PyTorch-Lightning Documentation

Release 0.7.1

William Falcon et al.

Mar 07, 2020

START HERE

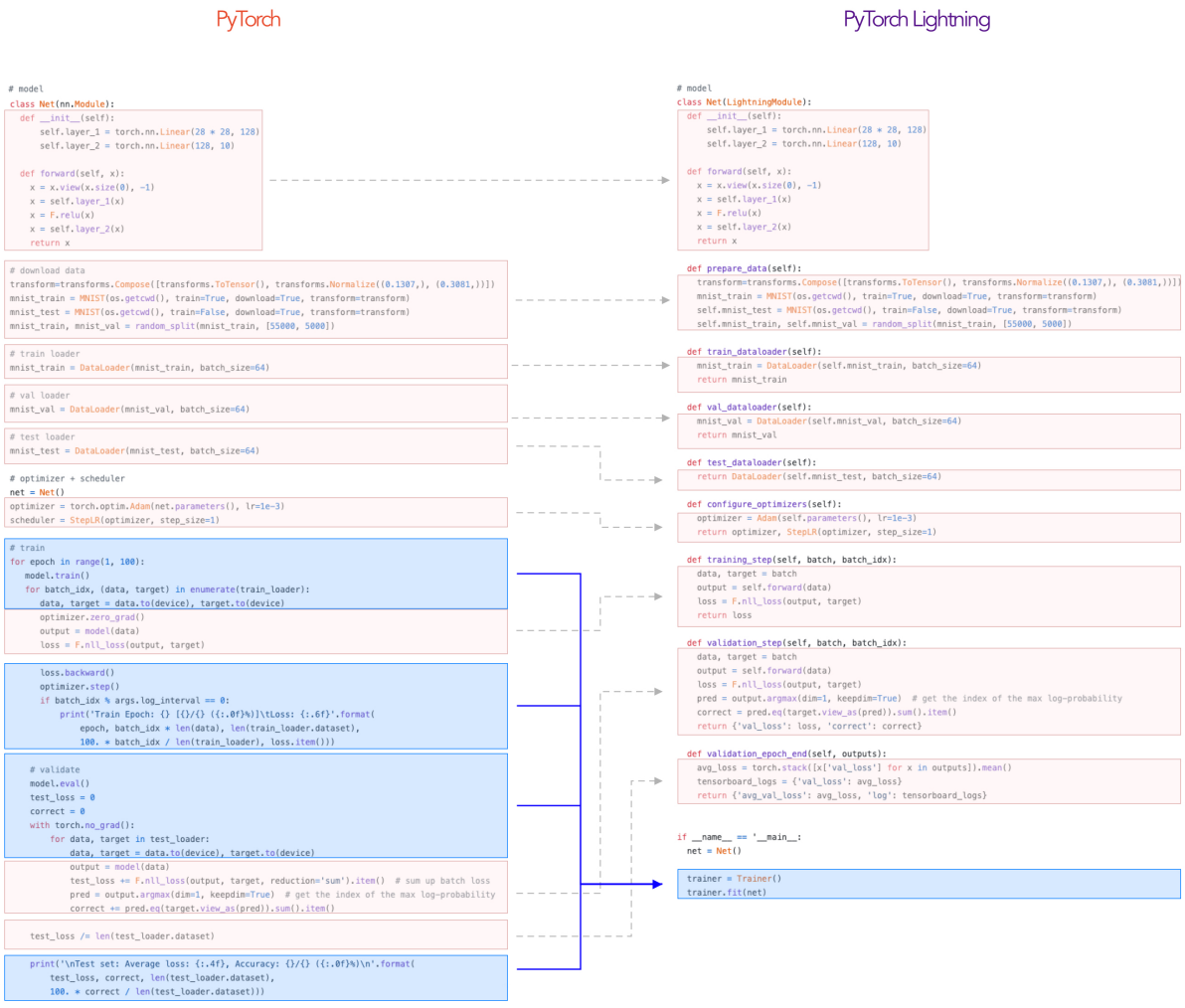
CHAPTER

ONE

INTRODUCTION GUIDE

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

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



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

1.1 Goal of this guide

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

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

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

1.2 Lightning Philosophy

Lightning factors DL/ML code into three types:

- Research code
- Engineerng code
- Non-essential code

1.2.1 Research code

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

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

x1 = l1(x)
x2 = l2(x2)
out = decoder(features, x)

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

1.2.2 Engineering code

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

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

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

distributed = DistributedParallel(model)

with gpu_zero:
```

(continues on next page)

(continued from previous page)

```
    download_data()

dist.barrier()
```

1.2.3 Non-essential code

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

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

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

1.3 Elements of a research project

Every research project requires the same core ingredients:

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

1.3.1 The Model

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

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

```
import torch
from torch.nn import functional as F
from torch import nn
import pytorch_lightning as pl

class LitMNIST(pl.LightningModule):

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

    # layer 3
    x = self.layer_3(x)

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

    return x
```

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

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

Out:

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

1.3.2 Data

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

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


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

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

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

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

class LitMNIST(pl.LightningModule):

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

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

Usually though, we want to separate the things that write to disk in data-processing from things like transforms which happen in memory.

```
class LitMNIST(pl.LightningModule):

  def prepare_data(self):
    # download only
    MNIST(os.getcwd(), train=True, download=True)

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

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

In general fill these methods with the following:

```
class LitMNIST(pl.LightningModule):

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

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

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

1.3.3 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. If you don't define this, Lightning will automatically use *Adam(self.parameters(), lr=1e-3)*.

```
class LitMNIST(pl.LightningModule):

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

1.3.4 Training step

The training step is what happens inside the training loop.

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

In the case of MNIST we do the following

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

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

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

```
class LitMNIST(pl.LightningModule):

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

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

1.4 Training

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

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

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

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

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

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

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

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

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

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

1.4.1 Train on CPU

```
from pytorch_lightning import Trainer

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

You should see the following weights summary and progress bar

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

1.4.2 Logging

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

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

Which will generate automatic tensorboard logs.

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

1.4.3 GPU training

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

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

1.4.4 Multi-GPU training

Or you can also train on multiple GPUs.

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

Or multiple nodes

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

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

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

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

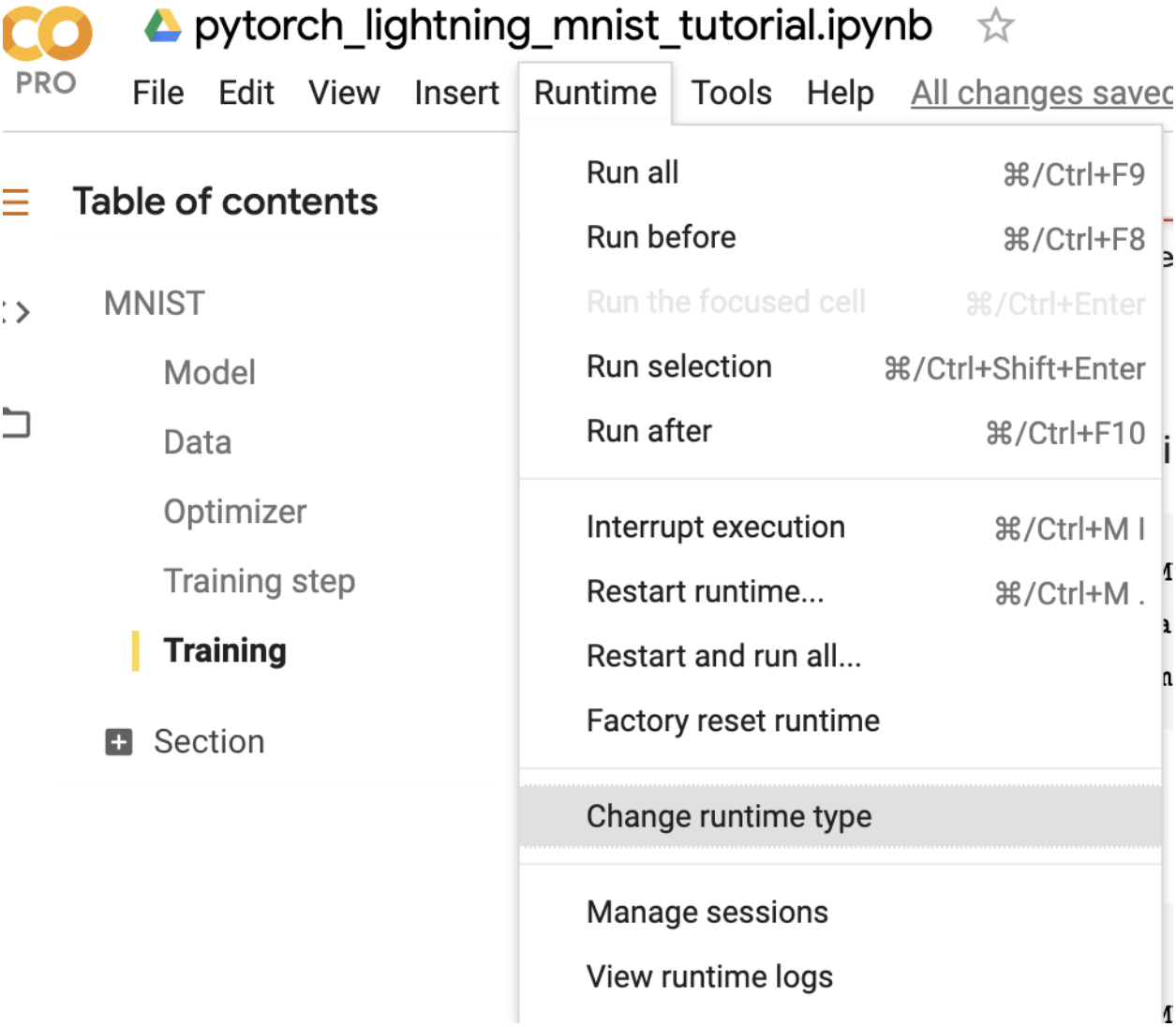

```
INFO:root:GPU available: True, used: True
INFO:root:VISIBLE GPUS: 0
INFO:root:
  | Name    | Type   | Params
-------------------------------
0 | layer_1 | Linear | 100 K
1 | layer_2 | Linear | 33 K
2 | layer_3 | Linear | 2 K
Epoch 1: 53% 500/938 [00:07<00:07, 61.53it/s, loss=0.206, v_num=3]
```

1.4.5 TPUs

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

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

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

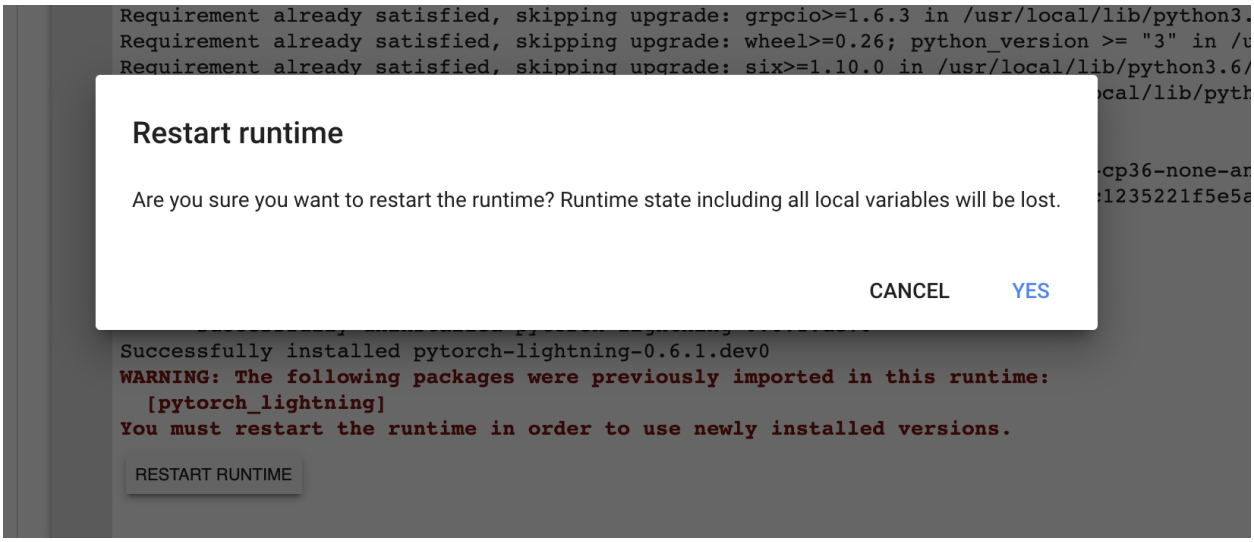


Next, install the required xla library (adds support for PyTorch on TPUs)

```
import collections
from datetime import datetime, timedelta
import os
import requests
import threading

_VersionConfig = collections.namedtuple('_VersionConfig', 'wheels,server')
VERSION = "torch_xla==nightly"  #@param ["xrt==1.15.0", "torch_xla==nightly"]
CONFIG = {
```

(continues on next page)



(continued from previous page)

```
    'xrt==1.15.0': _VersionConfig('1.15', '1.15.0'),
    'torch_xla==nightly': _VersionConfig('nightly', 'XRT-dev{}'.format(
        (datetime.today() - timedelta(1)).strftime('%Y%m%d'))),
}[VERSION]
DIST_BUCKET = 'gs://tpu-pytorch/wheels'
TORCH_WHEEL = 'torch-{}-cp36-cp36m-linux_x86_64.whl'.format(CONFIG.wheels)
TORCH_XLA_WHEEL = 'torch_xla-{}-cp36-cp36m-linux_x86_64.whl'.format(CONFIG.wheels)
TORCHVISION_WHEEL = 'torchvision-{}-cp36-cp36m-linux_x86_64.whl'.format(CONFIG.wheels)

# Update TPU XRT version
def update_server_xrt():
  print('Updating server-side XRT to {} ...'.format(CONFIG.server))
  url = 'http://{TPU_ADDRESS}:8475/requestversion/{XRT_VERSION}'.format(
      TPU_ADDRESS=os.environ['COLAB_TPU_ADDR'].split(':')[0],
      XRT_VERSION=CONFIG.server,
  )
  print('Done updating server-side XRT: {}'.format(requests.post(url)))

update = threading.Thread(target=update_server_xrt)
update.start()

# Install Colab TPU compat PyTorch/TPU wheels and dependencies
!pip uninstall -y torch torchvision
!gsutil cp "$DIST_BUCKET/$TORCH_WHEEL" .
!gsutil cp "$DIST_BUCKET/$TORCH_XLA_WHEEL" .
!gsutil cp "$DIST_BUCKET/$TORCHVISION_WHEEL" .
!pip install "$TORCH_WHEEL"
!pip install "$TORCH_XLA_WHEEL"
!pip install "$TORCHVISION_WHEEL"
!sudo apt-get install libomp5
update.join()
```

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

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

```
class LitMNIST(pl.LightningModule):
  def prepare_data(self):
    # transform
    transform=transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,
↪), (0.3081,))])

    # download
    mnist_train = MNIST(os.getcwd(), train=True, download=True, transform=transform)
    mnist_test = MNIST(os.getcwd(), train=False, download=True, 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=64)

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

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

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

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

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

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

You'll now see the TPU cores booting up.

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

Notice the epoch is MUCH faster!

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

1.5 Hyperparameters

Normally, we don't hard-code the values to a model. We usually use the command line to modify the network.

