From PyTorch to PyTorch Lightning — A gentle introduction

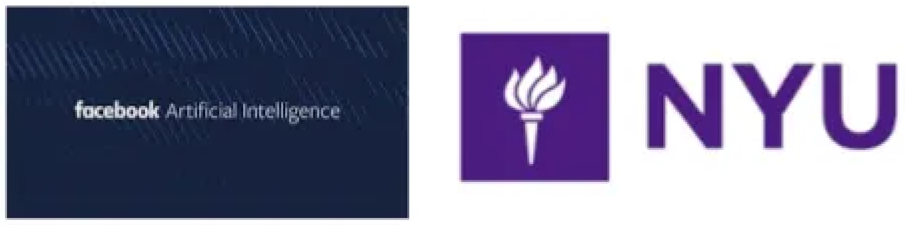
William Falcon Follow 10 min read · Feb 27, 2020

This post answers the most frequent question about why you need Lightning if you're using PyTorch.

PyTorch is extremely easy to use to build complex AI models. But once the research gets complicated and things like multi-GPU training, 16-bit precision and TPU training get mixed in, users are likely to introduce bugs.

PyTorch Lightning solves exactly this problem. Lightning structures your PyTorch code so it can abstract the details of training. This makes AI research scalable and fast to iterate on.

Who is PyTorch Lightning For?

