # automatically applies scaling, etc...
    self.manual_backward(loss, opt_a)
    opt_a.step()
```

Return type None

optimizer_step(*epoch=None*, *batch_idx=None*, *optimizer=None*, *optimizer_idx=None*, *optimizer_closure=None*, *on_tpu=None*, *using_native_amp=None*, *using_lbfgs=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.

Tip: With `Trainer(enable_pl_optimizer=True)`, you can use `optimizer.step()` directly and it will handle zero_grad, accumulated gradients, AMP, TPU and more automatically for you.

Warning: If you are overriding this method, make sure that you pass the `optimizer_closure` parameter to `optimizer.step()` function as shown in the examples. This ensures that `train_step_and_backward_closure` is called within `run_training_batch()`.

Parameters

- **epoch** (Optional[int]) – Current epoch
- **batch_idx** (Optional[int]) – Index of current batch
- **optimizer** (Optional[Optimizer]) – A PyTorch optimizer
- **optimizer_idx** (Optional[int]) – If you used multiple optimizers this indexes into that list.
- **optimizer_closure** (Optional[Callable]) – closure for all optimizers
- **on_tpu** (Optional[bool]) – true if TPU backward is required
- **using_native_amp** (Optional[bool]) – True if using native amp
- **using_lbfgs** (Optional[bool]) – True if the matching optimizer is lbfgs

Examples:

```
# DEFAULT
def optimizer_step(self, epoch, batch_idx, optimizer, optimizer_idx,
                   optimizer_closure, on_tpu, using_native_amp, using_lbfgs):
    optimizer.step(closure=optimizer_closure)

# Alternating schedule for optimizer steps (i.e.: GANs)
def optimizer_step(self, epoch, batch_idx, optimizer, optimizer_idx,
                   optimizer_closure, on_tpu, using_native_amp, using_lbfgs):
    # update generator opt every 2 steps
    if optimizer_idx == 0:
        if batch_idx % 2 == 0 :
            optimizer.step(closure=optimizer_closure)
            optimizer.zero_grad()

    # update discriminator opt every 4 steps
    if optimizer_idx == 1:
        if batch_idx % 4 == 0 :
            optimizer.step(closure=optimizer_closure)
            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, epoch, batch_idx, optimizer, optimizer_idx,
                   optimizer_closure, on_tpu, using_native_amp, using_lbfgs):
    # 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(closure=optimizer_closure)
    optimizer.zero_grad()
```

Return type None

predict(*batch*, *batch_idx*, *dataloader_idx=None*)

Use this function with trainer.predict(...). Override if you need to add any processing logic.

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 arguments from class *__init__*

```
>>> class ManuallyArgsModel(LightningModule):
...     def __init__(self, arg1, arg2, arg3):
...         super().__init__()
...         # manually assign 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_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** (List[Any]) – 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

Return type None

Returns None

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

Examples

With a single dataloader:

```
def test_epoch_end(self, outputs):
    # do something with the outputs of all test batches
    all_test_preds = test_step_outputs.predictions

    some_result = calc_all_results(all_test_preds)
    self.log(some_result)
```

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):
    final_value = 0
    for dataloader_outputs in outputs:
        for test_step_out in dataloader_outputs:
            # do something
            final_value += test_step_out

    self.log('final_metric', final_value)
```

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 dataloaders used).

Returns

Any of.

- Any object or value
- None - Testing will skip to the next batch

```
# 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)
    test_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'test_loss': loss, 'test_acc': test_acc})
```

If you pass in multiple test dataloaders, *test_step()* will have an additional argument.

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

Note: If you don't need to test 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.

Returns None or anything

```
# 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)
    self.log('test_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.encoder(x)
    return out

def test_step_end(self, output_results):
    # this out is now the full size of the batch
    all_test_step_outs = output_results.out
    loss = nce_loss(all_test_step_outs)
    self.log('test_loss', loss)
```

See also:

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

to_onnx (*file_path*, *input_sample=None*, ***kwargs*)
Saves the model in ONNX format

Parameters

- **file_path¶** (Union[str, Path]) – The path of the file the onnx model should be saved to.
- **input_sample¶** (Optional[Any]) – An input for tracing. Default: None (Use self.example_input_array)
- ****kwargs¶** – Will be passed to torch.onnx.export function.

Example

```
>>> class SimpleModel(LightningModule):
...     def __init__(self):
...         super().__init__()
...         self.l1 = torch.nn.Linear(in_features=64, out_features=4)
...
...     def forward(self, x):
...         return torch.relu(self.l1(x.view(x.size(0), -1)))
```

```
>>> with tempfile.NamedTemporaryFile(suffix='.onnx', delete=False) as tmpfile:
...     model = SimpleModel()
...     input_sample = torch.randn((1, 64))
...     model.to_onnx(tmpfile.name, input_sample, export_params=True)
...     os.path.isfile(tmpfile.name)
True
```

to_torchscript (*file_path=None*, *method='script'*, *example_inputs=None*, ***kwargs*)

By default compiles the whole model to a `ScriptModule`. If you want to use tracing, please provided the argument *method='trace'* and make sure that either the example_inputs argument is provided, or the model has self.example_input_array set. If you would like to customize the modules that are scripted you should override this method. In case you want to return multiple modules, we recommend using a dictionary.

Parameters

- **file_path** (`Union[str, Path, None]`) – Path where to save the torchscript. Default: None (no file saved).
- **method** (`Optional[str]`) – Whether to use TorchScript's script or trace method. Default: 'script'
- **example_inputs** (`Optional[Any]`) – An input to be used to do tracing when method is set to 'trace'. Default: None (Use self.example_input_array)
- ****kwargs** – Additional arguments that will be passed to the `torch.jit.script()` or `torch.jit.trace()` function.

Note:

- Requires the implementation of the *`forward()`* method.
- The exported script will be set to evaluation mode.
- It is recommended that you install the latest supported version of PyTorch to use this feature without limitations. See also the `torch.jit` documentation for supported features.

Example

```
>>> class SimpleModel(LightningModule):
...     def __init__(self):
...         super().__init__()
...         self.l1 = torch.nn.Linear(in_features=64, out_features=4)
...
...     def forward(self, x):
...         return torch.relu(self.l1(x.view(x.size(0), -1)))
...
>>> model = SimpleModel()
>>> torch.jit.save(model.to_torchscript(), "model.pt")
>>> os.path.isfile("model.pt")
>>> torch.jit.save(model.to_torchscript(file_path="model_trace.pt", method=
↪'trace',
...                                     example_inputs=torch.randn(1, 64)))
>>> os.path.isfile("model_trace.pt")
True
```

Return type `Union[ScriptModule, Dict[str, ScriptModule]]`

Returns This LightningModule as a torchscript, regardless of whether file_path is defined or not.

toggle_optimizer (*optimizer*, *optimizer_idx*)

Makes sure only the gradients of the current optimizer's parameters are calculated in the training step to prevent dangling gradients in multiple-optimizer setup.

Note: Only called when using multiple optimizers

Override for your own behavior

It works with `untoggle_optimizer` to make sure param_requires_grad_state is properly reset.

Parameters

- **optimizer** (`Optimizer`) – Current optimizer used in training_loop
- **optimizer_idx** (`int`) – Current optimizer idx in training_loop

training_epoch_end(*outputs*)

Called at the end of the training epoch with the outputs of all training steps. Use this in case you need to do something with all the outputs for every training_step.

```
# 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** (`List[Any]`) – 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 `None`

Returns None

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

Example:

```
def training_epoch_end(self, training_step_outputs):
    # do something with all training_step outputs
    return result
```

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, training_step_outputs):
    for out in training_step_outputs:
        # do something here
```

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.

Returns

Any of.

- Tensor - The loss tensor
- dict - A dictionary. Can include any keys, but must include the key 'loss'
- None - Training will skip to the next batch

Note: Returning None is currently not supported for multi-GPU or TPU, or with 16-bit precision enabled.

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.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

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': loss, 'hiddens': hiddens}
```

Note: 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.

Returns Anything

When using dp/ddp2 distributed backends, only a portion of the batch is inside the training_step:

```
def training_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self(x)

    # softmax uses only a portion of the batch in the denomintaor
    loss = self.softmax(out)
    loss = nce_loss(loss)
    return loss
```

If you wish to do something with all the parts of the batch, then use this method to do it:

```
def training_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self.encoder(x)
    return {'pred': out}

def training_step_end(self, training_step_outputs):
    gpu_0_pred = training_step_outputs[0]['pred']
    gpu_1_pred = training_step_outputs[1]['pred']
    gpu_n_pred = training_step_outputs[n]['pred']

    # this softmax now uses the full batch
    loss = nce_loss([gpu_0_pred, gpu_1_pred, gpu_n_pred])
    return 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

untoggle_optimizer(*optimizer_idx*)

Note: Only called when using multiple optimizers

Override for your own behavior

Parameters **optimizer_idx**¶ (int) – Current optimizer idx in training_loop

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**¶ (List[Any]) – 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 None

Returns None

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

Examples

With a single dataloader:

```
def validation_epoch_end(self, val_step_outputs):
    for out in val_step_outputs:
        # do something
```

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):
    for dataloader_output_result in outputs:
        dataloader_outs = dataloader_output_result.dataloader_i_outputs

    self.log('final_metric', final_value)
```

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(val_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 dataloaders used)

Returns

Any of.

- Any object or value
- None - Validation will skip to the next batch

```
# 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)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, *validation_step()* will have an additional argument.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx):
    # dataloader_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.

Returns None or anything

```
# 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.encoder(x)
    loss = self.softmax(out)
    loss = nce_loss(loss)
    self.log('val_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

def validation_step_end(self, val_step_outputs):
    for out in val_step_outputs:
        # do something with these
```

See also:

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

write_prediction(*name*, *value*, *filename='predictions.pt'*)
Write predictions to disk using `torch.save`

Example:

```
self.write_prediction('pred', torch.tensor(...), filename='my_predictions.pt')
```

Parameters

- **name** (`str`) – a string indicating the name to save the predictions under
- **value** (`Union[Tensor, List[Tensor]]`) – the predictions, either a single `Tensor` or a list of them
- **filename** (`str`) – name of the file to save the predictions to

---

Note: when running in distributed mode, calling `write_prediction` will create a file for each device with respective names: `filename_rank_0.pt`, `filename_rank_1.pt`, ...

---

write_prediction_dict(*predictions_dict*, *filename='predictions.pt'*)
Write a dictonary of predictions to disk at once using `torch.save`

Example:

```
pred_dict = {'pred1': torch.tensor(...), 'pred2': torch.tensor(...)}
self.write_prediction_dict(pred_dict)
```

Parameters **predictions_dict** (`Dict[str, Any]`) – dict containing predictions, where each prediction should either be single `Tensor` or a list of them

---

Note: when running in distributed mode, calling `write_prediction_dict` will create a file for each device with respective names: `filename_rank_0.pt`, `filename_rank_1.pt`, ...

---

property automatic_optimization
If False you are responsible for calling .backward, .step, zero_grad.

Return type `bool`

property current_epoch
The current epoch

Return type `int`

property global_rank
The index of the current process across all nodes and devices.

Return type `int`

property global_step
Total training batches seen across all epochs

Return type `int`

property local_rank
The index of the current process within a single node.

Return type `int`

property logger
Reference to the logger object in the Trainer.

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.

precision
The precision used

trainer
Pointer to the trainer object

use_amp
True if using amp

16.2 Callbacks API

base	Abstract base class used to build new callbacks.
early_stopping	Early Stopping
gpu_stats_monitor	GPU Stats Monitor
gradient_accumulation_scheduler	Gradient Accumulator
lr_monitor	Learning Rate Monitor
model_checkpoint	Model Checkpointing
progress	Progress Bars

16.2.1 base

Classes

Callback	Abstract base class used to build new callbacks.

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.

Subclass this class and override any of the relevant hooks

on_after_backward(*trainer*, *pl_module*)
Called after `loss.backward()` and before optimizers do anything.

Return type None

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

Return type None

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

Return type None

on_before_accelerator_backend_setup(*trainer*, *pl_module*)
Called before accelerator is being setup

Return type None

on_before_zero_grad(*trainer*, *pl_module*, *optimizer*)
Called after `optimizer.step()` and before `optimizer.zero_grad()`.

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

on_load_checkpoint(*checkpointed_state*)
Called when loading a model checkpoint, use to reload state.

Return type None

on_pretrain_routine_end(*trainer*, *pl_module*)
Called when the pretrain routine ends.

Return type None

on_pretrain_routine_start(*trainer*, *pl_module*)
Called when the pretrain routine begins.

Return type None

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

Return type None

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

Return type None

on_save_checkpoint(*trainer*, *pl_module*)
Called when saving a model checkpoint, use to persist state.

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

on_train_batch_end(*trainer*, *pl_module*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)
Called when the train batch ends.

Return type None

on_train_batch_start(*trainer*, *pl_module*, *batch*, *batch_idx*, *dataloader_idx*)
Called when the train batch begins.

Return type None

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

Return type None

on_train_epoch_end(*trainer*, *pl_module*, *outputs*)
Called when the train epoch ends.

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

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

Return type None

16.2.2 early_stopping

Classes

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

Early Stopping

Monitor a metric and stop training when it stops improving.

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

Bases: *pytorch_lightning.callbacks.base.Callback*

Monitor a metric and stop training when it stops improving.

Parameters

- **monitor** (str) – quantity to be monitored. Default: 'early_stop_on'.
- **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: 0.0.
- **patience** (int) – number of validation epochs with no improvement after which training will be stopped. Default: 3.
- **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.

  Warning: Setting mode='auto' has been deprecated in v1.1 and will be removed in v1.3.

- **strict** (bool) – whether to crash the training if *monitor* is not found in the validation metrics. Default: True.

Raises

- **MisconfigurationException** – If mode is none of "min", "max", and "auto".
- **RuntimeError** – If the metric monitor is not available.

Example:

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

on_load_checkpoint(*checkpointed_state*)
Called when loading a model checkpoint, use to reload state.

on_save_checkpoint(*trainer*, *pl_module*)
Called when saving a model checkpoint, use to persist state.

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

16.2.3 gpu_stats_monitor

Classes

GPUStatsMonitor	Automatically monitors and logs GPU stats during training stage.

GPU Stats Monitor

Monitor and logs GPU stats during training.

class pytorch_lightning.callbacks.gpu_stats_monitor.**GPUStatsMonitor**(*memory_utilization=True*, *gpu_utilization=True*, *intra_step_time=False*, *inter_step_time=False*, *fan_speed=False*, *temperature=False*)

Bases: *pytorch_lightning.callbacks.base.Callback*

Automatically monitors and logs GPU stats during training stage. `GPUStatsMonitor` is a callback and in order to use it you need to assign a logger in the `Trainer`.

Parameters

- **memory_utilization** (`bool`) – Set to `True` to monitor used, free and percentage of memory utilization at the start and end of each step. Default: `True`.
- **gpu_utilization** (`bool`) – Set to `True` to monitor percentage of GPU utilization at the start and end of each step. Default: `True`.
- **intra_step_time** (`bool`) – Set to `True` to monitor the time of each step. Default: `False`.
- **inter_step_time** (`bool`) – Set to `True` to monitor the time between the end of one step and the start of the next step. Default: `False`.
- **fan_speed** (`bool`) – Set to `True` to monitor percentage of fan speed. Default: `False`.
- **temperature** (`bool`) – Set to `True` to monitor the memory and gpu temperature in degree Celsius. Default: `False`.

Raises **MisconfigurationException** – If NVIDIA driver is not installed, not running on GPUs, or `Trainer` has no logger.

Example:

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.callbacks import GPUStatsMonitor
>>> gpu_stats = GPUStatsMonitor()
>>> trainer = Trainer(callbacks=[gpu_stats])
```

GPU stats are mainly based on *nvidia-smi –query-gpu* command. The description of the queries is as follows:

- **fan.speed** – The fan speed value is the percent of maximum speed that the device's fan is currently intended to run at. It ranges from 0 to 100 %. Note: The reported speed is the intended fan speed. If the fan is physically blocked and unable to spin, this output will not match the actual fan speed. Many parts do not report fan speeds because they rely on cooling via fans in the surrounding enclosure.
- **memory.used** – Total memory allocated by active contexts.
- **memory.free** – Total free memory.
- **utilization.gpu** – Percent of time over the past sample period during which one or more kernels was executing on the GPU. The sample period may be between 1 second and 1/6 second depending on the product.

- **utilization.memory** – Percent of time over the past sample period during which global (device) memory was being read or written. The sample period may be between 1 second and 1/6 second depending on the product.
- **temperature.gpu** – Core GPU temperature, in degrees C.
- **temperature.memory** – HBM memory temperature, in degrees C.

on_train_batch_end(*trainer*, **args*, ***kwargs*)
Called when the train batch ends.

on_train_batch_start(*trainer*, **args*, ***kwargs*)
Called when the train batch begins.

on_train_epoch_start(**args*, ***kwargs*)
Called when the train epoch begins.

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

16.2.4 gradient_accumulation_scheduler

Classes

GradientAccumulationScheduler	Change gradient accumulation factor according to scheduling.

Gradient Accumulator

Change gradient accumulation factor according to scheduling. Trainer also calls `optimizer.step()` for the last indivisible step number.

class pytorch_lightning.callbacks.gradient_accumulation_scheduler.**GradientAccumulationSched**
Bases: *pytorch_lightning.callbacks.base.Callback*

Change gradient accumulation factor according to scheduling.

Parameters **scheduling**¶ (Dict[int, int]) – scheduling in format {epoch: accumulation_factor}

Raises

- **TypeError** – If `scheduling` is an empty `dict`, or not all keys and values of `scheduling` are integers.
- **IndexError** – If `minimal_epoch` is less than 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.

16.2.5 lr_monitor

Classes

LearningRateMonitor	Automatically monitor and logs learning rate for learning rate schedulers during training.

Learning Rate Monitor

Monitor and logs learning rate for lr schedulers during training.

class pytorch_lightning.callbacks.lr_monitor.**LearningRateMonitor**(*logging_interval=None*, *log_momentum=False*)

Bases: *pytorch_lightning.callbacks.base.Callback*

Automatically monitor and logs learning rate for learning rate schedulers during training.

Parameters

- **logging_interval** (Optional[str]) – set to 'epoch' or 'step' to log lr of all optimizers at the same interval, set to None to log at individual interval according to the interval key of each scheduler. Defaults to None.
- **log_momentum** (bool) – option to also log the momentum values of the optimizer, if the optimizer has the momentum or betas attribute. Defaults to False.

Raises **MisconfigurationException** – If logging_interval is none of "step", "epoch", or None.

Example:

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.callbacks import LearningRateMonitor
>>> lr_monitor = LearningRateMonitor(logging_interval='step')
>>> trainer = Trainer(callbacks=[lr_monitor])
```

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_scheduler.LambdaLR(optimizer, ...)
        'name': 'my_logging_name'
    }
    return [optimizer], [lr_scheduler]
```

on_train_batch_start (*trainer*, **args*, ***kwargs*)
Called when the train batch begins.

on_train_epoch_start(*trainer*, **args*, ***kwargs*)
Called when the train epoch begins.

on_train_start(*trainer*, **args*, ***kwargs*)
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

Raises **MisconfigurationException** – If `Trainer` has no `logger`.

16.2.6 model_checkpoint

Classes

ModelCheckpoint	Save the model after every epoch by monitoring a quantity.

Model Checkpointing

Automatically save model checkpoints during training.

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

Bases: *pytorch_lightning.callbacks.base.Callback*

Save the model after every epoch by monitoring a quantity.

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

- **dirpath**¶ (`Union[str, Path, None]`) – directory to save the model file.

  Example:

  ```
  # custom path
  # saves a file like: my/path/epoch=0-step=10.ckpt
  >>> checkpoint_callback = ModelCheckpoint(dirpath='my/path/')
  ```

  By default, dirpath is `None` and will be set at runtime to the location specified by *Trainer*'s *default_root_dir* or *weights_save_path* arguments, and if the Trainer uses a logger, the path will also contain logger name and version.

- **filename**¶ (`Optional[str]`) – checkpoint filename. Can contain named formatting options to be auto-filled.

Example:

```
# save any arbitrary metrics like `val_loss`, etc. in name
# saves a file like: my/path/epoch=2-val_loss=0.02-other_metric=0.
↪03.ckpt
>>> checkpoint_callback = ModelCheckpoint(
...     dirpath='my/path',
...     filename='{epoch}-{val_loss:.2f}-{other_metric:.2f}'
... )
```

By default, filename is `None` and will be set to `'{epoch}-{step}'`.

- **monitor** (`Optional[str]`) – quantity to monitor. By default it is `None` which saves a checkpoint only for the last epoch.
- **verbose** (`bool`) – verbosity mode. Default: `False`.
- **save_last** (`Optional[bool]`) – When `True`, always saves the model at the end of the epoch to a file *last.ckpt*. Default: `None`.
- **save_top_k** (`Optional[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 `v1`.
- **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.

  Warning: Setting `mode='auto'` has been deprecated in v1.1 and will be removed in v1.3.

- **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.
- **prefix** (`str`) – A string to put at the beginning of checkpoint filename.

  Warning: This argument has been deprecated in v1.1 and will be removed in v1.3

Note: For extra customization, ModelCheckpoint includes the following attributes:

- `CHECKPOINT_JOIN_CHAR = "-"`
- `CHECKPOINT_NAME_LAST = "last"`
- `FILE_EXTENSION = ".ckpt"`
- `STARTING_VERSION = 1`

For example, you can change the default last checkpoint name by doing `checkpoint_callback.CHECKPOINT_NAME_LAST = "{epoch}-last"`

Raises

- **MisconfigurationException** – If `save_top_k` is neither `None` nor more than or equal to `-1`, if `monitor` is `None` and `save_top_k` is none of `None`, `-1`, and `0`, or if `mode` is none of `"min"`, `"max"`, and `"auto"`.
- **ValueError** – If `trainer.save_checkpoint` is `None`.

Example:

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

# saves checkpoints to 'my/path/' at every epoch
>>> checkpoint_callback = ModelCheckpoint(dirpath='my/path/')
>>> trainer = Trainer(callbacks=[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(
...     monitor='val_loss',
...     dirpath='my/path/',
...     filename='sample-mnist-{epoch:02d}-{val_loss:.2f}'
... )

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

file_exists(*filepath*, *trainer*)

Checks if a file exists on rank 0 and broadcasts the result to all other ranks, preventing the internal state to diverge between ranks.

Return type bool

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

Generate a filename according to the defined template.

Example:

```
>>> tmpdir = os.path.dirname(__file__)
>>> ckpt = ModelCheckpoint(dirpath=tmpdir, filename='{epoch}')
>>> os.path.basename(ckpt.format_checkpoint_name(0, 1, metrics={}))
'epoch=0.ckpt'
>>> ckpt = ModelCheckpoint(dirpath=tmpdir, filename='{epoch:03d}')
>>> os.path.basename(ckpt.format_checkpoint_name(5, 2, metrics={}))
'epoch=005.ckpt'
>>> ckpt = ModelCheckpoint(dirpath=tmpdir, filename='{epoch}-{val_loss:.2f}')
>>> os.path.basename(ckpt.format_checkpoint_name(2, 3, metrics=dict(val_
↪loss=0.123456)))
'epoch=2-val_loss=0.12.ckpt'
>>> ckpt = ModelCheckpoint(dirpath=tmpdir, filename='{missing:d}')
```

(continues on next page)

(continued from previous page)

```
>>> os.path.basename(ckpt.format_checkpoint_name(0, 4, metrics={}))
'missing=0.ckpt'
>>> ckpt = ModelCheckpoint(filename='{step}')
>>> os.path.basename(ckpt.format_checkpoint_name(0, 0, {}))
'step=0.ckpt'
```

Return type `str`

on_load_checkpoint(*checkpointed_state*)
Called when loading a model checkpoint, use to reload state.

on_pretrain_routine_start(*trainer*, *pl_module*)
When pretrain routine starts we build the ckpt dir on the fly

on_save_checkpoint(*trainer*, *pl_module*)
Called when saving a model checkpoint, use to persist state.

Return type `Dict[str, Any]`

on_validation_end(*trainer*, *pl_module*)
checkpoints can be saved at the end of the val loop

save_checkpoint(*trainer*, *pl_module*)
Performs the main logic around saving a checkpoint. This method runs on all ranks, it is the responsibility of *self.save_function* to handle correct behaviour in distributed training, i.e., saving only on rank 0.

to_yaml(*filepath=None*)
Saves the *best_k_models* dict containing the checkpoint paths with the corresponding scores to a YAML file.

16.2.7 progress

Functions

convert_inf	The tqdm doesn't support inf values.
reset	Resets the tqdm bar to 0 progress with a new total, unless it is disabled.

Classes

ProgressBar	This is the default progress bar used by Lightning.
ProgressBarBase	The base class for progress bars in Lightning.
tqdm	Custom tqdm progressbar where we append 0 to floating points/strings to prevent the progress bar from flickering

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_predict_tqdm()
Override this to customize the tqdm bar for predicting.

Return type *tqdm*

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_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*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)
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_batch_end(*trainer*, *pl_module*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)
Called when the train batch ends.

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*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)
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_train_batch_end(self, trainer, pl_module, outputs):
        super().on_train_batch_end(trainer, pl_module, outputs)  # 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_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*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)
Called when the test batch ends.

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

on_train_batch_end(*trainer*, *pl_module*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)
Called when the train batch ends.

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

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

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

property predict_batch_idx
The current batch index being processed during predicting. Use this to update your progress bar.

Return type int

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

Return type int

property total_predict_batches
The total number of predicting 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 predict dataloader is of infinite size.

Return type int

property total_test_batches
The total number of testing 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 validation 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

class pytorch_lightning.callbacks.progress.**tqdm**(**args*, ***kwargs*)
Bases: tqdm.

Custom tqdm progressbar where we append 0 to floating points/strings to prevent the progress bar from flickering

static format_num(*n*)
Add additional padding to the formatted numbers

Return type str

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

Return type Union[int, float, None]

pytorch_lightning.callbacks.progress.**reset**(*bar*, *total=None*)
Resets the tqdm bar to 0 progress with a new total, unless it is disabled.

Return type None

16.3 Loggers API

base	Abstract base class used to build new loggers.
comet	Comet Logger
csv_logs	CSV logger
mlflow	MLflow Logger
neptune	Neptune Logger
tensorboard	TensorBoard Logger
test_tube	Test Tube Logger
wandb	Weights and Biases Logger

16.3.1 base

Functions

merge_dicts	Merge a sequence with dictionaries into one dictionary by aggregating the same keys with some given function.
rank_zero_experiment	Returns the real experiment on rank 0 and otherwise the DummyExperiment.

Classes

DummyExperiment	Dummy experiment
DummyLogger	Dummy logger for internal use.
LightningLoggerBase	Base class for experiment loggers.
LoggerCollection	The *LoggerCollection* class is used to iterate all logging actions over the given *logger_iterable*.

Abstract base class used to build new loggers.

class pytorch_lightning.loggers.base.**DummyExperiment**
Bases: object

Dummy experiment

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.

---

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

log_graph(*model*, *input_array=None*)
Record model graph

Parameters

- **model**¶ (*LightningModule*) – lightning model
- **input_array**¶ – input passes to *model.forward*

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

property save_dir
Return the root directory where experiment logs get saved, or *None* if the logger does not save data locally.

Return type Optional[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

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

log_graph(*model*, *input_array=None*)
Record model graph

Parameters

- **model** (*LightningModule*) – lightning model
- **input_array** – input passes to *model.forward*

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

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.

property experiment
Return the experiment object associated with this logger.

Return type List[Any]

property name
Return the experiment name.

Return type str

property save_dir
Return the root directory where experiment logs get saved, or *None* if the logger does not save data locally.

Return type Optional[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,
```

(continues on next page)

(continued from previous page)

```
 'd': {'d1': 3, 'd2': 3, 'd3': 3, 'd4': {'d5': 1}},
 'v': 2.3}
```

pytorch_lightning.loggers.base.**rank_zero_experiment**(*fn*)

Returns the real experiment on rank 0 and otherwise the DummyExperiment.

Return type Callable

16.3.2 comet

Classes

CometLogger	Log using Comet.ml.

Comet Logger

class pytorch_lightning.loggers.comet.**CometLogger**(*api_key=None*, *save_dir=None*, *project_name=None*, *rest_api_key=None*, *experiment_name=None*, *experiment_key=None*, *offline=False*, *prefix=''*, ***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

```
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_key=os.environ.get('COMET_EXPERIMENT_KEY'),  # Optional
    experiment_name='default'  # Optional
)
trainer = Trainer(logger=comet_logger)
```

OFFLINE MODE

```
from pytorch_lightning.loggers import CometLogger
# arguments made to CometLogger are passed on to the comet_ml.Experiment class
```

(continues on next page)

(continued from previous page)

```
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. If not given, this will be loaded from the environment variable COMET_API_KEY or ~/.comet.config if either exists.
- **save_dir** (Optional[str]) – Required in offline mode. The path for the directory to save local comet logs. If given, this also sets the directory for saving checkpoints.
- **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.
- **offline** (bool) – If api_key and save_dir are both given, this determines whether the experiment will be in online or offline mode. This is useful if you use save_dir to control the checkpoints directory and have a ~/.comet.config file but still want to run offline experiments.
- **prefix** (str) – A string to put at the beginning of metric keys.
- ****kwargs** – Additional arguments like *workspace*, *log_code*, etc. used by CometExperiment can be passed as keyword arguments in this logger.

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_graph(*model*, *input_array=None*)

Record model graph

Parameters

- **model** (*LightningModule*) – lightning model
- **input_array** – input passes to *model.forward*

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()
```

property name
Return the experiment name.

Return type str

property save_dir
Return the root directory where experiment logs get saved, or *None* if the logger does not save data locally.

Return type Optional[str]

property version
Return the experiment version.

Return type str

16.3.3 csv_logs

Classes

CSVLogger	Log to local file system in yaml and CSV format.
ExperimentWriter	Experiment writer for CSVLogger.

CSV logger

CSV logger for basic experiment logging that does not require opening ports

class pytorch_lightning.loggers.csv_logs.**CSVLogger**(*save_dir*, *name='default'*, *version=None*, *prefix=''*)

Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Log to local file system in yaml and CSV format.

Logs are saved to `os.path.join(save_dir, name, version)`.

Example

```
>>> from pytorch_lightning import Trainer
>>> from pytorch_lightning.loggers import CSVLogger
>>> logger = CSVLogger("logs", name="my_exp_name")
>>> trainer = Trainer(logger=logger)
```

Parameters

- **save_dir** (`str`) – Save directory
- **name** (`Optional[str]`) – Experiment name. Defaults to `'default'`.
- **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.
- **prefix** (`str`) – A string to put at the beginning of metric keys.

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 ExperimentWriter object. To use ExperimentWriter features in your *LightningModule* do the following.

Example:

```
self.logger.experiment.some_experiment_writer_function()
```

Return type *ExperimentWriter*

property log_dir
The log directory for this run. 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 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 save_dir
Return the root directory where experiment logs get saved, or *None* if the logger does not save data locally.

Return type Optional[str]

property version
Return the experiment version.

Return type int

class pytorch_lightning.loggers.csv_logs.**ExperimentWriter**(*log_dir*)
Bases: object

Experiment writer for CSVLogger.

Currently supports to log hyperparameters and metrics in YAML and CSV format, respectively.

Parameters **log_dir**¶ (str) – Directory for the experiment logs

log_hparams(*params*)
Record hparams

Return type None

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

Return type None

save()
Save recorded hparams and metrics into files

Return type None

16.3.4 mlflow

Classes

MLFlowLogger	Log using MLflow.

MLflow Logger

class pytorch_lightning.loggers.mlflow.**MLFlowLogger**(*experiment_name='default'*, *tracking_uri=None*, *tags=None*, *save_dir='./mlruns'*, *prefix=''*)

Bases: *pytorch_lightning.loggers.base.LightningLoggerBase*

Log using MLflow.

Install it with pip:

```
pip install mlflow
```

```
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 your *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 *file:<save_dir>*.
- **tags** (Optional[Dict[str, Any]]) – A dictionary tags for the experiment.
- **save_dir** (Optional[str]) – A path to a local directory where the MLflow runs get saved. Defaults to *./mlflow* if *tracking_uri* is not provided. Has no effect if *tracking_uri* is provided.
- **prefix** (str) – A string to put at the beginning of metric keys.