```
from argparse import ArgumentParser

parser = ArgumentParser()

# parametrize the network
parser.add_argument('--layer_1_dim', type=int, default=128)
parser.add_argument('--layer_2_dim', type=int, default=256)
parser.add_argument('--batch_size', type=int, default=64)

args = parser.parse_args()
```

Now we can parametrize the LightningModule.

```
class LitMNIST(pl.LightningModule):
  def __init__(self, hparams):
    super(LitMNIST, self).__init__()
    self.hparams = hparams

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

  def forward(self, x):
    ...

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

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

hparams = parse_args()
model = LitMNIST(hparams)
```

Note: Bonus! if (hparams) is in your module, Lightning will save it into the checkpoint and restore your model using those hparams exactly.

And we can also add all the flags available in the Trainer to the Argparser.

```
# add all the available Trainer options to the ArgParser
parser = pl.Trainer.add_argparse_args(parser)
args = parser.parse_args()
```

And now you can start your program with

```
# now you can use any trainer flag
$ python main.py --num_nodes 2 --gpus 8
```

For a full guide on using hyperparameters, *check out the hyperparameters docs*.

1.6 Validating

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

```
from pytorch_lightning import Trainer

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

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

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

1.7 Testing

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

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

- test_step
- test_epoch_end
- test_dataloader

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

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

(continues on next page)

(continued from previous page)

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

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

```
from pytorch_lightning import Trainer

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

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

Out:

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

You can also run the test from a saved lightning model

```
model = LitMNIST.load_from_checkpoint(PATH)
trainer = Trainer(num_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.

1.8 Predicting

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

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

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

```
class MNISTClassifier(pl.LightningModule):

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

  def training_step(self, batch, batch_idx):
    x, y = batch
    logits = self.forward(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(pl.LightningModule):

  def forward(self, x):
    batch_size, channels, width, height = x.size()
    x = x.view(batch_size, -1)
    x = self.layer_1(x)
    x1 = torch.relu(x)
    x = self.layer_2(x1)
    x2 = torch.relu(x)
    x3 = self.layer_3(x2)
    return [x, x1, x2, x3]

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

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

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

```
class LitMNISTDreamer(pl.LightningModule):

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

(continues on next page)

(continued from previous page)

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

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

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

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

1.9 Extensibility

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

1.9.1 Training overrides

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

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

With your own

```
class LitMNIST(pl.LightningModule):

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

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

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

you could do your own:

```
class LitMNIST(pl.LightningModule):

    def configure_ddp(self, model, device_ids):

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

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

1.10 Callbacks

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

```
import pytorch_lightning as pl

class MyPrintingCallback(pl.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(callbacks=[MyPrintingCallback()])
```

Note: See full list of 12+ hooks in the Callback docs

1.11 Child Modules

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

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

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

class AutoEncoder(LitMNIST):
    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        self.encoder = Encoder()
        self.decoder = Decoder()

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

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

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

        loss = MSE(x, x_hat)
        return loss

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

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

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

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

and we can train this using the same trainer

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

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

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

1.12 Transfer Learning

1.12.1 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 pl.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(pl.LightningModule):
    def __init__(self):
        self.encoder = Encoder()
        self.decoder = Decoder()

class CIFAR10Classifier(pl.LightingModule):
    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!

1.12.2 Example: Imagenet (computer Vision)

```
import torchvision.models as models

class ImagenetTranferLearning(pl.LightingModule):
    def __init__(self):
        # init a pretrained resnet
        num_target_classes = 10
        self.feature_extractor = model.resnet50(
                                    pretrained=True,
                                    num_classes=num_target_classes)
        self.feature_extractor.eval()

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

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

Finetune

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

And use it to predict your data of interest

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

(continues on next page)

(continued from previous page)

```
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.

1.12.3 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.

```
from transformers import BertModel

class BertMNLIFinetuner(pl.LightningModule):

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

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


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

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

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

CHAPTER

TWO

CALLBACKS

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

An overall Lightning system should have:

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

Example

```
import pytorch_lightning as pl

class MyPrintingCallback(pl.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')

# pass to trainer
trainer = pl.Trainer(callbacks=[MyPrintingCallback()])
```

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

---

2.1 Callback Base

Abstract base class used to build new callbacks.

class pytorch_lightning.callbacks.base.**Callback**
Bases: abc.ABC

Abstract base class used to build new callbacks.

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

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

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

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

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

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

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

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

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

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

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

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

2.2 Early Stopping

Stop training when a monitored quantity has stopped improving.

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

Bases: pytorch_lightning.callbacks.base.Callback

Parameters

- **monitor** (*str*) – quantity to be monitored. Default: 'val_loss'.
- **min_delta** (*float*) – minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than *min_delta*, will count as no improvement. Default: 0.
- **patience** (*int*) – number of epochs with no improvement after which training will be stopped. Default: 0.
- **verbose** (*bool*) – verbosity mode. Default: False.

- **mode** (*str*) – one of {auto, min, max}. In *min* mode, training will stop when the quantity monitored has stopped decreasing; in *max* mode it will stop when the quantity monitored has stopped increasing; in *auto* mode, the direction is automatically inferred from the name of the monitored quantity. Default: `'auto'`.
- **strict** (*bool*) – whether to crash the training if *monitor* is not found in the metrics. Default: `True`.

Example:

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

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

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

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

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

2.3 Model Checkpointing

Automatically save model checkpoints during training.

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

Bases: `pytorch_lightning.callbacks.base.Callback`

Save the model after every epoch.

Parameters

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

  Example:

```
# no path
ModelCheckpoint()
#  saves like /my/path/epoch_0.ckpt

# save any arbitrary metrics like and val_loss, etc in name
```

(continues on next page)

(continued from previous page)

```
ModelCheckpoint(filepath='/my/path/{epoch}-{val_loss:.2f}-{other_
↪metric:.2f}')
# saves file like: /my/path/epoch=2-val_loss=0.2_other_metric=0.3.
↪ckpt
```

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

Example:

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

# saves checkpoints to my_path whenever 'val_loss' has a new min
checkpoint_callback = ModelCheckpoint(filepath='my_path')
Trainer(checkpoint_callback=checkpoint_callback)

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

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

Generate a filename according define template.

Examples

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

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

2.4 Gradient Accumulator

Change gradient accumulation factor according to scheduling.

class pytorch_lightning.callbacks.gradient_accumulation_scheduler.**GradientAccumulationSche**
Bases: pytorch_lightning.callbacks.base.Callback

Change gradient accumulation factor according to scheduling.

Parameters **scheduling** (dict) – scheduling in format {epoch: accumulation_factor} .. warning:: Epochs indexing starts from "1" until v0.6.x,

but will start from "0" in v0.8.0.

Example:

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

# at epoch 5 start accumulating every 2 batches
accumulator = GradientAccumulationScheduler(scheduling: {5: 2})
Trainer(accumulate_grad_batches=accumulator)
```

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

CHAPTER

THREE

HOOKS

3.1 Hooks

There are cases when you might want to do something different at different parts of the training/validation loop. To enable a hook, simply override the method in your LightningModule and the trainer will call it at the correct time.

Contributing If there's a hook you'd like to add, simply:

1. Fork PyTorchLightning.
2. Add the hook `pytorch_lightning.base_module.hooks.py`.
3. Add the correct place in the `pytorch_lightning.models.trainer` where it should be called.

class pytorch_lightning.core.hooks.**ModelHooks**(**args*, ***kwargs*)

Bases: `torch.nn.Module`

backward(*trainer*, *loss*, *optimizer*, *optimizer_idx*)

Override backward with your own implementation if you need to

Parameters

- **trainer** – Pointer to the trainer
- **loss** – Loss is already scaled by accumulated grads
- **optimizer** – Current optimizer being used
- **optimizer_idx** – Index of the current optimizer being used

Returns

Called to perform backward step. Feel free to override as needed.

The loss passed in has already been scaled for accumulated gradients if requested.

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

on_after_backward()

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

Returns

Called in the training loop after model.backward() This is the ideal place to inspect or log gradient information

```
def on_after_backward(self):
    # example to inspect gradient information in tensorboard
    if self.trainer.global_step % 25 == 0:  # don't make the tf file huge
        params = self.state_dict()
        for k, v in params.items():
            grads = v
            name = k
            self.logger.experiment.add_histogram(tag=name, values=grads,
                                                 global_step=self.trainer.
↪global_step)
```

on_batch_end()
Called in the training loop after the batch.

on_batch_start(*batch*)
Called in the training loop before anything happens for that batch.

Parameters **batch** –

Returns

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.

for optimizer in optimizers:

```
optimizer.step()
model.on_before_zero_grad(optimizer) # < ---- called here
optimizer.zero_grad
```

Parameters **optimizer** –

Returns

on_epoch_end()
Called in the training loop at the very end of the epoch.

on_epoch_start()
Called in the training loop at the very beginning of the epoch.

on_post_performance_check()
Called at the very end of the validation loop.

on_pre_performance_check()
Called at the very beginning of the validation loop.

on_sanity_check_start()
Called before starting evaluate .. warning:: will be deprecated. :return:

on_train_end()
Called at the end of training before logger experiment is closed :return:

on_train_start()
Called at the beginning of training before sanity check :return:

3.2 Hooks lifecycle

3.2.1 Training set-up

- init_ddp_connection
- init_optimizers
- configure_apex
- configure_ddp
- train_dataloader
- test_dataloaders
- val_dataloaders
- summarize
- restore_weights

3.2.2 Training loop

- on_epoch_start
- on_batch_start
- tbptt_split_batch
- training_step
- training_step_end (optional)
- backward
- on_after_backward
- optimizer.step()
- on_batch_end
- on_epoch_end

3.2.3 Validation loop

- model.zero_grad()
- model.eval()
- torch.set_grad_enabled(False)
- validation_step
- validation_end
- model.train()
- torch.set_grad_enabled(True)
- on_post_performance_check

3.2.4 Test loop

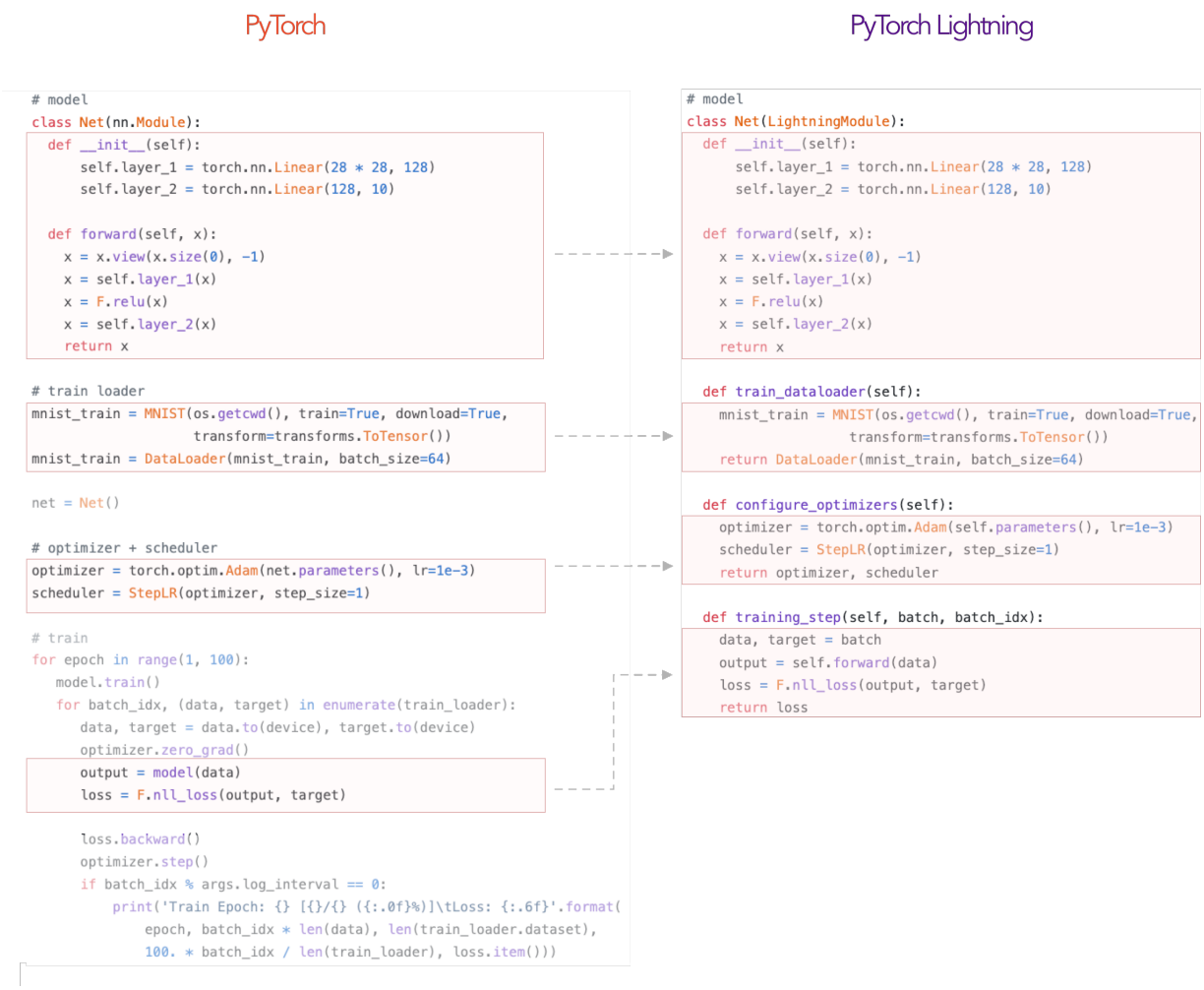
- model.zero_grad()
- model.eval()
- torch.set_grad_enabled(False)
- test_step
- test_end
- model.train()
- torch.set_grad_enabled(True)
- on_post_performance_check

CHAPTER

FOUR

LIGHTNINGMODULE

A LightningModule organizes your PyTorch code into the following sections:



Notice a few things.