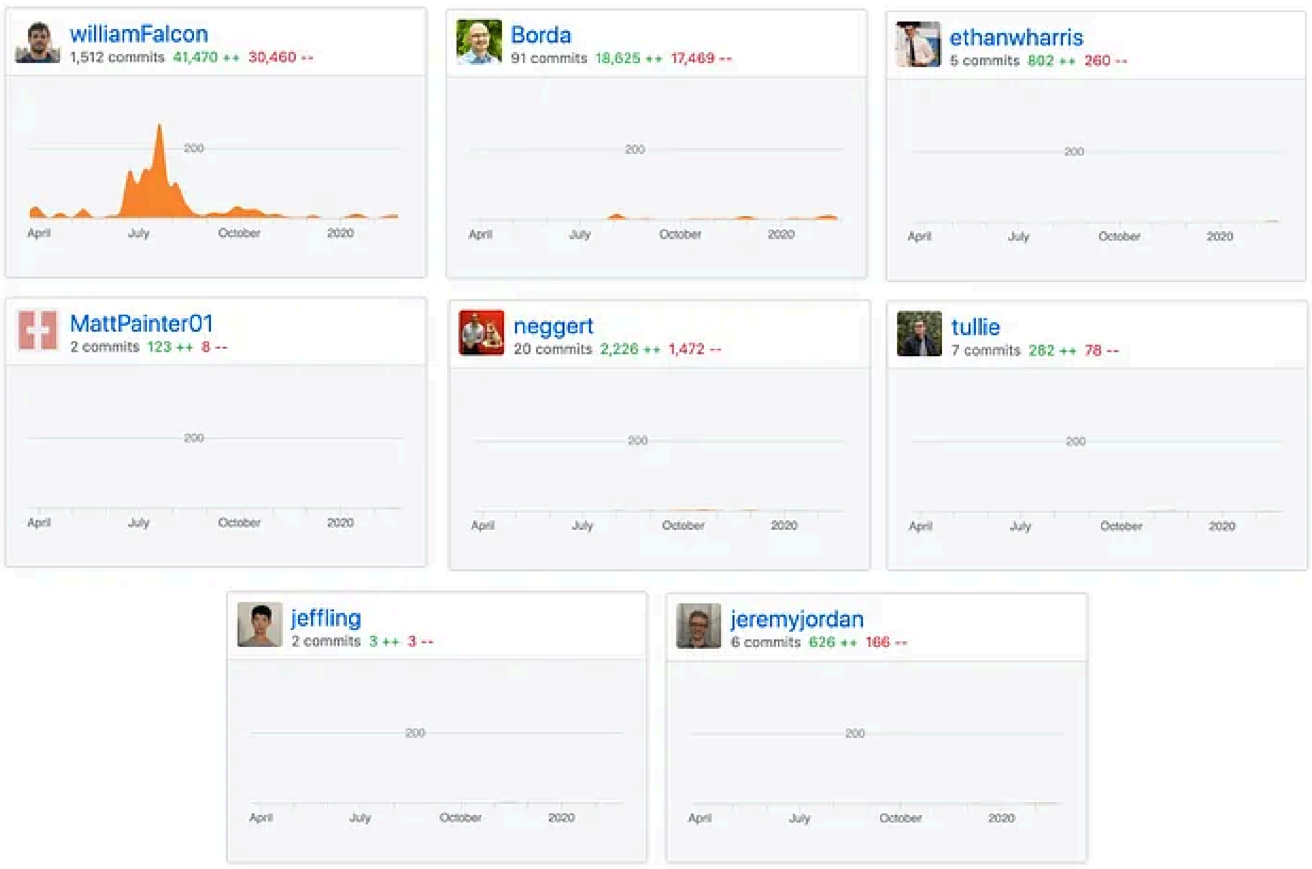


PyTorch Lightning was created while doing PhD research at both NYU and FAIR

PyTorch Lightning was created for professional researchers and PhD students working on AI research.

Lightning was born out of my Ph.D. AI research at NYU CILVR and Facebook AI Research. As a result, the framework is designed to be extremely extensible while making state of the art AI research techniques (like TPU training) trivial.

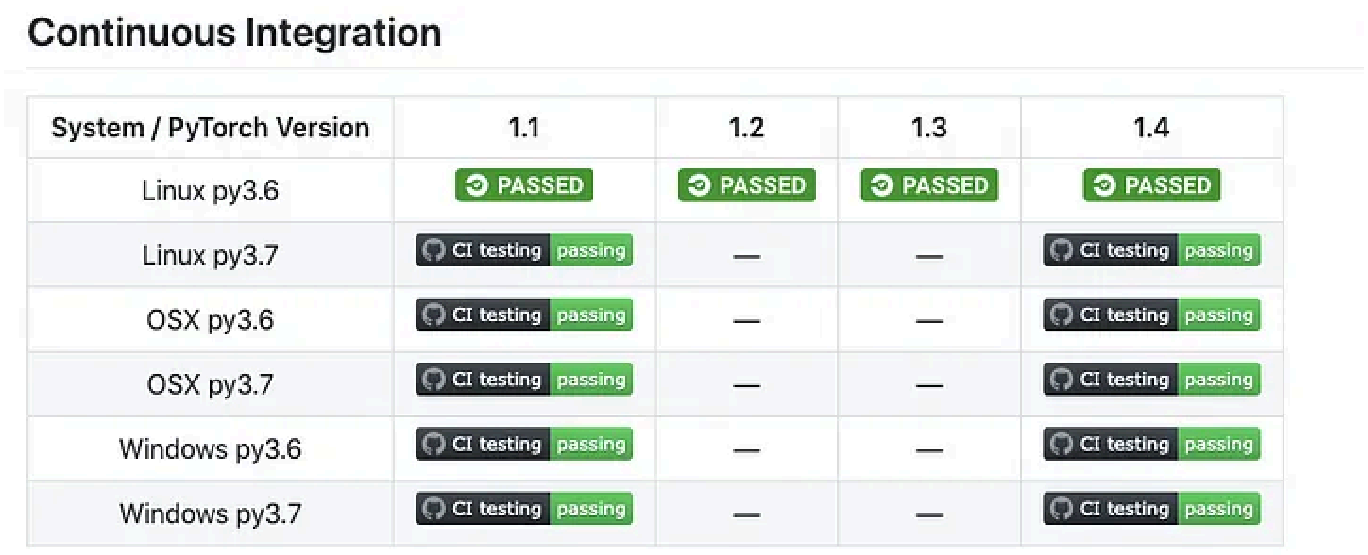
Now the core contributors are all pushing the state of the art in AI using Lightning and continue to add new cool features.



However, the simple interface gives **professional production teams** and **newcomers** access to the latest state of the art techniques developed by the Pytorch and PyTorch Lightning community.

Lightning counts with over 320 contributors, a core team of 11 research scientists, PhD students and professional deep learning engineers.