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 save_dir
The root file directory in which MLflow experiments are saved.

Return type Optional[str]

Returns Local path to the root experiment directory if the tracking uri is local. Otherwhise returns *None*.

property version
Return the experiment version.

Return type str

16.3.5 neptune

Classes

NeptuneLogger	Log using Neptune.

Neptune Logger

class pytorch_lightning.loggers.neptune.**NeptuneLogger**(*api_key=None*, *project_name=None*, *close_after_fit=True*, *offline_mode=False*, *experiment_name=None*, *experiment_id=None*, *prefix=''*, ***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

```
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

```
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',
```

(continues on next page)

(continued from previous page)

```
    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:

```
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))
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.
- **experiment_id** (Optional[str]) – Optional. Default is `None`. The ID of the existing experiment. If specified, connect to experiment with experiment_id in project_name. Input arguments "experiment_name", "params", "properties" and "tags" will be overriden based on fetched experiment data.
- **prefix** (str) – A string to put at the beginning of metric keys.
- ****kwargs** – Additional arguments like *params*, *tags*, *properties*, etc. used by `neptune.Session.create_experiment()` can be passed as keyword arguments in this logger.

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, 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, currently ignored

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 save_dir
Return the root directory where experiment logs get saved, or *None* if the logger does not save data locally.

Return type Optional[str]

property version
Return the experiment version.

Return type str

16.3.6 tensorboard

Classes

TensorBoardLogger	Log to local file system in TensorBoard format.

TensorBoard Logger

class pytorch_lightning.loggers.tensorboard.**TensorBoardLogger**(*save_dir*, *name='default'*, *version=None*, *log_graph=False*, *default_hp_metric=True*, *prefix=''*, ***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.
- **log_graph** (`bool`) – Adds the computational graph to tensorboard. This requires that the user has defined the *self.example_input_array* attribute in their model.
- **default_hp_metric** (`bool`) – Enables a placeholder metric with key *hp_metric* when *log_hyperparams* is called without a metric (otherwise calls to log_hyperparams without a metric are ignored).
- **prefix** (`str`) – A string to put at the beginning of metric keys.
- ****kwargs** – Additional arguments like *comment*, *filename_suffix*, etc. used by `SummaryWriter` can be passed as keyword arguments in this logger.

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_graph(*model*, *input_array=None*)

Record model graph

Parameters

- **model**¶ (*LightningModule*) – lightning model
- **input_array**¶ – input passes to *model.forward*

log_hyperparams(*params*, *metrics=None*)

Record hyperparameters. TensorBoard logs with and without saved hyperparameters are incompatible, the hyperparameters are then not displayed in the TensorBoard. Please delete or move the previously saved logs to display the new ones with hyperparameters.

Parameters

- **params**¶ (Union[Dict[str, Any], Namespace]) – a dictionary-like container with the hyperparameters
- **metrics**¶ (Optional[Dict[str, Any]]) – Dictionary with metric names as keys and measured quantities as values

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 save_dir
Return the root directory where experiment logs get saved, or *None* if the logger does not save data locally.

Return type Optional[str]

property version
Return the experiment version.

Return type int

16.3.7 test_tube

Classes

TestTubeLogger	Log to local file system in TensorBoard format but using a nicer folder structure (see full docs).

Test Tube Logger

class pytorch_lightning.loggers.test_tube.**TestTubeLogger**(*save_dir*, *name='default'*, *description=None*, *debug=False*, *version=None*, *create_git_tag=False*, *log_graph=False*, *prefix=''*)

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
```

```
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.
- **log_graph**¶ (bool) – Adds the computational graph to tensorboard. This requires that the user has defined the *self.example_input_array* attribute in their model.
- **prefix**¶ (str) – A string to put at the beginning of metric keys.

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_graph(*model*, *input_array=None*)
Record model graph

Parameters

- **model**¶ (*LightningModule*) – lightning model
- **input_array**¶ – input passes to *model.forward*

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 save_dir
Return the root directory where experiment logs get saved, or *None* if the logger does not save data locally.

Return type Optional[str]

property version
Return the experiment version.

Return type int

16.3.8 wandb

Classes

WandbLogger	Log using Weights and Biases.

Weights and Biases Logger

class pytorch_lightning.loggers.wandb.**WandbLogger**(*name=None*, *save_dir=None*, *offline=False*, *id=None*, *anonymous=False*, *version=None*, *project=None*, *log_model=False*, *experiment=None*, *prefix=''*, *sync_step=True*, ***kwargs*)

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 (wandb dir by default).
- **offline** (Optional[bool]) – Run offline (data can be streamed later to wandb servers).
- **id** (Optional[str]) – Sets the version, mainly used to resume a previous run.

- **version** (Optional[str]) – Same as id.
- **anonymous** (Optional[bool]) – Enables or explicitly disables anonymous logging.
- **project** (Optional[str]) – The name of the project to which this run will belong.
- **log_model** (Optional[bool]) – Save checkpoints in wandb dir to upload on W&B servers.
- **prefix** (Optional[str]) – A string to put at the beginning of metric keys.
- **sync_step** (Optional[bool]) – Sync Trainer step with wandb step.
- **experiment** – WandB experiment object. Automatically set when creating a run.
- ****kwargs** – Additional arguments like *entity*, *group*, *tags*, etc. used by `wandb.init()` can be passed as keyword arguments in this logger.

Example:

```
from pytorch_lightning.loggers import WandbLogger
from pytorch_lightning import Trainer
wandb_logger = WandbLogger()
trainer = Trainer(logger=wandb_logger)
```

Note: When logging manually through *wandb.log* or *trainer.logger.experiment.log*, make sure to use *commit=False* so the logging step does not increase.

See also:

- Tutorial on how to use W&B with PyTorch Lightning
- W&B Documentation

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

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 Optional[str]

property save_dir
Return the root directory where experiment logs get saved, or *None* if the logger does not save data locally.

Return type Optional[str]

property version
Return the experiment version.

Return type Optional[str]

16.4 Profiler API

profilers	Profiler to check if there are any bottlenecks in your code.

16.4.1 profilers

Classes

AdvancedProfiler	This profiler uses Python's cProfiler to record more detailed information about time spent in each function call recorded during a given action.
BaseProfiler	If you wish to write a custom profiler, you should inhereit from this class.
PassThroughProfiler	This class should be used when you don't want the (small) overhead of profiling.
PyTorchProfiler	This profiler uses PyTorch's Autograd Profiler and lets you inspect the cost of different operators inside your model - both on the CPU and GPU
SimpleProfiler	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.

Profiler to check if there are any bottlenecks in your code.

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.

Parameters **output_streams** (Union[list, tuple, None]) – 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.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.

Args: output_streams: 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.**PyTorchProfiler**(*output_filename=None*, *enabled=True*, *use_cuda=False*, *record_shapes=False*, *profile_memory=False*, *group_by_input_shapes=False*, *with_stack=False*, *use_kineto=False*, *use_cpu=True*, *emit_nvtx=False*, *export_to_chrome=False*, *path_to_export_trace=None*, *row_limit=20*, *sort_by_key=None*, *profiled_functions=None*, *local_rank=None*)
Bases: *pytorch_lightning.profiler.profilers.BaseProfiler*

This profiler uses PyTorch's Autograd Profiler and lets you inspect the cost of different operators inside your model - both on the CPU and GPU

Parameters

- **output_filename**¶ (Optional[str]) – optionally save profile results to file instead of printing to std out when training is finished. When using ddp, each rank will stream the profiled operation to their own file with the extension _{rank}.txt
- **enabled**¶ (bool) – Setting this to False makes this context manager a no-op.
- **use_cuda**¶ (bool) – Enables timing of CUDA events as well using the cudaEvent API. Adds approximately 4us of overhead to each tensor operation.

- **record_shapes** (bool) – If shapes recording is set, information about input dimensions will be collected.
- **profile_memory** (bool) – Whether to report memory usage, default: True (Introduced in PyTorch 1.6.0)
- **group_by_input_shapes** (bool) – Include operator input shapes and group calls by shape.
- **with_stack** (bool) – record source information (file and line number) for the ops (Introduced in PyTorch 1.7.0)
- **use_kineto** (bool) – experimental support for Kineto profiler (Introduced in PyTorch 1.8.0)
- **use_cpu** (bool) – use_kineto=True and can be used to lower the overhead for GPU-only profiling (Introduced in PyTorch 1.8.0)
- **emit_nvtx** (bool) – Context manager that makes every autograd operation emit an NVTX range Run:

```
nvprof --profile-from-start off -o trace_name.prof -- <regular
↪command here>
```

To visualize, you can either use:

```
nvvp trace_name.prof
torch.autograd.profiler.load_nvprof(path)
```

- **export_to_chrome** (bool) – Wether to export the sequence of profiled operators for Chrome. It will generate a `.json` file which can be read by Chrome.
- **path_to_export_trace** (Optional[str]) – Directory path to export `.json` traces when using `export_to_chrome=True`. By default, it will be save where the file being is being run.
- **row_limit** (int) – Limit the number of rows in a table, *0* is a special value that removes the limit completely.
- **sort_by_key** (Optional[str]) – Keys to sort out profiled table
- **profiled_functions** (Optional[List]) – list of profiled functions which will create a context manager on. Any other will be pass through.
- **local_rank** (Optional[int]) – When running in distributed setting, local_rank is used for each process to write to their own file if *output_fname* is provided.

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.**SimpleProfiler**(*output_filename=None*, *extended=True*)

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.

Parameters **output_filename**¶ (Optional[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

16.5 Trainer API

trainer	Trainer to automate the training.

16.5.1 trainer

Classes

Trainer	Customize every aspect of training via flags

Trainer to automate the training.

class pytorch_lightning.trainer.trainer.**Trainer**(*logger=True*, *checkpoint_callback=True*, *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=None*, *overfit_batches=0.0*, *track_grad_norm=-1*, *check_val_every_n_epoch=1*, *fast_dev_run=False*, *accumulate_grad_batches=1*, *max_epochs=None*, *min_epochs=None*, *max_steps=None*, *min_steps=None*, *limit_train_batches=1.0*, *limit_val_batches=1.0*, *limit_test_batches=1.0*, *limit_predict_batches=1.0*, *val_check_interval=1.0*, *flush_logs_every_n_steps=100*, *log_every_n_steps=50*, *accelerator=None*, *sync_batchnorm=False*, *precision=32*, *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*, *plugins=None*, *amp_backend='native'*, *amp_level='O2'*, *distributed_backend=None*, *automatic_optimization=None*, *move_metrics_to_cpu=False*, *enable_pl_optimizer=None*, *multiple_trainloader_mode='max_size_cycle'*, *stochastic_weight_avg=False*)

Bases: pytorch_lightning.trainer.properties.TrainerProperties, pytorch_lightning.trainer.callback_hook.TrainerCallbackHookMixin, pytorch_lightning.trainer.model_hooks.TrainerModelHooksMixin, pytorch_lightning.trainer.optimizers.TrainerOptimizersMixin, pytorch_lightning.trainer.logging.TrainerLoggingMixin, pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixin, pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin, pytorch_lightning.trainer.

deprecated_api.DeprecatedDistDeviceAttributes, pytorch_lightning.trainer.deprecated_api.DeprecatedTrainerAttributes

Customize every aspect of training via flags

Parameters

- **accelerator**¶ (Union[str, Accelerator, None]) – Previously known as distributed_backend (dp, ddp, ddp2, etc...). Can also take in an accelerator object for custom hardware.
- **accumulate_grad_batches**¶ (Union[int, Dict[int, int], List[list]]) – Accumulates grads every k batches or as set up in the dict.
- **amp_backend**¶ (str) – The mixed precision backend to use ("native" or "apex")
- **amp_level**¶ (str) – The optimization level to use (O1, O2, etc...).
- **auto_lr_find**¶ (Union[bool, str]) – If set to True, will make trainer.tune() run a learning rate finder, trying to optimize initial learning for faster convergence. trainer.tune() method will set the suggested 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.
- **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.
- **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.
- **benchmark**¶ (bool) – If true enables cudnn.benchmark.
- **callbacks**¶ (Union[List[*Callback*], *Callback*, None]) – Add a callback or list of callbacks.
- **checkpoint_callback**¶ (bool) – If True, enable checkpointing. It will configure a default ModelCheckpoint callback if there is no user-defined ModelCheckpoint in *callbacks*. Default: True.

  Warning: Passing a ModelCheckpoint instance to this argument is deprecated since v1.1 and will be unsupported from v1.3. Use *callbacks* argument instead.

- **check_val_every_n_epoch**¶ (int) – Check val every n train epochs.
- **default_root_dir**¶ (Optional[str]) – Default path for logs and weights when no logger/ckpt_callback passed. Default: os.getcwd(). Can be remote file paths such as *s3://mybucket/path* or 'hdfs://path/'
- **deterministic**¶ (bool) – If true enables cudnn.deterministic.
- **distributed_backend**¶ (Optional[str]) – deprecated. Please use 'accelerator'
- **fast_dev_run**¶ (Union[int, bool]) – runs n if set to n (int) else 1 if set to True batch(es) of train, val and test to find any bugs (ie: a sort of unit test).
- **flush_logs_every_n_steps**¶ (int) – How often to flush logs to disk (defaults to every 100 steps).

- **gpus**¶ (Union[int, str, List[int], None]) – number of gpus to train on (int) or which GPUs to train on (list or str) applied per node
- **gradient_clip_val**¶ (float) – 0 means don't clip.
- **limit_train_batches**¶ (Union[int, float]) – How much of training dataset to check (floats = percent, int = num_batches)
- **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)
- **logger**¶ (Union[*LightningLoggerBase*, Iterable[*LightningLoggerBase*], bool]) – Logger (or iterable collection of loggers) for experiment tracking.
- **log_gpu_memory**¶ (Optional[str]) – None, 'min_max', 'all'. Might slow performance
- **log_every_n_steps**¶ (int) – How often to log within steps (defaults to every 50 steps).
- **automatic_optimization**¶ (Optional[bool]) – If False you are responsible for calling .backward, .step, zero_grad in LightningModule. This argument has been moved to LightningModule. It is deprecated here in v1.1 and will be removed in v1.3.
- **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
- **process_position**¶ (int) – orders the progress bar when running multiple models on same machine.
- **progress_bar_refresh_rate**¶ (Optional[int]) – How often to refresh progress bar (in steps). Value 0 disables progress bar. Ignored when a custom progress bar is passed to *callbacks*. Default: None, means a suitable value will be chosen based on the environment (terminal, Google COLAB, etc.).
- **profiler**¶ (Union[*BaseProfiler*, bool, str, None]) – To profile individual steps during training and assist in identifying bottlenecks. Passing bool value is deprecated in v1.1 and will be removed in v1.3.
- **overfit_batches**¶ (Union[int, float]) – Overfit a percent of training data (float) or a set number of batches (int). Default: 0.0
- **plugins**¶ (Union[Plugin, str, list, None]) – Plugins allow modification of core behavior like ddp and amp, and enable custom lightning plugins.
- **precision**¶ (int) – Full precision (32), half precision (16). Can be used on CPU, GPU or TPUs.
- **max_epochs**¶ (Optional[int]) – Stop training once this number of epochs is reached. Disabled by default (None). If both max_epochs and max_steps are not specified, defaults to max_epochs = 1000.
- **min_epochs**¶ (Optional[int]) – Force training for at least these many epochs. Disabled by default (None). If both min_epochs and min_steps are not specified, defaults to min_epochs = 1.
- **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).
- **num_nodes**¶ (int) – number of GPU nodes for distributed training.
- **num_processes**¶ (int) – number of processes for distributed training with distributed_backend="ddp_cpu"
- **num_sanity_val_steps**¶ (int) – Sanity check runs n validation batches before starting the training routine. Set it to *-1* to run all batches in all validation dataloaders. Default: 2
- **reload_dataloaders_every_epoch**¶ (bool) – Set to True to reload dataloaders every epoch.
- **replace_sampler_ddp**¶ (bool) – Explicitly enables or disables sampler replacement. If not specified this will toggled automatically when DDP is used. By default it will add `shuffle=True` for train sampler and `shuffle=False` for val/test sampler. If you want to customize it, you can set `replace_sampler_ddp=False` and add your own distributed sampler.
- **resume_from_checkpoint**¶ (Union[str, Path, None]) – Path/URL of the checkpoint from which training is resumed. If there is no checkpoint file at the path, start from scratch. If resuming from mid-epoch checkpoint, training will start from the beginning of the next epoch.
- **sync_batchnorm**¶ (bool) – Synchronize batch norm layers between process groups/whole world.
- **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.
- **tpu_cores**¶ (Union[int, str, List[int], None]) – How many TPU cores to train on (1 or 8) / Single TPU to train on [1]
- **track_grad_norm**¶ (Union[int, float, str]) – -1 no tracking. Otherwise tracks that p-norm. May be set to 'inf' infinity-norm.
- **truncated_bptt_steps**¶ (Optional[int]) – Truncated back prop breaks performs backprop every k steps of much longer sequence.
- **val_check_interval**¶ (Union[int, float]) – How often to check the validation set. Use float to check within a training epoch, use int to check every n steps (batches).
- **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*. Can be remote file paths such as *s3://mybucket/path* or 'hdfs://path/' Defaults to *default_root_dir*.
- **move_metrics_to_cpu**¶ (bool) – Whether to force internal logged metrics to be moved to cpu. This can save some gpu memory, but can make training slower. Use with attention.
- **enable_pl_optimizer**¶ (Optional[bool]) – If True, each optimizer will be wrapped by *pytorch_lightning.core.optimizer.LightningOptimizer*. It allows Lightning to handle AMP, TPU, accumulated_gradients, etc. .. warning:: Currently deprecated and it will be removed in v1.3

- **multiple_trainloader_mode**¶ (str) – How to loop over the datasets when there are multiple train loaders. In 'max_size_cycle' mode, the trainer ends one epoch when the largest dataset is traversed, and smaller datasets reload when running out of their data. In 'min_size' mode, all the datasets reload when reaching the minimum length of datasets.
- **stochastic_weight_avg**¶ (bool) – Whether to use *Stochastic Weight Averaging (SWA) <https://pytorch.org/blog/pytorch-1.6-now-includes-stochastic-weight-averaging/>_*

fit(*model*, *train_dataloader=None*, *val_dataloaders=None*, *datamodule=None*)
Runs the full optimization routine.

Parameters

- **datamodule**¶ (Optional[*LightningDataModule*]) – A instance of LightningDataModule.
- **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

predict(*model=None*, *dataloaders=None*, *datamodule=None*)
Separates from fit to make sure you never run on your predictions set until you want to.

This will call the model forward function to compute predictions.

Parameters

- **model**¶ (Optional[*LightningModule*]) – The model to predict on.
- **dataloaders**¶ (Union[DataLoader, List[DataLoader], None]) – Either a single Pytorch Dataloader or a list of them, specifying inference samples.
- **datamodule**¶ (Optional[*LightningDataModule*]) – A instance of LightningDataModule.

Returns Returns a list of dictionaries, one for each provided dataloader containing their respective predictions.

setup_trainer(*model*)
Sanity check a few things before starting actual training or testing.

Parameters **model**¶ (*LightningModule*) – The model to run sanity test on.

test(*model=None*, *test_dataloaders=None*, *ckpt_path='best'*, *verbose=True*, *datamodule=None*)
Separates from fit to make sure you never run on your test set until you want to.

Parameters

- **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.
- **datamodule**¶ (Optional[*LightningDataModule*]) – A instance of LightningDataModule.
- **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.

- **verbose** (bool) – If True, prints the test results

Returns Returns a list of dictionaries, one for each test dataloader containing their respective metrics.

tune(*model*, *train_dataloader=None*, *val_dataloaders=None*, *datamodule=None*)
Runs routines to tune hyperparameters before training.

Parameters

- **datamodule** (Optional[*LightningDataModule*]) – A instance of LightningDataModule.
- **model** (*LightningModule*) – Model to tune.
- **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

16.6 Tuner API

batch_size_scaling
lr_finder

16.6.1 batch_size_scaling

Functions

scale_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.

pytorch_lightning.tuner.batch_size_scaling.**scale_batch_size**(*trainer*, *model*, *mode='power'*, *steps_per_trial=3*, *init_val=2*, *max_trials=25*, *batch_arg_name='batch_size'*, ***fit_kwargs*)
Will iteratively try to find the largest batch size for a given model that does not give an out of memory (OOM) error.

Parameters

- **trainer** – The Trainer
- **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
- **batch_arg_name**¶ (str) – name of the attribute that stores the batch size. It is expected that the user has provided a model or datamodule that has a hyperparameter with that name. We will look for this attribute name in the following places
  - `model`
  - `model.hparams`
  - `model.datamodule`
  - `trainer.datamodule` (the datamodule passed to the tune method)
- ****fit_kwargs**¶ – remaining arguments to be passed to .fit(), e.g., dataloader or datamodule.

16.6.2 lr_finder

Functions

lr_find	`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.

pytorch_lightning.tuner.lr_finder.**lr_find**(*trainer*, *model*, *train_dataloader=None*, *val_dataloaders=None*, *min_lr=1e-08*, *max_lr=1*, *num_training=100*, *mode='exponential'*, *early_stop_threshold=4.0*, *datamodule=None*, *update_attr=False*)

`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 to update learning rate after each batch:
  - `'exponential'` (default): Will increase the learning rate exponentially.

  - `'linear'`: Will increase the learning rate linearly.

- **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.
- **datamodule**¶ (`Optional[LightningDataModule]`) – An optional `LightningDataModule` which holds the training and validation dataloader(s). Note that the `train_dataloader` and `val_dataloaders` parameters cannot be used at the same time as this parameter, or a `MisconfigurationException` will be raised.
- **update_attr**¶ (`bool`) – Whether to update the learning rate attribute or not.

Example:

```
# Setup model and trainer
model = MyModelClass(hparams)
trainer = pl.Trainer()

# Run lr finder
lr_finder = trainer.tuner.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)
```

16.7 Utilities API

argparse_utils	
seed	Helper functions to help with reproducibility of models.

16.7.1 argparse_utils

16.7.2 seed

Functions

seed_everything	Function that sets seed for pseudo-random number generators in: pytorch, numpy, python.random In addition, sets the env variable *PL_GLOBAL_SEED* which will be passed to spawned subprocesses (e.g.

Helper functions to help with reproducibility of models.

pytorch_lightning.utilities.seed.**seed_everything**(*seed=None*)

Function that sets seed for pseudo-random number generators in: pytorch, numpy, python.random In addition, sets the env variable *PL_GLOBAL_SEED* which will be passed to spawned subprocesses (e.g. ddp_spawn backend).

Parameters **seed**¶ (Optional[int]) – the integer value seed for global random state in Lightning. If *None*, will read seed from *PL_GLOBAL_SEED* env variable or select it randomly.

Return type int

CHAPTER

SEVENTEEN

BOLTS

PyTorch Lightning Bolts, is our official collection of prebuilt models across many research domains.

```
pip install pytorch-lightning-bolts
```

In bolts we have:

- A collection of pretrained state-of-the-art models.
- A collection of models designed to bootstrap your research.
- A collection of callbacks, transforms, full datasets.
- All models work on CPUs, TPUs, GPUs and 16-bit precision.

17.1 Quality control

The Lightning community builds bolts and contributes them to Bolts. The lightning team guarantees that contributions are:

- Rigorously Tested (CPUs, GPUs, TPUs).
- Rigorously Documented.
- Standardized via PyTorch Lightning.
- Optimized for speed.
- Checked for correctness.

17.2 Example 1: Pretrained, prebuilt models

```
from pl_bolts.models import VAE, GPT2, ImageGPT, PixelCNN
from pl_bolts.models.self_supervised import AMDIM, CPCV2, SimCLR, MocoV2
from pl_bolts.models import LinearRegression, LogisticRegression
from pl_bolts.models.gans import GAN
from pl_bolts.callbacks import PrintTableMetricsCallback
from pl_bolts.datamodules import FashionMNISTDataModule, CIFAR10DataModule,␣
↪ImagenetDataModule
```

17.3 Example 2: Extend for faster research

Bolts are contributed with benchmarks and continuous-integration tests. This means you can trust the implementations and use them to bootstrap your research much faster.

```
from pl_bolts.models import ImageGPT
from pl_bolts.self_supervised import SimCLR

class VideoGPT(ImageGPT):

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

        logits = self.gpt(x)
        simclr_features = self.simclr(x)

        # ------------------
        # do something new with GPT logits + simclr_features
        # ------------------

        loss = self.criterion(logits.view(-1, logits.size(-1)), x.view(-1).long())

        logs = {"loss": loss}
        return {"loss": loss, "log": logs}
```

17.4 Example 3: Callbacks

We also have a collection of callbacks.

```
from pl_bolts.callbacks import PrintTableMetricsCallback
import pytorch_lightning as pl

trainer = pl.Trainer(callbacks=[PrintTableMetricsCallback()])

# loss│train_loss│val_loss│epoch
# ──────────────────────────────
# 2.2541470527648926│2.2541470527648926│2.2158432006835938│0
```

CHAPTER

EIGHTEEN

PYTORCH ECOSYSTEM EXAMPLES

- Pytorch Geometric: Deep learning on Graphs and other irregular structures.

CHAPTER

NINETEEN

COMMUNITY EXAMPLES

- Contextual Emotion Detection (DoubleDistilBert).
- Cotatron: Transcription-Guided Speech Encoder.
- FasterRCNN object detection + Hydra.
- Image Inpainting using Partial Convolutions.
- MNIST on TPU.
- NER (transformers, TPU).
- NeuralTexture (CVPR).
- Recurrent Attentive Neural Process.
- Siamese Nets for One-shot Image Recognition.
- Speech Transformers.
- Transformers transfer learning (Huggingface).
- Transformers text classification.
- VAE Library of over 18+ VAE flavors.
- Transformers Question Answering (SQuAD).
- Atlas: End-to-End 3D Scene Reconstruction from Posed Images.
- Self-Supervised Representation Learning (MoCo and BYOL).
- Pytorch-Forecasting: Time series forecasting package.
- Transformers masked language modeling.
- Pytorch Geometric Examples with Pytorch Lightning and Hydra.

CHAPTER

TWENTY

AWS/GCP TRAINING

Lightning has a native solution for training on AWS/GCP at scale (Lightning-Grid). Grid is in private early-access now but you can request access at grid.ai.

We've designed Grid to work for Lightning users without needing to make ANY changes to their code.

To use grid, take your regular command:

```
python my_model.py --learning_rate 1e-6 --layers 2 --gpus 4
```

And change it to use the grid train command:

```
grid train --grid_gpus 4 my_model.py --learning_rate 'uniform(1e-6, 1e-1, 20)' --
↪layers '[2, 4, 8, 16]'
```

The above command will launch (20 * 4) experiments each running on 4 GPUs (320 GPUs!) - by making ZERO changes to your code.

The *uniform* command is part of our new expressive syntax which lets you construct hyperparameter combinations using over 20+ distributions, lists, etc. Of course, you can also configure all of this using yamls which can be dynamically assembled at runtime.

Hint: Grid supports the search strategy of your choice! (and much more than just sweeps)

CHAPTER

TWENTYONE

16-BIT TRAINING

Lightning offers 16-bit training for CPUs, GPUs, and TPUs.

21.1 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

21.1.1 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, gpus=1)
```

21.1.2 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
```

(continues on next page)

(continued from previous page)

```
# depending on how you installed PyTorch

# see available modules
module avail

# 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+

21.1.3 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()`.

21.2 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

TWENTYTWO

COMPUTING CLUSTER (SLURM)

Lightning automates the details behind training on a SLURM-powered cluster.

22.1 Multi-node training

To train a model using multiple nodes, do the following:

1. Design your *lightning module*.
2. Enable DDP in the trainer

```
# train on 32 GPUs across 4 nodes
trainer = Trainer(gpus=8, num_nodes=4, accelerator='ddp')
```

3. It's a good idea to structure your training script like this:

```
# train.py
def main(hparams):
    model = LightningTemplateModel(hparams)

    trainer = Trainer(
        gpus=8,
        num_nodes=4,
        accelerator='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)
```

22.2 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
```

22.3 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 the 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.

22.4 Self-balancing architecture (COMING SOON)

Here Lightning distributes parts of your module across available GPUs to optimize for speed and memory.

CHAPTER

TWENTYTHREE

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. We are extending our Autoencoder from the *LitMNIST*-module which already defines all the dataloading. 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()
        self.metric = MSE()

    def forward(self, x):
        return self.encoder(x)

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

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

        loss = self.metric(x, x_hat)
        return loss

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

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

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

(continues on next page)

(continued from previous page)

```
        loss = self.metric(x, x_hat)
        self.log(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 should 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

TWENTYFOUR

DEBUGGING

The following are flags that make debugging much easier.

24.1 fast_dev_run

This flag runs a "unit test" by running n if set to `n` (int) else 1 if set to `True` training and validation batch(es). 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*)

```
# runs 1 train, val, test batch and program ends
trainer = Trainer(fast_dev_run=True)

# runs 7 train, val, test batches and program ends
trainer = Trainer(fast_dev_run=7)
```

Note: This argument will disable tuner, checkpoint callbacks, early stopping callbacks, loggers and logger callbacks like `LearningRateLogger` and runs for only 1 epoch.

24.2 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)
```

24.3 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)
```

24.4 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)

# similar, but with a fixed 10 batches no matter the size of the dataset
trainer = Trainer(overfit_batches=10)
```

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.

24.5 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 | 1.0 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`

24.6 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)
```

24.7 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=2)
```

CHAPTER

TWENTYFIVE

LOGGERS

Lightning supports the most popular logging frameworks (TensorBoard, Comet, etc...). TensorBoard is used by default, but you can pass to the *Trainer* any combination of the following loggers.

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')*

Read more about *logging* options.

To log arbitrary artifacts like images or audio samples use the *trainer.log_dir* property to resolve the path.