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

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

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

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

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

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

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

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

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

6. **A LightingModule 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).

4.1 Minimal Example

Here are the only required methods.

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

import pytorch_lightning as pl

class LitModel(pl.LightningModule):
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

Which you can train by doing:

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

trainer.fit(model)
```

4.2 Training loop structure

The general pattern is that each loop (training, validation, test loop) has 2 methods:

- ` ___step `
- ` ___epoch_end`

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

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

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

4.2.1 Add validation loop

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

```
class LitModel(pl.LightningModule):
    def validation_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self.forward(x)
        return {'val_loss': F.cross_entropy(y_hat, y)}

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

    def val_dataloader(self):
        # can also return a list of val dataloaders
        return DataLoader(...)
```

4.2.2 Add test loop

```
class LitModel(pl.LightningModule):
    def test_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self.forward(x)
        return {'test_loss': F.cross_entropy(y_hat, y)}

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

    def test_dataloader(self):
        # can also return a list of test dataloaders
        return DataLoader(...)
```

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

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

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

4.3 Training_step_end method

When using dataParallel or distributedDataParallel2, the training_step will be operating on a portion of the batch. This is normally ok but in special cases like calculating NCE loss using negative samples, we might want to perform a softmax across all samples in the batch.

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

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

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

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

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

4.4 Remove cuda calls

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

When you init a new tensor in your code, just use type_as

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

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

4.5 Data preparation

Data preparation in PyTorch follows 5 steps:

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

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

```
def prepare_data(self):
    # download
    mnist_train = MNIST(os.getcwd(), train=True, download=True,
                        transform=transforms.ToTensor())
    mnist_test = MNIST(os.getcwd(), train=False, download=True,
                        transform=transforms.ToTensor())

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

    # assign to use in dataloaders
    self.train_dataset = mnist_train
    self.val_dataset = mnist_val
    self.test_dataset = mnist_test

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

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

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

Note: `prepare_data` is called once.

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

4.6 Lifecycle

The methods in the LightningModule are called in this order:

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

If you define a validation loop then

5. `val_dataloader`

And if you define a test loop:

6. `test_dataloader`

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

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

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

4.7 Live demo

Check out this COLAB for a live demo.

4.8 LightningModule Class

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

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

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

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

Parameters

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

Returns Apex wrapped model and optimizers

Examples

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

    return model, optimizers
```

configure_ddp(*model*, *device_ids*)

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

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

Parameters

- **model** (LightningModule) – the LightningModule currently being optimized
- **device_ids** (*list*) – the list of GPU ids

Returns DDP wrapped model

Examples

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

configure_optimizers()

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

If you don't define this method Lightning will automatically use Adam(lr=1e-3)

Return: any of these 3 options:

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

Examples

```
# most cases (default if not defined)
def configure_optimizers(self):
    opt = Adam(self.parameters(), lr=1e-3)
    return opt

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

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

# 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]
```

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,

you can input this as 'frequency' key: dict(scheduler=lr_scheduler, interval='step' or 'epoch', frequency=x)

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

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

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

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

Returns Predicted output

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.forward(x)
    logits = self.classifier(feature_maps)

    # ...
    return loss

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

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

# -------------
```

(continues on next page)

(continued from previous page)

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

freeze()
Freeze all params for inference

Example

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

get_tqdm_dict()
Additional items to be displayed in the progress bar.

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

init_ddp_connection(*proc_rank*, *world_size*)
Override to define your custom way of setting up a distributed environment.

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

Parameters

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

Examples

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

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

    except Exception as e:
        default_port = 12910

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

(continues on next page)

(continued from previous page)

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

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

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

Primary way of loading model from a checkpoint. When Lightning saves a checkpoint it stores the hyperparameters in the checkpoint if you initialized your LightningModule with an argument called *hparams* which is a Namespace (output of using argparse to parse command line arguments).

Example

```
from argparse import Namespace
hparams = Namespace(**{'learning_rate': 0.1})

model = MyModel(hparams)

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

Parameters

- **checkpoint_path** (str) – Path to checkpoint.
- **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.
- **tags_csv** (Optional[str]) – Optional path to a .csv file with two columns (key, value) as in this example:

```
key,value
drop_prob,0.2
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 .csv file with the hparams you'd like to use. These will be converted into a argparse.Namespace and passed into your LightningModule for use.

Return type LightningModule

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',
    tags_csv='/path/to/hparams_file.csv'
)

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

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

> **Warning:**
>
> **Deprecated in version 0.7.0. You should use *load_from_checkpoint* instead.** Will be removed in v0.9.0.

on_load_checkpoint(*checkpoint*)

Called by lightning to restore your model. If you saved something with **on_save_checkpoint** this is your chance to restore this.

Parameters **checkpoint** (*dict*) – Loaded checkpoint

Example

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

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

on_save_checkpoint(*checkpoint*)

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

Parameters **checkpoint** (*dic*) – Checkpoint to be saved

Example

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

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

optimizer_step(*epoch*, *batch_idx*, *optimizer*, *optimizer_idx*, *second_order_closure=None*)
Override this method to adjust the default way the Trainer calls each optimizer. By default, Lightning calls .step() and zero_grad() as shown in the example once per optimizer.

Parameters

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

Examples

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

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

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

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

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

```
# learning rate warm-up
def optimizer_step(self, current_epoch, batch_idx, optimizer,
                   optimizer_idx, second_order_closure=None):
```

(continues on next page)

(continued from previous page)

```
    # warm up lr
    if self.trainer.global_step < 500:
        lr_scale = min(1., float(self.trainer.global_step + 1) / 500.)
        for pg in optimizer.param_groups:
            pg['lr'] = lr_scale * self.hparams.learning_rate

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

prepare_data()

Use this to download and prepare data. In distributed (GPU, TPU), this will only be called once

Returns PyTorch DataLoader

This is called before requesting the dataloaders

```
model.prepare_data()
model.train_dataloader()
model.val_dataloader()
model.test_dataloader()
```

Examples

```
def prepare_data(self):
    download_imagenet()
    clean_imagenet()
    cache_imagenet()
```

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

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

Parameters **x** (*object*) – The thing to print

Examples: .. code-block:: python

example if we were using this model as a feature extractor def forward(self, x):

self.print(x, 'in loader')

tbptt_split_batch(*batch*, *split_size*)

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

Parameters

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

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

Examples

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

          batch_split.append(split_x)

      splits.append(batch_split)

  return splits
```

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

---

test_dataloader()

Return a dataloader. It will not be called every epoch unless you set `` `Trainer(reload_dataloaders_every_epoch=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. No need to set yourself.

Returns PyTorch DataLoader

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.hparams.batch_size,
        shuffle=True
    )

    return loader
```

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

test_end(*outputs*)

Warning: Deprecated in v0.7.0. use test_epoch_end instead. Will be removed 1.0.0

test_epoch_end(*outputs*)
Called at end of 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*) – List of outputs you defined in test_step, or if there are multiple
- **a list containing a list of outputs for each dataloader** (*dataloaders,*) –

Returns Dict has the following optional keys: progress_bar -> Dict for progress bar display. Must have only tensors log -> Dict of metrics to add to logger. Must have only tensors (no images, etc)

Return type Dict or OrderedDict (dict)

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

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

Examples

With a single dataloader

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

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

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

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

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

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

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

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

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

Parameters

- **batch** (*torch.nn.Tensor | (Tensor, Tensor) | [Tensor, Tensor]*) – The output of your dataloader. A tensor, tuple or list
- **batch_idx** (*int*) – The index of this batch

- **dataloader_idx** (*int*) – The index of the dataloader that produced this batch (only if multiple test datasets used)

Returns Dict or OrderedDict - passed to the test_step_end

```
# 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.forward(x)
    loss = self.loss(out, y)

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

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

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

    # return an optional dict
    return output
```

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

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

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

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

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

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

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

---

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

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

Returns Dict or OrderedDict - passed to the test_epoch_end

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

Examples

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

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

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

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

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

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

See also:

see the multi-gpu guide for more details.

tng_dataloader()

Implement a PyTorch DataLoader.

> **Warning:** Deprecated in v0.5.0. use train_dataloader instead. Will be removed 1.0.0

train_dataloader()
Implement a PyTorch DataLoader

Returns PyTorch DataLoader

Return a dataloader. It will not be called every epoch unless you set `Trainer(reload_dataloaders_every_epoch=True)`.

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

Note: Lightning adds the correct sampler for distributed and arbitrary hardware. No need to set yourself.

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

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.hparams.batch_size,
        shuffle=True
    )
    return loader
```

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

> **Warning:** Deprecated in v0.7.0. use training_step_end instead

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

Parameters

- **batch** (*torch.nn.Tensor | (Tensor, 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*) – If using multiple optimizers, this argument will also be present.
- **hiddens** (– Tensor): Passed in if truncated_bptt_steps > 0.

Returns

dict with loss key and optional log, progress keys if implementing training_step, return whatever you need in that step:

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

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

Examples

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

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

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

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

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

    # return a dict
    return output
```

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

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

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

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

    return {
```

(continues on next page)

(continued from previous page)

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

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

training_step_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

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

Return type dictionary with loss key and optional log, progress keys

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

Examples

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

See also:

see the multi-gpu guide for more details.

unfreeze()

Unfreeze all params for training.

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

val_dataloader()

Return a dataloader. It will not be called every epoch unless you set `Trainer(reload_dataloaders_every_epoch=True)`.

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

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

Note: Lightning adds the correct sampler for distributed and arbitrary hardware No need to set yourself.

Returns PyTorch DataLoader

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.hparams.batch_size,
        shuffle=True
    )

    return loader

# can also return multiple dataloaders
```

(continues on next page)

(continued from previous page)

```
def val_dataloader(self):
    return [loader_a, loader_b, ..., loader_n]
```

```
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.hparams.batch_size,
        shuffle=True
    )

    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 *val_dataloaders*, the *validation_step* will have an argument *dataset_idx* which matches the order here.

validation_end(*outputs*)

Warning: Deprecated in v0.7.0. use validation_epoch_end instead. Will be removed 1.0.0

validation_epoch_end(*outputs*)

Called at end of validation epoch with the output of all validation_steps

```
# the pseudocode for these calls

val_outs = []
for val_batch in val_data:
    out = validation_step(train_batch)
    train_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters

- **outputs** (list) – List of outputs you defined in validation_step, or if there are multiple
- **a list containing a list of outputs for each dataloader** (*dataloaders,*) –

Returns Dict has the following optional keys: progress_bar -> Dict for progress bar display. Must have only tensors log -> Dict of metrics to add to logger. Must have only tensors (no images, etc)

Return type Dict or OrderedDict (dict)

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

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

Examples

With a single dataloader

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

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

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

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

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

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

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

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

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

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

Parameters

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

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

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

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

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

Examples

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

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

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

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

    # all optional...
    # return whatever you need for the collation function validation_end
    output = OrderedDict({
```

(continues on next page)

(continued from previous page)

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

    # return an optional dict
    return output
```

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

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

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

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

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

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

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

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

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

Returns Dict or OrderedDict - passed to the validation_epoch_end

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

Examples

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

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

(continues on next page)

(continued from previous page)

```
    loss = nce_loss(loss)
    return {'loss': loss}

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

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

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

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

See also:

see the multi-gpu guide for more details.

current_epoch = None
The current epoch

dtype = None
Current dtype

global_step = None
Total training batches seen across all epochs

logger = None
Pointer to the logger object

on_gpu = None
True if your model is currently running on GPUs. Useful to set flags around the LightningModule for different CPU vs GPU behavior.

trainer = None
Pointer to the trainer object

use_amp = None
True if using amp

use_ddp = None
True if using ddp

use_ddp2 = None
True if using ddp2

use_dp = None
True if using dp

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

Parameters **fn** –

Returns

CHAPTER

FIVE

LOGGERS

Lightning supports most popular logging frameworks (Tensorboard, comet, weights and biases, etc...). To use a logger, simply pass it into the trainer. To use multiple loggers, simply pass in a `list` or `tuple` of loggers.

```
from pytorch_lightning import loggers

# lightning uses tensorboard by default
tb_logger = loggers.TensorBoardLogger()
trainer = Trainer(logger=tb_logger)

# or choose from any of the others such as MLFlow, Comet, Neptune, Wandb
comet_logger = loggers.CometLogger()
trainer = Trainer(logger=comet_logger)

# or pass a list
tb_logger = loggers.TensorBoardLogger()
comet_logger = loggers.CometLogger()
trainer = Trainer(logger=[tb_logger, comet_logger])
```

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

5.1 Custom logger

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

```
from pytorch_lightning.loggers import LightningLoggerBase, rank_zero_only

class MyLogger(LightningLoggerBase):

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

5.2 Using loggers

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

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

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

Read more in the Experiment Logging use case.

5.3 Supported Loggers

class pytorch_lightning.loggers.**TensorBoardLogger**(*save_dir*, *name='default'*, *version=None*, ***kwargs*)

Bases: pytorch_lightning.loggers.base.LightningLoggerBase

Log to local file system in TensorBoard format

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

Example

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

Parameters

- **save_dir** (*str*) – Save directory
- **name** (*str*) – Experiment name. Defaults to "default". If it is the empty string then no per-experiment subdirectory is used.

- **version** (*int | str*) – Experiment version. If version is not specified the logger inspects the save directory for existing versions, then automatically assigns the next available version. If it is a string then it is used as the run-specific subdirectory name, otherwise version_${version} is used.
- ****kwargs** (*dict*) – Other arguments are passed directly to the `SummaryWriter` constructor.

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

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

Return type `None`

log_hyperparams(*params*)
Record hyperparameters.

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

Return type `None`

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

Parameters

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

Return type `None`

save()
Save log data.

Return type `None`

property experiment
Actual tensorboard object. To use tensorboard features 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 checkpoint will be saved in save_dir/version_dir

Return type str

property version
Return the experiment version.

Return type int

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

Bases: pytorch_lightning.loggers.base.LightningLoggerBase

Log using comet.ml.

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