```
def training_step(self, batch, batch_idx):
    img = ...
    log_image(img, self.trainer.log_dir)
```

25.1 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.

25.2 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.

25.3 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.

25.4 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.

25.5 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.

25.6 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(offline=True)
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.

25.7 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

TWENTYSIX

EARLY STOPPING

26.1 Stopping an epoch early

You can stop an epoch early by overriding *on_train_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.

26.2 Early stopping based on metric using the EarlyStopping Callback

The *EarlyStopping* callback can be used to monitor a validation metric and stop the training when no improvement is observed.

To enable it:

- Import *EarlyStopping* callback.
- Log the metric you want to monitor using *log()* method.
- Init the callback, and set *monitor* to the logged metric of your choice.
- Pass the *EarlyStopping* callback to the *Trainer* callbacks flag.

```
from pytorch_lightning.callbacks.early_stopping import EarlyStopping

def validation_step(...):
    self.log('val_loss', loss)

trainer = Trainer(callbacks=[EarlyStopping(monitor='val_loss')])
```

- You can customize the callbacks behaviour by changing its parameters.

```
early_stop_callback = EarlyStopping(
   monitor='val_accuracy',
   min_delta=0.00,
   patience=3,
   verbose=False,
   mode='max'
```

(continues on next page)

(continued from previous page)

```
)
trainer = Trainer(callbacks=[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 in 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*

See also:

- *Trainer*
- *EarlyStopping*

CHAPTER

TWENTYSEVEN

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.

27.1 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)
```

27.2 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)
```

27.3 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)
```

27.4 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
)
```

If you also pass `shuffle=True` to the dataloader, a different random subset of your dataset will be used for each epoch; otherwise the same subset will be used for all epochs.

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

TWENTYEIGHT

HYPERPARAMETERS

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

28.1 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
```

28.2 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 (i.e.: 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)
```

28.3 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 (i.e.: 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 of 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')

        # Now possible to access layer_1_dim from hparams
        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 = nn.Linear(28 * 28, self.hparams.layer_1_dim)
        self.layer_2 = nn.Linear(self.hparams.layer_1_dim, self.hparams.layer_2_
↪dim)
        self.layer_3 = nn.Linear(self.hparams.layer_2_dim, 10)
    def train_dataloader(self):
        return DataLoader(mnist_train, batch_size=self.hparams.batch_size)
```

> **Warning:** Deprecated since v1.1.0. This method of assigning hyperparameters to the LightningModule will no longer be supported from v1.3.0. Use the `self.save_hyperparameters()` method from above instead.

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.save_hyperparameters(conf)

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

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

# Now possible to access any stored variables from hparams
model.hparams.anything
```

28.4 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=[...
↪])
```

28.5 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 = nn.Linear(28 * 28, layer_1_dim)
```

(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('--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)

    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
```

CHAPTER

TWENTYNINE

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 Finder support for DDP is not implemented yet, it is coming soon.

29.1 Using Lightning's built-in LR finder

To enable the learning rate finder, your *lightning module* needs to have a `learning_rate` or `lr` property. Then, set `Trainer(auto_lr_find=True)` during trainer construction, and then call `trainer.tune(model)` to run the LR finder. The suggested `learning_rate` will be written to the console and will be automatically set to your *lightning module*, which can be accessed via `self.learning_rate` or `self.lr`.

```
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))

model = LitModel()

# finds learning rate automatically
# sets hparams.lr or hparams.learning_rate to that learning rate
trainer = Trainer(auto_lr_find=True)

trainer.tune(model)
```

If your model is using an arbitrary value instead of `self.lr` or `self.learning_rate`, set that value as `auto_lr_find`:

```
model = LitModel()

# to set to your own hparams.my_value
trainer = Trainer(auto_lr_find='my_value')

trainer.tune(model)
```

If you want to inspect the results of the learning rate finder 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.tuner.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 achieves 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()`.

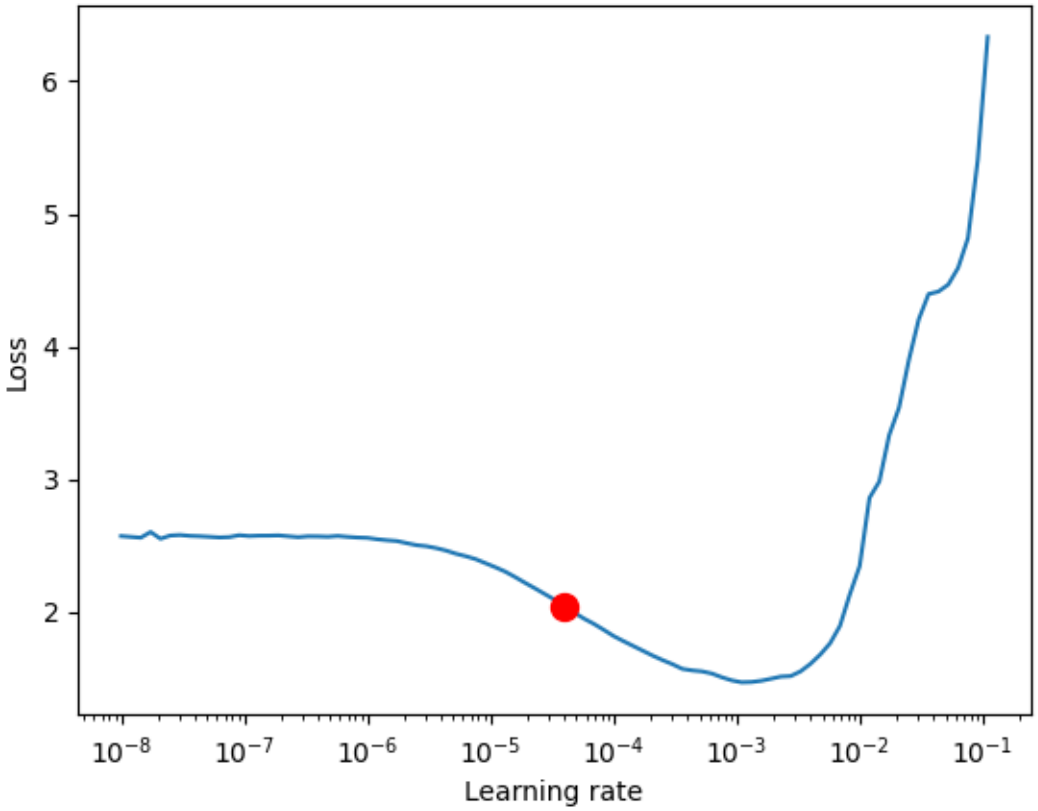
The parameters of the algorithm can be seen below.

pytorch_lightning.tuner.lr_finder.**lr_find**(*trainer*, *model*, *train_dataloader=None*, *val_dataloaders=None*, *min_lr=1e-08*, *max_lr=1*, *num_training=100*, *mode='exponential'*, *early_stop_threshold=4.0*, *datamodule=None*, *update_attr=False*)

`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


Loss
6
5
4
3
2
10^{-8}
10^{-7}
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
Learning rate

- **num_training**¶ (int) – number of learning rates to test
- **mode**¶ (str) – Search strategy to update learning rate after each batch:
  - 'exponential' (default): Will increase the learning rate exponentially.
  - 'linear': Will increase the learning rate linearly.
- **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.
- **datamodule**¶ (Optional[*LightningDataModule*]) – An optional LightningDataModule which holds the training and validation dataloader(s). Note that the train_dataloader and val_dataloaders parameters cannot be used at the same time as this parameter, or a MisconfigurationException will be raised.
- **update_attr**¶ (bool) – Whether to update the learning rate attribute or not.

Example:

```
# Setup model and trainer
model = MyModelClass(hparams)
trainer = pl.Trainer()

# Run lr finder
lr_finder = trainer.tuner.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)
```

CHAPTER

THIRTY

MULTI-GPU TRAINING

Lightning supports multiple ways of doing distributed training.

30.1 Preparing your code

To train on CPU/GPU/TPU without changing your code, we need to build a few good habits :)

30.1.1 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)
```

30.1.2 Init tensors using type_as and register_buffer

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):
```

(continues on next page)

(continued from previous page)

```
    z = torch.Tensor(2, 3)
    z = z.type_as(x)
```

The *LightningModule* knows what device it is on. You can access the reference via `self.device`. Sometimes it is necessary to store tensors as module attributes. However, if they are not parameters they will remain on the CPU even if the module gets moved to a new device. To prevent that and remain device agnostic, register the tensor as a buffer in your modules's `__init__` method with `register_buffer()`.

```
class LitModel(LightningModule):

    def __init__(self):
        ...
        self.register_buffer("sigma", torch.eye(3))
        # you can now access self.sigma anywhere in your module
```

30.1.3 Remove samplers

In PyTorch, you must use `DistributedSampler` for multi-node or TPU training. 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: By default it will add `shuffle=True` for train sampler and `shuffle=False` for val/test sampler. `drop_last` in `DistributedSampler` will be set to its default value in PyTorch.

Note: You can disable this behavior with `Trainer(replace_sampler_ddp=False)`

Note: For iterable datasets, we don't do this automatically.

30.1.4 Synchronize validation and test logging

When running in distributed mode, we have to ensure that the validation and test step logging calls are synchronized across processes. This is done by adding `sync_dist=True` to all `self.log` calls in the validation and test step. This ensures that each GPU worker has the same behaviour when tracking model checkpoints, which is important for later downstream tasks such as testing the best checkpoint across all workers.

Note if you use any built in metrics or custom metrics that use the *Metrics API*, these do not need to be updated and are automatically handled for you.

```
def validation_step(self, batch, batch_idx):
    x, y = batch
    logits = self(x)
    loss = self.loss(logits, y)
    # Add sync_dist=True to sync logging across all GPU workers
    self.log('validation_loss', loss, on_step=True, on_epoch=True, sync_dist=True)

def test_step(self, batch, batch_idx):
    x, y = batch
    logits = self(x)
    loss = self.loss(logits, y)
    # Add sync_dist=True to sync logging across all GPU workers
    self.log('test_loss', loss, on_step=True, on_epoch=True, sync_dist=True)
```

30.1.5 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)
```

(continues on next page)

(continued from previous page)

```
_pickle.PicklingError: Can't pickle [THIS IS THE THING TO FIND AND DELETE]:
attribute lookup <lambda> on __main__ failed
```

30.2 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 per node
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.device_count()))
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

Note: When specifying number of gpus as an integer `gpus=k`, setting the trainer flag `auto_select_gpus=True` will automatically help you find `k` gpus that are not occupied by other processes. This is especially useful when GPUs are configured to be in "exclusive mode", such that only one process at a time can access them. For more details see the *trainer guide*.

30.3 Select torch distributed backend

By default, Lightning will select the `nccl` backend over `gloo` when running on GPUs. Find more information about PyTorch's supported backends here.

Lightning exposes an environment variable `PL_TORCH_DISTRIBUTED_BACKEND` for the user to change the backend.

```
PL_TORCH_DISTRIBUTED_BACKEND=gloo python train.py ...
```

30.4 Distributed modes

Lightning allows multiple ways of training

- Data Parallel (`accelerator='dp'`) (multiple-gpus, 1 machine)
- DistributedDataParallel (`accelerator='ddp'`) (multiple-gpus across many machines (python script based)).
- DistributedDataParallel (`accelerator='ddp_spawn'`) (multiple-gpus across many machines (spawn based)).
- DistributedDataParallel 2 (`accelerator='ddp2'`) (DP in a machine, DDP across machines).
- Horovod (`accelerator='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.

30.4.1 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, accelerator='dp')
```

30.4.2 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.
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, accelerator='ddp')

# train on 32 GPUs (4 nodes)
trainer = Trainer(gpus=8, accelerator='ddp', num_nodes=4)
```

This Lightning implementation of DDP calls your script under the hood multiple times with the correct environment variables:

```
# 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.

There are cases in which it is NOT possible to use DDP. Examples are:

- Jupyter Notebook, Google COLAB, Kaggle, etc.
- You have a nested script without a root package
- Your script needs to invoke both *.fit* and *.test*, or one of them multiple times

In these situations you should use *dp* or *ddp_spawn* instead.

30.4.3 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, accelerator='ddp2', num_nodes=4)
```

30.4.4 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, accelerator='ddp_spawn')
```

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'
```

(continues on next page)

(continued from previous page)

```
      ],
      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 --accelerator 'ddp' --gpus 8
```

30.4.5 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(accelerator='horovod', gpus=1)

# train Horovod on CPU (number of processes / machines provided on command-line)
trainer = Trainer(accelerator='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.

30.4.6 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)

    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)
    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):
    ...
```

30.4.7 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, precision=16)*
	Y	Y			*Trainer(gpus=k, accelerator='dp')*
	Y		Y		*Trainer(gpus=k, accelerator='ddp')*
	Y		Y	Y	*Trainer(gpus=k, accelerator='ddp', precision=16)*

30.4.8 Implement Your Own Distributed (DDP) training

If you need your own way to init PyTorch DDP you can override `pytorch_lightning.plugins.training_type.ddp.DDPPlugin.init_ddp_connection()`.

If you also need to use your own DDP implementation, override `pytorch_lightning.plugins.training_type.ddp.DDPPlugin.configure_ddp()`.

30.5 Model Parallelism [BETA]

Model Parallelism tackles training large models on distributed systems, by modifying distributed communications and memory management of the model. Unlike data parallelism, the model is partitioned in various ways across the GPUs, in most cases to reduce the memory overhead when training large models. This is useful when dealing with large Transformer based models, or in environments where GPU memory is limited.

Lightning currently offers the following methods to leverage model parallelism:

- Sharded Training (partitioning your gradients and optimizer state across multiple GPUs, for reduced memory overhead with **no performance loss**)
- Sequential Model Parallelism with Checkpointing (partition your `nn.Sequential` module across multiple GPUs, leverage checkpointing and microbatching for further memory improvements and device utilization)

30.5.1 Sharded Training

Lightning integration of optimizer sharded training provided by FairScale. The technique can be found within DeepSpeed ZeRO and ZeRO-2, however the implementation is built from the ground up to be pytorch compatible and standalone. Sharded Training allows you to maintain GPU scaling efficiency, whilst reducing memory overhead drastically. In short, expect normal linear scaling, and significantly reduced memory usage when training large models.

Sharded Training still utilizes Data Parallel Training under the hood, except optimizer states and gradients are sharded across GPUs. This means the memory overhead per GPU is lower, as each GPU only has to maintain a partition of your optimizer state and gradients.

The benefits vary by model and parameter sizes, but we've recorded up to a 63% memory reduction per GPU allowing us to double our model sizes. Because of extremely efficient communication, these benefits in multi-GPU setups are almost free and throughput scales well with multi-node setups.

Below we use the NeMo Transformer Lightning Language Modeling example to benchmark the maximum batch size and model size that can be fit on 8 A100 GPUs for DDP vs Sharded Training. Note that the benefits can still be obtained using 2 or more GPUs, and for even larger batch sizes you can scale to multiple nodes.

Increase Your Batch Size

Use Sharded Training to scale your batch size further using the same compute. This will reduce your overall epoch time.

Distributed Training	Model Size (Millions)	Max Batch Size	Percentage Gain (%)
Native DDP	930	32	•
Sharded DDP	930	**52**	**48%**

Increase Your Model Size

Use Sharded Training to scale your model size further using the same compute.

Distributed Training	Batch Size	Max Model Size (Millions)	Percentage Gain (%)
Native DDP	32	930	•
Sharded DDP	32	**1404**	**41%**
Native DDP	8	1572	•
Sharded DDP	8	**2872**	**59%**

It is highly recommended to use Sharded Training in multi-GPU environments where memory is limited, or where training larger models are beneficial (500M+ parameter models). A technical note: as batch size scales, storing activations for the backwards pass becomes the bottleneck in training. As a result, sharding optimizer state and gradients becomes less impactful. Work within the future will bring optional sharding to activations and model parameters to reduce memory further, but come with a speed cost.

To use Sharded Training, you need to first install FairScale using the command below.

```
pip install fairscale
```

```
# train using Sharded DDP
trainer = Trainer(accelerator='ddp', plugins='ddp_sharded')
```

Sharded Training can work across all DDP variants by adding the additional `--plugins ddp_sharded` flag.

Internally we re-initialize your optimizers and shard them across your machines and processes. We handle all communication using PyTorch distributed, so no code changes are required.

30.5.2 DeepSpeed

Note: The DeepSpeed plugin is in beta and the API is subject to change. Please create an issue if you run into any issues.

DeepSpeed offers additional CUDA deep learning training optimizations, similar to FairScale. DeepSpeed offers lower level training optimizations, and useful efficient optimizers such as 1-bit Adam. Using the plugin, we were able to **train model sizes of 10 Billion parameters and above**, with a lot of useful information in this benchmark and the DeepSpeed docs. We recommend using DeepSpeed in environments where speed and memory optimizations are important (such as training large billion parameter models). In addition, we recommend trying *Sharded Training* first before trying DeepSpeed's further optimizations, primarily due to FairScale Sharded ease of use in scenarios such as multiple optimizers/schedulers.

To use DeepSpeed, you first need to install DeepSpeed using the commands below.

```
pip install deepspeed mpi4py
```

If you run into an issue with the install or later in training, ensure that the CUDA version of the pytorch you've installed matches your locally installed CUDA (you can see which one has been recognized by running `nvcc --version`). Additionally if you run into any issues installing m4py, ensure you have openmpi installed using `sudo apt install libopenmpi-dev` or `brew install mpich` before running `pip install mpi4py`.

Note: Currently `resume_from_checkpoint` and manual optimization are not supported.

DeepSpeed only supports single optimizer, single scheduler.

ZeRO-Offload

Below we show an example of running ZeRO-Offload. ZeRO-Offload leverages the host CPU to offload optimizer memory/computation, reducing the overall memory consumption. For even more speed benefit, they offer an optimized CPU version of ADAM to run the offloaded computation, which is faster than the standard PyTorch implementation. By default we enable ZeRO-Offload.

Note: To use ZeRO-Offload, you must use `precision=16` or set precision via the DeepSpeed config..

```
from pytorch_lightning import Trainer

model = MyModel()
trainer = Trainer(gpus=4, plugins='deepspeed', precision=16)
trainer.fit(model)
```

This can also be done via the command line using a Pytorch Lightning script:

```
python train.py --plugins deepspeed --precision 16 --gpus 4
```

You can also modify the ZeRO-Offload parameters via the plugin as below.

```
from pytorch_lightning import Trainer
from pytorch_lightning.plugins import DeepSpeedPlugin

model = MyModel()
trainer = Trainer(gpus=4, plugins=DeepSpeedPlugin(allgather_bucket_size=5e8, reduce_
↪bucket_size=5e8), precision=16)
trainer.fit(model)
```

Note: We suggest tuning the `allgather_bucket_size` parameter and `reduce_bucket_size` parameter to find optimum parameters based on your model size. These control how large a buffer we limit the model to using when reducing gradients/gathering updated parameters. Smaller values will result in less memory, but tradeoff with speed.

DeepSpeed allocates a reduce buffer size multiplied by 4.5x so take that into consideration when tweaking the parameters.

The plugin sets a reasonable default of `2e8`, which should work for most low VRAM GPUs (less than `7GB`), allocating roughly `3.6GB` of VRAM as buffer. Higher VRAM GPUs should aim for values around `5e8`.

Custom DeepSpeed Config

DeepSpeed allows use of custom DeepSpeed optimizers and schedulers defined within a config file. This allows you to enable optimizers such as 1-bit Adam.

Note: All plugin default parameters will be ignored when a config object is passed. All compatible arguments can be seen in the DeepSpeed docs.

```
from pytorch_lightning import Trainer
from pytorch_lightning.plugins import DeepSpeedPlugin

deepspeed_config = {
    "zero_allow_untested_optimizer": True,
    "optimizer": {
        "type": "OneBitAdam",
        "params": {
            "lr": 3e-5,
            "betas": [0.998, 0.999],
            "eps": 1e-5,
            "weight_decay": 1e-9,
            "cuda_aware": True,
        },
    },
    'scheduler': {
        "type": "WarmupLR",
        "params": {
            "last_batch_iteration": -1,
            "warmup_min_lr": 0,
            "warmup_max_lr": 3e-5,
            "warmup_num_steps": 100,
        }
    },
    "zero_optimization": {
        "stage": 2, # Enable Stage 2 ZeRO (Optimizer/Gradient state partitioning)
        "cpu_offload": True, # Enable Offloading optimizer state/calculation to the␣
↪host CPU
        "contiguous_gradients": True, # Reduce gradient fragmentation.
        "overlap_comm": True, # Overlap reduce/backward operation of gradients for␣
↪speed.
        "allgather_bucket_size": 2e8, # Number of elements to all gather at once.
        "reduce_bucket_size": 2e8, # Number of elements we reduce/allreduce at once.
    }
}

model = MyModel()
trainer = Trainer(gpus=4, plugins=DeepSpeedPlugin(deepspeed_config), precision=16)
trainer.fit(model)
```

We support taking the config as a json formatted file:

```
from pytorch_lightning import Trainer
from pytorch_lightning.plugins import DeepSpeedPlugin

model = MyModel()
trainer = Trainer(gpus=4, plugins=DeepSpeedPlugin("/path/to/deepspeed_config.json"),␣
↪precision=16)
trainer.fit(model)
```

You can use also use an environment variable via your PyTorch Lightning script:

```
PL_DEEPSPEED_CONFIG_PATH=/path/to/deepspeed_config.json python train.py --plugins␣
↪deepspeed
```

30.5.3 Sequential Model Parallelism with Checkpointing

PyTorch Lightning integration for Sequential Model Parallelism using FairScale. Sequential Model Parallelism splits a sequential module onto multiple GPUs, reducing peak GPU memory requirements substantially. We also provide auto-balancing techniques through FairScale, to find optimal balances for the model across GPUs. In addition, we use Gradient Checkpointing to reduce GPU memory requirements further, and micro-batches to minimizing device under-utilization automatically.

Reference: https://arxiv.org/abs/1811.06965

Note: RPCSequentialPlugin is currently supported only for Pytorch 1.6.

To get started, install FairScale using the command below. We install a specific branch which contains PyTorch related fixes for Sequential Parallelism.

```
pip install https://github.com/PyTorchLightning/fairscale/archive/pl_1.2.0.zip
```

To use Sequential Model Parallelism, you must define a `nn.Sequential` module that defines the layers you wish to parallelize across GPUs. This should be kept within the `sequential_module` variable within your `LightningModule` like below.

```
from pytorch_lightning.plugins.training_type.rpc_sequential import RPCSequentialPlugin
from pytorch_lightning import LightningModule

class MyModel(LightningModule):
    def __init__(self):
        ...
        self.sequential_module = nn.Sequential(my_layers)

# Split my module across 4 gpus, one layer each
model = MyModel()
plugin = RPCSequentialPlugin(balance=[1, 1, 1, 1])
trainer = Trainer(accelerator='ddp', gpus=4, plugins=[plugin])
trainer.fit(model)
```

We provide a minimal example of Sequential Model Parallelism using a convolutional model training on cifar10, split onto GPUs here. To run the example, you need to install Bolts. Install with `pip install pytorch-lightning-bolts`.

When running the Sequential Model Parallelism example on 2 GPUS we achieve these memory savings.

Table 1: GPU Memory Utilization

GPUS	Without Balancing	With Balancing
Gpu 0	4436 MB	1554 MB
Gpu 1	~0	994 MB

To run the example with Sequential Model Parallelism:

```
python pl_examples/basic_examples/conv_sequential_example.py --batch_size 1024 --gpus␣
↪2 --accelerator ddp --use_ddp_sequential
```

To run the same example without Sequential Model Parallelism:

```
python pl_examples/basic_examples/conv_sequential_example.py --batch_size 1024 --gpus␣
↪1
```

30.6 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, accelerator='ddp|horovod')

# effective batch size = 7 * 8 * 10
Trainer(gpus=8, num_nodes=10, accelerator='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, accelerator='ddp2')

# effective batch size = 7 * 10
Trainer(gpus=8, num_nodes=10, accelerator='ddp2')
```

Note: Huge batch sizes are actually really bad for convergence. Check out: Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

---

30.7 TorchElastic

Lightning supports the use of TorchElastic to enable fault-tolerant 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, accelerator='ddp')
```

Following the TorchElastic 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
```

(continues on next page)

(continued from previous page)

```
        --rdzv_backend=etcd
        --rdzv_endpoint=ETCD_HOST:ETCD_PORT
        YOUR_LIGHTNING_TRAINING_SCRIPT.py (--arg1 ... train script args...)
```

See the official TorchElastic documentation for details on installation and more use cases.

30.8 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!

30.9 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

THIRTYONE

MULTIPLE DATASETS

Lightning supports multiple dataloaders in a few ways.

1. Create a dataloader that iterates multiple datasets under the hood.
2. In the training loop you can pass multiple loaders as a dict or list/tuple and lightning will automatically combine the batches from different loaders.
3. In the validation and test loop you also have the option to return multiple dataloaders which lightning will call sequentially.

31.1 Multiple training dataloaders

For training, the usual way to use multiple dataloaders is to create a `DataLoader` class which wraps your multiple 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
```

(continues on next page)

(continued from previous page)

```
        )
        return loader

    def val_dataloader(self):
        # SAME
        ...

    def test_dataloader(self):
        # SAME
        ...
```

However, with lightning you can also return multiple loaders and lightning will take care of batch combination.

For more details please have a look at `multiple_trainloader_mode`

```
class LitModel(LightningModule):

    def train_dataloader(self):

        loader_a = torch.utils.data.DataLoader(range(6), batch_size=4)
        loader_b = torch.utils.data.DataLoader(range(15), batch_size=5)

        # pass loaders as a dict. This will create batches like this:
        # {'a': batch from loader_a, 'b': batch from loader_b}
        loaders = {'a': loader_a,
                   'b': loader_b}

        # OR:
        # pass loaders as sequence. This will create batches like this:
        # [batch from loader_a, batch from loader_b]
        loaders = [loader_a, loader_b]

        return loaders
```

31.2 Test/Val dataloaders

For validation and test dataloaders, lightning also gives you the additional option of passing 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

THIRTYTWO

SAVING AND LOADING WEIGHTS

Lightning automates saving and loading checkpoints. Checkpoints capture the exact value of all parameters used by a model.

Checkpointing your training allows you to resume a training process in case it was interrupted, fine-tune a model or use a pre-trained model for inference without having to retrain the model.

32.1 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)

32.1.1 Automatic saving

Lightning automatically saves a checkpoint for you in your current working directory, with the state of your last training epoch. This makes sure you can resume training in case it was interrupted.

To change the checkpoint path pass in:

```
# saves checkpoints to '/your/path/to/save/checkpoints' at every epoch end
trainer = Trainer(default_root_dir='/your/path/to/save/checkpoints')
```

You can customize the checkpointing behavior to monitor any quantity of your training or validation steps. For example, if you want to update your checkpoints based on your validation loss:

1. Calculate any metric or other quantity you wish to monitor, such as validation loss.
2. Log the quantity using *log()* method, with a key such as *val_loss*.
3. Initializing the *ModelCheckpoint* callback, and set *monitor* to be the key of your quantity.
4. Pass the callback to the *callbacks* `Trainer` flag.

```
from pytorch_lightning.callbacks import ModelCheckpoint

class LitAutoEncoder(LightningModule):
    def validation_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self.backbone(x)

        # 1. calculate loss
        loss = F.cross_entropy(y_hat, y)

        # 2. log `val_loss`
        self.log('val_loss', loss)

# 3. Init ModelCheckpoint callback, monitoring 'val_loss'
checkpoint_callback = ModelCheckpoint(monitor='val_loss')

# 4. Add your callback to the callbacks list
trainer = Trainer(callbacks=[checkpoint_callback])
```

You can also control more advanced options, like *save_top_k*, to save the best k models and the *mode* of the monitored quantity (min/max), *save_weights_only* or *period* to set the interval of epochs between checkpoints, to avoid slowdowns.

```
from pytorch_lightning.callbacks import ModelCheckpoint

class LitAutoEncoder(LightningModule):
    def validation_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self.backbone(x)
        loss = F.cross_entropy(y_hat, y)
        self.log('val_loss', loss)

# saves a file like: my/path/sample-mnist-epoch=02-val_loss=0.32.ckpt
checkpoint_callback = ModelCheckpoint(
    monitor='val_loss',
    dirpath='my/path/',
    filename='sample-mnist-{epoch:02d}-{val_loss:.2f}',
    save_top_k=3,
    mode='min',
)

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

You can retrieve the checkpoint after training by calling

```
checkpoint_callback = ModelCheckpoint(dirpath='my/path/')
trainer = Trainer(callbacks=[checkpoint_callback])
trainer.fit(model)
checkpoint_callback.best_model_path
```

Disabling checkpoints

You can disable checkpointing by passing

```
trainer = Trainer(checkpoint_callback=False)
```

The Lightning checkpoint also saves the arguments passed into the LightningModule init under the *hyper_parameters* key in the checkpoint.

```
class MyLightningModule(LightningModule):

   def __init__(self, learning_rate, *args, **kwargs):
        super().__init__()
        self.save_hyperparameters()

# all init args were saved to the checkpoint
checkpoint = torch.load(CKPT_PATH)
print(checkpoint['hyper_parameters'])
# {'learning_rate': the_value}
```

32.1.2 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")
```

32.1.3 Manual saving with accelerators

Lightning also handles accelerators where multiple processes are running, such as DDP. For example, when using the DDP accelerator our training script is running across multiple devices at the same time. Lightning automatically ensures that the model is saved only on the main process, whilst other processes do not interfere with saving checkpoints. This requires no code changes as seen below.

```
trainer = Trainer(accelerator="ddp")
model = MyLightningModule(hparams)
trainer.fit(model)
# Saves only on the main process
trainer.save_checkpoint("example.ckpt")
```

Not using *trainer.save_checkpoint* can lead to unexpected behaviour and potential deadlock. Using other saving functions will result in all devices attempting to save the checkpoint. As a result, we highly recommend using the trainer's save functionality. If using custom saving functions cannot be avoided, we recommend using `rank_zero_only()` to ensure saving occurs only on the main process.

32.2 Checkpoint loading

To load a model along with its weights, biases and hyperparameters use the 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.save_hyperparameters()
        self.l1 = nn.Linear(self.hparams.in_dim, self.hparams.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)

# 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)
```

classmethod LightningModule.**load_from_checkpoint**(*checkpoint_path*, *map_location=None*, *hparams_file=None*, *strict=True*, ***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 *hyper_parameters*

Any arguments specified through *args and **kwargs will override args stored in *hyper_parameters*.

Parameters

- **checkpoint_path** (Union[str, IO]) – Path to checkpoint. This can also be a URL, or file-like object
- **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`.

- **strict** (`bool`) – Whether to strictly enforce that the keys in `checkpoint_path` match the keys returned by this module's state dict. Default: *True*.
- **kwargs** – Any extra keyword args needed to init the model. Can also be used to override saved hyperparameter values.

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)
```

32.2.1 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

THIRTYTHREE

OPTIMIZATION

Lightning offers two modes for managing the optimization process:

- automatic optimization (AutoOpt)
- manual optimization

For the majority of research cases, **automatic optimization** will do the right thing for you and it is what most users should use.

For advanced/expert users who want to do esoteric optimization schedules or techniques, use **manual optimization**.

33.1 Manual optimization

For advanced research topics like reinforcement learning, sparse coding, or GAN research, it may be desirable to manually manage the optimization process. To do so, do the following:

- Override your LightningModule `automatic_optimization` property to return `False`
- Drop or ignore the optimizer_idx argument
- Use *self.manual_backward(loss)* instead of *loss.backward()*.

Note: This is only recommended for experts who need ultimate flexibility. Lightning will handle only precision and accelerators logic. The users are left with zero_grad, accumulated_grad_batches, model toggling, etc..

Warning: Before 1.2, `optimzer.step` was calling `zero_grad` internally. From 1.2, it is left to the users expertize.

Tip: To perform `accumulate_grad_batches` with one optimizer, you can do as such.

Tip: `self.optimizers()` will return `LightningOptimizer` objects. You can access your own optimizer with `optimizer.optimizer`. However, if you use your own optimizer to perform a step, Lightning won't be able to support accelerators and precision for you.

```
def training_step(batch, batch_idx, optimizer_idx):
    opt = self.optimizers()

    loss = self.compute_loss(batch)
    self.manual_backward(loss)
    opt.step()

    # accumulate gradient batches
    if batch_idx % 2 == 0:
        opt.zero_grad()
```

Tip: It is a good practice to provide the optimizer with a `closure` function that performs a `forward` and `backward` pass of your model. It is optional for most optimizers, but makes your code compatible if you switch to an optimizer which requires a closure.

Here is the same example as above using a `closure`.

```
def training_step(batch, batch_idx, optimizer_idx):
    opt = self.optimizers()

    def forward_and_backward():
        loss = self.compute_loss(batch)
        self.manual_backward(loss)

    opt.step(closure=forward_and_backward)

    # accumulate gradient batches
    if batch_idx % 2 == 0:
        opt.zero_grad()
```

```
# Scenario for a GAN.

def training_step(...):
    opt_gen, opt_dis = self.optimizers()

    # compute generator loss
    loss_gen = self.compute_generator_loss(...)

    # zero_grad needs to be called before backward
    opt_gen.zero_grad()
    self.manual_backward(loss_gen)
    opt_gen.step()

    # compute discriminator loss
    loss_dis = self.compute_discriminator_loss(...)

    # zero_grad needs to be called before backward
    opt_dis.zero_grad()
    self.manual_backward(loss_dis)
    opt_dis.step()
```

Note: `LightningOptimizer` provides a `toggle_model` function as a `@context_manager` for advanced users. It can be useful when performing gradient accumulation with several optimizers or training in a distributed

setting.

Here is an explanation of what it does:

Considering the current optimizer as A and all other optimizers as B. Toggling means that all parameters from B exclusive to A will have their `requires_grad` attribute set to `False`. Their original state will be restored when exiting the context manager.

When performing gradient accumulation, there is no need to perform grad synchronization during the accumulation phase. Setting `sync_grad` to `False` will block this synchronization and improve your training speed.

Here is an example on how to use it:

```
# Scenario for a GAN with gradient accumulation every 2 batches and optimized for␣
↪multiple gpus.

def training_step(self, batch, batch_idx, ...):
    opt_gen, opt_dis = self.optimizers()

    accumulated_grad_batches = batch_idx % 2 == 0

    # compute generator loss
    def closure_gen():
        loss_gen = self.compute_generator_loss(...)
        self.manual_backward(loss_gen)
        if accumulated_grad_batches:
            opt_gen.zero_grad()

    with opt_gen.toggle_model(sync_grad=accumulated_grad_batches):
        opt_gen.step(closure=closure_gen)

    def closure_dis():
        loss_dis = self.compute_discriminator_loss(...)
        self.manual_backward(loss_dis)
        if accumulated_grad_batches:
            opt_dis.zero_grad()

    with opt_dis.toggle_model(sync_grad=accumulated_grad_batches):
        opt_dis.step(closure=closure_dis)
```

33.2 Automatic optimization

With Lightning most users don't have to think about when to call .backward(), .step(), .zero_grad(), since Lightning automates that for you.

Under the hood Lightning does the following:

```
for epoch in epochs:
    for batch in data:
        loss = model.training_step(batch, batch_idx, ...)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

(continues on next page)

(continued from previous page)

```
    for scheduler in schedulers:
        scheduler.step()
```

In the case of multiple optimizers, Lightning does the following:

```
for epoch in epochs:
  for batch in data:
     for opt in optimizers:
        disable_grads_for_other_optimizers()
        train_step(opt)
        opt.step()

  for scheduler in schedulers:
     scheduler.step()
```

33.2.1 Learning rate scheduling

Every optimizer you use can be paired with any LearningRateScheduler. In the basic use-case, the scheduler (or multiple schedulers) should be returned as the second output from the `.configure_optimizers` method:

```
# no LR scheduler
def configure_optimizers(self):
   return Adam(...)

# Adam + LR scheduler
def configure_optimizers(self):
   optimizer = Adam(...)
   scheduler = LambdaLR(optimizer, ...)
   return [optimizer], [scheduler]

# Two optimizers each with a scheduler
def configure_optimizers(self):
   optimizer1 = Adam(...)
   optimizer2 = SGD(...)
   scheduler1 = LambdaLR(optimizer1, ...)
   scheduler2 = LambdaLR(optimizer2, ...)
   return [optimizer1, optimizer2], [scheduler1, scheduler2]
```

When there are schedulers in which the `.step()` method is conditioned on a metric value (for example the `ReduceLROnPlateau` scheduler), Lightning requires that the output from `configure_optimizers` should be dicts, one for each optimizer, with the keyword `monitor` set to metric that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
   return {
       'optimizer': Adam(...),
       'lr_scheduler': ReduceLROnPlateau(optimizer, ...),
       'monitor': 'metric_to_track'
   }

# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
   optimizer1 = Adam(...)
```

(continues on next page)

(continued from previous page)

```
   optimizer2 = SGD(...)
   scheduler1 = ReduceLROnPlateau(optimizer1, ...)
   scheduler2 = LambdaLR(optimizer2, ...)
   return (
       {'optimizer': optimizer1, 'lr_scheduler': scheduler1, 'monitor': 'metric_to_
↪track'},
       {'optimizer': optimizer2, 'lr_scheduler': scheduler2},
   )
```

Note: Metrics can be made availble to condition on by simply logging it using `self.log('metric_to_track', metric_val)` in your lightning module.

By default, all schedulers will be called after each epoch ends. To change this behaviour, a scheduler configuration should be returned as a dict which can contain the following keywords:

- `scheduler` (required): the actual scheduler object
- `monitor` (optional): metric to condition
- `interval` (optional): either `epoch` (default) for stepping after each epoch ends or `step` for stepping after each optimization step
- `frequency` (optional): how many epochs/steps should pass between calls to `scheduler.step()`. Default is 1, corresponding to updating the learning rate after every epoch/step.
- `strict` (optional): if set to `True` will enforce that value specified in `monitor` is available while trying to call `scheduler.step()`, and stop training if not found. If `False` will only give a warning and continue training (without calling the scheduler).
- `name` (optional): if using the *`LearningRateMonitor`* callback to monitor the learning rate progress, this keyword can be used to specify a specific name the learning rate should be logged as.

```
# Same as the above example with additional params passed to the first scheduler
# In this case the ReduceLROnPlateau will step after every 10 processed batches
def configure_optimizers(self):
   optimizers = [Adam(...), SGD(...)]
   schedulers = [
      {
         'scheduler': ReduceLROnPlateau(optimizers[0], ...),
         'monitor': 'metric_to_track',
         'interval': 'step',
         'frequency': 10,
         'strict': True,
      },
      LambdaLR(optimizers[1], ...)
   ]
   return optimizers, schedulers
```

33.2.2 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(...)], {'scheduler': ReduceLROnPlateau(), 'monitor':
↪'metric_to_track'}
```

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 schedulers:
      scheduler.step()
```

33.2.3 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

---

Note: When using Trainer(enable_pl_optimizer=True), there is no need to call *.zero_grad()*.

```
def optimizer_zero_grad(self, current_epoch, batch_idx, optimizer, opt_idx):
  optimizer.zero_grad()

# Alternating schedule for optimizer steps (ie: GANs)
def optimizer_step(self, current_epoch, batch_nb, optimizer, optimizer_idx, closure,␣
↪on_tpu=False, using_native_amp=False, using_lbfgs=False):
    # update generator opt every 2 steps
    if optimizer_idx == 0:
        if batch_nb % 2 == 0 :
           optimizer.step(closure=closure)

    # update discriminator opt every 4 steps
    if optimizer_idx == 1:
        if batch_nb % 4 == 0 :
           optimizer.step(closure=closure)
```

Here we add a learning-rate warm up

```
# learning rate warm-up
def optimizer_step(self, current_epoch, batch_nb, optimizer, optimizer_idx, closure,␣
↪on_tpu=False, using_native_amp=False, using_lbfgs=False):
    # 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(closure=closure)
```

Note: The default `optimizer_step` is relying on the internal `LightningOptimizer` to properly perform a step. It handles TPUs, AMP, accumulate_grad_batches, zero_grad, and much more ...

```
# function hook in LightningModule
def optimizer_step(self, current_epoch, batch_nb, optimizer, optimizer_idx, closure,␣
↪on_tpu=False, using_native_amp=False, using_lbfgs=False):
  optimizer.step(closure=closure)
```

Note: To access your wrapped Optimizer from `LightningOptimizer`, do as follow.

```
# function hook in LightningModule
def optimizer_step(self, current_epoch, batch_nb, optimizer, optimizer_idx, closure,␣
↪on_tpu=False, using_native_amp=False, using_lbfgs=False):

  # `optimizer is a ``LightningOptimizer`` wrapping the optimizer.
  # To access it, do as follow:
  optimizer = optimizer.optimizer

  # run step. However, it won't work on TPU, AMP, etc...
  optimizer.step(closure=closure)
```

33.2.4 Using the closure functions for optimization

When using optimization schemes such as LBFGS, the *second_order_closure* needs to be enabled. By default, this function is defined by wrapping the *training_step* and the backward steps as follows

```
def second_order_closure(pl_module, split_batch, batch_idx, opt_idx, optimizer,␣
↪hidden):
    # Model training step on a given batch
    result = pl_module.training_step(split_batch, batch_idx, opt_idx, hidden)

    # Model backward pass
    pl_module.backward(result, optimizer, opt_idx)

    # on_after_backward callback
    pl_module.on_after_backward(result.training_step_output, batch_idx, result.loss)
```

(continues on next page)

(continued from previous page)

```
    return result

# This default `second_order_closure` function can be enabled by passing it directly␣
↪into the `optimizer.step`
def optimizer_step(self, current_epoch, batch_nb, optimizer, optimizer_idx, second_
↪order_closure, on_tpu=False, using_native_amp=False, using_lbfgs=False):
    # update params
    optimizer.step(second_order_closure)
```

CHAPTER

THIRTYFOUR

PERFORMANCE AND BOTTLENECK PROFILER

Profiling your training run can help you understand if there are any bottlenecks in your code.

34.1 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

34.2 Enable simple profiling

If you only wish to profile the standard actions, you can set *profiler="simple"* when constructing your *Trainer* object.

```
trainer = Trainer(..., profiler="simple")
```

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
```

34.3 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.

```
trainer = Trainer(..., profiler="advanced")

or

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()
```

(continues on next page)

(continued from previous page)

```
    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)
```

34.4 PyTorch Profiling

Autograd includes a profiler that lets you inspect the cost of different operators inside your model - both on the CPU and GPU.

Find the Pytorch Profiler doc at [PyTorch Profiler](https://pytorch-lightning.readthedocs.io/en/stable/profiler.html)

```
trainer = Trainer(..., profiler="pytorch")

or

profiler = PyTorchProfiler(...)
trainer = Trainer(..., profiler=profiler)
```

This profiler works with PyTorch `DistributedDataParallel`. If `output_filename` is provided, each rank will save their profiled operation to their own file.

The profiler's results will be printed on 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.

This profiler will record only for *training_step_and_backward*, *evaluation_step* and *test_step* functions by default. The output below shows the profiling for the action *training_step_and_backward*. The user can provide `PyTorchProfiler(profiled_functions=[...])` to extend the scope of profiled functions.

Note: When using the PyTorch Profiler, wall clock time will not not be representative of the true wall clock time. This is due to forcing profiled operations to be measured synchronously, when many CUDA ops happen asynchronously. It is recommended to use this Profiler to find bottlenecks/breakdowns, however for end to end wall clock time use the *SimpleProfiler*. # noqa E501

```
Profiler Report

Profile stats for: training_step_and_backward
---------------------  ---------------  ---------------  ---------------  ------------
↪---  ---------------
Name                   Self CPU total %  Self CPU total   CPU total %      CPU total ␣
↪      CPU time avg
---------------------  ---------------  ---------------  ---------------  ------------
↪---  ---------------
t                      62.10%           1.044ms          62.77%           1.055ms   ␣
↪     1.055ms
addmm                  32.32%           543.135us        32.69%           549.362us ␣
↪     549.362us
mse_loss               1.35%            22.657us         3.58%            60.105us  ␣
↪     60.105us
```

(continues on next page)

(continued from previous page)

```
mean                    0.22%            3.694us          2.05%            34.523us
↪      34.523us
div_                    0.64%            10.756us         1.90%            32.001us
↪      16.000us
ones_like               0.21%            3.461us          0.81%            13.669us
↪      13.669us
sum_out                 0.45%            7.638us          0.74%            12.432us
↪      12.432us
transpose               0.23%            3.786us          0.68%            11.393us
↪      11.393us
as_strided              0.60%            10.060us         0.60%            10.060us
↪      3.353us
to                      0.18%            3.059us          0.44%            7.464us
↪      7.464us
empty_like              0.14%            2.387us          0.41%            6.859us
↪      6.859us
empty_strided           0.38%            6.351us          0.38%            6.351us
↪      3.175us
fill_                   0.28%            4.782us          0.33%            5.566us
↪      2.783us
expand                  0.20%            3.336us          0.28%            4.743us
↪      4.743us
empty                   0.27%            4.456us          0.27%            4.456us
↪      2.228us
copy_                   0.15%            2.526us          0.15%            2.526us
↪      2.526us
broadcast_tensors       0.15%            2.492us          0.15%            2.492us
↪      2.492us
size                    0.06%            0.967us          0.06%            0.967us
↪      0.484us
is_complex              0.06%            0.961us          0.06%            0.961us
↪      0.481us
stride                  0.03%            0.517us          0.03%            0.517us
↪      0.517us
----------------------  ---------------  ---------------  ---------------  ------------
↪---  ---------------
Self CPU time total: 1.681ms
```

When running with *PyTorchProfiler(emit_nvtx=True)*. You should run as following:

```
nvprof --profile-from-start off -o trace_name.prof -- <regular command here>
```

To visualize the profiled operation, you can either:

- Use:

  ```
  nvvp trace_name.prof
  ```

- Use:

  ```
  python -c 'import torch; print(torch.autograd.profiler.load_nvprof("trace_name.
  ↪prof"))'
  ```

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.**BaseProfiler**(*output_streams=None*)
Bases: abc.ABC

If you wish to write a custom profiler, you should inhereit from this class.

Parameters **output_streams** (Union[list, tuple, None]) – 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.**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.

Args: output_streams: 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.**PyTorchProfiler**(*output_filename=None*, *enabled=True*, *use_cuda=False*, *record_shapes=False*, *profile_memory=False*, *group_by_input_shapes=False*, *with_stack=False*, *use_kineto=False*, *use_cpu=True*, *emit_nvtx=False*, *export_to_chrome=False*, *path_to_export_trace=None*, *row_limit=20*, *sort_by_key=None*, *profiled_functions=None*, *local_rank=None*)
Bases: *pytorch_lightning.profiler.profilers.BaseProfiler*

This profiler uses PyTorch's Autograd Profiler and lets you inspect the cost of different operators inside your model - both on the CPU and GPU

Parameters

- **output_filename**¶ (Optional[str]) – optionally save profile results to file instead of printing to std out when training is finished. When using ddp, each rank will stream the profiled operation to their own file with the extension _{rank}.txt
- **enabled**¶ (bool) – Setting this to False makes this context manager a no-op.
- **use_cuda**¶ (bool) – Enables timing of CUDA events as well using the cudaEvent API. Adds approximately 4us of overhead to each tensor operation.
- **record_shapes**¶ (bool) – If shapes recording is set, information about input dimensions will be collected.
- **profile_memory**¶ (bool) – Whether to report memory usage, default: True (Introduced in PyTorch 1.6.0)
- **group_by_input_shapes**¶ (bool) – Include operator input shapes and group calls by shape.

- **with_stack**¶ (bool) – record source information (file and line number) for the ops (Introduced in PyTorch 1.7.0)
- **use_kineto**¶ (bool) – experimental support for Kineto profiler (Introduced in PyTorch 1.8.0)
- **use_cpu**¶ (bool) – use_kineto=True and can be used to lower the overhead for GPU-only profiling (Introduced in PyTorch 1.8.0)
- **emit_nvtx**¶ (bool) – Context manager that makes every autograd operation emit an NVTX range Run:

```
nvprof --profile-from-start off -o trace_name.prof -- <regular␣
↪command here>
```

To visualize, you can either use:

```
nvvp trace_name.prof
torch.autograd.profiler.load_nvprof(path)
```

- **export_to_chrome**¶ (bool) – Wether to export the sequence of profiled operators for Chrome. It will generate a `.json` file which can be read by Chrome.
- **path_to_export_trace**¶ (Optional[str]) – Directory path to export `.json` traces when using `export_to_chrome=True`. By default, it will be save where the file being is being run.
- **row_limit**¶ (int) – Limit the number of rows in a table, *0* is a special value that removes the limit completely.
- **sort_by_key**¶ (Optional[str]) – Keys to sort out profiled table
- **profiled_functions**¶ (Optional[List]) – list of profiled functions which will create a context manager on. Any other will be pass through.
- **local_rank**¶ (Optional[int]) – When running in distributed setting, local_rank is used for each process to write to their own file if *output_fname* is provided.

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.**SimpleProfiler**(*output_filename=None*, *extended=True*)
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.

Parameters **output_filename** (Optional[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

CHAPTER

THIRTYFIVE

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

THIRTYSIX

SEQUENTIAL DATA

Lightning has built in support for dealing with sequential data.

36.1 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 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)
```

36.2 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.

36.3 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 an int (specifying the number of training batches to run before validation) when initializing the Trainer. This is because the IterableDataset does not have a `__len__` and Lightning requires this to calculate the validation interval when `val_check_interval` is less than one. Similarly, you can set `limit_{mode}_batches` to a float or an int. If it is set to 0.0 or 0 it will set `num_{mode}_batches` to 0, if it is an int it will set `num_{mode}_batches` to `limit_{mode}_batches`, if it is set to 1.0 it will run for the whole dataset, otherwise it will throw an exception. Here mode can be train/val/test.

```
# 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)

# Set limit_val_batches to 0.0 or 0
trainer = Trainer(limit_val_batches=0.0)

# Set limit_val_batches as an int
trainer = Trainer(limit_val_batches=100)
```

CHAPTER

THIRTYSEVEN

TRAINING TRICKS

Lightning implements various tricks to help during training

37.1 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)
```

37.2 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)
```

37.3 Stochastic Weight Averaging

Stochastic Weight Averaging (SWA) can make your models generalize better at virtually no additional cost. This can be used with both non-trained and trained models. The SWA procedure smooths the loss landscape thus making it harder to end up in a local minimum during optimization.

For a more detailed explanation of SWA and how it works, read this post by the PyTorch team.

See also:

StochasticWeightAveraging (Callback)

```
# Enable Stochastic Weight Averaging
trainer = Trainer(stochastic_weight_avg=True)
```

37.4 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')

# find the batch size
trainer.tune(model)
```

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'* will initially also try doubling the batch size until it encounters an OOM, after which it will do a binary search that will finetune the batch size. Additionally, it should be noted that the batch size scaler cannot search for batch sizes larger than the size of the training dataset.

Note: This feature expects that a *batch_size* field is either located as a model attribute i.e. *model.batch_size* or as a field in your *hparams* i.e. *model.hparams.batch_size*. The field 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|self.hparams.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()
tuner = Tuner(trainer)

# Invoke method
new_batch_size = tuner.scale_batch_size(model, *extra_parameters_here)

# 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, etc.) 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 chosen strategy.
3. The found batch size is saved to either *model.batch_size* or *model.hparams.batch_size*
4. Restore the initial state of model and trainer

class pytorch_lightning.tuner.tuning.**Tuner**(*trainer*)

Bases: object

scale_batch_size(*model*, *mode='power'*, *steps_per_trial=3*, *init_val=2*, *max_trials=25*, *batch_arg_name='batch_size'*, ***fit_kwargs*)

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** – 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
- **batch_arg_name** (str) – name of the attribute that stores the batch size. It is expected that the user has provided a model or datamodule that has a hyperparameter with that name. We will look for this attribute name in the following places

- `model`
- `model.hparams`
- `model.datamodule`
- `trainer.datamodule` (the datamodule passed to the tune method)

- **`**fit_kwargs`** – remaining arguments to be passed to .fit(), e.g., dataloader or datamodule.

Warning: Batch size finder is not supported for DDP yet, it is coming soon.

37.5 Sequential Model Parallelism with Checkpointing

PyTorch Lightning integration for Sequential Model Parallelism using FairScale. Sequential Model Parallelism splits a sequential module onto multiple GPUs, reducing peak GPU memory requirements substantially.

For more information, refer to *Sequential Model Parallelism with Checkpointing*.

CHAPTER

THIRTYEIGHT

PRUNING AND QUANTIZATION

Pruning and Quantization are techniques to compress model size for deployment, allowing inference speed up and energy saving without significant accuracy losses.

38.1 Pruning

Warning: Pruning is in beta and subject to change.

Pruning is a technique which focuses on eliminating some of the model weights to reduce the model size and decrease inference requirements.

Pruning has been shown to achieve significant efficiency improvements while minimizing the drop in model performance (prediction quality). Model pruning is recommended for cloud endpoints, deploying models on edge devices, or mobile inference (among others).

To enable pruning during training in Lightning, simply pass in the *ModelPruning* callback to the Lightning Trainer. PyTorch's native pruning implementation is used under the hood.

This callback supports multiple pruning functions: pass any torch.nn.utils.prune function as a string to select which weights to prune (random_unstructured, RandomStructured, etc) or implement your own by subclassing BasePruningMethod.

```
from pytorch_lightning.callbacks import ModelPruning

# set the amount to be the fraction of parameters to prune
trainer = Trainer(callbacks=[ModelPruning("l1_unstructured", amount=0.5)])
```

You can also perform iterative pruning, apply the lottery ticket hypothesis, and more!

```
def compute_amount(epoch):
    # the sum of all returned values need to be smaller than 1
    if epoch == 10:
        return 0.5

    elif epoch == 50:
        return 0.25

   elif 75 < epoch < 99 :
        return 0.01
```

(continues on next page)

(continued from previous page)

```
# the amount can be also be a callable
trainer = Trainer(callbacks=[ModelPruning("l1_unstructured", amount=compute_amount)])
```

38.2 Quantization

Warning: Quantization is in beta and subject to change.

Model quantization is another performance optimization technique that allows speeding up inference and decreasing memory requirements by performing computations and storing tensors at lower bitwidths (such as INT8 or FLOAT16) than floating-point precision. This is particularly beneficial during model deployment.

Quantization Aware Training (QAT) mimics the effects of quantization during training: The computations are carried-out in floating-point precision but the subsequent quantization effect is taken into account. The weights and activations are quantized into lower precision only for inference, when training is completed.

Quantization is useful when it is required to serve large models on machines with limited memory, or when there's a need to switch between models and reducing the I/O time is important. For example, switching between monolingual speech recognition models across multiple languages.

Lightning includes *QuantizationAwareTraining* callback (using PyTorch's native quantization, read more here), which allows creating fully quantized models (compatible with torchscript).

```
from pytorch_lightning.callbacks import QuantizationAwareTraining

class RegressionModel(LightningModule):

    def __init__(self):
        super().__init__()
        self.layer_0 = nn.Linear(16, 64)
        self.layer_0a = torch.nn.ReLU()
        self.layer_1 = nn.Linear(64, 64)
        self.layer_1a = torch.nn.ReLU()
        self.layer_end = nn.Linear(64, 1)

    def forward(self, x):
        x = self.layer_0(x)
        x = self.layer_0a(x)
        x = self.layer_1(x)
        x = self.layer_1a(x)
        x = self.layer_end(x)
        return x

trainer = Trainer(callbacks=[QuantizationAwareTraining()])
qmodel = RegressionModel()
trainer.fit(qmodel, ...)

batch = iter(my_dataloader()).next()
qmodel(qmodel.quant(batch[0]))

tsmodel = qmodel.to_torchscript()
tsmodel(tsmodel.quant(batch[0]))
```

You can further customize the callback:

```
    qcb = QuantizationAwareTraining(
        # specification of quant estimation quality
        observer_type='histogram',
        # specify which layers shall be merged together to increase efficiency
        modules_to_fuse=[(f'layer_{i}', f'layer_{i}a') for i in range(2)]
        # make your model compatible with all original input/outputs, in such case␣
↪the model is wrapped in a shell with entry/exit layers.
        input_compatible=True
    )

batch = iter(my_dataloader()).next()
qmodel(batch[0])
```

CHAPTER

THIRTYNINE

TRANSFER LEARNING

39.1 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!

39.2 Example: Imagenet (computer Vision)

```
import torchvision.models as models

class ImagenetTransferLearning(LightningModule):
    def __init__(self):
        super().__init__()

        # init a pretrained resnet
        backbone = models.resnet50(pretrained=True)
        num_filters = backbone.fc.in_features
        layers = list(backbone.children())[:-1]
        self.feature_extractor = nn.Sequential(*layers)

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

    def forward(self, x):
        self.feature_extractor.eval()
        with torch.no_grad():
            representations = self.feature_extractor(x).flatten(1)
        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.

39.3 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)
```

(continues on next page)

(continued from previous page)

```
        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

FORTY

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.

40.1 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.

40.2 How to access TPUs

To access TPUs, there are three main ways.

1. Using Google Colab.
2. Using Google Cloud (GCP).
3. Using Kaggle.

40.3 Kaggle TPUs

For starting Kaggle projects with TPUs, refer to this kernel.

40.4 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 1.7 --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.

40.5 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.

40.6 TPU core training

Lightning supports training on a single TPU core or 8 TPU cores.

The Trainer parameters `tpu_cores` defines how many TPU cores to train on (1 or 8) / Single TPU to train on [1].

For Single TPU training, Just pass the TPU core ID [1-8] in a list.

Single TPU core training. Model will train on TPU core ID 5.

```
trainer = pl.Trainer(tpu_cores=[5])
```

8 TPU cores training. Model will train on 8 TPU cores.

```
trainer = pl.Trainer(tpu_cores=8)
```

40.7 Distributed Backend with TPU

The `accelerator` option used for GPUs does not apply to TPUs. TPUs work in DDP mode by default (distributing over each core)

40.8 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
```

See this guide on how to set up the instance groups and VMs needed to run TPU Pods.

40.9 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, 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.

40.10 Weight Sharing/Tying

Weight Tying/Sharing is a technique where in the module weights are shared among two or more layers. This is a common method to reduce memory consumption and is utilized in many State of the Art architectures today.

PyTorch XLA requires these weights to be tied/shared after moving the model to the TPU device. To support this requirement Lightning provides a model hook which is called after the model is moved to the device. Any weights that require to be tied should be done in the *on_post_move_to_device* model hook. This will ensure that the weights among the modules are shared and not copied.

PyTorch Lightning has an inbuilt check which verifies that the model parameter lengths match once the model is moved to the device. If the lengths do not match Lightning throws a warning message.

Example:

```
from pytorch_lightning.core.lightning import LightningModule
from torch import nn
from pytorch_lightning.trainer.trainer import Trainer


class WeightSharingModule(LightningModule):
    def __init__(self):
        super().__init__()
        self.layer_1 = nn.Linear(32, 10, bias=False)
        self.layer_2 = nn.Linear(10, 32, bias=False)
```

(continues on next page)

(continued from previous page)

```
        self.layer_3 = nn.Linear(32, 10, bias=False)
        # TPU shared weights are copied independently
        # on the XLA device and this line won't have any effect.
        # However, it works fine for CPU and GPU.
        self.layer_3.weight = self.layer_1.weight

    def forward(self, x):
        x = self.layer_1(x)
        x = self.layer_2(x)
        x = self.layer_3(x)
        return x

    def on_post_move_to_device(self):
        # Weights shared after the model has been moved to TPU Device
        self.layer_3.weight = self.layer_1.weight


model = WeightSharingModule()
trainer = Trainer(max_epochs=1, tpu_cores=8)
```

See XLA Documentation

40.11 Performance considerations

The TPU was designed for specific workloads and operations to carry out large volumes of matrix multiplication, convolution operations and other commonly used ops in applied deep learning. The specialization makes it a strong choice for NLP tasks, sequential convolutional networks, and under low precision operation. There are cases in which training on TPUs is slower when compared with GPUs, for possible reasons listed:

- Too small batch size.
- Explicit evaluation of tensors during training, e.g. `tensor.item()`
- Tensor shapes (e.g. model inputs) change often during training.
- Limited resources when using TPU's with PyTorch Link
- XLA Graph compilation during the initial steps Reference
- Some tensor ops are not fully supported on TPU, or not supported at all. These operations will be performed on CPU (context switch).
- PyTorch integration is still experimental. Some performance bottlenecks may simply be the result of unfinished implementation.

The official PyTorch XLA performance guide has more detailed information on how PyTorch code can be optimized for TPU. In particular, the metrics report allows one to identify operations that lead to context switching.

40.12 About XLA

XLA is the library that interfaces PyTorch with the TPUs. For more information check out XLA.

Guide for troubleshooting XLA

CHAPTER

FORTYONE

COMPUTING CLUSTER

With Lightning it is easy to run your training script on a computing cluster without almost any modifications to the script. This guide shows how to run a training job on a general purpose cluster.

Also, check *Accelerators* as a new and more general approach to a cluster setup.

41.1 Cluster setup

To setup a multi-node computing cluster you need:

1) Multiple computers with PyTorch Lightning installed

2) A network connectivity between them with firewall rules that allow traffic flow on a specified *MASTER_PORT*.