```
# ONLINE MODE
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["COMET_API_KEY"],
    workspace=os.environ["COMET_WORKSPACE"], # Optional
    project_name="default_project", # Optional
    rest_api_key=os.environ["COMET_REST_API_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=".",
    workspace=os.environ["COMET_WORKSPACE"], # Optional
    project_name="default_project", # Optional
    rest_api_key=os.environ["COMET_REST_API_KEY"], # Optional
    experiment_name="default" # Optional
)
trainer = Trainer(logger=comet_logger)
```

Parameters

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

- **rest_api_key** (*str*) – Optional. Rest API key found in Comet.ml settings. This is used to determine version number
- **experiment_name** (*str*) – Optional. String representing the name for this particular experiment on Comet.ml.
- **experiment_key** (*str*) – Optional. If set, restores from existing experiment.

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

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

Return type None

log_hyperparams(*params*)
Record hyperparameters.

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

Return type None

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

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 do the following.

Example:

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

Return type BaseExperiment

property name
Return the experiment name.

Return type str

property version
Return the experiment version.

Return type str

class pytorch_lightning.loggers.**MLFlowLogger**(*experiment_name*, *tracking_uri=None*, *tags=None*)
Bases: pytorch_lightning.loggers.base.LightningLoggerBase

Logs using MLFlow

Parameters

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

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

Parameters **status** (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]) – argparse.Namespace containing the hyperparameters

Return type None

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

Parameters

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

Return type None

save()
Save log data.

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

Example:

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

Return type MlflowClient

property name
Return the experiment name.

Return type str

property version
Return the experiment version.

Return type str

class pytorch_lightning.loggers.**NeptuneLogger**(*api_key=None*, *project_name=None*, *offline_mode=False*, *experiment_name=None*, *upload_source_files=None*, *params=None*, *properties=None*, *tags=None*, ***kwargs*)
Bases: pytorch_lightning.loggers.base.LightningLoggerBase

Neptune logger can be used in the online mode or offline (silent) mode. To log experiment data in online mode, NeptuneLogger requries an API key:

Initialize a neptune.ml logger.

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

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

neptune_logger = NeptuneLogger(
    api_key=os.environ["NEPTUNE_API_TOKEN"],
    project_name="USER_NAME/PROJECT_NAME",
    experiment_name="default", # Optional,
    params={"max_epochs": 10}, # Optional,
    tags=["pytorch-lightning","mlp"] # Optional,
)
trainer = Trainer(max_epochs=10, logger=neptune_logger)
```

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

neptune_logger = NeptuneLogger(
    project_name="USER_NAME/PROJECT_NAME",
    experiment_name="default", # Optional,
    params={"max_epochs": 10}, # Optional,
    tags=["pytorch-lightning","mlp"] # Optional,
)
trainer = Trainer(max_epochs=10, logger=neptune_logger)
```

Use the logger anywhere in you LightningModule as follows:

```
def train_step(...):
    # example
    self.logger.experiment.log_metric("acc_train", acc_train) # log metrics
    self.logger.experiment.log_image("worse_predictions", prediction_image) # log␣
↪images
    self.logger.experiment.log_artifact("model_checkpoint.pt", prediction_image)
↪# log model checkpoint
    self.logger.experiment.whatever_neptune_supports(...)

def any_lightning_module_function_or_hook(...):
    self.logger.experiment.log_metric("acc_train", acc_train) # log metrics
    self.logger.experiment.log_image("worse_predictions", prediction_image) # log␣
↪images
    self.logger.experiment.log_artifact("model_checkpoint.pt", prediction_image)
↪# log model checkpoint
    self.logger.experiment.whatever_neptune_supports(...)
```

Parameters

- **api_key** (*str | None*) – Required in online mode. Neputne API token, found on https://neptune.ml. Read how to get your API key https://docs.neptune.ml/python-api/tutorials/get-started.html#copy-api-token.

- **project_name** (*str*) – Required in online mode. Qualified name of a project in a form of "namespace/project_name" for example "tom/minst-classification". If None, the value of NEPTUNE_PROJECT environment variable will be taken. You need to create the project in https://neptune.ml first.
- **offline_mode** (*bool*) – Optional default False. If offline_mode=True no logs will be send to neptune. Usually used for debug purposes.
- **experiment_name** (*str|None*) – Optional. Editable name of the experiment. Name is displayed in the experiment's Details (Metadata section) and in experiments view as a column.
- **upload_source_files** (*list|None*) – Optional. List of source files to be uploaded. Must be list of str or single str. Uploaded sources are displayed in the experiment's Source code tab. If None is passed, Python file from which experiment was created will be uploaded. Pass empty list ([]) to upload no files. Unix style pathname pattern expansion is supported. For example, you can pass '*.py'

  to upload all python source files from the current directory.

  For recursion lookup use '**/*.py' (for Python 3.5 and later). For more information see glob library.
- **params** (*dict|None*) – Optional. Parameters of the experiment. After experiment creation params are read-only. Parameters are displayed in the experiment's Parameters section and each key-value pair can be viewed in experiments view as a column.
- **properties** (*dict|None*) – Optional default is {}. Properties of the experiment. They are editable after experiment is created. Properties are displayed in the experiment's Details and each key-value pair can be viewed in experiments view as a column.
- **tags** (*list|None*) – Optional default []. Must be list of str. Tags of the experiment. They are editable after experiment is created (see: append_tag() and remove_tag()). Tags are displayed in the experiment's Details and can be viewed in experiments view as a column.

append_tags(*tags*)

appends tags to neptune experiment

Parameters **tags** (Union[str, Iterable[str]]) – Tags to add to the current experiment. If str is passed, singe tag is added. If multiple - comma separated - str are passed, all of them are added as tags. If list of str is passed, all elements of the list are added as tags.

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 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]) – argparse.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** (*str|PIL.Image|matplotlib.figure.Figure*) – The value of the log (data-point). Can be one of the following types: PIL image, matplotlib.figure.Figure, path to image file (str)
- **step** (Optional[int]) – Step number at which the metrics should be recorded, must be strictly increasing

Return type None

log_metric(*metric_name*, *metric_value*, *step=None*)
Log metrics (numeric values) in Neptune experiments

Parameters

- **metric_name** (str) – The name of log, i.e. mse, loss, accuracy.
- **metric_value** (Union[Tensor, float, str]) – The value of the log (data-point).
- **step** (Optional[int]) – Step number at which the metrics should be recorded, must be strictly increasing

Return type None

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

Parameters

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

Return type None

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

Parameters

- **log_name** (str) – The name of log, i.e. mse, my_text_data, timing_info.
- **text** (str) – The value of the log (data-point).
- **step** (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 do the following.

Example:

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

Return type Experiment

property name

Return the experiment name.

Return type str

property version

Return the experiment version.

Return type str

class pytorch_lightning.loggers.**TestTubeLogger**(*save_dir*, *name='default'*, *description=None*, *debug=False*, *version=None*, *create_git_tag=False*)

Bases: pytorch_lightning.loggers.base.LightningLoggerBase

Log to local file system in TensorBoard format but using a nicer folder structure. (see full docs).

Example

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

Use the logger anywhere in you LightningModule as follows:

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

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

Parameters

- **save_dir** (*str*) – Save directory
- **name** (*str*) – Experiment name. Defaults to “default”.
- **description** (*str*) – A short snippet about this experiment
- **debug** (*bool*) – If True, it doesn’t log anything
- **version** (*int*) – Experiment version. If version is not specified the logger inspects the save

- **for existing versions, then automatically assigns the next available version.** (*directory*) –
- **create_git_tag** (*bool*) – If True creates a git tag to save the code used in this experiment

close()
Do any cleanup that is necessary to close an experiment.

Return type None

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

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

Return type None

log_hyperparams(*params*)
Record hyperparameters.

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

Return type None

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

Parameters

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

Return type None

save()
Save log data.

Return type None

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

Example:

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

Return type Experiment

property name
Return the experiment name.

Return type str

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

Return type int

property version
Return the experiment version.

Return type int

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

Bases: pytorch_lightning.loggers.base.LightningLoggerBase

Logger for W&B.

Parameters

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

Example

```
from pytorch_lightning.loggers import WandbLogger
from pytorch_lightning import Trainer

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

finalize(*status='success'*)

Do any processing that is necessary to finalize an experiment.

Parameters **status** (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]) – argparse.Namespace containing the hyperparameters

Return type None

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

Record metrics.

Parameters

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

Return type None

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

Example:

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

Return type Run

property name
Return the experiment name.

Return type str

property version
Return the experiment version.

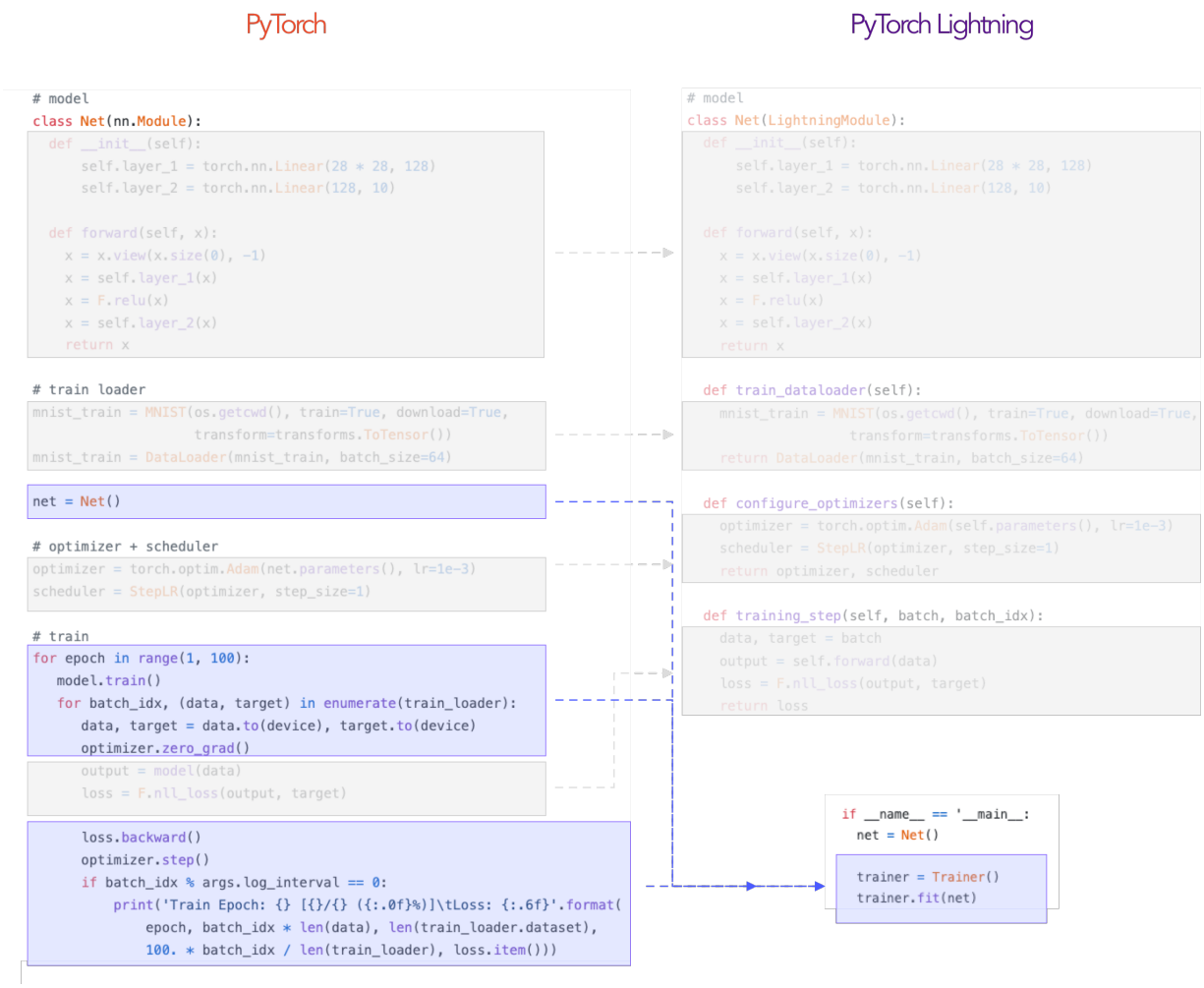
Return type str

CHAPTER

SIX

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.

6.1 Basic use

This is the basic use of the trainer:

```
from pytorch_lightning import Trainer

model = MyLightningModule()

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

6.2 Best Practices

For cluster computing, it's recommended you structure your main.py file this way

```
from argparser import AugumentParser

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:distributed_backend

```
$ python main.py --gpus 2
```

6.3 Testing

Once you're done training, feel free to run the test set! (Only right before publishing your paper or pushing to production)

```
trainer.test()
```

6.4 Deployment / prediction

You just trained a LightningModule which is also just a torch.nn.Module. Use it to do whatever!

```
# load model
pretrained_model = LightningModule.load_from_checkpoint(PATH)
pretrained_model.freeze()

# use it for finetuning
def forward(self, x):
    features = pretrained_model(x)
    classes = classifier(features)

# or for prediction
out = pretrained_model(x)
api_write({'response': out}
```

6.5 Trainer flags

6.5.1 accumulate_grad_batches

Accumulates grads every k batches or as set up in the dict.

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

Example:

```
# accumulate every 4 batches (effective batch size is batch*4)
trainer = Trainer(accumulate_grad_batches=4)

# no accumulation for epochs 1-4. accumulate 3 for epochs 5-10. accumulate 20 after␣
↪that
trainer = Trainer(accumulate_grad_batches={5: 3, 10: 20})
```

6.5.2 amp_level

The optimization level to use (O1, O2, etc...) for 16-bit GPU precision (using NVIDIA apex under the hood).