3) Defined environment variables on each node required for the PyTorch Lightning multi-node distributed training

PyTorch Lightning follows the design of PyTorch distributed communication package. and requires the following environment variables to be defined on each node:

- *MASTER_PORT* - required; has to be a free port on machine with NODE_RANK 0
- *MASTER_ADDR* - required (except for NODE_RANK 0); address of NODE_RANK 0 node
- *WORLD_SIZE* - required; how many nodes are in the cluster
- *NODE_RANK* - required; id of the node in the cluster

41.2 Training script design

To train a model using multiple nodes, do the following:

1. Design your *LightningModule* (no need to add anything specific here).
2. Enable DDP in the trainer

```
# train on 32 GPUs across 4 nodes
trainer = Trainer(gpus=8, num_nodes=4, accelerator='ddp')
```

41.3 Submit a job to the cluster

To submit a training job to the cluster you need to run the same training script on each node of the cluster. This means that you need to:

1. Copy all third-party libraries to each node (usually means - distribute requirements.txt file and install it).
2. Copy all your import dependencies and the script itself to each node.
3. Run the script on each node.

CHAPTER

FORTYTWO

TEST SET

Lightning forces the user to run the test set separately to make sure it isn't evaluated by mistake. Testing is performed using the `trainer` object's `.test()` method.

`Trainer.test`(*model=None*, *test_dataloaders=None*, *ckpt_path='best'*, *verbose=True*, *datamodule=None*)

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

Parameters

- **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`.
- **datamodule** (Optional[*LightningDataModule*]) – A instance of `LightningDataModule`.
- **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.
- **verbose** (bool) – If True, prints the test results

Returns Returns a list of dictionaries, one for each test dataloader containing their respective metrics.

42.1 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)
```

42.2 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)
```

42.3 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...)

42.4 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 *lightning module* 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. Additionally, you can also pass in an *datamodules* that have overridden the *test_dataloader* method.

```
class MyDataModule(pl.LightningDataModule):
    ...
    def test_dataloader(self):
        return DataLoader(...)

# setup your datamodule
dm = MyDataModule(...)

# test (pass in datamodule)
trainer.test(datamodule=dm)
```

CHAPTER

FORTYTHREE

INFERENCE IN PRODUCTION

PyTorch Lightning eases the process of deploying models into production.

43.1 Exporting to ONNX

PyTorch Lightning provides a handy function to quickly export your model to ONNX format, which allows the model to be independent of PyTorch and run on an ONNX Runtime.

To export your model to ONNX format call the `to_onnx` function on your Lightning Module with the filepath and input_sample.

```
filepath = 'model.onnx'
model = SimpleModel()
input_sample = torch.randn((1, 64))
model.to_onnx(filepath, input_sample, export_params=True)
```

You can also skip passing the input sample if the ` example_input_array ` property is specified in your LightningModule.

Once you have the exported model, you can run it on your ONNX runtime in the following way:

```
ort_session = onnxruntime.InferenceSession(filepath)
input_name = ort_session.get_inputs()[0].name
ort_inputs = {input_name: np.random.randn(1, 64).astype(np.float32)}
ort_outs = ort_session.run(None, ort_inputs)
```

43.2 Exporting to TorchScript

TorchScript allows you to serialize your models in a way that it can be loaded in non-Python environments. The LightningModule has a handy method *to_torchscript()* that returns a scripted module which you can save or directly use.

```
model = SimpleModel()
script = model.to_torchscript()

# save for use in production environment
torch.jit.save(script, "model.pt")
```

It is recommended that you install the latest supported version of PyTorch to use this feature without limitations.

CHAPTER

FORTYFOUR

CONVERSATIONAL AI

These are amazing ecosystems to help with Automatic Speech Recognition (ASR), Natural Language Processing (NLP), and Text to speech (TTS).

44.1 NeMo

NVIDIA NeMo is a toolkit for building new State-of-the-Art Conversational AI models. NeMo has separate collections for Automatic Speech Recognition (ASR), Natural Language Processing (NLP), and Text-to-Speech (TTS) models. Each collection consists of prebuilt modules that include everything needed to train on your data. Every module can easily be customized, extended, and composed to create new Conversational AI model architectures.

Conversational AI architectures are typically very large and require a lot of data and compute for training. NeMo uses PyTorch Lightning for easy and performant multi-GPU/multi-node mixed-precision training.

Note: Every NeMo model is a LightningModule that comes equipped with all supporting infrastructure for training and reproducibility.

44.1.1 NeMo Models

NeMo Models contain everything needed to train and reproduce state of the art Conversational AI research and applications, including:

- neural network architectures
- datasets/data loaders
- data preprocessing/postprocessing
- data augmentors
- optimizers and schedulers
- tokenizers
- language models

NeMo uses Hydra for configuring both NeMo models and the PyTorch Lightning Trainer. Depending on the domain and application, many different AI libraries will have to be configured to build the application. Hydra makes it easy to bring all of these libraries together so that each can be configured from .yaml or the Hydra CLI.

Note: Every NeMo model has an example configuration file and a corresponding script that contains all configurations needed for training.

The end result of using NeMo, Pytorch Lightning, and Hydra is that NeMo models all have the same look and feel. This makes it easy to do Conversational AI research across multiple domains. NeMo models are also fully compatible with the PyTorch ecosystem.

Installing NeMo

Before installing NeMo, please install Cython first.

```
pip install Cython
```

For ASR and TTS models, also install these linux utilities.

```
apt-get update && apt-get install -y libsndfile1 ffmpeg
```

Then installing the latest NeMo release is a simple pip install.

```
pip install nemo_toolkit[all]==1.0.0b1
```

To install the main branch from GitHub:

```
python -m pip install git+https://github.com/NVIDIA/NeMo.git@main#egg=nemo_
↪toolkit[all]
```

To install from a local clone of NeMo:

```
./reinstall.sh # from cloned NeMo's git root
```

For Docker users, the NeMo container is available on NGC.

```
docker pull nvcr.io/nvidia/nemo:v1.0.0b1
```

```
docker run --runtime=nvidia -it --rm -v --shm-size=8g -p 8888:8888 -p 6006:6006 --
↪ulimit memlock=-1 --ulimit stack=67108864 nvcr.io/nvidia/nemo:v1.0.0b1
```

Experiment Manager

NeMo's Experiment Manager leverages PyTorch Lightning for model checkpointing, TensorBoard Logging, and Weights and Biases logging. The Experiment Manager is included by default in all NeMo example scripts.

```
exp_manager(trainer, cfg.get("exp_manager", None))
```

And is configurable via .yaml with Hydra.

```
exp_manager:
    exp_dir: null
    name: *name
    create_tensorboard_logger: True
    create_checkpoint_callback: True
```

Optionally launch Tensorboard to view training results in ./nemo_experiments (by default).

```
tensorboard --bind_all --logdir nemo_experiments
```

44.1.2 Automatic Speech Recognition (ASR)

Everything needed to train Convolutional ASR models is included with NeMo. NeMo supports multiple Speech Recognition architectures, including Jasper and QuartzNet. NeMo Speech Models can be trained from scratch on custom datasets or fine-tuned using pre-trained checkpoints trained on thousands of hours of audio that can be restored for immediate use.

Some typical ASR tasks are included with NeMo:

- Audio transcription
- Byte Pair/Word Piece Training
- Speech Commands
- Voice Activity Detection
- Speaker Recognition

See this asr notebook for a full tutorial on doing ASR with NeMo, PyTorch Lightning, and Hydra.

Specify ASR Model Configurations with YAML File

NeMo Models and the PyTorch Lightning Trainer can be fully configured from .yaml files using Hydra.

See this asr config for the entire speech to text .yaml file.

```
# configure the PyTorch Lightning Trainer
trainer:
    gpus: 0 # number of gpus
    max_epochs: 5
    max_steps: null # computed at runtime if not set
    num_nodes: 1
    distributed_backend: ddp
    ...
# configure the ASR model
model:
    ...
    encoder:
        cls: nemo.collections.asr.modules.ConvASREncoder
        params:
            feat_in: *n_mels
            activation: relu
            conv_mask: true
```

(continues on next page)

(continued from previous page)

```
        jasper:
            - filters: 128
            repeat: 1
            kernel: [11]
            stride: [1]
            dilation: [1]
            dropout: *dropout
            ...
    # all other configuration, data, optimizer, preprocessor, etc
    ...
```

Developing ASR Model From Scratch

speech_to_text.py

```
# hydra_runner calls hydra.main and is useful for multi-node experiments
@hydra_runner(config_path="conf", config_name="config")
def main(cfg):
    trainer = Trainer(**cfg.trainer)
    asr_model = EncDecCTCModel(cfg.model, trainer)
    trainer.fit(asr_model)
```

Hydra makes every aspect of the NeMo model, including the PyTorch Lightning Trainer, customizable from the command line.

```
python NeMo/examples/asr/speech_to_text.py --config-name=quartznet_15x5 \
    trainer.gpus=4 \
    trainer.max_epochs=128 \
    +trainer.precision=16 \
    model.train_ds.manifest_filepath=<PATH_TO_DATA>/librispeech-train-all.json \
    model.validation_ds.manifest_filepath=<PATH_TO_DATA>/librispeech-dev-other.json \
    model.train_ds.batch_size=64 \
    +model.validation_ds.num_workers=16 \
    +model.train_ds.num_workers=16
```

Note: Training NeMo ASR models can take days/weeks so it is highly recommended to use multiple GPUs and multiple nodes with the PyTorch Lightning Trainer.

Using State-Of-The-Art Pre-trained ASR Model

Transcribe audio with QuartzNet model pretrained on ~3300 hours of audio.

```
quartznet = EncDecCTCModel.from_pretrained('QuartzNet15x5Base-En')

files = ['path/to/my.wav'] # file duration should be less than 25 seconds

for fname, transcription in zip(files, quartznet.transcribe(paths2audio_files=files)):
    print(f"Audio in {fname} was recognized as: {transcription}")
```

To see the available pretrained checkpoints:

```
EncDecCTCModel.list_available_models()
```

NeMo ASR Model Under the Hood

Any aspect of ASR training or model architecture design can easily be customized with PyTorch Lightning since every NeMo model is a Lightning Module.

```
class EncDecCTCModel(ASRModel):
    """Base class for encoder decoder CTC-based models."""
...
    @typecheck()
    def forward(self, input_signal, input_signal_length):
        processed_signal, processed_signal_len = self.preprocessor(
            input_signal=input_signal, length=input_signal_length,
        )
        # Spec augment is not applied during evaluation/testing
        if self.spec_augmentation is not None and self.training:
            processed_signal = self.spec_augmentation(input_spec=processed_signal)
        encoded, encoded_len = self.encoder(audio_signal=processed_signal,␣
↪length=processed_signal_len)
        log_probs = self.decoder(encoder_output=encoded)
        greedy_predictions = log_probs.argmax(dim=-1, keepdim=False)
        return log_probs, encoded_len, greedy_predictions

    # PTL-specific methods
    def training_step(self, batch, batch_nb):
        audio_signal, audio_signal_len, transcript, transcript_len = batch
        log_probs, encoded_len, predictions = self.forward(
            input_signal=audio_signal, input_signal_length=audio_signal_len
        )
        loss_value = self.loss(
            log_probs=log_probs, targets=transcript, input_lengths=encoded_len,␣
↪target_lengths=transcript_len
        )
        wer_num, wer_denom = self._wer(predictions, transcript, transcript_len)
        tensorboard_logs = {
            'train_loss': loss_value,
            'training_batch_wer': wer_num / wer_denom,
            'learning_rate': self._optimizer.param_groups[0]['lr'],
        }
        return {'loss': loss_value, 'log': tensorboard_logs}
```

Neural Types in NeMo ASR

NeMo Models and Neural Modules come with Neural Type checking. Neural type checking is extremely useful when combining many different neural network architectures for a production-grade application.

```
@property
def input_types(self) -> Optional[Dict[str, NeuralType]]:
    if hasattr(self.preprocessor, '_sample_rate'):
        audio_eltype = AudioSignal(freq=self.preprocessor._sample_rate)
    else:
        audio_eltype = AudioSignal()
    return {
```

(continues on next page)

(continued from previous page)

```
        "input_signal": NeuralType(('B', 'T'), audio_eltype),
        "input_signal_length": NeuralType(tuple('B'), LengthsType()),
    }

@property
def output_types(self) -> Optional[Dict[str, NeuralType]]:
    return {
        "outputs": NeuralType(('B', 'T', 'D'), LogprobsType()),
        "encoded_lengths": NeuralType(tuple('B'), LengthsType()),
        "greedy_predictions": NeuralType(('B', 'T'), LabelsType()),
    }
```

44.1.3 Natural Language Processing (NLP)

Everything needed to finetune BERT-like language models for NLP tasks is included with NeMo. NeMo NLP Models include HuggingFace Transformers and NVIDIA Megatron-LM BERT and Bio-Megatron models. NeMo can also be used for pretraining BERT-based language models from HuggingFace.

Any of the HuggingFace encoders or Megatron-LM encoders can easily be used for the NLP tasks that are included with NeMo:

- Glue Benchmark (All tasks)
- Intent Slot Classification
- Language Modeling (BERT Pretraining)
- Question Answering
- Text Classification (including Sentiment Analysis)
- Token Classification (including Named Entity Recognition)
- Punctuation and Capitalization

Named Entity Recognition (NER)

NER (or more generally token classification) is the NLP task of detecting and classifying key information (entities) in text. This task is very popular in Healthcare and Finance. In finance, for example, it can be important to identify geographical, geopolitical, organizational, persons, events, and natural phenomenon entities. See this NER notebook for a full tutorial on doing NER with NeMo, PyTorch Lightning, and Hydra.

Specify NER Model Configurations with YAML File

Note: NeMo Models and the PyTorch Lightning Trainer can be fully configured from .yaml files using Hydra.

See this token classification config for the entire NER (token classification) .yaml file.

```
# configure any argument of the PyTorch Lightning Trainer
trainer:
    gpus: 1 # the number of gpus, 0 for CPU
```

(continues on next page)

(continued from previous page)

```
    num_nodes: 1
    max_epochs: 5
    ...
# configure any aspect of the token classification model here
model:
    dataset:
        data_dir: ??? # /path/to/data
        class_balancing: null # choose from [null, weighted_loss]. Weighted_loss␣
↪enables the weighted class balancing of the loss, may be used for handling␣
↪unbalanced classes
        max_seq_length: 128
        ...
  tokenizer:
    tokenizer_name: ${model.language_model.pretrained_model_name} # or sentencepiece
    vocab_file: null # path to vocab file
    ...
# the language model can be from HuggingFace or Megatron-LM
language_model:
    pretrained_model_name: bert-base-uncased
    lm_checkpoint: null
    ...
# the classifier for the downstream task
  head:
    num_fc_layers: 2
    fc_dropout: 0.5
    activation: 'relu'
    ...
# all other configuration: train/val/test/ data, optimizer, experiment manager, etc
...
```

Developing NER Model From Scratch

token_classification.py

```
# hydra_runner calls hydra.main and is useful for multi-node experiments
@hydra_runner(config_path="conf", config_name="token_classification_config")
def main(cfg: DictConfig) -> None:
    trainer = pl.Trainer(**cfg.trainer)
    model = TokenClassificationModel(cfg.model, trainer=trainer)
    trainer.fit(model)
```

After training, we can do inference with the saved NER model using PyTorch Lightning.

Inference from file:

```
gpu = 1 if cfg.trainer.gpus != 0 else 0
trainer = pl.Trainer(gpus=gpu)
model.set_trainer(trainer)
model.evaluate_from_file(
    text_file=os.path.join(cfg.model.dataset.data_dir, cfg.model.validation_ds.text_
↪file),
    labels_file=os.path.join(cfg.model.dataset.data_dir, cfg.model.validation_ds.
↪labels_file),
    output_dir=exp_dir,
    add_confusion_matrix=True,
```

(continues on next page)

(continued from previous page)

```
    normalize_confusion_matrix=True,
)
```

Or we can run inference on a few examples:

```
queries = ['we bought four shirts from the nvidia gear store in santa clara.',
↪'Nvidia is a company in Santa Clara.']
results = model.add_predictions(queries)

for query, result in zip(queries, results):
    logging.info(f'Query : {query}')
    logging.info(f'Result: {result.strip()}\n')
```

Hydra makes every aspect of the NeMo model, including the PyTorch Lightning Trainer, customizable from the command line.

```
python token_classification.py \
    model.language_model.pretrained_model_name=bert-base-cased \
    model.head.num_fc_layers=2 \
    model.dataset.data_dir=/path/to/my/data  \
    trainer.max_epochs=5 \
    trainer.gpus=[0,1]
```

Tokenizers

Tokenization is the process of converting natural language text into integer arrays which can be used for machine learning. For NLP tasks, tokenization is an essential part of data preprocessing. NeMo supports all BERT-like model tokenizers from HuggingFace's AutoTokenizer and also supports Google's SentencePieceTokenizer which can be trained on custom data.

To see the list of supported tokenizers:

```
from nemo.collections import nlp as nemo_nlp

nemo_nlp.modules.get_tokenizer_list()
```

See this tokenizer notebook for a full tutorial on using tokenizers in NeMo.

Language Models

Language models are used to extract information from (tokenized) text. Much of the state-of-the-art in natural language processing is achieved by fine-tuning pretrained language models on the downstream task.

With NeMo, you can either pretrain a BERT model on your data or use a pretrained language model from HuggingFace Transformers or NVIDIA Megatron-LM.

To see the list of language models available in NeMo:

```
nemo_nlp.modules.get_pretrained_lm_models_list(include_external=True)
```

Easily switch between any language model in the above list by using *.get_lm_model*.

```
nemo_nlp.modules.get_lm_model(pretrained_model_name='distilbert-base-uncased')
```

See this language model notebook for a full tutorial on using pretrained language models in NeMo.

Using a Pre-trained NER Model

NeMo has pre-trained NER models that can be used to get started with Token Classification right away. Models are automatically downloaded from NGC, cached locally to disk, and loaded into GPU memory using the *.from_pretrained* method.

```
# load pre-trained NER model
pretrained_ner_model = TokenClassificationModel.from_pretrained(model_name="NERModel")

# define the list of queries for inference
queries = [
    'we bought four shirts from the nvidia gear store in santa clara.',
    'Nvidia is a company.',
    'The Adventures of Tom Sawyer by Mark Twain is an 1876 novel about a young boy␣
↪growing '
    + 'up along the Mississippi River.',
]
results = pretrained_ner_model.add_predictions(queries)

for query, result in zip(queries, results):
    print()
    print(f'Query : {query}')
    print(f'Result: {result.strip()}\n')
```

NeMo NER Model Under the Hood

Any aspect of NLP training or model architecture design can easily be customized with PyTorch Lightning since every NeMo model is a Lightning Module.

```
class TokenClassificationModel(ModelPT):
    """
    Token Classification Model with BERT, applicable for tasks such as Named Entity␣
↪Recognition
    """
    ...
    @typecheck()
    def forward(self, input_ids, token_type_ids, attention_mask):
        hidden_states = self.bert_model(
            input_ids=input_ids, token_type_ids=token_type_ids, attention_
↪mask=attention_mask
        )
        logits = self.classifier(hidden_states=hidden_states)
        return logits

    # PTL-specfic methods
    def training_step(self, batch, batch_idx):
        """
        Lightning calls this inside the training loop with the data from the training␣
↪dataloader
        passed in as `batch`.
```

(continues on next page)

(continued from previous page)

```
        """
        input_ids, input_type_ids, input_mask, subtokens_mask, loss_mask, labels =␣
↪batch
        logits = self(input_ids=input_ids, token_type_ids=input_type_ids, attention_
↪mask=input_mask)

        loss = self.loss(logits=logits, labels=labels, loss_mask=loss_mask)
        tensorboard_logs = {'train_loss': loss, 'lr': self._optimizer.param_groups[0][
↪'lr']}
        return {'loss': loss, 'log': tensorboard_logs}
    ...
```

Neural Types in NeMo NLP

NeMo Models and Neural Modules come with Neural Type checking. Neural type checking is extremely useful when combining many different neural network architectures for a production-grade application.

```
@property
def input_types(self) -> Optional[Dict[str, NeuralType]]:
    return self.bert_model.input_types

@property
def output_types(self) -> Optional[Dict[str, NeuralType]]:
    return self.classifier.output_types
```

44.1.4 Text-To-Speech (TTS)

Everything needed to train TTS models and generate audio is included with NeMo. NeMo TTS Models can be trained from scratch on your own data or pretrained models can be downloaded automatically. NeMo currently supports a two step inference procedure. First, a model is used to generate a mel spectrogram from text. Second, a model is used to generate audio from a mel spectrogram.

Mel Spectrogram Generators:

- Tacotron 2
- Glow-TTS

Audio Generators:

- Griffin-Lim
- WaveGlow
- SqueezeWave

Specify TTS Model Configurations with YAML File

Note: NeMo Models and PyTorch Lightning Trainer can be fully configured from .yaml files using Hydra.

tts/conf/glow_tts.yaml

```
# configure the PyTorch Lightning Trainer
trainer:
    gpus: -1 # number of gpus
    max_epochs: 350
    num_nodes: 1
    distributed_backend: ddp
    ...

# configure the TTS model
model:
    ...
    encoder:
        cls: nemo.collections.tts.modules.glow_tts.TextEncoder
            params:
            n_vocab: 148
            out_channels: *n_mels
            hidden_channels: 192
            filter_channels: 768
            filter_channels_dp: 256
            ...
# all other configuration, data, optimizer, parser, preprocessor, etc
...
```

Developing TTS Model From Scratch

tts/glow_tts.py

```
# hydra_runner calls hydra.main and is useful for multi-node experiments
@hydra_runner(config_path="conf", config_name="glow_tts")
def main(cfg):
    trainer = pl.Trainer(**cfg.trainer)
    model = GlowTTSModel(cfg=cfg.model, trainer=trainer)
    trainer.fit(model)
```

Hydra makes every aspect of the NeMo model, including the PyTorch Lightning Trainer, customizable from the command line.

```
python NeMo/examples/tts/glow_tts.py \
    trainer.gpus=4 \
    trainer.max_epochs=400 \
    ...
    train_dataset=/path/to/train/data \
    validation_datasets=/path/to/val/data \
    model.train_ds.batch_size = 64 \
```

Note: Training NeMo TTS models from scratch can take days or weeks so it is highly recommended to use multiple GPUs and multiple nodes with the PyTorch Lightning Trainer.

Using State-Of-The-Art Pre-trained TTS Model

Generate speech using models trained on *LJSpeech <https://keithito.com/LJ-Speech-Dataset/>*, around 24 hours of single speaker data.

See this TTS notebook for a full tutorial on generating speech with NeMo, PyTorch Lightning, and Hydra.

```
# load pretrained spectrogram model
spec_gen = SpecModel.from_pretrained('GlowTTS-22050Hz').cuda()

# load pretrained Generators
vocoder = WaveGlowModel.from_pretrained('WaveGlow-22050Hz').cuda()

def infer(spec_gen_model, vocder_model, str_input):
    with torch.no_grad():
        parsed = spec_gen.parse(text_to_generate)
        spectrogram = spec_gen.generate_spectrogram(tokens=parsed)
        audio = vocoder.convert_spectrogram_to_audio(spec=spectrogram)
    if isinstance(spectrogram, torch.Tensor):
        spectrogram = spectrogram.to('cpu').numpy()
    if len(spectrogram.shape) == 3:
        spectrogram = spectrogram[0]
    if isinstance(audio, torch.Tensor):
        audio = audio.to('cpu').numpy()
    return spectrogram, audio

text_to_generate = input("Input what you want the model to say: ")
spec, audio = infer(spec_gen, vocoder, text_to_generate)
```

To see the available pretrained checkpoints:

```
# spec generator
GlowTTSModel.list_available_models()

# vocoder
WaveGlowModel.list_available_models()
```

NeMo TTS Model Under the Hood

Any aspect of TTS training or model architecture design can easily be customized with PyTorch Lightning since every NeMo model is a LightningModule.

glow_tts.py

```
class GlowTTSModel(SpectrogramGenerator):
    """
    GlowTTS model used to generate spectrograms from text
    Consists of a text encoder and an invertible spectrogram decoder
    """
    ...
    # NeMo models come with neural type checking
    @typecheck(
        input_types={
            "x": NeuralType(('B', 'T'), TokenIndex()),
            "x_lengths": NeuralType(('B'), LengthsType()),
            "y": NeuralType(('B', 'D', 'T'), MelSpectrogramType(), optional=True),
            "y_lengths": NeuralType(('B'), LengthsType(), optional=True),
```

(continues on next page)

(continued from previous page)

```
            "gen": NeuralType(optional=True),
            "noise_scale": NeuralType(optional=True),
            "length_scale": NeuralType(optional=True),
        }
    )
    def forward(self, *, x, x_lengths, y=None, y_lengths=None, gen=False, noise_
↪scale=0.3, length_scale=1.0):
        if gen:
            return self.glow_tts.generate_spect(
                text=x, text_lengths=x_lengths, noise_scale=noise_scale, length_
↪scale=length_scale
            )
        else:
            return self.glow_tts(text=x, text_lengths=x_lengths, spect=y, spect_
↪lengths=y_lengths)
    ...
    def step(self, y, y_lengths, x, x_lengths):
        z, y_m, y_logs, logdet, logw, logw_, y_lengths, attn = self(
            x=x, x_lengths=x_lengths, y=y, y_lengths=y_lengths, gen=False
        )

        l_mle, l_length, logdet = self.loss(
            z=z,
            y_m=y_m,
            y_logs=y_logs,
            logdet=logdet,
            logw=logw,
            logw_=logw_,
            x_lengths=x_lengths,
            y_lengths=y_lengths,
        )

        loss = sum([l_mle, l_length])

        return l_mle, l_length, logdet, loss, attn

    # PTL-specfic methods
    def training_step(self, batch, batch_idx):
        y, y_lengths, x, x_lengths = batch

        y, y_lengths = self.preprocessor(input_signal=y, length=y_lengths)

        l_mle, l_length, logdet, loss, _ = self.step(y, y_lengths, x, x_lengths)

        output = {
            "loss": loss,  # required
            "progress_bar": {"l_mle": l_mle, "l_length": l_length, "logdet": logdet},
            "log": {"loss": loss, "l_mle": l_mle, "l_length": l_length, "logdet":␣
↪logdet},
        }

        return output
    ...
```

Neural Types in NeMo TTS

NeMo Models and Neural Modules come with Neural Type checking. Neural type checking is extremely useful when combining many different neural network architectures for a production-grade application.

```
@typecheck(
    input_types={
        "x": NeuralType(('B', 'T'), TokenIndex()),
        "x_lengths": NeuralType(('B'), LengthsType()),
        "y": NeuralType(('B', 'D', 'T'), MelSpectrogramType(), optional=True),
        "y_lengths": NeuralType(('B'), LengthsType(), optional=True),
        "gen": NeuralType(optional=True),
        "noise_scale": NeuralType(optional=True),
        "length_scale": NeuralType(optional=True),
    }
)
def forward(self, *, x, x_lengths, y=None, y_lengths=None, gen=False, noise_scale=0.3,
↪ length_scale=1.0):
    ...
```

44.1.5 Learn More

- Watch the NVIDIA NeMo Intro Video
- Watch the PyTorch Lightning and NVIDIA NeMo Discussion Video
- Visit the NVIDIA NeMo Developer Website
- Read the NVIDIA NeMo PyTorch Blog
- Download pre-trained ASR, NLP, and TTS models on NVIDIA NGC to quickly get started with NeMo.
- Become an expert on Building Conversational AI applications with our tutorials, and example scripts,
- See our developer guide for more information on core NeMo concepts, ASR/NLP/TTS collections, and the NeMo API.

Note: NeMo tutorial notebooks can be run on Google Colab.

NVIDIA NeMo is actively being developed on GitHub. Contributions are welcome!

CHAPTER

FORTYFIVE

CONTRIBUTOR COVENANT CODE OF CONDUCT

45.1 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.

45.2 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

45.3 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.

45.4 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.

45.5 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.

45.6 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

FORTYSIX

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!

46.1 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.

46.2 Lightning Design Principles

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

46.2.1 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.

46.2.2 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 :)

46.2.3 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.

46.2.4 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.

46.2.5 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 backward compatible with good deprecation warnings.

You shouldn't be afraid to upgrade Lightning :)

46.2.6 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.

46.2.7 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.

46.3 Contribution Types

We are always looking for help implementing new features or fixing bugs.

A lot of good work has already been done in project mechanics (requirements.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)!

46.3.1 Bug Fixes:

1. If you find a bug please submit a github issue.
    - Make sure the title explains the issue.
    - Describe your setup, what you are trying to do, expected vs. actual behaviour. Please add configs and code samples.
    - Add details on how to reproduce the issue - a minimal test case is always best, colab is also great. Note, that the sample code shall be minimal and if needed with publicly available data.
2. Try to fix it or recommend a solution. We highly recommend to use test-driven approach:
    - Convert your minimal code example to a unit/integration test with assert on expected results.
    - Start by debugging the issue... You can run just this particular test in your IDE and draft a fix.
    - Verify that your test case fails on the master branch and only passes with the fix applied.
3. Submit a PR!

***Note**, even if you do not find the solution, sending a PR with a test covering the issue is a valid contribution and we can help you or finish it with you :]*

46.3.2 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! We recommend test driven approach to adding new features as well:
    - Write a test for the functionality you want to add.
    - Write the functional code until the test passes.
4. Add/update the relevant tests!

- This PR is a good example for adding a new metric, and this one for a new logger.

46.3.3 Test cases:

Want to keep Lightning healthy? Love seeing those green tests? So do we! How to we keep it that way? We write tests! We value tests contribution even more than new features.

Most of the tests in PyTorch Lightning train a trial MNIST model under various trainer conditions (ddp, ddp2+amp, etc...). The tests expect the model to perform to a reasonable degree of testing accuracy to pass. Want to add a new test case and not sure how? Talk to us!

46.4 Guidelines

46.4.1 Developments scripts

To build the documentation locally, simply execute the following commands from project root (only for Unix):

- `make clean` cleans repo from temp/generated files
- `make docs` builds documentation under *docs/build/html*
- `make test` runs all project's tests with coverage

46.4.2 Original code

All added or edited code shall be the own original work of the particular contributor. If you use some third-party implementation, all such blocks/functions/modules shall be properly referred and if possible also agreed by code's author. For example - `This code is inspired from http://...`. In case you adding new dependencies, make sure that they are compatible with the actual PyTorch Lightning license (ie. dependencies should be *at least* as permissive as the PyTorch Lightning license).

46.4.3 Coding Style

1. Use f-strings for output formation (except logging when we stay with lazy `logging.info("Hello %s!", name)`.
2. You can use `pre-commit` to make sure your code style is correct.