Check NVIDIA apex docs for level

Example:

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

6.5.3 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)
```

6.5.4 callbacks

Add a list of user defined callbacks.

Note: Only user defined callbacks (ie: Not EarlyStopping or ModelCheckpoint)

```
# a list of callbacks
callbacks = [PrintCallback()]
trainer = Trainer(callbacks=callbacks)
```

Example:

```
from pytorch_lightning.callbacks import Callback

class PrintCallback(Callback):
    def on_train_start(self):
        print("Training is started!")
    def on_train_end(self):
        print(f"Training is done. The logs are: {self.trainer.logs}")
```

6.5.5 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)
```

6.5.6 checkpoint_callback

Callback for checkpointing.

```
trainer = Trainer(checkpoint_callback=checkpoint_callback)
```

Example:

```
from pytorch_lightning.callbacks import ModelCheckpoint

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

6.5.7 default_save_path

Default path for logs and weights when no logger or `pytorch_lightning.callbacks.ModelCheckpoint` callback passed. On certain clusters you might want to separate where logs and checkpoints are stored. If you don't then use this method for convenience.

Example:

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

6.5.8 distributed_backend

The distributed backend to use.

- (`dp`) is DataParallel (split batch among GPUs of same machine)
- (`ddp`) is DistributedDataParallel (each gpu on each node trains, and syncs grads)
- **(`ddp2`) dp on node, ddp across nodes. Useful for things like increasing** the number of negative samples

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

Example:

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

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

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

6.5.9 early_stop_callback

Callback for early stopping. early_stop_callback (`pytorch_lightning.callbacks.EarlyStopping`)

- **`True`: A default callback monitoring `'val_loss'` is created.** Will raise an error if `'val_loss'` is not found.
- `False`: Early stopping will be disabled.
- `None`: The default callback monitoring `'val_loss'` is created.
- Default: `None`.

```
trainer = Trainer(early_stop_callback=early_stop_callback)
```

Example:

```
from pytorch_lightning.callbacks import EarlyStopping

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

Note: If `'val_loss'` is not found will work as if early stopping is disabled.

6.5.10 fast_dev_run

Runs 1 batch of train, test and val to find any bugs (ie: a sort of unit test).

Under the hood the pseudocode looks like this:

```
# loading
__init__()
prepare_data

# test training step
training_batch = next(train_dataloader)
training_step(training_batch)

# test val step
val_batch = next(val_dataloader)
out = validation_step(val_batch)
validation_epoch_end([out])
```

Example:

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

# runs 1 train, val, test  batch and program ends
trainer = Trainer(fast_dev_run=True)
```

6.5.11 gpus

- Number of GPUs to train on
- or Which GPUs to train on
- can handle strings

Example:

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

# int: train on 2 gpus
trainer = Trainer(gpus=2)

# list: train on GPUs 1, 4 (by bus ordering)
trainer = Trainer(gpus=[1, 4])
trainer = Trainer(gpus='1, 4') # equivalent

# -1: train on all gpus
trainer = Trainer(gpus=-1)
trainer = Trainer(gpus='-1') # equivalent

# combine with num_nodes to train on multiple GPUs across nodes
# uses 8 gpus in total
trainer = Trainer(gpus=2, num_nodes=4)
```

Note: See the *multi-gpu computing guide*

6.5.12 gradient_clip_val

Gradient clipping value

- 0 means don't clip.

Example:

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

gradient_clip:

Warning: Deprecated since version 0.5.0.

Use *gradient_clip_val* instead. Will remove 0.8.0.

6.5.13 log_gpu_memory

Options:

- None
- 'min_max'
- 'all'

Example:

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

# log all the GPUs (on master node only)
trainer = Trainer(log_gpu_memory='all')

# log only the min and max memory on the master node
trainer = Trainer(log_gpu_memory='min_max')
```

Note: Might slow performance because it uses the output of nvidia-smi.

6.5.14 log_save_interval

Writes logs to disk this often.

Example:

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

6.5.15 logger

Logger (or iterable collection of loggers) for experiment tracking.

```
Trainer(logger=logger)
```

Example:

```
from pytorch_lightning.loggers import TensorBoardLogger

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

6.5.16 max_epochs

Stop training once this number of epochs is reached

Example:

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

max_nb_epochs:

Warning: Deprecated since version 0.5.0.

Use *max_epochs* instead. Will remove 0.8.0.

6.5.17 min_epochs

Force training for at least these many epochs

Example:

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

min_nb_epochs:

Warning: deprecated:: 0.5.0 Use *min_epochs* instead. Will remove 0.8.0.

6.5.18 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)
```

Example:

```
# Stop after 100 steps
trainer = Trainer(max_steps=100)
```

6.5.19 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)
```

Example:

```
# Run at least for 100 steps (disable min_epochs)
trainer = Trainer(min_steps=100, min_epochs=0)
```

6.5.20 num_nodes

Number of GPU nodes for distributed training.

Example:

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

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

nb_gpu_nodes:

> **Warning:** Deprecated since version 0.5.0.
>
> Use *num_nodes* instead. Will remove 0.8.0.

6.5.21 num_sanity_val_steps

Sanity check runs n batches of val before starting the training routine. This catches any bugs in your validation without having to wait for the first validation check. The Trainer uses 5 steps by default. Turn it off or modify it here.

Example:

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

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

nb_sanity_val_steps:

> **Warning:** Deprecated since version 0.5.0.
>
> Use *num_sanity_val_steps* instead. Will remove 0.8.0.

6.5.22 num_tpu_cores

How many TPU cores to train on (1 or 8).

A single TPU v2 or v3 has 8 cores. A TPU pod has up to 2048 cores. A slice of a POD means you get as many cores as you request.

Your effective batch size is batch_size * total tpu cores.

Note: No need to add a DistributedDataSampler, Lightning automatically does it for you.

This parameter can be either 1 or 8.

Example:

```
# your_trainer_file.py

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

# int: train on a single core
trainer = Trainer(num_tpu_cores=1)

# int: train on all cores few cores
trainer = Trainer(num_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(num_tpu_cores=8)

# -1: train on all available TPUs
trainer = Trainer(num_tpu_cores=-1)
```

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
```

6.5.23 overfit_pct

Uses this much data of all datasets. Useful for quickly debugging or trying to overfit on purpose

Example:

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

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

6.5.24 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.

Example:

```
# default used by the Trainer
trainer = Trainer(precision=32)

# 16-bit precision
trainer = Trainer(precision=16)
```

(continues on next page)

(continued from previous page)

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

6.5.25 print_nan_grads

Prints gradients with nan values

Example:

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

6.5.26 process_position

Orders the tqdm bar. Useful when running multiple trainers on the same node.

Example:

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

6.5.27 profiler

To profile individual steps during training and assist in identifying bottlenecks.

See the *profiler documentation*. for more details.

Example:

```
from pytorch_lightning.profiler import Profiler, AdvancedProfiler

# default used by the Trainer
trainer = Trainer(profiler=None)

# to profile standard training events
trainer = Trainer(profiler=True)

# equivalent to profiler=True
profiler = Profiler()
trainer = Trainer(profiler=profiler)

# advanced profiler for function-level stats
profiler = AdvancedProfiler()
trainer = Trainer(profiler=profiler)
```

6.5.28 progress_bar_refresh_rate

How often to refresh progress bar (in steps). Faster refresh rates (lower number), in notebooks is known to crash them because of their screen refresh rates. 50 is optimal for those cases.

Example:

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

6.5.29 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:
```

6.5.30 resume_from_checkpoint

To resume training from a specific checkpoint pass in the path here.k

Example:

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

# resume from a specific checkpoint
trainer = Trainer(resume_from_checkpoint='some/path/to/my_checkpoint.ckpt')
```

6.5.31 row_log_interval

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

Example:

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

add_row_log_interval:

Warning: Deprecated since version 0.5.0.

Use *row_log_interval* instead. Will remove 0.8.0.

use_amp:

> **Warning:** Deprecated since version 0.7.0.
>
> Use *precision* instead. Will remove 0.9.0.

6.5.32 show_progress_bar

If true shows tqdm progress bar

Example:

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

6.5.33 test_percent_check

How much of test dataset to check.

Example:

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

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

6.5.34 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)
```

Example:

```
# 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)
```

6.5.35 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)
```

Example:

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

6.5.36 train_percent_check

How much of training dataset to check. Useful when debugging or testing something that happens at the end of an epoch.

Example:

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

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

6.5.37 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.")

Example:

```
# default used by the Trainer (ie: disabled)
trainer = Trainer(truncated_bptt_steps=None)

# backprop every 5 steps in a batch
trainer = Trainer(truncated_bptt_steps=5)
```

Note: Make sure your batches have a sequence dimension.

Lightning takes care to split your batch along the time-dimension.

```
# we use the second as the time dimension
# (batch, time, ...)
sub_batch = batch[0, 0:t, ...]
```

Using this feature requires updating your LightningModule's `pytorch_lightning.core.LightningModule.training_step()` to include a *hiddens* arg with the hidden

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

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

To modify how the batch is split, override `pytorch_lightning.core.LightningModule.tbptt_split_batch()`:

```
class LitMNIST(pl.LightningModule):
    def tbptt_split_batch(self, batch, split_size):
        # do your own splitting on the batch
        return splits
```

6.5.38 val_percent_check

How much of validation dataset to check. Useful when debugging or testing something that happens at the end of an epoch.

Example:

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

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

6.5.39 weights_save_path

Directory of where to save weights if specified.

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

Example:

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

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

6.5.40 weights_summary

Prints a summary of the weights when training begins. Options: 'full', 'top', None.

Example:

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

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

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

6.6 Trainer class

class pytorch_lightning.trainer.**Trainer**(*logger=True*, *checkpoint_callback=True*, *early_stop_callback=False*, *callbacks=[]*, *default_save_path=None*, *gradient_clip_val=0*, *gradient_clip=None*, *process_position=0*, *nb_gpu_nodes=None*, *num_nodes=1*, *gpus=None*, *num_tpu_cores=None*, *log_gpu_memory=None*, *show_progress_bar=True*, *progress_bar_refresh_rate=50*, *overfit_pct=0.0*, *track_grad_norm=-1*, *check_val_every_n_epoch=1*, *fast_dev_run=False*, *accumulate_grad_batches=1*, *max_nb_epochs=None*, *min_nb_epochs=None*, *max_epochs=1000*, *min_epochs=1*, *max_steps=None*, *min_steps=None*, *train_percent_check=1.0*, *val_percent_check=1.0*, *test_percent_check=1.0*, *val_check_interval=1.0*, *log_save_interval=100*, *row_log_interval=10*, *add_row_log_interval=None*, *distributed_backend=None*, *use_amp=False*, *precision=32*, *print_nan_grads=False*, *weights_summary='full'*, *weights_save_path=None*, *amp_level='O1'*, *nb_sanity_val_steps=None*, *num_sanity_val_steps=5*, *truncated_bptt_steps=None*, *resume_from_checkpoint=None*, *profiler=None*, *benchmark=False*, *reload_dataloaders_every_epoch=False*, ***kwargs*)

Bases: pytorch_lightning.trainer.training_io.TrainerIOMixin, pytorch_lightning.trainer.distrib_parts.TrainerDPMixin, pytorch_lightning.trainer.distrib_data_parallel.TrainerDDPMixin, pytorch_lightning.trainer.logging.TrainerLoggingMixin, pytorch_lightning.trainer.model_hooks.TrainerModelHooksMixin, pytorch_lightning.trainer.training_tricks.TrainerTrainingTricksMixin, pytorch_lightning.trainer.data_loading.TrainerDataLoadingMixin, pytorch_lightning.trainer.auto_mix_precision.

```
TrainerAMPMixin,                    pytorch_lightning.trainer.evaluation_loop.
TrainerEvaluationLoopMixin,           pytorch_lightning.trainer.training_loop.
TrainerTrainLoopMixin,              pytorch_lightning.trainer.callback_config.
TrainerCallbackConfigMixin,           pytorch_lightning.trainer.callback_hook.
TrainerCallbackHookMixin,           pytorch_lightning.trainer.deprecated_api.
TrainerDeprecatedAPITillVer0_8
```

Customize every aspect of training via flags

Parameters

- **logger** (Union[LightningLoggerBase, Iterable[LightningLoggerBase], bool]) – Logger (or iterable collection of loggers) for experiment tracking.
- **checkpoint_callback** (Union[ModelCheckpoint, bool]) – Callback for checkpointing.
- **early_stop_callback** (pytorch_lightning.callbacks.EarlyStopping) –
- **callbacks** (List[Callback]) – Add a list of callbacks.
- **default_save_path** (Optional[str]) – Default path for logs and weights when no logger/ckpt_callback passed
- **gradient_clip_val** (float) – 0 means don't clip.
- **gradient_clip** –

  Warning: deprecated 0.7.0 Use *gradient_clip_val* instead. Will remove 0.9.0.

- **process_position** (int) – orders the tqdm bar when running multiple models on same machine.
- **num_nodes** (int) – number of GPU nodes for distributed training.
- **nb_gpu_nodes** –

  Warning: Deprecated since version 0.7.0.

  Use *num_nodes* instead. Will remove 0.9.0.

- **gpus** (Union[List[int], str, int, None]) – Which GPUs to train on.
- **num_tpu_cores** (Optional[int]) – How many TPU cores to train on (1 or 8).
- **log_gpu_memory** (Optional[str]) – None, 'min_max', 'all'. Might slow performance
- **show_progress_bar** (bool) – If true shows tqdm progress bar
- **progress_bar_refresh_rate** (int) – How often to refresh progress bar (in steps)
- **track_grad_norm** (int) – -1 no tracking. Otherwise tracks that norm
- **check_val_every_n_epoch** (int) – Check val every n train epochs.
- **fast_dev_run** (bool) – runs 1 batch of train, test and val to find any bugs (ie: a sort of unit test).

- **accumulate_grad_batches** (Union[int, Dict[int, int], List[list]]) – Accumulates grads every k batches or as set up in the dict.
- **max_epochs** (int) – Stop training once this number of epochs is reached.
- **max_nb_epochs** –

  Warning: Deprecated since version 0.7.0.

  Use *max_epochs* instead. Will remove 0.9.0.

- **min_epochs** (int) – Force training for at least these many epochs
- **min_nb_epochs** –

  Warning: Deprecated since version 0.7.0.

  Use *min_epochs* instead. Will remove 0.9.0.

- **max_steps** (Optional[int]) – Stop training after this number of steps. Disabled by default (None).
- **min_steps** (Optional[int]) – Force training for at least these number of steps. Disabled by default (None).
- **train_percent_check** (float) – How much of training dataset to check.
- **val_percent_check** (float) – How much of validation dataset to check.
- **test_percent_check** (float) – How much of test dataset to check.
- **val_check_interval** (float) – How often within one training epoch to check the validation set
- **log_save_interval** (int) – Writes logs to disk this often
- **row_log_interval** (int) – How often to add logging rows (does not write to disk)
- **add_row_log_interval** –

  Warning: Deprecated since version 0.7.0.

  Use *row_log_interval* instead. Will remove 0.9.0.

- **distributed_backend** (Optional[str]) – The distributed backend to use.
- **use_amp** –

  Warning: Deprecated since version 0.7.0.

  Use *precision* instead. Will remove 0.9.0.

- **precision** (int) – Full precision (32), half precision (16).
- **print_nan_grads** (bool) – Prints gradients with nan values
- **weights_summary** (str) – Prints a summary of the weights when training begins.

- **weights_save_path** (Optional[str]) – Where to save weights if specified.
- **amp_level** (str) – The optimization level to use (O1, O2, etc...).
- **num_sanity_val_steps** (int) – Sanity check runs n batches of val before starting the training routine.
- **nb_sanity_val_steps** –

  Warning: Deprecated since version 0.7.0.

  Use *num_sanity_val_steps* instead. Will remove 0.8.0.

- **truncated_bptt_steps** (Optional[int]) – Truncated back prop breaks performs backprop every k steps of
- **resume_from_checkpoint** (Optional[str]) – To resume training from a specific checkpoint pass in the path here.k
- **profiler** (Optional[BaseProfiler]) – To profile individual steps during training and assist in
- **reload_dataloaders_every_epoch** (bool) – Set to True to reload dataloaders every epoch
- **benchmark** (bool) – If true enables cudnn.benchmark.

classmethod add_argparse_args(*parent_parser*)
Extend existing argparse by default *Trainer* attributes.

Return type ArgumentParser

fit(*model*, *train_dataloader=None*, *val_dataloaders=None*, *test_dataloaders=None*)
Runs the full optimization routine.

Parameters

- **model** (LightningModule) – Model to fit.
- **train_dataloader** (Optional[DataLoader]) – A Pytorch DataLoader with training samples. If the model has a predefined train_dataloader method this will be skipped.
- **val_dataloaders** (Optional[DataLoader]) – 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_dataloaders** (Optional[DataLoader]) – Either a single Pytorch Dataloader or a list of them, specifying validation samples. If the model has a predefined test_dataloaders method this will be skipped

Example:

```
# Option 1,
# Define the train_dataloader(), test_dataloader() and val_dataloader() fxs
# in the lightningModule
# RECOMMENDED FOR MOST RESEARCH AND APPLICATIONS TO MAINTAIN READABILITY
trainer = Trainer()
model = LightningModule()
trainer.fit(model)
```

(continues on next page)

(continued from previous page)

```
# Option 2
# in production cases we might want to pass different datasets to the same␣
↪model
# Recommended for PRODUCTION SYSTEMS
train, val, test = DataLoader(...), DataLoader(...), DataLoader(...)
trainer = Trainer()
model = LightningModule()
trainer.fit(model, train_dataloader=train,
            val_dataloader=val, test_dataloader=test)

# Option 1 & 2 can be mixed, for example the training set can be
# defined as part of the model, and validation/test can then be
# feed to .fit()
```

test(*model=None*)

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

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

Example:

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

trainer.fit()
trainer.test()

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

CHAPTER

SEVEN

16-BIT TRAINING

Lightning offers 16-bit training for CPUs, GPUs and TPUs.

7.1 GPU 16-bit

Lightning uses NVIDIA apex to handle 16-bit precision training.

To use 16-bit precision, do two things:

1. Install Apex
2. Set the "precision" trainer flag.

7.1.1 Install apex

```
$ git clone https://github.com/NVIDIA/apex
$ cd apex

# ------------------------
# OPTIONAL: on your cluster you might need to load cuda 10 or 9
# depending on how you installed PyTorch

# see available modules
module avail

# 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" ./
```

7.1.2 Enable 16-bit

```
# turn on 16-bit
trainer = Trainer(amp_level='O1', 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()`.

7.2 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(num_tpu_cores=8, precision=32)

# turn on 16-bit
trainer = Trainer(num_tpu_cores=8, precision=16)
```

CHAPTER

EIGHT

COMPUTING CLUSTER (SLURM)

Lightning automates job the details behind training on a SLURM powered cluster.

8.1 Multi-node training

To train a model using multiple-nodes do the following:

1. Design your LightningModule.
2. Add torch.DistributedSampler which enables access to a subset of your full dataset to each GPU.
3. Enable ddp in the trainer

```
# train on 32 GPUs across 4 nodes
trainer = Trainer(gpus=8, num_nodes=4, distributed_backend='ddp')
```

4. It's a good idea to structure your train.py file like this:

```
# train.py
def main(hparams):
    model = LightningTemplateModel(hparams)

    trainer = pl.Trainer(
        gpus=8,
        num_nodes=4,
        distributed_backend='ddp'
    )

    trainer.fit(model)


if __name__ == '__main__':
    root_dir = os.path.dirname(os.path.realpath(__file__))
    parent_parser = ArgumentParser(add_help=False)
    hyperparams = parser.parse_args()

   # TRAIN
    main(hyperparams)
```

4. Submit the appropriate SLURM job

```
#!/bin/bash -l

# SLURM SUBMIT SCRIPT
```

(continues on next page)

(continued from previous page)

```
#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
```

8.2 Walltime auto-resubmit

When you use Lightning in a SLURM cluster, lightning automatically detects when it is about to run into the walltime, and it does the following:

1. Saves a temporary checkpoint.
2. Requeues the job.
3. When the job starts, it loads the temporary checkpoint.

Note: To get this behavior you have to do nothing.

CHAPTER

NINE

CHILD MODULES

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

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

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

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

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

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

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

        loss = MSE(x, x_hat)
        return loss

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

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

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

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

and we can train this using the same trainer

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

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

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

CHAPTER

TEN

DEBUGGING

The following are flags that make debugging much easier.

10.1 Fast dev run

This flag runs a "unit test" by running 1 training batch and 1 validation batch. The point is to detect any bugs in the training/validation loop without having to wait for a full epoch to crash.

```
trainer = pl.Trainer(fast_dev_run=True)
```

10.2 Inspect gradient norms

Logs (to a logger), the norm of each weight matrix.

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

10.3 Log GPU usage

Logs (to a logger) the GPU usage for each GPU on the master machine.

(See: trainer)

```
trainer = pl.Trainer(log_gpu_memory=True)
```

10.4 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: trainer)

```
trainer = pl.Trainer(overfit_pct=0.01)
```

10.5 Print the parameter count by layer

Whenever the .fit() function gets called, the Trainer will print the weights summary for the lightningModule. To disable this behavior, turn off this flag:

(See: trainer.weights_summary)

```
trainer = pl.Trainer(weights_summary=None)
```

10.6 Print which gradients are nan

Prints the tensors with nan gradients.

(See: `trainer.print_nan_grads()`)

```
trainer = pl.Trainer(print_nan_grads=False)
```

10.7 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.

```
# DEFAULT
trainer = Trainer(nb_sanity_val_steps=5)
```

CHAPTER

ELEVEN

EXPERIMENT LOGGING

11.1 Comet.ml

Comet.ml is a third-party logger. To use CometLogger as your logger do the following.

Note: See: comet docs.

```
from pytorch_lightning.loggers import CometLogger

 comet_logger = CometLogger(
     api_key=os.environ["COMET_KEY"],
     workspace=os.environ["COMET_WORKSPACE"], # Optional
     project_name="default_project", # Optional
     rest_api_key=os.environ["COMET_REST_KEY"], # Optional
     experiment_name="default" # Optional
 )
trainer = Trainer(logger=comet_logger)
```

The CometLogger is available anywhere except __init__ in your LightningModule

```
class MyModule(pl.LightningModule):

   def any_lightning_module_function_or_hook(self, ...):
      some_img = fake_image()
      self.logger.experiment.add_image('generated_images', some_img, 0)
```

11.2 Neptune.ai

Neptune.ai is a third-party logger. To use Neptune.ai as your logger do the following.

Note: See: neptune docs.

```
from pytorch_lightning.loggers import NeptuneLogger

 neptune_logger = NeptuneLogger(
     project_name="USER_NAME/PROJECT_NAME",
     experiment_name="default", # Optional,
```

(continues on next page)

(continued from previous page)

```
    params={"max_epochs": 10}, # Optional,
    tags=["pytorch-lightning","mlp"] # Optional,
)
trainer = Trainer(logger=neptune_logger)
```

The Neptune.ai is available anywhere except __init__ in your LightningModule

```
class MyModule(pl.LightningModule):

   def any_lightning_module_function_or_hook(self, ...):
      some_img = fake_image()
      self.logger.experiment.add_image('generated_images', some_img, 0)
```

11.3 Tensorboard

To use Tensorboard as your logger do the following.

Note: See: TensorBoardLogger tf-logger

```
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(pl.LightningModule):

   def any_lightning_module_function_or_hook(self, ...):
      some_img = fake_image()
      self.logger.experiment.add_image('generated_images', some_img, 0)
```

11.4 Test Tube

Test Tube is a tensorboard logger but with nicer file structure. To use TestTube as your logger do the following.

Note: See: TestTube testTube

```
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(pl.LightningModule):

   def any_lightning_module_function_or_hook(self, ...):
```

(continues on next page)

(continued from previous page)

```
      some_img = fake_image()
      self.logger.experiment.add_image('generated_images', some_img, 0)
```

11.5 Wandb

Wandb is a third-party logger. To use Wandb as your logger do the following.

Note: See: wandb docs

```
from pytorch_lightning.loggers import WandbLogger

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

The Wandb logger is available anywhere except __init__ in your LightningModule

```
class MyModule(pl.LightningModule):

   def any_lightning_module_function_or_hook(self, ...):
      some_img = fake_image()
      self.logger.experiment.add_image('generated_images', some_img, 0)
```

11.6 Multiple Loggers

PyTorch-Lightning supports 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("tt_logs", name="my_model")
trainer = Trainer(logger=[logger1, logger2])
```

The loggers are available as a list anywhere except __init__ in your LightningModule

```
class MyModule(pl.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

TWELVE

EXPERIMENT REPORTING

Lightning supports many different experiment loggers. These loggers allow you to monitor losses, images, text, etc... as training progresses. They usually provide a GUI to visualize and can sometimes even snapshot hyperparameters used in each experiment.

12.1 Control logging frequency

It may slow training down to log every single batch. Trainer has an option to log every k batches instead.

```
# k = 10
Trainer(row_log_interval=10)
```

12.2 Control log writing frequency

Writing to a logger can be expensive. In Lightning you can set the interval at which you want to log using this trainer flag.

Note: See: trainer

```
k = 100
Trainer(log_save_interval=k)
```

12.3 Log metrics

To plot metrics into whatever logger you passed in (tensorboard, comet, neptune, etc...)

1. training_epoch_end, validation_epoch_end, test_epoch_end will all log anything in the "log" key of the return dict.

```
def training_epoch_end(self, outputs):
   loss = some_loss()
   ...

   logs = {'train_loss': loss}
   results = {'log': logs}
   return results
```

(continues on next page)

(continued from previous page)

```
def validation_epoch_end(self, outputs):
    loss = some_loss()
    ...

    logs = {'val_loss': loss}
    results = {'log': logs}
    return results

def test_epoch_end(self, outputs):
    loss = some_loss()
    ...

    logs = {'test_loss': loss}
    results = {'log': logs}
    return results
```

2. In addition, you can also use any arbitrary functionality from a particular logger from within your LightningModule. For instance, here we log images using tensorboard.

```
def training_step(self, batch, batch_idx):
    self.generated_imgs = self.decoder.generate()

    sample_imgs = self.generated_imgs[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('generated_images', grid, 0)

    ...
    return results
```

12.4 Modify progress bar

Each return dict from the training_end, validation_end, testing_end and training_step also has a key called "progress_bar".

Here we show the validation loss in the progress bar

```
def validation_epoch_end(self, outputs):
    loss = some_loss()
    ...

    logs = {'val_loss': loss}
    results = {'progress_bar': logs}
    return results
```

12.5 Snapshot hyperparameters

When training a model, it's useful to know what hyperparams went into that model. When Lightning creates a checkpoint, it stores a key "hparams" with the hyperparams.

```
lightning_checkpoint = torch.load(filepath, map_location=lambda storage, loc: storage)
hyperparams = lightning_checkpoint['hparams']
```

Some loggers also allow logging the hyperparams used in the experiment. For instance, when using the TestTubeLogger or the TensorBoardLogger, all hyperparams will show in the hparams tab.

12.6 Snapshot code

Loggers also allow you to snapshot a copy of the code used in this experiment. For example, TestTubeLogger does this with a flag:

```
from pytorch_lightning.loggers import TestTubeLogger

logger = TestTubeLogger(create_git_tag=True)
```

CHAPTER

THIRTEEN

EARLY STOPPING

13.1 Default behavior

By default early stopping will be enabled if *'val_loss'* is found in *validation_epoch_end()* return dict. Otherwise training will proceed with early stopping disabled.

13.2 Enable Early Stopping

There are two ways to enable early stopping.

Note: See: trainer

```
# A) Set early_stop_callback to True. Will look for 'val_loss'
# in validation_epoch_end() return dict. If it is not found an error is raised.
trainer = Trainer(early_stop_callback=True)

# B) Or configure your own callback
early_stop_callback = EarlyStopping(
    monitor='val_loss',
    min_delta=0.00,
    patience=3,
    verbose=False,
    mode='min'
)
trainer = Trainer(early_stop_callback=early_stop_callback)
```

13.3 Disable Early Stopping

To disable early stopping pass `False` to the *early_stop_callback*. Note that `None` will not disable early stopping but will lead to the default behaviour.

Note: See: trainer

CHAPTER

FOURTEEN

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.

14.1 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)
```

14.2 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.

Note: See: trainer

```
# DEFAULT
trainer = Trainer(min_epochs=1, max_epochs=1000)
```

14.3 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)

# check every 100 train batches (ie: for IterableDatasets or fixed frequency)
trainer = Trainer(val_check_interval=100)
```

14.4 Use training data subset

If you don't want to check 100% of the training set (for debugging or if it's huge), set this flag.

```
# DEFAULT
trainer = Trainer(train_percent_check=1.0)

# check 10% only
trainer = Trainer(train_percent_check=0.1)
```

Note: train_percent_check will be overwritten by overfit_pct if overfit_pct > 0

14.5 Use test data subset

If you don't want to check 100% of the test set (for debugging or if it's huge), set this flag test_percent_check will be overwritten by overfit_pct if overfit_pct > 0.

```
# DEFAULT
trainer = Trainer(test_percent_check=1.0)

# check 10% only
trainer = Trainer(test_percent_check=0.1)
```

14.6 Use validation data subset

If you don't want to check 100% of the validation set (for debugging or if it's huge), set this flag val_percent_check will be overwritten by overfit_pct if overfit_pct > 0

```
# DEFAULT
trainer = Trainer(val_percent_check=1.0)

# check 10% only
trainer = Trainer(val_percent_check=0.1)
```

CHAPTER

FIFTEEN

HYPERPARAMETERS

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

15.1 LightiningModule hparams

Normally, we don't hard-code the values to a model. We usually use the command line to modify the network. The *Trainer* can add all the available options to an ArgumentParser.

```
from argparse import ArgumentParser

parser = ArgumentParser()

# parametrize the network
parser.add_argument('--layer_1_dim', type=int, default=128)
parser.add_argument('--layer_2_dim', type=int, default=256)
parser.add_argument('--batch_size', type=int, default=64)

# add all the available options to the trainer
parser = pl.Trainer.add_argparse_args(parser)

args = parser.parse_args()
```

Now we can parametrize the LightningModule.

```
class LitMNIST(pl.LightningModule):
  def __init__(self, hparams):
    super(LitMNIST, self).__init__()
    self.hparams = hparams

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

  def forward(self, x):
    ...

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

  def configure_optimizers(self):
```

(continues on next page)

(continued from previous page)

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

hparams = parse_args()
model = LitMNIST(hparams)
```

Note: Bonus! if (hparams) is in your module, Lightning will save it into the checkpoint and restore your model using those hparams exactly.

And we can also add all the flags available in the Trainer to the Argparser.

```
# add all the available Trainer options to the ArgParser
parser = pl.Trainer.add_argparse_args(parser)
args = parser.parse_args()
```

And now you can start your program with

```
# now you can use any trainer flag
$ python main.py --num_nodes 2 --gpus 8
```

15.2 Trainer args

It also gets annoying to map each argument into the Argparser. Luckily we have a default parser

```
parser = ArgumentParser()

# add all options available in the trainer such as (max_epochs, etc...)
parser = Trainer.add_argparse_args(parser)
```

We set up the main training entry point file like this:

```
def main(args):
    model = LitMNIST(hparams=args)
    trainer = Trainer(max_epochs=args.max_epochs)
    trainer.fit(model)

if __name__ == '__main__':
    parser = ArgumentParser()

    # adds all the trainer options as default arguments (like max_epochs)
    parser = Trainer.add_argparse_args(parser)

    # parametrize the network
    parser.add_argument('--layer_1_dim', type=int, default=128)
    parser.add_argument('--layer_1_dim', type=int, default=256)
    parser.add_argument('--batch_size', type=int, default=64)
    args = parser.parse_args()

    # train
    main(args)
```

And now we can train like this:

```
$ python main.py --layer_1_dim 128 --layer_2_dim 256 --batch_size 64 --max_epochs 64
```

But it would also be nice to pass in any arbitrary argument to the trainer. We can do it by changing how we init the trainer.

```
def main(args):
    model = LitMNIST(hparams=args)

    # makes all trainer options available from the command line
    trainer = Trainer.from_argparse_args(args)
```

and now we can do this:

```
$ python main.py --gpus 1 --min_epochs 12 --max_epochs 64 --arbitrary_trainer_arg␣
↪some_value
```

15.3 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(pl.LightningModule):
  def __init__(self, hparams):
    super(LitMNIST, self).__init__()
    self.layer_1 = torch.nn.Linear(28 * 28, hparams.layer_1_dim)

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

class GoodGAN(pl.LightningModule):
  def __init__(self, hparams):
    super(GoodGAN, self).__init__()
    self.encoder = Encoder(layers=hparams.encoder_layers)

    @staticmethod
    def add_model_specific_args(parent_parser):
        parser = ArgumentParser(parents=[parent_parser])
        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):

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

    model = LitMNIST(hparams=args)
```

(continues on next page)

(continued from previous page)

```
    trainer = Trainer(max_epochs=args.max_epochs)
    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')
    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
```

15.4 Hyperparameter Optimization

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

- Hydra
- Optuna

CHAPTER

SIXTEEN

MULTI-GPU TRAINING

Lightning supports multiple ways of doing distributed training.

16.1 Preparing your code

To train on CPU/GPU/TPU without changing your code, we need to build a few good habits :)

16.1.1 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)
```

16.1.2 Init using type_as

When you need to create a new tensor, use *type_as*. This will make your code scale to any arbitrary number of GPUs or TPUs with Lightning

```
# before lightning
def forward(self, x):
    z = torch.Tensor(2, 3)
    z = z.cuda(0)

# with lightning
def forward(self, x):
    z = torch.Tensor(2, 3)
    z = z.type_as(x.type())
```

16.1.3 Remove samplers

For multi-node or TPU training, in PyTorch we must use *torch.nn.DistributedSampler*. 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)
```

With Lightning, you don't need to do this because it takes care of adding the correct samplers when needed.

```
# with lightning
def train_dataloader(self):
    dataset = MNIST(...)
    return DataLoader(dataset)
```

16.2 Distributed modes

Lightning allows multiple ways of training

- Data Parallel (*distributed_backend='dp'*) (multiple-gpus, 1 machine)
- DistributedDataParallel (*distributed_backend='ddp'*) (multiple-gpus across many machines).
- DistributedDataParallel2 (*distributed_backend='ddp2'*) (dp in a machine, ddp across machines).
- TPUs (*num_tpu_cores=8|x*) (tpu or TPU pod)

16.2.1 Data Parallel (dp)

DataParallel 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.

```
# train on 1 GPU (using dp mode)
trainer = pl.Trainer(gpus=2, distributed_backend='dp')
```

16.2.2 Distributed Data Parallel

DistributedDataParallel works as follows.

1. Each GPU across every node gets its own process.
2. Each GPU gets visibility into a subset of the overall dataset. It will only ever see that subset.
3. Each process inits the model.

Note: Make sure to set the random seed so that each model inits with the same weights

4. Each process performs a full forward and backward pass in parallel.
5. The gradients are synced and averaged across all processes.
6. Each process updates its optimizer.

```
# train on 8 GPUs (same machine (ie: node))
trainer = pl.Trainer(gpus=8, distributed_backend='ddp')

# train on 32 GPUs (4 nodes)
trainer = pl.Trainer(gpus=8, distributed_backend='ddp', num_nodes=4)
```

16.2.3 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 = pl.Trainer(gpus=8, distributed_backend='ddp2', num_nodes=4)
```

16.2.4 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_0(gpu_1_batch)
    y_2 = model_copy_gpu_0(gpu_2_batch)
    y_3 = model_copy_gpu_0(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.forward(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 no matter the backend.

Validation and test step also have the same option when using dp

```
def validation_step_end(self, batch_parts_outputs):
    ...
```

(continues on next page)

(continued from previous page)

```
def test_step_end(self, batch_parts_outputs):
    ...
```

16.2.5 Implement Your Own Distributed (DDP) training

If you need your own way to init PyTorch DDP you can override `pytorch_lightning.core.LightningModule.()`.

If you also need to use your own DDP implementation, override: `pytorch_lightning.core.LightningModule.configure_ddp()`.

CHAPTER

SEVENTEEN

SAVING AND LOADING WEIGHTS

Lightning can automate saving and loading checkpoints.

17.1 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)

17.1.1 Automatic saving

Checkpointing is enabled by default to the current working directory. To change the checkpoint path pass in:

```
Trainer(default_save_path='/your/path/to/save/checkpoints')
```

To modify the behavior of checkpointing pass in your own callback.

```
from pytorch_lightning.callbacks import ModelCheckpoint

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

trainer = Trainer(checkpoint_callback=checkpoint_callback)
```

Or disable it by passing

```
trainer = Trainer(checkpoint_callback=False)
```

The Lightning checkpoint also saves the hparams (hyperparams) passed into the LightningModule init.

Note: hparams is a Namespace.

```
from argparse import Namespace

# usually these come from command line args
args = Namespace(learning_rate=0.001)

# define you module to have hparams as the first arg
# this means your checkpoint will have everything that went into making
# this model (in this case, learning rate)
class MyLightningModule(pl.LightningModule):

    def __init__(self, hparams, ...):
        self.hparams = hparams
```

17.1.2 Manual saving

To save your own checkpoint call:

```
model.save_checkpoint(PATH)
```

17.2 Checkpoint Loading

You might want to not only load a model but also continue training it. Use this method to restore the trainer state as well. This will continue from the epoch and global step you last left off. However, the dataloaders will start from the first batch again (if you shuffled it shouldn't matter).

```
model = MyLightingModule.load_from_checkpoint(PATH)
model.eval()
y_hat = model(x)
```

A LightningModule is no different than a nn.Module. This means you can load it and use it for predictions as you would a nn.Module.

CHAPTER

EIGHTEEN

OPTIMIZATION

18.1 Learning rate scheduling

Every optimizer you use can be paired with any LearningRateScheduler.

```
# no LR scheduler
def configure_optimizers(self):
   return Adam(...)

# Adam + LR scheduler
def configure_optimizers(self):
   return [Adam(...)], [ReduceLROnPlateau()]

# Two optimziers each with a scheduler
def configure_optimizers(self):
   return [Adam(...), SGD(...)], [ReduceLROnPlateau(), LambdaLR()]
```

18.2 Use multiple optimizers (like GANs)

To use multiple optimizers return > 1 optimizers from `pytorch_lightning.core.LightningModule.configure_optimizers()`

```
# one optimizer
def configure_optimizers(self):
   return Adam(...)

# two optimizers, no schedulers
def configure_optimizers(self):
   return Adam(...), SGD(...)

# Two optimizers, one scheduler for adam only
def configure_optimizers(self):
   return [Adam(...), SGD(...)], [ReduceLROnPlateau()]
```

Lightning will call each optimizer sequentially:

```
for epoch in epochs:
   for batch in data:
      for opt in optimizers:
         train_step(opt)
         opt.step()
```

(continues on next page)

(continued from previous page)

```
   for scheduler in scheduler:
      scheduler.step()
```

18.3 Step optimizers at arbitrary intervals

To do more interesting things with your optimizers such as learning rate warm-up or odd scheduling, override the `optimizer_step()` function.

For example, here step optimizer A every 2 batches and optimizer B every 4 batches

```
def optimizer_step(self, current_epoch, batch_nb, optimizer, optimizer_i, second_
↪order_closure=None):
    optimizer.step()
    optimizer.zero_grad()

# Alternating schedule for optimizer steps (ie: GANs)
def optimizer_step(self, current_epoch, batch_nb, optimizer, optimizer_i, second_
↪order_closure=None):
    # update generator opt every 2 steps
    if optimizer_i == 0:
        if batch_nb % 2 == 0 :
            optimizer.step()
            optimizer.zero_grad()

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

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

Here we add a learning-rate warm up

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

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

CHAPTER

NINETEEN

PERFORMANCE AND BOTTLENECK PROFILER

Profiling your training run can help you understand if there are any bottlenecks in your code.

19.1 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

19.2 Enable simple profiling

If you only wish to profile the standard actions, you can set *profiler=True* when constructing your *Trainer* object.

```
trainer = Trainer(..., profiler=True)
```

The profiler's results will be printed at the completion of a training *fit()*.

```
Profiler Report

Action                  |  Mean duration (s)    |  Total time (s)
-----------------------------------------------------------------
on_epoch_start          |  5.993e-06            |  5.993e-06
get_train_batch         |  0.0087412            |  16.398
on_batch_start          |  5.0865e-06           |  0.0095372
```

(continues on next page)

(continued from previous page)

```
model_forward           |  0.0017818           |  3.3408
model_backward          |  0.0018283           |  3.4282
on_after_backward       |  4.2862e-06          |  0.0080366
optimizer_step          |  0.0011072           |  2.0759
on_batch_end            |  4.5202e-06          |  0.0084753
on_epoch_end            |  3.919e-06           |  3.919e-06
on_train_end            |  5.449e-06           |  5.449e-06
```

19.3 Advanced Profiling

If you want more information on the functions called during each event, you can use the *AdvancedProfiler*. This option uses Python's cProfiler to provide a report of time spent on *each* function called within your code.

```
profiler = AdvancedProfiler()
trainer = Trainer(..., profiler=profiler)
```

The profiler's results will be printed at the completion of a training *fit()*. This profiler report can be quite long, so you can also specify an *output_filename* to save the report instead of logging it to the output in your terminal. The output below shows the profiling for the action *get_train_batch*.

```
Profiler Report

Profile stats for: get_train_batch
        4869394 function calls (4863767 primitive calls) in 18.893 seconds
Ordered by: cumulative time
List reduced from 76 to 10 due to restriction <10>
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
3752/1876    0.011    0.000   18.887    0.010 {built-in method builtins.next}
    1876     0.008    0.000   18.877    0.010 dataloader.py:344(__next__)
    1876     0.074    0.000   18.869    0.010 dataloader.py:383(_next_data)
    1875     0.012    0.000   18.721    0.010 fetch.py:42(fetch)
    1875     0.084    0.000   18.290    0.010 fetch.py:44(<listcomp>)
    60000    1.759    0.000   18.206    0.000 mnist.py:80(__getitem__)
    60000    0.267    0.000   13.022    0.000 transforms.py:68(__call__)
    60000    0.182    0.000    7.020    0.000 transforms.py:93(__call__)
    60000    1.651    0.000    6.839    0.000 functional.py:42(to_tensor)
    60000    0.260    0.000    5.734    0.000 transforms.py:167(__call__)
```

You can also reference this profiler in your LightningModule to profile specific actions of interest. If you don't want to always have the profiler turned on, you can optionally pass a *PassThroughProfiler* which will allow you to skip profiling without having to make any code changes. Each profiler has a method *profile()* which returns a context handler. Simply pass in the name of your action that you want to track and the profiler will record performance for code executed within this context.

```
from pytorch_lightning.profiler import Profiler, PassThroughProfiler

class MyModel(LightningModule):
    def __init__(self, hparams, profiler=None):
        self.hparams = hparams
        self.profiler = profiler or PassThroughProfiler()

    def custom_processing_step(self, data):
        with profiler.profile('my_custom_action'):
```

(continues on next page)

(continued from previous page)

```
            # custom processing step
        return data

profiler = Profiler()
model = MyModel(hparams, profiler)
trainer = Trainer(profiler=profiler, max_epochs=1)
```

class pytorch_lightning.profiler.**Profiler**
Bases: pytorch_lightning.profiler.profiler.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.

describe()
Logs a profile report after the conclusion of the training run.

start(*action_name*)
Defines how to start recording an action.

stop(*action_name*)
Defines how to record the duration once an action is complete.

class pytorch_lightning.profiler.**AdvancedProfiler**(*output_filename=None*, *line_count_restriction=1.0*)
Bases: pytorch_lightning.profiler.profiler.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

- **(str)** (*output_filename*) – optionally save profile results to file instead of printing to std out when training is finished.
- **(int|float)** (*line_count_restriction*) – 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.

stop(*action_name*)
Defines how to record the duration once an action is complete.

class pytorch_lightning.profiler.**PassThroughProfiler**
Bases: pytorch_lightning.profiler.profiler.BaseProfiler

This class should be used when you don't want the (small) overhead of profiling. The Trainer uses this class by default.

start(*action_name*)
Defines how to start recording an action.

stop(*action_name*)
Defines how to record the duration once an action is complete.