46.4.4 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

```
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
```

(continues on next page)

(continued from previous page)

```
    .. 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.

46.4.5 Testing

Local: Testing your work locally will help you speed up the process since it allows you to focus on particular (failing) test-cases. To setup a local development environment, install both local and test dependencies:

```
python -m pip install ".[dev, examples]"
python -m pip install pre-commit
```

You can run the full test-case in your terminal via this make script:

```
make test
```

Note: if your computer does not have multi-GPU nor TPU these tests are skipped.

GitHub Actions: For convenience, you can also use your own GHActions building which will be triggered with each commit. This is useful if you do not test against all required dependency versions.

Docker: Another option is utilize the pytorch lightning cuda base docker image. You can then run:

```
python -m pytest pytorch_lightning tests pl_examples -v
```

You can also run a single test as follows:

```
python -m pytest -v tests/trainer/test_trainer_cli.py::test_default_args
```

46.4.6 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. If you want to implement a new feature or enhance an existing one, start by opening a GitHub issue to explain the feature and the motivation. Members from core-contributors will take a look (it might take some time - we are often overloaded with issues!) and discuss it. Once an agreement was reached - start coding.
2. Start your work locally (usually until you need our CI testing).
    - Create a branch and prepare your changes.
    - Tip: do not work with your master directly, it may become complicated when you need to rebase.
    - Tip: give your PR a good name! It will be useful later when you may work on multiple tasks/PRs.
3. Test your code!
    - It is always good practice to start coding by creating a test case, verifying it breaks with current behaviour, and passes with your new changes.
    - Make sure your new tests cover all different edge cases.
    - Make sure all exceptions are handled.
4. Create a "Draft PR" which is clearly marked, to let us know you don't need feedback yet.
5. When you feel ready for integrating your work, mark your PR "Ready for review".
    - Your code should be readable and follow the project's design principles.
    - Make sure all tests are passing.
    - Make sure you add a GitHub issue to your PR.
6. 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.

46.4.7 Question & Answer

How can I help/contribute?

All types of contributions are welcome - reporting bugs, fixing documentation, adding test cases, solving issues, and preparing bug fixes. To get started with code contributions, look for issues marked with the label good first issue or chose something close to your domain with the label help wanted. Before coding, make sure that the issue description is clear and comment on the issue so that we can assign it to you (or simply self-assign if you can).

Is there a recommendation for branch names?

We recommend you follow this convention `<type>/<issue-id>_<short-name>` where the types are: `bugfix`, `feature`, `docs`, or `tests` (but if you are using your own fork that's optional).

How to rebase my PR?

We recommend creating a PR in a separate branch other than `master`, especially if you plan to submit several changes and do not want to wait until the first one is resolved (we can work on them in parallel).

First, make sure you have set upstream by running:

```
git remote add upstream https://github.com/PyTorchLightning/pytorch-lightning.git
```

You'll know its set up right if you run `git remote -v` and see something similar to this:

```
origin  https://github.com/{YOUR_USERNAME}/pytorch-lightning.git (fetch)
origin  https://github.com/{YOUR_USERNAME}/pytorch-lightning.git (push)
upstream        https://github.com/PyTorchLightning/pytorch-lightning.git (fetch)
upstream        https://github.com/PyTorchLightning/pytorch-lightning.git (push)
```

Checkout your feature branch and rebase it with upstream's master before pushing up your feature branch:

```
git fetch --all --prune
git rebase upstream/master
# follow git instructions to resolve conflicts
git push -f
```

How to add new tests?**

We are using pytest in Pytorch Lightning.

Here are tutorials:

- (recommended) Visual Testing with pytest from JetBrains on YouTube
- Effective Python Testing With Pytest article on realpython.com

Here is the process to create a new test

- 1. Optional: Follow tutorials !
- 1. Find a file in tests/ which match what you want to test. If none, create one.
- 1. Use this template to get started !
- 1. Use `BoringModel and derivates to test out your code.`

```
# TEST SHOULD BE IN YOUR FILE: tests/..../...py
# TEST CODE TEMPLATE

# [OPTIONAL] pytest decorator
# @pytest.mark.skipif(not torch.cuda.is_available(), reason="test requires GPU machine
↪")
def test_explain_what_is_being_tested(tmpdir):
    """
    Test description about text reason to be
    """
```

(continues on next page)

(continued from previous page)

```
    # os.environ["PL_DEV_DEBUG"] = '1' # [OPTIONAL] When activated, you can use␣
↪internal trainer.dev_debugger

    class ExtendedModel(BoringModel):
        ...

    model = ExtendedModel()

    # BoringModel is a functional model. You might want to set methods to None to␣
↪test your behaviour
    # Example: model.training_step_end = None

    trainer = Trainer(
        default_root_dir=tmpdir, # will save everything within a tmpdir generated for␣
↪this test
        ...
    )
    trainer.fit(model)
    trainer.test()  # [OPTIONAL]

    # assert the behaviour is correct.
    assert ...
```

run our/your test with

```
python -m pytest tests/..../...py::test_explain_what_is_being_tested --verbose --
↪capture=no
```

How to contribute bugfixes/features?

Currently we have separate streams/branches for bugfixes/features and release from the default branch (`master`). Bugfixes should land in this `master` branch and features should land in `release/X.y-dev`. This means that when starting your contribution and creating a branch according to question 2) you should start this new branch from master or future release dev branch. Later in PR creation also pay attention to properly set the target branch, usually the starting (base) and target branch are the same.

Note, that this flow may change after the 1.2 release as we will adjust releasing strategy.

How to fix PR with mixed base and target branches?

Sometimes you start your PR as a bug-fix but it turns out to be more of a feature (or the other way around). Do not panic, the solution is very straightforward and quite simple. All you need to do are these two steps in arbitrary order:

- Ask someone from Core to change the base/target branch to the correct one
- Rebase or cherry-pick your commits onto the correct base branch...

Let's show how to deal with the git... the sample case is moving a PR from `master` to `release/1.2-dev` assuming my branch name is `my-branch` and the last true master commit is `ccc111` and your first commit is `mmm222`.

- **Cherry-picking** way

```
git checkout my-branch
# create a local backup of your branch
git checkout -b my-branch-backup
# reset your branch to the correct base
git reset release/1.2-dev --hard
# ACTION: this step is much easier to do with IDE
#  so open one and cherry-pick your last commits from `my-branch-backup`
#  resolve all eventual conflict as the new base may contain different code
# when all done, push back to the open PR
git push -f
```

- **Rebasing way**, see more about rebase onto usage

```
git checkout my-branch
# rebase your commits on the correct branch
git rebase --onto release/1.2-dev ccc111
# if there is no collision you shall see just success
#  eventually you would need to resolve collision and in such case follow the␣
↪instruction in terminal
# when all done, push back to the open PR
git push -f
```

46.4.8 Bonus Workflow Tip

If you don't want to remember all the commands above every time you want to push some code/setup a Lightning Dev environment on a new VM, you can set up bash aliases for some common commands. You can add these to one of your `~/.bashrc`, `~/.zshrc`, or `~/.bash_aliases` files.

NOTE: Once you edit one of these files, remember to `source` it or restart your shell. (ex. `source ~/.bashrc` if you added these to your `~/.bashrc` file).

```
plclone (){
    git clone https://github.com/{YOUR_USERNAME}/pytorch-lightning.git
    cd pytorch-lightning
    git remote add upstream https://github.com/PyTorchLightning/pytorch-lightning.git
    # This is just here to print out info about your remote upstream/origin
    git remote -v
}

plfetch (){
    git fetch --all --prune
    git checkout master
    git merge upstream/master
}