CHAPTER

TWENTY

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 = pl.Trainer(gpus=1)
```

CHAPTER

TWENTYONE

SEQUENTIAL DATA

Lightning has built in support for dealing with sequential data.

21.1 Packed sequences as inputs

When using PackedSequence, do 2 things:

1. return either a padded tensor in dataset or a list of variable length tensors in the dataloader collate_fn (example above shows the list implementation).
2. Pack the sequence in forward or training and validation steps depending on use case.

```
# For use in dataloader
 def collate_fn(batch):
     x = [item[0] for item in batch]
     y = [item[1] for item in batch]
     return x, y

 # In module
 def training_step(self, batch, batch_nb):
     x = rnn.pack_sequence(batch[0], enforce_sorted=False)
     y = rnn.pack_sequence(batch[1], enforce_sorted=False)
```

21.2 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.

CHAPTER

TWENTYTWO

TRAINING TRICKS

Lightning implements various tricks to help during training

22.1 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.

Note: See: trainer

```
# DEFAULT (ie: no accumulated grads)
trainer = Trainer(accumulate_grad_batches=1)
```

22.2 Gradient Clipping

Gradient clipping may be enabled to avoid exploding gradients. Specifically, this will clip the gradient norm computed over all model parameters together.

Note: See: 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)
```

CHAPTER

TWENTYTHREE

TRANSFER LEARNING

23.1 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 pl.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(pl.LightningModule):
    def __init__(self):
        self.encoder = Encoder()
        self.decoder = Decoder()

class CIFAR10Classifier(pl.LightingModule):
    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!

23.2 Example: Imagenet (computer Vision)

```
import torchvision.models as models

class ImagenetTranferLearning(pl.LightingModule):
    def __init__(self):
        # init a pretrained resnet
        num_target_classes = 10
        self.feature_extractor = model.resnet50(
                                    pretrained=True,
                                    num_classes=num_target_classes)
        self.feature_extractor.eval()

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

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

Finetune

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

And use it to predict your data of interest

```
model = ImagenetTranferLearning.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.

23.3 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.

```
from transformers import BertModel

class BertMNLIFinetuner(pl.LightningModule):

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

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

(continues on next page)

(continued from previous page)

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

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

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

CHAPTER

TWENTYFOUR

TPU SUPPORT

Lightning supports running on TPUs. At this moment, TPUs are only available on Google Cloud (GCP). For more information on TPUs watch this video.

24.1 Live demo

Check out this Google Colab to see how to train MNIST on TPUs.

24.2 TPU Terminology

A TPU is a Tensor processing unit. Each TPU has 8 cores where each core is optimized for 128x128 matrix multiplies. In general, a single TPU is about as fast as 5 V100 GPUs!

A TPU pod hosts many TPUs on it. Currently, TPU pod v2 has 2048 cores! You can request a full pod from Google cloud or a "slice" which gives you some subset of those 2048 cores.

24.3 How to access TPUs

To access TPUs there are two main ways.

1. Using google colab.
2. Using Google Cloud (GCP).

24.4 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.

```
import collections
from datetime import datetime, timedelta
import os
import requests
import threading

_VersionConfig = collections.namedtuple('_VersionConfig', 'wheels,server')
VERSION = "xrt==1.15.0"  #@param ["xrt==1.15.0", "torch_xla==nightly"]
CONFIG = {
    'xrt==1.15.0': _VersionConfig('1.15', '1.15.0'),
    'torch_xla==nightly': _VersionConfig('nightly', 'XRT-dev{}'.format(
        (datetime.today() - timedelta(1)).strftime('%Y%m%d'))),
}[VERSION]
DIST_BUCKET = 'gs://tpu-pytorch/wheels'
TORCH_WHEEL = 'torch-{}-cp36-cp36m-linux_x86_64.whl'.format(CONFIG.wheels)
TORCH_XLA_WHEEL = 'torch_xla-{}-cp36-cp36m-linux_x86_64.whl'.format(CONFIG.wheels)
TORCHVISION_WHEEL = 'torchvision-{}-cp36-cp36m-linux_x86_64.whl'.format(CONFIG.wheels)

# Update TPU XRT version
def update_server_xrt():
  print('Updating server-side XRT to {} ...'.format(CONFIG.server))
  url = 'http://{TPU_ADDRESS}:8475/requestversion/{XRT_VERSION}'.format(
      TPU_ADDRESS=os.environ['COLAB_TPU_ADDR'].split(':')[0],
      XRT_VERSION=CONFIG.server,
  )
  print('Done updating server-side XRT: {}'.format(requests.post(url)))

update = threading.Thread(target=update_server_xrt)
update.start()

# Install Colab TPU compat PyTorch/TPU wheels and dependencies
!pip uninstall -y torch torchvision
!gsutil cp "$DIST_BUCKET/$TORCH_WHEEL" .
!gsutil cp "$DIST_BUCKET/$TORCH_XLA_WHEEL" .
!gsutil cp "$DIST_BUCKET/$TORCHVISION_WHEEL" .
!pip install "$TORCH_WHEEL"
!pip install "$TORCH_XLA_WHEEL"
!pip install "$TORCHVISION_WHEEL"
!sudo apt-get install libomp5
update.join()
```

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.

7. TPUs require a DistributedSampler. That means you should change your train_dataloader (and val, train) code as follows.

```
import torch_xla.core.xla_model as xm

def train_dataloader(self):
    dataset = MNIST(
        os.getcwd(),
        train=True,
        download=True,
        transform=transforms.ToTensor()
```

(continues on next page)

(continued from previous page)

```
    )

    # required for TPU support
    sampler = None
    if use_tpu:
        sampler = torch.utils.data.distributed.DistributedSampler(
            dataset,
            num_replicas=xm.xrt_world_size(),
            rank=xm.get_ordinal(),
            shuffle=True
        )

    loader = DataLoader(
        dataset,
        sampler=sampler,
        batch_size=32
    )

    return loader
```

8. 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(num_tpu_cores=8)
trainer.fit(my_model)
```

That's it! Your model will train on all 8 TPU cores.

24.5 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
```

24.6 16 bit precision

Lightning also supports training in 16-bit precision with TPUs. By default, TPU training will use 32-bit precision. To enable 16-bit, also set the 16-bit flag.

```
import pytorch_lightning as pl

my_model = MyLightningModule()
trainer = pl.Trainer(num_tpu_cores=8, precision=16)
trainer.fit(my_model)
```

Under the hood the xla library will use the bfloat16 type.

24.7 About XLA

XLA is the library that interfaces PyTorch with the TPUs. For more information check out XLA.

Guide for troubleshooting XLA

CHAPTER

TWENTYFIVE

TEST SET

Lightning forces the user to run the test set separately to make sure it isn't evaluated by mistake

25.1 Test after fit

To run the test set after training completes, use this method

```
# run full training
trainer.fit(model)

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

25.2 Test pre-trained model

To run the test set on a pretrained model, use this method.

```
model = MyLightningModule.load_from_metrics(
    weights_path='/path/to/pytorch_checkpoint.ckpt',
    tags_csv='/path/to/test_tube/experiment/version/meta_tags.csv',
    on_gpu=True,
    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...

CHAPTER

TWENTYSIX

CONTRIBUTOR COVENANT CODE OF CONDUCT

26.1 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.

26.2 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

26.3 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.

26.4 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.

26.5 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.

26.6 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

TWENTYSEVEN

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!

27.1 Main Core Value: One less thing to remember

Simplify the API as much as possible from the user perspective. Any additions or improvements should minimize things the user needs to remember.

For example: One benefit of the validation_step is that the user doesn't have to remember to set the model to .eval(). This avoids all sorts of subtle errors the user could make.

27.2 Lightning Design Principles

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

27.2.1 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.

27.2.2 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 :)

27.2.3 Force User Decisions To Best Practices

There are 1,000 ways to do something. However, something eventually becomes standard practice that everyone does. Thus we pick one way of doing it and force everyone to do it this way. A good example is accumulated gradients. There are many ways to implement, we just pick one and force users to use that one. A bad forced decision would be to make users use a specific library to do something.

When something becomes a best practice, we add it to the framework. This likely looks like code in utils or in the model file that everyone keeps adding over and over again across projects. When this happens, bring that code inside the trainer and add a flag for it.

27.2.4 Simple External API

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

27.2.5 Backward-compatible API

We all hate updating our deep learning packages because we don't want to refactor a bunch of stuff. In Lightning, we make sure every change we make which could break an API is backwards compatible with good deprecation warnings.

You shouldn't be afraid to upgrade Lightning :)

27.2.6 Gain User Trust

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

27.2.7 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.

27.3 Contribution Types

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

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

27.3.1 Bug Fixes:

1. Submit a github issue - try to decried what happen so other can reproduce it too.
2. Try to ix it or recommend a solution. . .
3. Submit a PR!

27.3.2 New Features:

1. Submit a github issue - describe what is motivation of such feature (plus an use-case).
2. Let's discuss to agree on the feature scope.
3. Submit a PR! (with updated docs and tests).

27.4 Guidelines

27.4.1 Coding Style

1. Use f-strings for output formation (except logging when we stay with lazy `logging.info("Hello %s!`, name).
2. Test the code with flake8, run locally PEP8 fixes:

```
autopep8 -v -r --max-line-length 120 --in-place .
```

27.4.2 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...
```

(continues on next page)

(continued from previous page)

```
        >>> my_func(1, 2)
        3

    .. note:: If you want to add something.
    """
    p = param_b if param_b else 0
    return str(param_a + p)
```

27.4.3 Testing

Test your work locally to speed up your work since so you can focus only in particular (failing) test-cases. To setup a local development environment, install both local and test dependencies:

```
pip install -r requirements.txt
pip install -r tests/requirements.txt
```

You can run the full test-case in your terminal via this bash script:

```
bash .run_local_tests.sh
```

Note: if your computer does not have multi-GPU nor TPU these tests are skipped.

For convenience, you can use also your own CircleCI building which will be triggered with each commit. This is useful if you do not test against all required dependencies version. To do so, login to CircleCI and enable your forked project in the dashboard. It will just work after that.

27.4.4 Pull Request

We welcome any useful contribution! For convinece here's a recommended workflow:

1. Think about what you want to do - fix a bug, repair docs, etc.
2. Start your work locally (usually until you need our CI testing)
    - create a branch and prepare your changes
    - hint: do not work with your master directly, it may become complicated when you need to rebase
    - hint: give your PR a good name! it will be useful later when you may work on multiple tasks/PRs
3. Create a "Draft PR" which is clearly marked which lets us know you don't need feedback yet.
4. When you feel like you are ready for integrating your work, turn your PR to "Ready for review".
5. Use tags in PR name for following cases:
    - **[blocked by #]** if you work is depending on others changes
    - **[wip]** when you start to re-edit your work, mark it so no one will accidentally merge it in meantime

CHAPTER

TWENTYEIGHT

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.

28.1 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.

28.2 The bar for joining the team

Lightning is being used to solve really hard problems at the top AI labs in the world. As such, the bar for adding team members is extremely high. Candidates must have solid engineering skills, have a good eye for user experience, and must be a power user of Lightning and PyTorch.

With that said, the Lightning team will be diverse and a reflection of an inclusive AI community. You don't have to be an engineer to conntribute! Scientists with great usability intuition and PyTorch ninja skills are welcomed!

28.3 Responsibilities:

The responsibilities mainly revolve around 3 things.

28.3.1 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.

28.3.2 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 :)

28.3.3 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.

28.3.4 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.

28.3.5 Summary: Requirements to apply

- Solve 10 Github issues. The goal is to be inline with expectations for solving issues by the last one so you can do them on your own. If not, I might ask you to solve a few more specific ones.
- Do 10 PR reviews. The goal is to be inline with expectations for solving issues by the last one so you can do them on your own. If not, I might ask you to solve a few more specific ones.

If you want to be considered, ping me on gitter and start tracking your progress here.

CHAPTER

TWENTYNINE

BEFORE SUBMITTING

- [ ] Was this discussed/approved via a Github issue? (no need for typos and docs improvements)
- [ ] Did you read the contributor guideline, Pull Request section?
- [ ] Did you make sure to update the docs?
- [ ] Did you write any new necessary tests?
- [ ] If you made a notable change (that affects users), did you update the CHANGELOG?

29.1 What does this PR do?

Fixes # (issue).

29.2 PR review

Anyone in the community is free to review the PR once the tests have passed.If we didn't discuss your PR in Github issues there's a high chance it will not be merged.

29.3 Did you have fun?

Make sure you had fun coding

CHAPTER

THIRTY

PYTORCH LIGHTNING GOVERNANCE | PERSONS OF INTEREST

30.1 Leads

- William Falcon (williamFalcon) (Lightning founder)
- Jirka Borovec (Borda)
- Ethan Harris (ethanwharris) (Torchbearer founder)
- Matthew Painter (MattPainter01) (Torchbearer founder)

30.2 Core Maintainers

- Nic Eggert (neggert)
- Jeff Ling (jeffling)
- Tullie Murrell (tullie)

CHAPTER

THIRTYONE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

B

M

O

P