# Rebase your branch with upstream's master
# plrebase <your-branch-name>
plrebase (){
    git checkout $@
    git rebase master
}
```

Now, you can:

- clone your fork and set up upstream by running `plclone` from your terminal
- fetch upstream and update your local master branch with it by running `plfetch`

- rebase your feature branch (after running `plfetch`) by running `plrebase your-branch-name`

CHAPTER

FORTYSEVEN

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.

47.1 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.

47.2 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 contribute! Scientists with great usability intuition and PyTorch ninja skills are welcomed!

47.3 Responsibilities:

The responsibilities mainly revolve around 3 things.

47.3.1 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.

47.3.2 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 :)

47.3.3 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.

47.3.4 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.

47.3.5 Summary: Requirements to apply

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.

- Solve 10+ Github issues.
- Create 5+ meaningful PRs which solves some reported issue - bug,
- Perform 10+ PR reviews from other contributors.

If you want to be considered, ping me on Slack.

CHAPTER

FORTYEIGHT

PYTORCH LIGHTNING GOVERNANCE | PERSONS OF INTEREST

48.1 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)

48.2 Core Maintainers

- Nic Eggert (neggert)
- Jeff Ling (jeffling)
- Jeremy Jordan (jeremyjordan)
- Tullie Murrell (tullie)
- Adrian Wälchli (awaelchli)
- Nicki Skafte (skaftenicki)
- Peter Yu (yukw777)
- Rohit Gupta (rohitgr7)
- Lezwon Castelino (lezwon)
- Jeff Yang (ydcjeff)
- Roger Shieh (s-rog)
- Carlos Mocholí (carmocca)
- Ananth Subramaniam (ananthsub)
- Thomas Chaton (tchaton)
- Sean Narenthiran (SeanNaren)

CHAPTER

FORTYNINE

CHANGELOG

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog.

49.1 [1.2.1] - 2021-02-23

49.1.1 [1.2.1] - Fixed

- Fixed incorrect yield logic for the amp autocast context manager (#6080)
- Fixed priority of plugin/accelerator when setting distributed mode (#6089)
- Fixed error message for AMP + CPU incompatibility (#6107)

49.2 [1.2.0] - 2021-02-18

49.2.1 [1.2.0] - Added

- Added `DataType`, `AverageMethod` and `MDMCAverageMethod` enum in metrics (#5657)
- Added support for summarized model total params size in megabytes (#5590)
- Added support for multiple train loaders (#1959)
- Added `Accuracy` metric now generalizes to Top-k accuracy for (multi-dimensional) multi-class inputs using the `top_k` parameter (#4838)
- Added `Accuracy` metric now enables the computation of subset accuracy for multi-label or multi-dimensional multi-class inputs with the `subset_accuracy` parameter (#4838)
- Added `HammingDistance` metric to compute the hamming distance (loss) (#4838)
- Added `max_fpr` parameter to `auroc` metric for computing partial auroc metric (#3790)
- Added `StatScores` metric to compute the number of true positives, false positives, true negatives and false negatives (#4839)
- Added `R2Score` metric (#5241)
- Added `LambdaCallback` (#5347)
- Added `BackboneLambdaFinetuningCallback` (#5377)
- Accelerator `all_gather` supports collection (#5221)

- Added `image_gradients` functional metric to compute the image gradients of a given input image. (#5056)
- Added `MetricCollection` (#4318)
- Added `.clone()` method to metrics (#4318)
- Added `IoU` class interface (#4704)
- Support to tie weights after moving model to TPU via `on_post_move_to_device` hook
- Added missing val/test hooks in `LightningModule` (#5467)
- The `Recall` and `Precision` metrics (and their functional counterparts `recall` and `precision`) can now be generalized to Recall@K and Precision@K with the use of `top_k` parameter (#4842)
- Added `ModelPruning` Callback (#5618, #5825, #6045)
- Added `PyTorchProfiler` (#5560)
- Added compositional metrics (#5464)
- Added Trainer method `predict(...)` for high performence predictions (#5579)
- Added `on_before_batch_transfer` and `on_after_batch_transfer` data hooks (#3671)
- Added AUC/AUROC class interface (#5479)
- Added `PredictLoop` object (#5752)
- Added `QuantizationAwareTraining` callback (#5706, #6040)
- Added `LightningModule.configure_callbacks` to enable the definition of model-specific callbacks (#5621)
- Added `dim` to `PSNR` metric for mean-squared-error reduction (#5957)
- Added promxial policy optimization template to pl_examples (#5394)
- Added `log_graph` to `CometLogger` (#5295)
- Added possibility for nested loaders (#5404)
- Added `sync_step` to Wandb logger (#5351)
- Added `StochasticWeightAveraging` callback (#5640)
- Added `LightningDataModule.from_datasets(...)` (#5133)
- Added `PL_TORCH_DISTRIBUTED_BACKEND` env variable to select backend (#5981)
- Added `Trainer` flag to activate Stochastic Weight Averaging (SWA) `Trainer(stochastic_weight_avg=True)` (#6038)
- Added DeepSpeed integration (#5954, #6042)

49.2.2 [1.2.0] - Changed

- Changed `stat_scores` metric now calculates stat scores over all classes and gains new parameters, in line with the new `StatScores` metric (#4839)
- Changed `computer_vision_fine_tunning` example to use `BackboneLambdaFinetuningCallback` (#5377)
- Changed `automatic casting` for LoggerConnector `metrics` (#5218)
- Changed `iou` [func] to allow float input (#4704)
- Metric `compute()` method will no longer automatically call `reset()` (#5409)

- Set PyTorch 1.4 as min requirements, also for testing and examples `torchvision>=0.5` and `torchtext>=0.5` (#5418)
- Changed `callbacks` argument in `Trainer` to allow `Callback` input (#5446)
- Changed the default of `find_unused_parameters` to `False` in DDP (#5185)
- Changed `ModelCheckpoint` version suffixes to start at 1 (#5008)
- Progress bar metrics tensors are now converted to float (#5692)
- Changed the default value for the `progress_bar_refresh_rate` Trainer argument in Google COLAB notebooks to 20 (#5516)
- Extended support for purely iteration-based training (#5726)
- Made `LightningModule.global_rank`, `LightningModule.local_rank` and `LightningModule.logger` read-only properties (#5730)
- Forced `ModelCheckpoint` callbacks to run after all others to guarantee all states are saved to the checkpoint (#5731)
- Refactored Accelerators and Plugins (#5743)
  - Added base classes for plugins (#5715)
  - Added parallel plugins for DP, DDP, DDPSpawn, DDP2 and Horovod (#5714)
  - Precision Plugins (#5718)
  - Added new Accelerators for CPU, GPU and TPU (#5719)
  - Added Plugins for TPU training (#5719)
  - Added RPC and Sharded plugins (#5732)
  - Added missing `LightningModule`-wrapper logic to new plugins and accelerator (#5734)
  - Moved device-specific teardown logic from training loop to accelerator (#5973)
  - Moved accelerator_connector.py to the connectors subfolder (#6033)
  - Trainer only references accelerator (#6039)
  - Made parallel devices optional across all plugins (#6051)
  - Cleaning (#5948, #5949, #5950)
- Enabled `self.log` in callbacks (#5094)
- Renamed xxx_AVAILABLE as protected (#5082)
- Unified module names in Utils (#5199)
- Separated utils: imports & enums (#5256 #5874)
- Refactor: clean trainer device & distributed getters (#5300)
- Simplified training phase as LightningEnum (#5419)
- Updated metrics to use LightningEnum (#5689)
- Changed the seq of `on_train_batch_end`, `on_batch_end` & `on_train_epoch_end`, `on_epoch_end hooks` (#5688)
- Refactored `setup_training` and remove `test_mode` (#5388)
- Disabled training with zero `num_training_batches` when insufficient `limit_train_batches` (#5703)

- Refactored `EpochResultStore` (#5522)
- Update `lr_finder` to check for attribute if not running `fast_dev_run` (#5990)
- LightningOptimizer manual optimizer is more flexible and expose `toggle_model` (#5771)
- `MlflowLogger` limit parameter value length to 250 char (#5893)
- Re-introduced fix for Hydra directory sync with multiple process (#5993)

49.2.3 [1.2.0] - Deprecated

- Function `stat_scores_multiple_classes` is deprecated in favor of `stat_scores` (#4839)
- Moved accelerators and plugins to its `legacy` pkg (#5645)
- Deprecated `LightningDistributedDataParallel` in favor of new wrapper module `LightningDistributedModule` (#5185)
- Deprecated `LightningDataParallel` in favor of new wrapper module `LightningParallelModule` (#5670)
- Renamed utils modules (#5199)
  - `argparse_utils` >> `argparse`
  - `model_utils` >> `model_helpers`
  - `warning_utils` >> `warnings`
  - `xla_device_utils` >> `xla_device`
- Deprecated using `'val_loss'` to set the `ModelCheckpoint` monitor (#6012)
- Deprecated `.get_model()` with explicit `.lightning_module` property (#6035)
- Deprecated Trainer attribute `accelerator_backend` in favor of `accelerator` (#6034)

49.2.4 [1.2.0] - Removed

- Removed deprecated checkpoint argument `filepath` (#5321)
- Removed deprecated `Fbeta`, `f1_score` and `fbeta_score` metrics (#5322)
- Removed deprecated `TrainResult` (#5323)
- Removed deprecated `EvalResult` (#5633)
- Removed `LoggerStages` (#5673)

49.2.5 [1.2.0] - Fixed

- Fixed distributed setting and `ddp_cpu` only with `num_processes>1` (#5297)
- Fixed the saved filename in `ModelCheckpoint` when it already exists (#4861)
- Fixed `DDPHPCAccelerator` hangs in DDP construction by calling `init_device` (#5157)
- Fixed `num_workers` for Windows example (#5375)
- Fixed loading yaml (#5619)
- Fixed support custom DataLoader with DDP if they can be re-instantiated (#5745)

- Fixed repeated `.fit()` calls ignore max_steps iteration bound (#5936)
- Fixed throwing `MisconfigurationError` on unknown mode (#5255)
- Resolve bug with Finetuning (#5744)
- Fixed `ModelCheckpoint` race condition in file existence check (#5155)
- Fixed some compatibility with PyTorch 1.8 (#5864)
- Fixed forward cache (#5895)
- Fixed recursive detach of tensors to CPU (#6007)
- Fixed passing wrong strings for scheduler interval doesn't throw an error (#5923)
- Fixed wrong `requires_grad` state after `return None` with multiple optimizers (#5738)
- Fixed add `on_epoch_end` hook at the end of `validation`, `test` epoch (#5986)
- Fixed missing `process_dataloader` call for `TPUSpawn` when in distributed mode (#6015)
- Fixed progress bar flickering by appending 0 to floats/strings (#6009)
- Fixed synchronization issues with TPU training (#6027)
- Fixed `hparams.yaml` saved twice when using `TensorBoardLogger` (#5953)
- Fixed basic examples (#5912, #5985)
- Fixed `fairscale` compatible with PT 1.8 (#5996)
- Ensured `process_dataloader` is called when `tpu_cores > 1` to use Parallel DataLoader (#6015)
- Attempted SLURM auto resume call when non-shell call fails (#6002)
- Fixed wrapping optimizers upon assignment (#6006)
- Fixed allowing hashing of metrics with lists in their state (#5939)

49.3 [1.1.8] - 2021-02-08

49.3.1 [1.1.8] - Fixed

- Separate epoch validation from step validation (#5208)
- Fixed `toggle_optimizers` not handling all optimizer parameters (#5775)

49.4 [1.1.7] - 2021-02-03

49.4.1 [1.1.7] - Fixed

- Fixed `TensorBoardLogger` not closing `SummaryWriter` on `finalize` (#5696)
- Fixed filtering of pytorch "unsqueeze" warning when using DP (#5622)
- Fixed `num_classes` argument in F1 metric (#5663)
- Fixed `log_dir` property (#5537)
- Fixed a race condition in `ModelCheckpoint` when checking if a checkpoint file exists (#5144)

- Remove unnecessary intermediate layers in Dockerfiles (#5697)
- Fixed auto learning rate ordering (#5638)

49.5 [1.1.6] - 2021-01-26

49.5.1 [1.1.6] - Changed

- Increased TPU check timeout from 20s to 100s (#5598)
- Ignored `step` param in Neptune logger's log_metric method (#5510)
- Pass batch outputs to `on_train_batch_end` instead of `epoch_end` outputs (#4369)

49.5.2 [1.1.6] - Fixed

- Fixed `toggle_optimizer` to reset `requires_grad` state (#5574)
- Fixed FileNotFoundError for best checkpoint when using DDP with Hydra (#5629)
- Fixed an error when logging a progress bar metric with a reserved name (#5620)
- Fixed `Metric`'s `state_dict` not included when child modules (#5614)
- Fixed Neptune logger creating multiple experiments when GPUs > 1 (#3256)
- Fixed duplicate logs appearing in console when using the python logging module (#5509)
- Fixed tensor printing in `trainer.test()` (#5138)
- Fixed not using dataloader when `hparams` present (#4559)

49.6 [1.1.5] - 2021-01-19

49.6.1 [1.1.5] - Fixed

- Fixed a visual bug in the progress bar display initialization (#4579)
- Fixed logging `on_train_batch_end` in a callback with multiple optimizers (#5521)
- Fixed `reinit_scheduler_properties` with correct optimizer (#5519)
- Fixed `val_check_interval` with `fast_dev_run` (#5540)

49.7 [1.1.4] - 2021-01-12

49.7.1 [1.1.4] - Added

- Add automatic optimization property setter to lightning module (#5169)

49.7.2 [1.1.4] - Changed

- Changed deprecated `enable_pl_optimizer=True` (#5244)

49.7.3 [1.1.4] - Fixed

- Fixed `transfer_batch_to_device` for DDP with `len(devices_ids) == 1` (#5195)
- Logging only on `not should_accumulate()` during training (#5417)
- Resolve interpolation bug with Hydra (#5406)
- Check environ before selecting a seed to prevent warning message (#4743)
- Fixed signature mismatch in `model_to_device` of `DDPCPUHPCAccelerator` (#5505)

49.8 [1.1.3] - 2021-01-05

49.8.1 [1.1.3] - Added

- Added a check for optimizer attached to `lr_scheduler` (#5338)
- Added support for passing non-existing filepaths to `resume_from_checkpoint` (#4402)

49.8.2 [1.1.3] - Changed

- Skip restore from `resume_from_checkpoint` while `testing` (#5161)
- Allowed `log_momentum` for adaptive optimizers in `LearningRateMonitor` (#5333)
- Disabled checkpointing, earlystopping and logging with `fast_dev_run` (#5277)
- Distributed group defaults to `WORLD` if `None` (#5125)

49.8.3 [1.1.3] - Fixed

- Fixed `trainer.test` returning non-test metrics (#5214)
- Fixed metric state reset (#5273)
- Fixed `--num-nodes` on `DDPSequentialPlugin` (#5327)
- Fixed invalid value for `weights_summary` (#5296)
- Fixed `Trainer.test` not using the latest `best_model_path` (#5161)
- Fixed existence check for hparams not using underlying filesystem (#5250)
- Fixed `LightningOptimizer` AMP bug (#5191)
- Fixed casted key to string in `_flatten_dict` (#5354)

49.9 [1.1.2] - 2020-12-23

49.9.1 [1.1.2] - Added

- Support number for logging with `sync_dist=True` (#5080)
- Added offset logging step when resuming for Wandb logger (#5050)

49.9.2 [1.1.2] - Removed

- `enable_pl_optimizer=False` by default to temporarily fix AMP issues (#5163)

49.9.3 [1.1.2] - Fixed

- Metric reduction with Logging (#5150)
- Remove nan loss in manual optimization (#5121)
- Un-balanced logging properly supported (#5119)
- Fix hanging in DDP HPC accelerators (#5157)
- Fix saved filename in `ModelCheckpoint` if it already exists (#4861)
- Fix reset `TensorRunningAccum` (#5106)
- Updated `DALIClassificationLoader` to not use deprecated arguments (#4925)
- Corrected call to `torch.no_grad` (#5124)

49.10 [1.1.1] - 2020-12-15

49.10.1 [1.1.1] - Added

- Add a notebook example to reach a quick baseline of ~94% accuracy on CIFAR10 using Resnet in Lightning (#4818)

49.10.2 [1.1.1] - Changed

- Simplify accelerator steps (#5015)
- Refactor load in checkpoint connector (#4593)
- Fixed the saved filename in `ModelCheckpoint` when it already exists (#4861)

49.10.3 [1.1.1] - Removed

- Drop duplicate metrics (#5014)
- Remove beta arg from F1 class and functional (#5076)

49.10.4 [1.1.1] - Fixed

- Fixed trainer by default `None` in `DDPAccelerator` (#4915)
- Fixed `LightningOptimizer` to expose optimizer attributes (#5095)
- Do not warn when the `name` key is used in the `lr_scheduler` dict (#5057)
- Check if optimizer supports closure (#4981)
- Extend LightningOptimizer to exposure underlying Optimizer attributes + update doc (#5095)
- Add deprecated metric utility functions back to functional (#5067, #5068)
- Allow any input in `to_onnx` and `to_torchscript` (#4378)
- Do not warn when the name key is used in the `lr_scheduler` dict (#5057)
- Fixed `DDPHPCAccelerator` hangs in DDP construction by calling `init_device` (#5157)

49.11 [1.1.0] - 2020-12-09

49.11.1 [1.1.0] - Added

- Added "monitor" key to saved `ModelCheckpoints` (#4383)
- Added `ConfusionMatrix` class interface (#4348)
- Added multiclass AUROC metric (#4236)
- Added global step indexing to the checkpoint name for a better sub-epoch checkpointing experience (#3807)
- Added optimizer hooks in callbacks (#4379)
- Added option to log momentum (#4384)
- Added `current_score` to `ModelCheckpoint.on_save_checkpoint` (#4721)
- Added logging using `self.log` in train and evaluation for epoch end hooks (#4552, #4495, #4439, #4684, #4913)
- Added ability for DDP plugin to modify optimizer state saving (#4675)
- Added casting to python types for numpy scalars when logging hparams (#4647)
- Added `prefix` argument in loggers (#4557)
- Added printing of total num of params, trainable and non-trainable params in ModelSummary (#4521)
- Added `PrecisionRecallCurve, ROC, AveragePrecision` class metric (#4549)
- Added custom `Apex` and `NativeAMP` as `Precision plugins` (#4355)
- Added `DALI MNIST` example (#3721)
- Added `sharded plugin` for DDP for multi-gpu training memory optimizations (#4639, #4686, #4675, #4737, #4773)

- Added `experiment_id` to the NeptuneLogger (#3462)
- Added `Pytorch Geometric` integration example with Lightning (#4568)
- Added `all_gather` method to `LightningModule` which allows gradient based tensor synchronizations for use-cases such as negative sampling. (#5012)
- Enabled `self.log` in most functions (#4969)
- Added changeable extension variable for `ModelCheckpoint` (#4977)

49.11.2 [1.1.0] - Changed

- Tuner algorithms will be skipped if `fast_dev_run=True` (#3903)
- `WandbLogger` does not force wandb `reinit` arg to True anymore and creates a run only when needed (#4648)
- Changed `automatic_optimization` to be a model attribute (#4602)
- Changed `Simple Profiler` report to order by percentage time spent + num calls (#4880)
- Simplify optimization Logic (#4984)
- Classification metrics overhaul (#4837)
- Updated `fast_dev_run` to accept integer representing num_batches (#4629)
- Refactored optimizer (#4658)

49.11.3 [1.1.0] - Deprecated

- Deprecated `prefix` argument in `ModelCheckpoint` (#4765)
- Deprecated the old way of assigning hyper-parameters through `self.hparams = ...` (#4813)
- Deprecated `mode='auto'` from `ModelCheckpoint` and `EarlyStopping` (#4695)

49.11.4 [1.1.0] - Removed

- Removed `reorder` parameter of the `auc` metric (#5004)
- Removed `multiclass_roc` and `multiclass_precision_recall_curve`, use `roc` and `precision_recall_curve` instead (#4549)

49.11.5 [1.1.0] - Fixed

- Added feature to move tensors to CPU before saving (#4309)
- Fixed `LoggerConnector` to have logged metrics on root device in DP (#4138)
- Auto convert tensors to contiguous format when `gather_all` (#4907)
- Fixed `PYTHONPATH` for ddp test model (#4528)
- Fixed allowing logger to support indexing (#4595)
- Fixed DDP and manual_optimization (#4976)

49.12 [1.0.8] - 2020-11-24

49.12.1 [1.0.8] - Added

- Added casting to python types for numpy scalars when logging `hparams` (#4647)
- Added warning when progress bar refresh rate is less than 20 on Google Colab to prevent crashing (#4654)
- Added `F1` class metric (#4656)

49.12.2 [1.0.8] - Changed

- Consistently use `step=trainer.global_step` in `LearningRateMonitor` independently of `logging_interval` (#4376)
- Metric states are no longer as default added to `state_dict` (#4685)
- Renamed class metric `Fbeta` >> `FBeta` (#4656)
- Model summary: add 1 decimal place (#4745)
- Do not override `PYTHONWARNINGS` (#4700)
- Changed `init_ddp_connection` moved from `DDP` to `DDPPlugin` (#4407)

49.12.3 [1.0.8] - Fixed

- Fixed checkpoint `hparams` dict casting when `omegaconf` is available (#4770)
- Fixed incomplete progress bars when total batches not divisible by refresh rate (#4577)
- Updated SSIM metric (#4566)
- Fixed batch_arg_name - add `batch_arg_name` to all calls to `_adjust_batch_size`bug (#4812)
- Fixed `torchtext` data to GPU (#4785)
- Fixed a crash bug in MLFlow logger (#4716)

49.13 [1.0.7] - 2020-11-17

49.13.1 [1.0.7] - Added

- Added lambda closure to `manual_optimizer_step` (#4618)

49.13.2 [1.0.7] - Changed

- Change Metrics `persistent` default mode to `False` (#4685)
- LoggerConnector log_metrics will use `total_batch_idx` instead of `global_step` when logging on `training step` (#4738)

49.13.3 [1.0.7] - Fixed

- Prevent crash if `sync_dist=True` on CPU (#4626)
- Fixed average pbar Metrics (#4534)
- Fixed `setup` callback hook to correctly pass the LightningModule through (#4608)
- Allowing decorate model init with saving `hparams` inside (#4662)
- Fixed `split_idx` set by `LoggerConnector` in `on_trainer_init` to `Trainer` (#4697)

49.14 [1.0.6] - 2020-11-11

49.14.1 [1.0.6] - Added

- Added metrics aggregation in Horovod and fixed early stopping (#3775)
- Added `manual_optimizer_step` which work with `AMP Native` and `accumulated_grad_batches` (#4485)
- Added `persistent(mode)` method to metrics, to enable and disable metric states being added to `state_dict` (#4482)
- Added congratulations at the end of our notebooks (#4555)
- Added parameters `move_metrics_to_cpu` in Trainer to disable gpu leak (#4592)

49.14.2 [1.0.6] - Changed

- Changed `fsspec` to tuner (#4458)
- Unify SLURM/TorchElastic under backend plugin (#4578, #4580, #4581, #4582, #4583)

49.14.3 [1.0.6] - Fixed

- Fixed feature-lack in `hpc_load` (#4526)
- Fixed metrics states being overridden in DDP mode (#4482)
- Fixed `lightning_getattr`, `lightning_hasattr` not finding the correct attributes in datamodule (#4347)
- Fixed automatic optimization AMP by `manual_optimization_step` (#4485)
- Replace `MisconfigurationException` with warning in `ModelCheckpoint` Callback (#4560)
- Fixed logged keys in mlflow logger (#4412)
- Fixed `is_picklable` by catching `AttributeError` (#4508)

- Fixed multi test dataloaders dict `AttributeError` error (#4480)
- Fixed show progress bar only for `progress_rank 0` on `DDP_SLURM` (#4437)

49.15 [1.0.5] - 2020-11-03

49.15.1 [1.0.5] - Added

- Added PyTorch 1.7 Stable support (#3821)
- Added timeout for `tpu_device_exists` to ensure process does not hang indefinitely (#4340)

49.15.2 [1.0.5] - Changed

- W&B log in sync with `Trainer` step (#4405)
- Hook `on_after_backward` is called only when `optimizer_step` is being called (#4439)
- Moved `track_and_norm_grad` into `training loop` and called only when `optimizer_step` is being called (#4439)
- Changed type checker with explicit cast of `ref_model` object (#4457)
- Changed `distributed_backend` -> `accelerator` (#4429)

49.15.3 [1.0.5] - Deprecated

- Deprecated passing `ModelCheckpoint` instance to `checkpoint_callback` Trainer argument (#4336)

49.15.4 [1.0.5] - Fixed

- Disable saving checkpoints if not trained (#4372)
- Fixed error using `auto_select_gpus=True` with `gpus=-1` (#4209)
- Disabled training when `limit_train_batches=0` (#4371)
- Fixed that metrics do not store computational graph for all seen data (#4313)
- Fixed AMP unscale for `on_after_backward` (#4439)
- Fixed TorchScript export when module includes Metrics (#4428)
- Fixed TorchScript trace method's data to device and docstring (#4360)
- Fixed CSV logger warning (#4419)
- Fixed skip DDP parameter sync (#4301)
- Fixed `WandbLogger` _sanitize_callable function (#4422)
- Fixed `AMP Native` `_unscale` gradient (#4441)

49.16 [1.0.4] - 2020-10-27

49.16.1 [1.0.4] - Added

- Added `dirpath` and `filename` parameter in `ModelCheckpoint` (#4213)
- Added plugins docs and DDPPlugin to customize ddp across all accelerators (#4258)
- Added `strict` option to the scheduler dictionary (#3586)
- Added `fsspec` support for profilers (#4162)
- Added autogenerated helptext to `Trainer.add_argparse_args` (#4344)
- Added support for string values in `Trainer`'s `profiler` parameter (#3656)
- Added support for string values in `Trainer`'s `profiler` parameter (#3656)
- Added `optimizer_closure` to `optimizer.step` when supported (#4190)
- Added unification of regression metrics (#4166)
- Added checkpoint load from Bytes (#4314)

49.16.2 [1.0.4] - Changed

- Improved error messages for invalid `configure_optimizers` returns (#3587)
- Allow changing the logged step value in `validation_step` (#4130)
- Allow setting `replace_sampler_ddp=True` with a distributed sampler already added (#4273)
- Fixed santized parameters for `WandbLogger.log_hyperparams` (#4320)

49.16.3 [1.0.4] - Deprecated

- Deprecated `filepath` in `ModelCheckpoint` (#4213)
- Deprecated `reorder` parameter of the `auc` metric (#4237)
- Deprecated bool values in `Trainer`'s `profiler` parameter (#3656)

49.16.4 [1.0.4] - Fixed

- Fixed setting device ids in DDP (#4297)
- Fixed synchronization of best model path in `ddp_accelerator` (#4323)
- Fixed `WandbLogger` not uploading checkpoint artifacts at the end of training (#4341)
- Fixed `FBeta` computation (#4183)
- Fixed `accumulation across batches` has completed `before breaking training loop` (#4278)
- Fixed `ModelCheckpoint` don't increase current_epoch and global_step when not training (#4291)
- Fixed `COMET_EXPERIMENT_KEY` environment variable usage in comet logger (#4230)

49.17 [1.0.3] - 2020-10-20

49.17.1 [1.0.3] - Added

- Added persistent flag to `Metric.add_state` (#4195)

49.17.2 [1.0.3] - Changed

- Used `checkpoint_connector.hpc_save` in SLURM (#4217)
- Moved base req. to root (#4219)

49.17.3 [1.0.3] - Fixed

- Fixed `hparams` assign in init (#4189)
- Fixed overwrite check for model hooks (#4010)

49.18 [1.0.2] - 2020-10-15

49.18.1 [1.0.2] - Added

- Added trace functionality to the function `to_torchscript` (#4142)

49.18.2 [1.0.2] - Changed

- Called `on_load_checkpoint` before loading `state_dict` (#4057)

49.18.3 [1.0.2] - Removed

- Removed duplicate metric vs step log for train loop (#4173)

49.18.4 [1.0.2] - Fixed

- Fixed the `self.log` problem in `validation_step()` (#4169)
- Fixed `hparams` saving - save the state when `save_hyperparameters()` is called [in `__init__`] (#4163)
- Fixed runtime failure while exporting `hparams` to yaml (#4158)

49.19 [1.0.1] - 2020-10-14

49.19.1 [1.0.1] - Added

- Added getstate/setstate method for torch.save serialization (#4127)

49.20 [1.0.0] - 2020-10-13

49.20.1 [1.0.0] - Added

- Added Explained Variance Metric + metric fix (#4013)
- Added Metric <-> Lightning Module integration tests (#4008)
- Added parsing OS env vars in `Trainer` (#4022)
- Added classification metrics (#4043)
- Updated explained variance metric (#4024)
- Enabled plugins (#4041)
- Enabled custom clusters (#4048)
- Enabled passing in custom accelerators (#4050)
- Added `LightningModule.toggle_optimizer` (#4058)
- Added `LightningModule.manual_backward` (#4063)
- Added `output` argument to `*_batch_end` hooks (#3965, #3966)
- Added `output` argument to `*_epoch_end` hooks (#3967)

49.20.2 [1.0.0] - Changed

- Integrated metrics API with self.log (#3961)
- Decoupled Apex (#4052, #4054, #4055, #4056, #4058, #4060, #4061, #4062, #4063, #4064, #4065)
- Renamed all backends to `Accelerator` (#4066)
- Enabled manual returns (#4089)

49.20.3 [1.0.0] - Removed

- Removed support for EvalResult and TrainResult (#3968)
- Removed deprecated trainer flags: `overfit_pct`, `log_save_interval`, `row_log_interval` (#3969)
- Removed deprecated early_stop_callback (#3982)
- Removed deprecated model hooks (#3980)
- Removed deprecated callbacks (#3979)
- Removed `trainer` argument in `LightningModule.backward` #4056)

49.20.4 [1.0.0] - Fixed

- Fixed `current_epoch` property update to reflect true epoch number inside `LightningDataModule`, when `reload_dataloaders_every_epoch=True`. (#3974)
- Fixed to print scaler value in progress bar (#4053)
- Fixed mismatch between docstring and code regarding when `on_load_checkpoint` hook is called (#3996)

49.21 [0.10.0] - 2020-10-07

49.21.1 [0.10.0] - Added

- Added new Metrics API. (#3868, #3921)
- Enable PyTorch 1.7 compatibility (#3541)
- Added `LightningModule.to_torchscript` to support exporting as `ScriptModule` (#3258)
- Added warning when dropping unpicklable `hparams` (#2874)
- Added EMB similarity (#3349)
- Added `ModelCheckpoint.to_yaml` method (#3048)
- Allow `ModelCheckpoint` monitor to be `None`, meaning it will always save (#3630)
- Disabled optimizers setup during testing (#3059)
- Added support for datamodules to save and load checkpoints when training (#3563)
- Added support for datamodule in learning rate finder (#3425)
- Added gradient clip test for native AMP (#3754)
- Added dist lib to enable syncing anything across devices (#3762)
- Added `broadcast` to `TPUBackend` (#3814)
- Added `XLADeviceUtils` class to check XLA device type (#3274)

49.21.2 [0.10.0] - Changed

- Refactored accelerator backends:
  - moved TPU `xxx_step` to backend (#3118)
  - refactored DDP backend `forward` (#3119)
  - refactored GPU backend `__step` (#3120)
  - refactored Horovod backend (#3121, #3122)
  - remove obscure forward call in eval + CPU backend `___step` (#3123)
  - reduced all simplified forward (#3126)
  - added hook base method (#3127)
  - refactor eval loop to use hooks - use `test_mode` for if so we can split later (#3129)
  - moved `___step_end` hooks (#3130)
  - training forward refactor (#3134)

- training AMP scaling refactor (#3135)
- eval step scaling factor (#3136)
- add eval loop object to streamline eval loop (#3138)
- refactored dataloader process hook (#3139)
- refactored inner eval loop (#3141)
- final inner eval loop hooks (#3154)
- clean up hooks in `run_evaluation` (#3156)
- clean up data reset (#3161)
- expand eval loop out (#3165)
- moved hooks around in eval loop (#3195)
- remove `_evaluate` fx (#3197)
- `Trainer.fit` hook clean up (#3198)
- DDPs train hooks (#3203)
- refactor DDP backend (#3204, #3207, #3208, #3209, #3210)
- reduced accelerator selection (#3211)
- group prepare data hook (#3212)
- added data connector (#3285)
- modular is_overridden (#3290)
- adding `Trainer.tune()` (#3293)
- move `run_pretrain_routine` -> `setup_training` (#3294)
- move train outside of setup training (#3297)
- move `prepare_data` to data connector (#3307)
- moved accelerator router (#3309)
- train loop refactor - moving train loop to own object (#3310, #3312, #3313, #3314)
- duplicate data interface definition up into DataHooks class (#3344)
- inner train loop (#3359, #3361, #3362, #3363, #3365, #3366, #3367, #3368, #3369, #3370, #3371, #3372, #3373, #3374, #3375, #3376, #3385, #3388, #3397)
- all logging related calls in a connector (#3395)
- device parser (#3400, #3405)
- added model connector (#3407)
- moved eval loop logging to loggers (#3408)
- moved eval loop (#3412#3408)
- trainer/separate argparse (#3421, #3428, #3432)
- move `lr_finder` (#3434)
- organize args (##3435, #3442, #3447, #3448, #3449, #3456)
- move specific accelerator code (#3457)

  - group connectors (#3472)
  - accelerator connector methods x/n (#3469, #3470, #3474)
  - merge backends x/n (#3476, #3477, #3478, #3480, #3482)
  - apex plugin (#3502)
  - precision plugins (#3504)
  - Result - make monitor default to `checkpoint_on` to simplify (#3571)
  - reference to the Trainer on the `LightningDataModule` (#3684)
  - add `.log` to lightning module (#3686, #3699, #3701, #3704, #3715)
  - enable tracking original metric when step and epoch are both true (#3685)
  - deprecated results obj, added support for simpler comms (#3681)
  - move backends back to individual files (#3712)
  - fixes logging for eval steps (#3763)
  - decoupled DDP, DDP spawn (#3733, #3766, #3767, #3774, #3802, #3806)
  - remove weight loading hack for ddp_cpu (#3808)
  - separate `torchelastic` from DDP (#3810)
  - separate SLURM from DDP (#3809)
  - decoupled DDP2 (#3816)
  - bug fix with logging val epoch end + monitor (#3812)
  - decoupled DDP, DDP spawn (#3733, #3817, #3819, #3927)
  - callback system and init DDP (#3836)
  - adding compute environments (#3837, #3842)
  - epoch can now log independently (#3843)
  - test selecting the correct backend. temp backends while slurm and TorchElastic are decoupled (#3848)
  - fixed `init_slurm_connection` causing hostname errors (#3856)
  - moves init apex from LM to apex connector (#3923)
  - moves sync bn to each backend (#3925)
  - moves configure ddp to each backend (#3924)
- Deprecation warning (#3844)
- Changed `LearningRateLogger` to `LearningRateMonitor` (#3251)
- Used `fsspec` instead of `gfile` for all IO (#3320)
  - Swaped `torch.load` for `fsspec` load in DDP spawn backend (#3787)
  - Swaped `torch.load` for `fsspec` load in cloud_io loading (#3692)
  - Added support for `to_disk()` to use remote filepaths with `fsspec` (#3930)
  - Updated model_checkpoint's to_yaml to use `fsspec` open (#3801)
  - Fixed `fsspec` is inconsistant when doing `fs.ls` (#3805)
- Refactor `GPUStatsMonitor` to improve training speed (#3257)

- Changed IoU score behavior for classes absent in target and pred (#3098)
- Changed IoU `remove_bg` bool to `ignore_index` optional int (#3098)
- Changed defaults of `save_top_k` and `save_last` to `None` in ModelCheckpoint (#3680)
- `row_log_interval` and `log_save_interval` are now based on training loop's `global_step` instead of epoch-internal batch index (#3667)
- Silenced some warnings. verified ddp refactors (#3483)
- Cleaning up stale logger tests (#3490)
- Allow `ModelCheckpoint` monitor to be `None` (#3633)
- Enable `None` model checkpoint default (#3669)
- Skipped `best_model_path` if `checkpoint_callback` is `None` (#2962)
- Used `raise .. from ..` to explicitly chain exceptions (#3750)
- Mocking loggers (#3596, #3617, #3851, #3859, #3884, #3853, #3910, #3889, #3926)
- Write predictions in LightningModule instead of EvalResult #3882

49.21.3 [0.10.0] - Deprecated

- Deprecated `TrainResult` and `EvalResult`, use `self.log` and `self.write` from the `LightningModule` to log metrics and write predictions. `training_step` can now only return a scalar (for the loss) or a dictionary with anything you want. (#3681)
- Deprecate `early_stop_callback` Trainer argument (#3845)
- Rename Trainer arguments `row_log_interval` >> `log_every_n_steps` and `log_save_interval` >> `flush_logs_every_n_steps` (#3748)

49.21.4 [0.10.0] - Removed

- Removed experimental Metric API (#3868, #3943, #3949, #3946), listed changes before final removal:
  - Added `EmbeddingSimilarity` metric (#3349, #3358)
  - Added hooks to metric module interface (#2528)
  - Added error when AUROC metric is used for multiclass problems (#3350)
  - Fixed `ModelCheckpoint` with `save_top_k=-1` option not tracking the best models when a monitor metric is available (#3735)
  - Fixed counter-intuitive error being thrown in `Accuracy` metric for zero target tensor (#3764)
  - Fixed aggregation of metrics (#3517)
  - Fixed Metric aggregation (#3321)
  - Fixed RMSLE metric (#3188)
  - Renamed `reduction` to `class_reduction` in classification metrics (#3322)
  - Changed `class_reduction` similar to sklearn for classification metrics (#3322)
  - Renaming of precision recall metric (#3308)

49.21.5 [0.10.0] - Fixed

- Fixed `on_train_batch_start` hook to end epoch early (#3700)
- Fixed `num_sanity_val_steps` is clipped to `limit_val_batches` (#2917)
- Fixed ONNX model save on GPU (#3145)
- Fixed `GpuUsageLogger` to work on different platforms (#3008)
- Fixed auto-scale batch size not dumping `auto_lr_find` parameter (#3151)
- Fixed `batch_outputs` with optimizer frequencies (#3229)
- Fixed setting batch size in `LightningModule.datamodule` when using `auto_scale_batch_size` (#3266)
- Fixed Horovod distributed backend compatibility with native AMP (#3404)
- Fixed batch size auto scaling exceeding the size of the dataset (#3271)
- Fixed getting `experiment_id` from MLFlow only once instead of each training loop (#3394)
- Fixed `overfit_batches` which now correctly disables shuffling for the training loader. (#3501)
- Fixed gradient norm tracking for `row_log_interval > 1` (#3489)
- Fixed `ModelCheckpoint` name formatting (3164)
- Fixed auto-scale batch size (#3151)
- Fixed example implementation of AutoEncoder (#3190)
- Fixed invalid paths when remote logging with TensorBoard (#3236)
- Fixed change `t()` to `transpose()` as XLA devices do not support `.t()` on 1-dim tensor (#3252)
- Fixed (weights only) checkpoints loading without PL (#3287)
- Fixed `gather_all_tensors` cross GPUs in DDP (#3319)
- Fixed CometML save dir (#3419)
- Fixed forward key metrics (#3467)
- Fixed normalize mode at confusion matrix (replace NaNs with zeros) (#3465)
- Fixed global step increment in training loop when `training_epoch_end` hook is used (#3673)
- Fixed dataloader shuffling not getting turned off with `overfit_batches > 0` and `distributed_backend = "ddp"` (#3534)
- Fixed determinism in `DDPSpawnBackend` when using `seed_everything` in main process (#3335)
- Fixed `ModelCheckpoint` `period` to actually save every `period` epochs (#3630)
- Fixed `val_progress_bar` total with `num_sanity_val_steps` (#3751)
- Fixed Tuner dump: add `current_epoch` to dumped_params (#3261)
- Fixed `current_epoch` and `global_step` properties mismatch between `Trainer` and `LightningModule` (#3785)
- Fixed learning rate scheduler for optimizers with internal state (#3897)
- Fixed `tbptt_reduce_fx` when non-floating tensors are logged (#3796)
- Fixed model checkpoint frequency (#3852)
- Fixed logging non-tensor scalar with result breaks subsequent epoch aggregation (#3855)

- Fixed `TrainerEvaluationLoopMixin` activates `model.train()` at the end (#3858)
- Fixed `overfit_batches` when using with multiple val/test_dataloaders (#3857)
- Fixed enables `training_step` to return `None` (#3862)
- Fixed init nan for checkpointing (#3863)
- Fixed for `load_from_checkpoint` (#2776)
- Fixes incorrect `batch_sizes` when Dataloader returns a dict with multiple tensors (#3668)
- Fixed unexpected signature for `validation_step` (#3947)

49.22 [0.9.0] - 2020-08-20

49.22.1 [0.9.0] - Added

- Added SyncBN for DDP (#2801, #2838)
- Added basic `CSVLogger` (#2721)
- Added SSIM metrics (#2671)
- Added BLEU metrics (#2535)
- Added support to export a model to ONNX format (#2596)
- Added support for `Trainer(num_sanity_val_steps=-1)` to check all validation data before training (#2246)
- Added struct. output:
  - tests for val loop flow (#2605)
  - `EvalResult` support for train and val. loop (#2615, #2651)
  - weighted average in results obj (#2930)
  - fix result obj DP auto reduce (#3013)
- Added class `LightningDataModule` (#2668)
- Added support for PyTorch 1.6 (#2745)
- Added call DataModule hooks implicitly in trainer (#2755)
- Added support for Mean in DDP Sync (#2568)
- Added remaining `sklearn` metrics: `AveragePrecision`, `BalancedAccuracy`, `CohenKappaScore`, `DCG`, `Hamming`, `Hinge`, `Jaccard`, `MeanAbsoluteError`, `MeanSquaredError`, `MeanSquaredLogError`, `MedianAbsoluteError`, `R2Score`, `MeanPoissonDeviance`, `MeanGammaDeviance`, `MeanTweedieDeviance`, `ExplainedVariance` (#2562)
- Added support for `limit_{mode}_batches (int)` to work with infinite dataloader (IterableDataset) (#2840)
- Added support returning python scalars in DP (#1935)
- Added support to Tensorboard logger for OmegaConf `hparams` (#2846)
- Added tracking of basic states in `Trainer` (#2541)
- Tracks all outputs including TBPTT and multiple optimizers (#2890)

- Added GPU Usage Logger (#2932)
- Added `strict=False` for `load_from_checkpoint` (#2819)
- Added saving test predictions on multiple GPUs (#2926)
- Auto log the computational graph for loggers that support this (#3003)
- Added warning when changing monitor and using results obj (#3014)
- Added a hook `transfer_batch_to_device` to the `LightningDataModule` (#3038)

49.22.2 [0.9.0] - Changed

- Truncated long version numbers in progress bar (#2594)
- Enabling val/test loop disabling (#2692)
- Refactored into `accelerator` module:
  - GPU training (#2704)
  - TPU training (#2708)
  - DDP(2) backend (#2796)
  - Retrieve last logged val from result by key (#3049)
- Using `.comet.config` file for `CometLogger` (#1913)
- Updated hooks arguments - breaking for `setup` and `teardown` (#2850)
- Using `gfile` to support remote directories (#2164)
- Moved optimizer creation after device placement for DDP backends (#2904)
- Support `**DictConfig` for `hparam` serialization (#2519)
- Removed callback metrics from test results obj (#2994)
- Re-enabled naming metrics in ckpt name (#3060)
- Changed progress bar epoch counting to start from 0 (#3061)

49.22.3 [0.9.0] - Deprecated

- Deprecated Trainer attribute `ckpt_path`, which will now be set by `weights_save_path` (#2681)

49.22.4 [0.9.0] - Removed

- Removed deprecated: (#2760)
  - core decorator `data_loader`
  - Module hook `on_sanity_check_start` and loading `load_from_metrics`
  - package `pytorch_lightning.logging`
  - Trainer arguments: `show_progress_bar`, `num_tpu_cores`, `use_amp`, `print_nan_grads`
  - LR Finder argument `num_accumulation_steps`

49.22.5 [0.9.0] - Fixed

- Fixed `accumulate_grad_batches` for last batch (#2853)
- Fixed setup call while testing (#2624)
- Fixed local rank zero casting (#2640)
- Fixed single scalar return from training (#2587)
- Fixed Horovod backend to scale LR schedlers with the optimizer (#2626)
- Fixed `dtype` and `device` properties not getting updated in submodules (#2657)
- Fixed `fast_dev_run` to run for all dataloaders (#2581)
- Fixed `save_dir` in loggers getting ignored by default value of `weights_save_path` when user did not specify `weights_save_path` (#2681)
- Fixed `weights_save_path` getting ignored when `logger=False` is passed to Trainer (#2681)
- Fixed TPU multi-core and Float16 (#2632)
- Fixed test metrics not being logged with `LoggerCollection` (#2723)
- Fixed data transfer to device when using `torchtext.data.Field` and `include_lengths is True` (#2689)
- Fixed shuffle argument for distributed sampler (#2789)
- Fixed logging interval (#2694)
- Fixed loss value in the progress bar is wrong when `accumulate_grad_batches > 1` (#2738)
- Fixed correct CWD for ddp sub-processes when using Hydra (#2719)
- Fixed selecting GPUs using `CUDA_VISIBLE_DEVICES` (#2739, #2796)
- Fixed false `num_classes` warning in metrics (#2781)
- Fixed shell injection vulnerability in subprocess call (#2786)
- Fixed LR finder and `hparams` compatibility (#2821)
- Fixed `ModelCheckpoint` not saving the latest information when `save_last=True` (#2881)
- Fixed ImageNet example: learning rate scheduler, number of workers and batch size when using DDP (#2889)
- Fixed apex gradient clipping (#2829)
- Fixed save apex scaler states (#2828)
- Fixed a model loading issue with inheritance and variable positional arguments (#2911)
- Fixed passing `non_blocking=True` when transferring a batch object that does not support it (#2910)
- Fixed checkpointing to remote file paths (#2925)
- Fixed adding val step argument to metrics (#2986)
- Fixed an issue that caused `Trainer.test()` to stall in ddp mode (#2997)
- Fixed gathering of results with tensors of varying shape (#3020)
- Fixed batch size auto-scaling feature to set the new value on the correct model attribute (#3043)
- Fixed automatic batch scaling not working with half precision (#3045)
- Fixed setting device to root gpu (#3042)

49.23 [0.8.5] - 2020-07-09

49.23.1 [0.8.5] - Added

- Added a PSNR metric: peak signal-to-noise ratio (#2483)
- Added functional regression metrics (#2492)

49.23.2 [0.8.5] - Removed

- Removed auto val reduce (#2462)

49.23.3 [0.8.5] - Fixed

- Flattening Wandb Hyperparameters (#2459)
- Fixed using the same DDP python interpreter and actually running (#2482)
- Fixed model summary input type conversion for models that have input dtype different from model parameters (#2510)
- Made `TensorBoardLogger` and `CometLogger` pickleable (#2518)
- Fixed a problem with `MLflowLogger` creating multiple run folders (#2502)
- Fixed global_step increment (#2455)
- Fixed TPU hanging example (#2488)
- Fixed `argparse` default value bug (#2526)
- Fixed Dice and IoU to avoid NaN by adding small eps (#2545)
- Fixed accumulate gradients schedule at epoch 0 (continued) (#2513)
- Fixed Trainer `.fit()` returning last not best weights in "ddp_spawn" (#2565)
- Fixed passing (do not pass) TPU weights back on test (#2566)
- Fixed DDP tests and `.test()` (#2512, #2570)

49.24 [0.8.4] - 2020-07-01

49.24.1 [0.8.4] - Added

- Added reduce ddp results on eval (#2434)
- Added a warning when an `IterableDataset` has `__len__` defined (#2437)

49.24.2 [0.8.4] - Changed

- Enabled no returns from eval (#2446)

49.24.3 [0.8.4] - Fixed

- Fixes train outputs (#2428)
- Fixes Conda dependencies (#2412)
- Fixed Apex scaling with decoupled backward (#2433)
- Fixed crashing or wrong displaying progressbar because of missing ipywidgets (#2417)
- Fixed TPU saving dir (fc26078e, 04e68f02)
- Fixed logging on rank 0 only (#2425)

49.25 [0.8.3] - 2020-06-29

49.25.1 [0.8.3] - Fixed

- Fixed AMP wrong call (593837e)
- Fixed batch typo (92d1e75)

49.26 [0.8.2] - 2020-06-28

49.26.1 [0.8.2] - Added

- Added TorchText support for moving data to GPU (#2379)

49.26.2 [0.8.2] - Changed

- Changed epoch indexing from 0 instead of 1 (#2289)
- Refactor Model `backward` (#2276)
- Refactored `training_batch` + tests to verify correctness (#2327, #2328)
- Refactored training loop (#2336)
- Made optimization steps for hooks (#2363)
- Changed default apex level to 'O2' (#2362)

49.26.3 [0.8.2] - Removed

- Moved `TrainsLogger` to Bolts (#2384)

49.26.4 [0.8.2] - Fixed

- Fixed parsing TPU arguments and TPU tests (#2094)
- Fixed number batches in case of multiple dataloaders and `limit_{*}_batches` (#1920, #2226)
- Fixed an issue with forward hooks not being removed after model summary (#2298)
- Fix for `load_from_checkpoint()` not working with absolute path on Windows (#2294)
- Fixed an issue how _has_len handles `NotImplementedError` e.g. raised by `torchtext.data.Iterator` (#2293), (#2307)
- Fixed `average_precision` metric (#2319)
- Fixed ROC metric for CUDA tensors (#2304)
- Fixed `average_precision` metric (#2319)
- Fixed lost compatibility with custom datatypes implementing `.to` (#2335)
- Fixed loading model with kwargs (#2387)
- Fixed sum(0) for `trainer.num_val_batches` (#2268)
- Fixed checking if the parameters are a `DictConfig` Object (#2216)
- Fixed SLURM weights saving (#2341)
- Fixed swaps LR scheduler order (#2356)
- Fixed adding tensorboard `hparams` logging test (#2342)
- Fixed use model ref for tear down (#2360)
- Fixed logger crash on DDP (#2388)
- Fixed several issues with early stopping and checkpoint callbacks (#1504, #2391)
- Fixed loading past checkpoints from v0.7.x (#2405)
- Fixed loading model without arguments (#2403)
- Fixed Windows compatibility issue (#2358)

49.27 [0.8.1] - 2020-06-19

49.27.1 [0.8.1] - Fixed

- Fixed the `load_from_checkpoint` path detected as URL bug (#2244)
- Fixed hooks - added barrier (#2245, #2257, #2260)
- Fixed `hparams` - remove frame inspection on `self.hparams` (#2253)
- Fixed setup and on fit calls (#2252)
- Fixed GPU template (#2255)

49.28 [0.8.0] - 2020-06-18

49.28.1 [0.8.0] - Added

- Added `overfit_batches`, `limit_{val|test}_batches` flags (overfit now uses training set for all three) (#2213)
- Added metrics
  - Base classes (#1326, #1877)
  - Sklearn metrics classes (#1327)
  - Native torch metrics (#1488, #2062)
  - docs for all Metrics (#2184, #2209)
  - Regression metrics (#2221)
- Added type hints in `Trainer.fit()` and `Trainer.test()` to reflect that also a list of dataloaders can be passed in (#1723)
- Allow dataloaders without sampler field present (#1907)
- Added option `save_last` to save the model at the end of every epoch in `ModelCheckpoint` (#1908)
- Early stopping checks `on_validation_end` (#1458)
- Attribute `best_model_path` to `ModelCheckpoint` for storing and later retrieving the path to the best saved model file (#1799)
- Speed up single-core TPU training by loading data using `ParallelLoader` (#2033)
- Added a model hook `transfer_batch_to_device` that enables moving custom data structures to the target device (1756)
- Added black formatter for the code with code-checker on pull (1610)
- Added back the slow spawn ddp implementation as `ddp_spawn` (#2115)
- Added loading checkpoints from URLs (#1667)
- Added a callback method `on_keyboard_interrupt` for handling KeyboardInterrupt events during training (#2134)
- Added a decorator `auto_move_data` that moves data to the correct device when using the LightningModule for inference (#1905)
- Added `ckpt_path` option to `LightningModule.test(...)` to load particular checkpoint (#2190)
- Added `setup` and `teardown` hooks for model (#2229)

49.28.2 [0.8.0] - Changed

- Allow user to select individual TPU core to train on (#1729)
- Removed non-finite values from loss in `LRFinder` (#1862)
- Allow passing model hyperparameters as complete kwarg list (#1896)
- Renamed `ModelCheckpoint`'s attributes `best` to `best_model_score` and `kth_best_model` to `kth_best_model_path` (#1799)
- Re-Enable Logger's `ImportErrors` (#1938)

- Changed the default value of the Trainer argument `weights_summary` from `full` to `top` (#2029)
- Raise an error when lightning replaces an existing sampler (#2020)
- Enabled `prepare_data` from correct processes - clarify local vs global rank (#2166)
- Remove explicit flush from tensorboard logger (#2126)
- Changed epoch indexing from 1 instead of 0 (#2206)

49.28.3 [0.8.0] - Deprecated

- Deprecated flags: (#2213)
  - `overfit_pct` in favour of `overfit_batches`
  - `val_percent_check` in favour of `limit_val_batches`
  - `test_percent_check` in favour of `limit_test_batches`
- Deprecated `ModelCheckpoint`'s attributes `best` and `kth_best_model` (#1799)
- Dropped official support/testing for older PyTorch versions <1.3 (#1917)
- Deprecated Trainer `proc_rank` in favour of `global_rank` (#2166, #2269)

49.28.4 [0.8.0] - Removed

- Removed unintended Trainer argument `progress_bar_callback`, the callback should be passed in by `Trainer(callbacks=[...])` instead (#1855)
- Removed obsolete `self._device` in Trainer (#1849)
- Removed deprecated API (#2073)
  - Packages: `pytorch_lightning.pt_overrides`, `pytorch_lightning.root_module`
  - Modules: `pytorch_lightning.logging.comet_logger`, `pytorch_lightning.logging.mlflow_logger`, `pytorch_lightning.logging.test_tube_logger`, `pytorch_lightning.overrides.override_data_parallel`, `pytorch_lightning.core.model_saving`, `pytorch_lightning.core.root_module`
  - Trainer arguments: `add_row_log_interval`, `default_save_path`, `gradient_clip`, `nb_gpu_nodes`, `max_nb_epochs`, `min_nb_epochs`, `nb_sanity_val_steps`
  - Trainer attributes: `nb_gpu_nodes`, `num_gpu_nodes`, `gradient_clip`, `max_nb_epochs`, `min_nb_epochs`, `nb_sanity_val_steps`, `default_save_path`, `tng_tqdm_dic`

49.28.5 [0.8.0] - Fixed

- Run graceful training teardown on interpreter exit (#1631)
- Fixed user warning when apex was used together with learning rate schedulers (#1873)
- Fixed multiple calls of `EarlyStopping` callback (#1863)
- Fixed an issue with `Trainer.from_argparse_args` when passing in unknown Trainer args (#1932)
- Fixed bug related to logger not being reset correctly for model after tuner algorithms (#1933)
- Fixed root node resolution for SLURM cluster with dash in host name (#1954)

- Fixed `LearningRateLogger` in multi-scheduler setting (#1944)
- Fixed test configuration check and testing (#1804)
- Fixed an issue with Trainer constructor silently ignoring unknown/misspelled arguments (#1820)
- Fixed `save_weights_only` in ModelCheckpoint (#1780)
- Allow use of same `WandbLogger` instance for multiple training loops (#2055)
- Fixed an issue with `_auto_collect_arguments` collecting local variables that are not constructor arguments and not working for signatures that have the instance not named `self` (#2048)
- Fixed mistake in parameters' grad norm tracking (#2012)
- Fixed CPU and hanging GPU crash (#2118)
- Fixed an issue with the model summary and `example_input_array` depending on a specific ordering of the submodules in a LightningModule (#1773)
- Fixed Tpu logging (#2230)
- Fixed Pid port + duplicate `rank_zero` logging (#2140, #2231)

49.29 [0.7.6] - 2020-05-16

49.29.1 [0.7.6] - Added

- Added callback for logging learning rates (#1498)
- Added transfer learning example (for a binary classification task in computer vision) (#1564)
- Added type hints in `Trainer.fit()` and `Trainer.test()` to reflect that also a list of dataloaders can be passed in (#1723).
- Added auto scaling of batch size (#1638)
- The progress bar metrics now also get updated in `training_epoch_end` (#1724)
- Enable `NeptuneLogger` to work with `distributed_backend=ddp` (#1753)
- Added option to provide seed to random generators to ensure reproducibility (#1572)
- Added override for hparams in `load_from_ckpt` (#1797)
- Added support multi-node distributed execution under `torchelastic` (#1811, #1818)
- Added using `store_true` for bool args (#1822, #1842)
- Added dummy logger for internally disabling logging for some features (#1836)

49.29.2 [0.7.6] - Changed

- Enable `non-blocking` for device transfers to GPU (#1843)
- Replace mata_tags.csv with hparams.yaml (#1271)
- Reduction when `batch_size < num_gpus` (#1609)
- Updated LightningTemplateModel to look more like Colab example (#1577)
- Don't convert `namedtuple` to `tuple` when transferring the batch to target device (#1589)
- Allow passing hparams as keyword argument to LightningModule when loading from checkpoint (#1639)

- Args should come after the last positional argument (#1807)
- Made ddp the default if no backend specified with multiple GPUs (#1789)

49.29.3 [0.7.6] - Deprecated

- Deprecated `tags_csv` in favor of `hparams_file` (#1271)

49.29.4 [0.7.6] - Fixed

- Fixed broken link in PR template (#1675)
- Fixed ModelCheckpoint not None checking filepath (#1654)
- Trainer now calls `on_load_checkpoint()` when resuming from a checkpoint (#1666)
- Fixed sampler logic for ddp with iterable dataset (#1734)
- Fixed `_reset_eval_dataloader()` for IterableDataset (#1560)
- Fixed Horovod distributed backend to set the `root_gpu` property (#1669)
- Fixed wandb logger `global_step` affects other loggers (#1492)
- Fixed disabling progress bar on non-zero ranks using Horovod backend (#1709)
- Fixed bugs that prevent lr finder to be used together with early stopping and validation dataloaders (#1676)
- Fixed a bug in Trainer that prepended the checkpoint path with `version_` when it shouldn't (#1748)
- Fixed lr key name in case of param groups in LearningRateLogger (#1719)
- Fixed saving native AMP scaler state (introduced in #1561)
- Fixed accumulation parameter and suggestion method for learning rate finder (#1801)
- Fixed num processes wasn't being set properly and auto sampler was ddp failing (#1819)
- Fixed bugs in semantic segmentation example (#1824)
- Fixed saving native AMP scaler state (#1561, #1777)
- Fixed native amp + ddp (#1788)
- Fixed `hparam` logging with metrics (#1647)

49.30 [0.7.5] - 2020-04-27

49.30.1 [0.7.5] - Changed

- Allow logging of metrics together with `hparams` (#1630)
- Allow metrics logged together with hparams (#1630)

49.30.2 [0.7.5] - Removed

- Removed Warning from trainer loop (#1634)

49.30.3 [0.7.5] - Fixed

- Fixed ModelCheckpoint not being fixable (#1632)
- Fixed CPU DDP breaking change and DDP change (#1635)
- Tested pickling (#1636)

49.31 [0.7.4] - 2020-04-26

49.31.1 [0.7.4] - Added

- Added flag `replace_sampler_ddp` to manually disable sampler replacement in DDP (#1513)
- Added speed parity tests (max 1 sec difference per epoch)(#1482)
- Added `auto_select_gpus` flag to trainer that enables automatic selection of available GPUs on exclusive mode systems.
- Added learning rate finder (#1347)
- Added support for ddp mode in clusters without SLURM (#1387)
- Added `test_dataloaders` parameter to `Trainer.test()` (#1434)
- Added `terminate_on_nan` flag to trainer that performs a NaN check with each training iteration when set to `True` (#1475)
- Added speed parity tests (max 1 sec difference per epoch)(#1482)
- Added `terminate_on_nan` flag to trainer that performs a NaN check with each training iteration when set to `True`. (#1475)
- Added `ddp_cpu` backend for testing ddp without GPUs (#1158)
- Added Horovod support as a distributed backend `Trainer(distributed_backend='horovod')` (#1529)
- Added support for 8 core distributed training on Kaggle TPU's (#1568)
- Added support for native AMP (#1561, #1580)

49.31.2 [0.7.4] - Changed

- Changed the default behaviour to no longer include a NaN check with each training iteration. (#1475)
- Decoupled the progress bar from trainer` it is a callback now and can be customized or even be replaced entirely (#1450).
- Changed lr schedule step interval behavior to update every backwards pass instead of every forwards pass (#1477)
- Defines shared proc. rank, remove rank from instances (e.g. loggers) (#1408)
- Updated semantic segmentation example with custom U-Net and logging (#1371)

- Disabled val and test shuffling (#1600)

49.31.3 [0.7.4] - Deprecated

- Deprecated `training_tqdm_dict` in favor of `progress_bar_dict` (#1450).

49.31.4 [0.7.4] - Removed

- Removed `test_dataloaders` parameter from `Trainer.fit()` (#1434)

49.31.5 [0.7.4] - Fixed

- Added the possibility to pass nested metrics dictionaries to loggers (#1582)
- Fixed memory leak from opt return (#1528)
- Fixed saving checkpoint before deleting old ones (#1453)
- Fixed loggers - flushing last logged metrics even before continue, e.g. `trainer.test()` results (#1459)
- Fixed optimizer configuration when `configure_optimizers` returns dict without `lr_scheduler` (#1443)
- Fixed `LightningModule` - mixing hparams and arguments in `LightningModule.__init__()` crashes load_from_checkpoint() (#1505)
- Added a missing call to the `on_before_zero_grad` model hook (#1493).
- Allow use of sweeps with `WandbLogger` (#1512)
- Fixed a bug that caused the `callbacks` Trainer argument to reference a global variable (#1534).
- Fixed a bug that set all boolean CLI arguments from `Trainer.add_argparse_args` always to True (#1571)
- Fixed do not copy the batch when training on a single GPU (#1576, #1579)
- Fixed soft checkpoint removing on DDP (#1408)
- Fixed automatic parser bug (#1585)
- Fixed bool conversion from string (#1606)

49.32 [0.7.3] - 2020-04-09

49.32.1 [0.7.3] - Added

- Added `rank_zero_warn` for warning only in rank 0 (#1428)

49.32.2 [0.7.3] - Fixed

- Fixed default `DistributedSampler` for DDP training (#1425)
- Fixed workers warning not on windows (#1430)
- Fixed returning tuple from `run_training_batch` (#1431)
- Fixed gradient clipping (#1438)
- Fixed pretty print (#1441)

49.33 [0.7.2] - 2020-04-07

49.33.1 [0.7.2] - Added

- Added same step loggers' metrics aggregation (#1278)
- Added parity test between a vanilla MNIST model and lightning model (#1284)
- Added parity test between a vanilla RNN model and lightning model (#1351)
- Added Reinforcement Learning - Deep Q-network (DQN) lightning example (#1232)
- Added support for hierarchical `dict` (#1152)
- Added `TrainsLogger` class (#1122)
- Added type hints to `pytorch_lightning.core` (#946)
- Added support for `IterableDataset` in validation and testing (#1104)
- Added support for non-primitive types in `hparams` for `TensorboardLogger` (#1130)
- Added a check that stops the training when loss or weights contain `NaN` or `inf` values. (#1097)
- Added support for `IterableDataset` when `val_check_interval=1.0` (default), this will trigger validation at the end of each epoch. (#1283)
- Added `summary` method to Profilers. (#1259)
- Added informative errors if user defined dataloader has zero length (#1280)
- Added testing for python 3.8 (#915)
- Added a `training_epoch_end` method which is the mirror of `validation_epoch_end`. (#1357)
- Added model configuration checking (#1199)
- Added support for optimizer frequencies through `LightningModule.configure_optimizers()` (#1269)
- Added option to run without an optimizer by returning `None` from `configure_optimizers`. (#1279)
- Added a warning when the number of data loader workers is small. (#1378)

49.33.2 [0.7.2] - Changed

- Changed (renamed and refatored) `TensorRunningMean` -> `TensorRunningAccum`: running accumulations were generalized. (#1278)
- Changed `progress_bar_refresh_rate` trainer flag to disable progress bar when set to 0. (#1108)
- Enhanced `load_from_checkpoint` to also forward params to the model (#1307)
- Updated references to `self.forward()` to instead use the `__call__` interface. (#1211)
- Changed default behaviour of `configure_optimizers` to use no optimizer rather than Adam. (#1279)
- Allow to upload models on W&B (#1339)
- On DP and DDP2 unsqueeze is automated now (#1319)
- Did not always create a DataLoader during reinstantiation, but the same type as before (if subclass of DataLoader) (#1346)
- Did not interfere with a default sampler (#1318)
- Remove default Adam optimizer (#1317)
- Give warnings for unimplemented required lightning methods (#1317)
- Made `evaluate` method private >> `Trainer._evaluate(...)`. (#1260)
- Simplify the PL examples structure (shallower and more readable) (#1247)
- Changed min max gpu memory to be on their own plots (#1358)
- Remove `.item` which causes sync issues (#1254)
- Changed smoothing in TQDM to decrease variability of time remaining between training / eval (#1194)
- Change default logger to dedicated one (#1064)

49.33.3 [0.7.2] - Deprecated

- Deprecated Trainer argument `print_nan_grads` (#1097)
- Deprecated Trainer argument `show_progress_bar` (#1108)

49.33.4 [0.7.2] - Removed

- Removed test for no test dataloader in .fit (#1495)
- Removed duplicated module `pytorch_lightning.utilities.arg_parse` for loading CLI arguments (#1167)
- Removed wandb logger's `finalize` method (#1193)
- Dropped `torchvision` dependency in tests and added own MNIST dataset class instead (#986)

49.33.5 [0.7.2] - Fixed

- Fixed `model_checkpoint` when saving all models (#1359)
- `Trainer.add_argparse_args` classmethod fixed. Now it adds a type for the arguments (#1147)
- Fixed bug related to type checking of `ReduceLROnPlateau` lr schedulers(#1126)
- Fixed a bug to ensure lightning checkpoints to be backward compatible (#1132)
- Fixed a bug that created an extra dataloader with active `reload_dataloaders_every_epoch` (#1196)
- Fixed all warnings and errors in the docs build process (#1191)
- Fixed an issue where `val_percent_check=0` would not disable validation (#1251)
- Fixed average of incomplete `TensorRunningMean` (#1309)
- Fixed `WandbLogger.watch` with `wandb.init()` (#1311)
- Fixed an issue with early stopping that would prevent it from monitoring training metrics when validation is disabled / not implemented (#1235).
- Fixed a bug that would cause `trainer.test()` to run on the validation set when overloading `validation_epoch_end` and `test_end` (#1353)
- Fixed `WandbLogger.watch` - use of the watch method without importing `wandb` (#1311)
- Fixed `WandbLogger` to be used with 'ddp' - allow reinits in sub-processes (#1149, #1360)
- Made `training_epoch_end` behave like `validation_epoch_end` (#1357)
- Fixed `fast_dev_run` running validation twice (#1365)
- Fixed pickle error from quick patch `__code__` (#1352)
- Fixed memory leak on GPU0 (#1094, #1349)
- Fixed checkpointing interval (#1272)
- Fixed validation and training loops run the partial dataset (#1192)
- Fixed running `on_validation_end` only on main process in DDP (#1125)
- Fixed `load_spawn_weights` only in proc rank 0 (#1385)
- Fixes `use_amp` issue (#1145)
- Fixes using deprecated `use_amp` attribute (#1145)
- Fixed Tensorboard logger error: lightning_logs directory not exists in multi-node DDP on nodes with rank != 0 (#1377)
- Fixed `Unimplemented backend XLA` error on TPU (#1387)

49.34 [0.7.1] - 2020-03-07

49.34.1 [0.7.1] - Fixed

- Fixes `print` issues and `data_loader` (#1080)

49.35 [0.7.0] - 2020-03-06

49.35.1 [0.7.0] - Added

- Added automatic sampler setup. Depending on DDP or TPU, lightning configures the sampler correctly (user needs to do nothing) (#926)
- Added `reload_dataloaders_every_epoch=False` flag for trainer. Some users require reloading data every epoch (#926)
- Added `progress_bar_refresh_rate=50` flag for trainer. Throttle refresh rate on notebooks (#926)
- Updated governance docs
- Added a check to ensure that the metric used for early stopping exists before training commences (#542)
- Added `optimizer_idx` argument to `backward` hook (#733)
- Added `entity` argument to `WandbLogger` to be passed to `wandb.init` (#783)
- Added a tool for profiling training runs (#782)
- Improved flexibility for naming of TensorBoard logs, can now set `version` to a `str` to just save to that directory, and use `name=''` to prevent experiment-name directory (#804)
- Added option to specify `step` key when logging metrics (#808)
- Added `train_dataloader`, `val_dataloader` and `test_dataloader` arguments to `Trainer.fit()`, for alternative data parsing (#759)
- Added Tensor Processing Unit (TPU) support (#868)
- Added semantic segmentation example (#751,#876, #881)
- Split callbacks in multiple files (#849)
- Support for user defined callbacks (#889 and #950)
- Added support for multiple loggers to be passed to `Trainer` as an iterable (e.g. list, tuple, etc.) (#903)
- Added support for step-based learning rate scheduling (#941)
- Added support for logging `hparams` as dict (#1029)
- Checkpoint and early stopping now work without val. step (#1041)
- Support graceful training cleanup after Keyboard Interrupt (#856, #1019)
- Added type hints for function arguments (#912,)
- Added default `argparser` for `Trainer` (#952, #1023)
- Added TPU gradient clipping (#963)
- Added max/min number of steps in `Trainer` (#728)

49.35.2 [0.7.0] - Changed

- Improved `NeptuneLogger` by adding `close_after_fit` argument to allow logging after training(#908)
- Changed default TQDM to use `tqdm.auto` for prettier outputs in IPython notebooks (#752)
- Changed `pytorch_lightning.logging` to `pytorch_lightning.loggers` (#767)
- Moved the default `tqdm_dict` definition from Trainer to `LightningModule`, so it can be overridden by the user (#749)
- Moved functionality of `LightningModule.load_from_metrics` into `LightningModule.load_from_checkpoint` (#995)
- Changed Checkpoint path parameter from `filepath` to `dirpath` (#1016)
- Freezed models `hparams` as `Namespace` property (#1029)
- Dropped `logging` config in package init (#1015)
- Renames model steps (#1051)
  - `training_end` >> `training_epoch_end`
  - `validation_end` >> `validation_epoch_end`
  - `test_end` >> `test_epoch_end`
- Refactor dataloading, supports infinite dataloader (#955)
- Create single file in `TensorBoardLogger` (#777)

49.35.3 [0.7.0] - Deprecated

- Deprecated `pytorch_lightning.logging` (#767)
- Deprecated `LightningModule.load_from_metrics` in favour of `LightningModule.load_from_checkpoint` (#995, #1079)
- Deprecated `@data_loader` decorator (#926)
- Deprecated model steps `training_end`, `validation_end` and `test_end` (#1051, #1056)

49.35.4 [0.7.0] - Removed

- Removed dependency on `pandas` (#736)
- Removed dependency on `torchvision` (#797)
- Removed dependency on `scikit-learn` (#801)

49.35.5 [0.7.0] - Fixed

- Fixed a bug where early stopping `on_end_epoch` would be called inconsistently when `check_val_every_n_epoch == 0` (#743)
- Fixed a bug where the model checkpointer didn't write to the same directory as the logger (#771)
- Fixed a bug where the `TensorBoardLogger` class would create an additional empty log file during fitting (#777)
- Fixed a bug where `global_step` was advanced incorrectly when using `accumulate_grad_batches > 1` (#832)
- Fixed a bug when calling `self.logger.experiment` with multiple loggers (#1009)
- Fixed a bug when calling `logger.append_tags` on a `NeptuneLogger` with a single tag (#1009)
- Fixed sending back data from `.spawn` by saving and loading the trained model in/out of the process (#1017
- Fixed port collision on DDP (#1010)
- Fixed/tested pass overrides (#918)
- Fixed comet logger to log after train (#892)
- Remove deprecated args to learning rate step function (#890)

49.36 [0.6.0] - 2020-01-21

49.36.1 [0.6.0] - Added

- Added support for resuming from a specific checkpoint via `resume_from_checkpoint` argument (#516)
- Added support for `ReduceLROnPlateau` scheduler (#320)
- Added support for Apex mode `O2` in conjunction with Data Parallel (#493)
- Added option (`save_top_k`) to save the top k models in the `ModelCheckpoint` class (#128)
- Added `on_train_start` and `on_train_end` hooks to `ModelHooks` (#598)
- Added `TensorBoardLogger` (#607)
- Added support for weight summary of model with multiple inputs (#543)
- Added `map_location` argument to `load_from_metrics` and `load_from_checkpoint` (#625)
- Added option to disable validation by setting `val_percent_check=0` (#649)
- Added `NeptuneLogger` class (#648)
- Added `WandbLogger` class (#627)

49.36.2 [0.6.0] - Changed

- Changed the default progress bar to print to stdout instead of stderr (#531)
- Renamed `step_idx` to `step`, `epoch_idx` to `epoch`, `max_num_epochs` to `max_epochs` and `min_num_epochs` to `min_epochs` (#589)
- Renamed `total_batch_nb` to `total_batches`, `nb_val_batches` to `num_val_batches`, `nb_training_batches` to `num_training_batches`, `max_nb_epochs` to `max_epochs`, `min_nb_epochs` to `min_epochs`, `nb_test_batches` to `num_test_batches`, and `nb_val_batches` to `num_val_batches` (#567)
- Changed gradient logging to use parameter names instead of indexes (#660)
- Changed the default logger to `TensorBoardLogger` (#609)
- Changed the directory for tensorboard logging to be the same as model checkpointing (#706)

49.36.3 [0.6.0] - Deprecated

- Deprecated `max_nb_epochs` and `min_nb_epochs` (#567)
- Deprecated the `on_sanity_check_start` hook in `ModelHooks` (#598)

49.36.4 [0.6.0] - Removed

- Removed the `save_best_only` argument from `ModelCheckpoint`, use `save_top_k=1` instead (#128)

49.36.5 [0.6.0] - Fixed

- Fixed a bug which ocurred when using Adagrad with cuda (#554)
- Fixed a bug where training would be on the GPU despite setting `gpus=0` or `gpus=[]` (#561)
- Fixed an error with `print_nan_gradients` when some parameters do not require gradient (#579)
- Fixed a bug where the progress bar would show an incorrect number of total steps during the validation sanity check when using multiple validation data loaders (#597)
- Fixed support for PyTorch 1.1.0 (#552)
- Fixed an issue with early stopping when using a `val_check_interval < 1.0` in `Trainer` (#492)
- Fixed bugs relating to the `CometLogger` object that would cause it to not work properly (#481)
- Fixed a bug that would occur when returning `-1` from `on_batch_start` following an early exit or when the batch was `None` (#509)
- Fixed a potential race condition with several processes trying to create checkpoint directories (#530)
- Fixed a bug where batch 'segments' would remain on the GPU when using `truncated_bptt > 1` (#532)
- Fixed a bug when using `IterableDataset` (#547)
- Fixed a bug where `.item` was called on non-tensor objects (#602)
- Fixed a bug where `Trainer.train` would crash on an uninitialized variable if the trainer was run after resuming from a checkpoint that was already at `max_epochs` (#608)
- Fixed a bug where early stopping would begin two epochs early (#617)

- Fixed a bug where `num_training_batches` and `num_test_batches` would sometimes be rounded down to zero (#649)
- Fixed a bug where an additional batch would be processed when manually setting `num_training_batches` (#653)
- Fixed a bug when batches did not have a `.copy` method (#701)
- Fixed a bug when using `log_gpu_memory=True` in Python 3.6 (#715)
- Fixed a bug where checkpoint writing could exit before completion, giving incomplete checkpoints (#689)
- Fixed a bug where `on_train_end` was not called when ealy stopping (#723)

49.37 [0.5.3] - 2019-11-06

49.37.1 [0.5.3] - Added

- Added option to disable default logger, checkpointer, and early stopping by passing `logger=False`, `checkpoint_callback=False` and `early_stop_callback=False` respectively
- Added `CometLogger` for use with Comet.ml
- Added `val_check_interval` argument to `Trainer` allowing validition to be performed at every given number of batches
- Added functionality to save and load hyperparameters using the standard checkpoint mechanism
- Added call to `torch.cuda.empty_cache` before training starts
- Added option for user to override the call t `backward`
- Added support for truncated backprop through time via the `truncated_bptt_steps` argument in `Trainer`
- Added option to operate on all outputs from `training_step` in DDP2
- Added a hook for modifying DDP init
- Added a hook for modifying Apex

49.37.2 [0.5.3] - Changed

- Changed experiment version to be padded with zeros (e.g. `/dir/version_9` becomes `/dir/version_0009`)
- Changed callback metrics to include any metrics given in logs or progress bar
- Changed the default for `save_best_only` in `ModelCheckpoint` to `True`
- Added `tng_data_loader` for backwards compatibility
- Renamed `MLFlowLogger.client` to `MLFlowLogger.experiment` for consistency
- Moved `global_step` increment to happen after the batch has been processed
- Changed weights restore to first attempt HPC weights before restoring normally, preventing both weights being restored and running out of memory
- Changed progress bar functionality to add multiple progress bars for train/val/test
- Changed calls to `print` to use `logging` instead

49.37.3 [0.5.3] - Deprecated

- Deprecated `tng_dataloader`

49.37.4 [0.5.3] - Fixed

- Fixed an issue where the number of batches was off by one during training
- Fixed a bug that occured when setting a ckeckpoint callback and `early_stop_callback=False`
- Fixed an error when importing CometLogger
- Fixed a bug where the `gpus` argument had some unexpected behaviour
- Fixed a bug where the computed total number of batches was sometimes incorrect
- Fixed a bug where the progress bar would sometimes not show the total number of batches in test mode
- Fixed a bug when using the `log_gpu_memory='min_max'` option in `Trainer`
- Fixed a bug where checkpointing would sometimes erase the current directory

49.38 [0.5.2] - 2019-10-10

49.38.1 [0.5.2] - Added

- Added `weights_summary` argument to `Trainer` to be set to `full` (full summary), `top` (just top level modules) or other
- Added `tags` argument to `MLFlowLogger`

49.38.2 [0.5.2] - Changed

- Changed default for `amp_level` to `O1`

49.38.3 [0.5.2] - Removed

- Removed the `print_weights_summary` argument from `Trainer`

49.38.4 [0.5.2] - Fixed

- Fixed a bug where logs were not written properly
- Fixed a bug where `logger.finalize` wasn't called after training is complete
- Fixed callback metric errors in DDP
- Fixed a bug where `TestTubeLogger` didn't log to the correct directory

49.39 [0.5.1] - 2019-10-05

49.39.1 [0.5.1] - Added

- Added the `LightningLoggerBase` class for experiment loggers
- Added `MLFlowLogger` for logging with `mlflow`
- Added `TestTubeLogger` for logging with `test_tube`
- Added a different implementation of DDP (`distributed_backed='ddp2'`) where every node has one model using all GPUs
- Added support for optimisers which require a closure (e.g. LBFGS)
- Added automatic `MASTER_PORT` defualt for DDP when not set manually
- Added new GPU memory logging options `'min_max'` (log only the min/max utilization) and `'all'` (log all the GPU memory)

49.39.2 [0.5.1] - Changed

- Changed schedulers to always be called with the current epoch
- Changed `test_tube` to an optional dependency
- Changed data loaders to internally use a getter instead of a python property
- Disabled auto GPU loading when restoring weights to prevent out of memory errors
- Changed logging, early stopping and checkpointing to occur by default

49.39.3 [0.5.1] - Fixed

- Fixed a bug with samplers that do not specify `set_epoch`
- Fixed a bug when using the `MLFlowLogger` with unsupported data types, this will now raise a warning
- Fixed a bug where gradient norms were alwasy zero using `track_grad_norm`
- Fixed a bug which causes a crash when logging memory

49.40 [0.5.0] - 2019-09-26

49.40.1 [0.5.0] - Changed

- Changed `data_batch` argument to `batch` throughout
- Changed `batch_i` argument to `batch_idx` throughout
- Changed `tng_dataloader` method to `train_dataloader`
- Changed `on_tng_metrics` method to `on_training_metrics`
- Changed `gradient_clip` argument to `gradient_clip_val`
- Changed `add_log_row_interval` to `row_log_interval`

49.40.2 [0.5.0] - Fixed

- Fixed a bug with tensorboard logging in multi-gpu setup

49.41 [0.4.9] - 2019-09-16

49.41.1 [0.4.9] - Added

- Added the flag `log_gpu_memory` to `Trainer` to deactivate logging of GPU memory utilization
- Added SLURM resubmit functionality (port from test-tube)
- Added optional weight_save_path to trainer to remove the need for a checkpoint_callback when using cluster training
- Added option to use single gpu per node with `DistributedDataParallel`

49.41.2 [0.4.9] - Changed

- Changed functionality of `validation_end` and `test_end` with multiple dataloaders to be given all of the dataloaders at once rather than in seperate calls
- Changed print_nan_grads to only print the parameter value and gradients when they contain NaN
- Changed gpu API to take integers as well (e.g. `gpus=2` instead of `gpus=[0, 1]`)
- All models now loaded on to CPU to avoid device and out of memory issues in PyTorch

49.41.3 [0.4.9] - Fixed

- Fixed a bug where data types that implement `.to` but not `.cuda` would not be properly moved onto the GPU
- Fixed a bug where data would not be re-shuffled every epoch when using a `DistributedSampler`

49.42 [0.4.8] - 2019-08-31

49.42.1 [0.4.8] - Added

- Added `test_step` and `test_end` methods, used when `Trainer.test` is called
- Added `GradientAccumulationScheduler` callback which can be used to schedule changes to the number of accumulation batches
- Added option to skip the validation sanity check by setting `nb_sanity_val_steps = 0`

49.42.2 [0.4.8] - Fixed

- Fixed a bug when setting `nb_sanity_val_steps = 0`

49.43 [0.4.7] - 2019-08-24

49.43.1 [0.4.7] - Changed

- Changed the default `val_check_interval` to `1.0`
- Changed defaults for `nb_val_batches`, `nb_tng_batches` and `nb_test_batches` to 0

49.43.2 [0.4.7] - Fixed

- Fixed a bug where the full validation set as used despite setting `val_percent_check`
- Fixed a bug where an `Exception` was thrown when using a data set containing a single batch
- Fixed a bug where an `Exception` was thrown if no `val_dataloader` was given
- Fixed a bug where tuples were not properly transfered to the GPU
- Fixed a bug where data of a non standard type was not properly handled by the trainer
- Fixed a bug when loading data as a tuple
- Fixed a bug where `AttributeError` could be suppressed by the `Trainer`

49.44 [0.4.6] - 2019-08-15

49.44.1 [0.4.6] - Added

- Added support for data to be given as a `dict` or `list` with a single gpu
- Added support for `configure_optimizers` to return a single optimizer, two list (optimizers and schedulers), or a single list

49.44.2 [0.4.6] - Fixed

- Fixed a bug where returning just an optimizer list (i.e. without schedulers) from `configure_optimizers` would throw an `Exception`

49.45 [0.4.5] - 2019-08-13

49.45.1 [0.4.5] - Added

- Added `optimizer_step` method that can be overridden to change the standard optimizer behaviour

49.46 [0.4.4] - 2019-08-12

49.46.1 [0.4.4] - Added

- Added supoort for multiple validation dataloaders
- Added support for latest test-tube logger (optimised for `torch==1.2.0`)

49.46.2 [0.4.4] - Changed

- `validation_step` and `val_dataloader` are now optional
- `lr_scheduler` is now activated after epoch

49.46.3 [0.4.4] - Fixed

- Fixed a bug where a warning would show when using `lr_scheduler` in `torch>1.1.0`
- Fixed a bug where an `Exception` would be thrown if using `torch.DistributedDataParallel` without using a `DistributedSampler`, this now throws a `Warning` instead

49.47 [0.4.3] - 2019-08-10

49.47.1 [0.4.3] - Fixed

- Fixed a bug where accumulate gradients would scale the loss incorrectly

49.48 [0.4.2] - 2019-08-08

49.48.1 [0.4.2] - Changed

- Changed install requirement to `torch==1.2.0`

49.49 [0.4.1] - 2019-08-08

49.49.1 [0.4.1] - Changed

- Changed install requirement to `torch==1.1.0`

49.50 [0.4.0] - 2019-08-08

49.50.1 [0.4.0] - Added

- Added 16-bit support for a single GPU
- Added support for training continuation (preserves epoch, global step etc.)

49.50.2 [0.4.0] - Changed

- Changed `training_step` and `validation_step`, outputs will no longer be automatically reduced

49.50.3 [0.4.0] - Removed

- Removed need for `Experiment` object in `Trainer`

49.50.4 [0.4.0] - Fixed

- Fixed issues with reducing outputs from generative models (such as images and text)

49.51 [0.3.6] - 2019-07-25

49.51.1 [0.3.6] - Added

- Added a decorator to do lazy data loading internally

49.51.2 [0.3.6] - Fixed

- Fixed a bug where `Experiment` object was not process safe, potentially causing logs to be overwritten

49.52 [0.3.5] - 2019-07-25

49.53 [0.3.4] - 2019-07-22

49.54 [0.3.3] - 2019-07-22

49.55 [0.3.2] - 2019-07-21

49.56 [0.3.1] - 2019-07-21

49.57 [0.2.x] - 2019-07-09

49.58 [0.1.x] - 2019-06-DD

CHAPTER

FIFTY

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

p

INDEX

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W