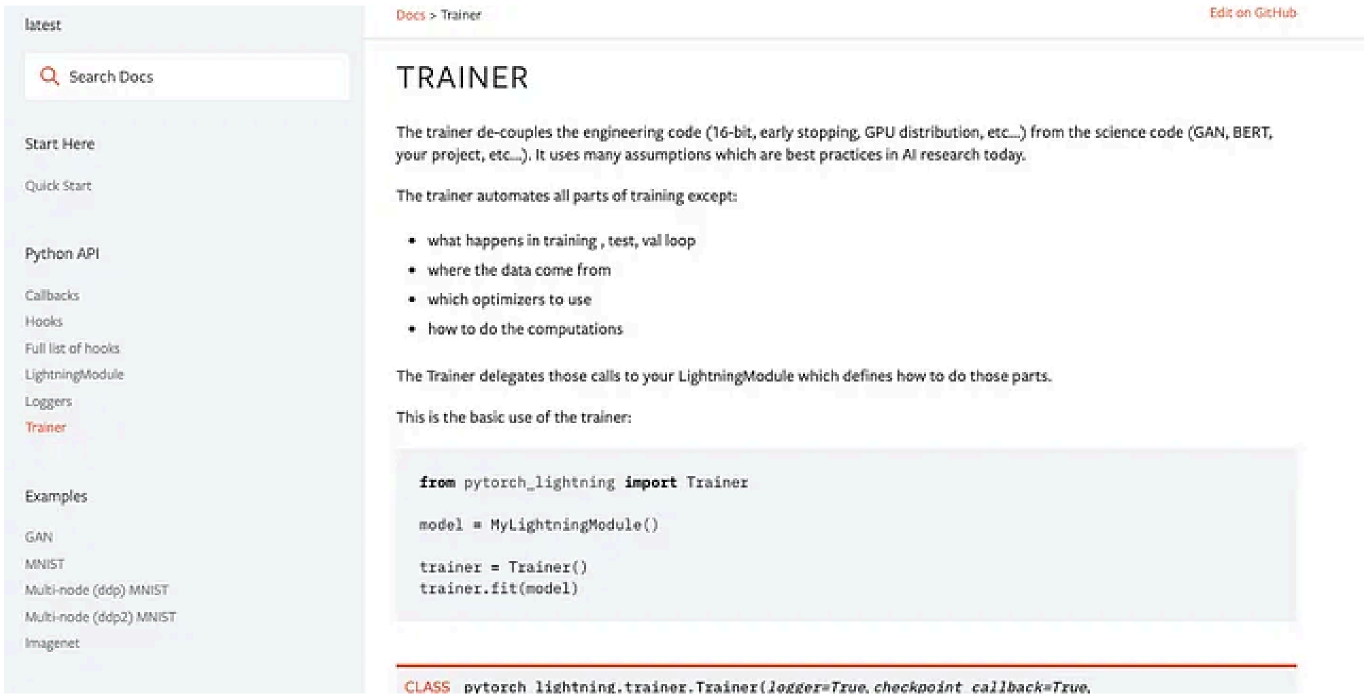
it is rigorously tested



Continuous Integration

System / PyTorch Version	1.1	1.2	1.3	1.4
Linux py3.6	PASSED	PASSED	PASSED	PASSED
Linux py3.7	CI testing passing	—	—	CI testing passing
OSX py3.6	CI testing passing	—	—	CI testing passing
OSX py3.7	CI testing passing	—	—	CI testing passing
Windows py3.6	CI testing passing	—	—	CI testing passing
Windows py3.7	CI testing passing	—	—	CI testing passing

and thoroughly documented



TRAINER

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

The trainer automates all parts of training except:

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

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

This is the basic use of the trainer:

```
from pytorch_lightning import Trainer

model = MyLightningModule()

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

CLASS pytorch_lightning.trainer.Trainer(*logger=True, checkpoint_callback=True,*

Outline

This tutorial will walk you through building a simple MNIST classifier showing PyTorch and PyTorch Lightning code side-by-side. While Lightning can build any arbitrarily complicated system, we use MNIST to illustrate how to refactor PyTorch code into PyTorch Lightning.

The full code is available at this Colab Notebook.

The Typical AI Research project

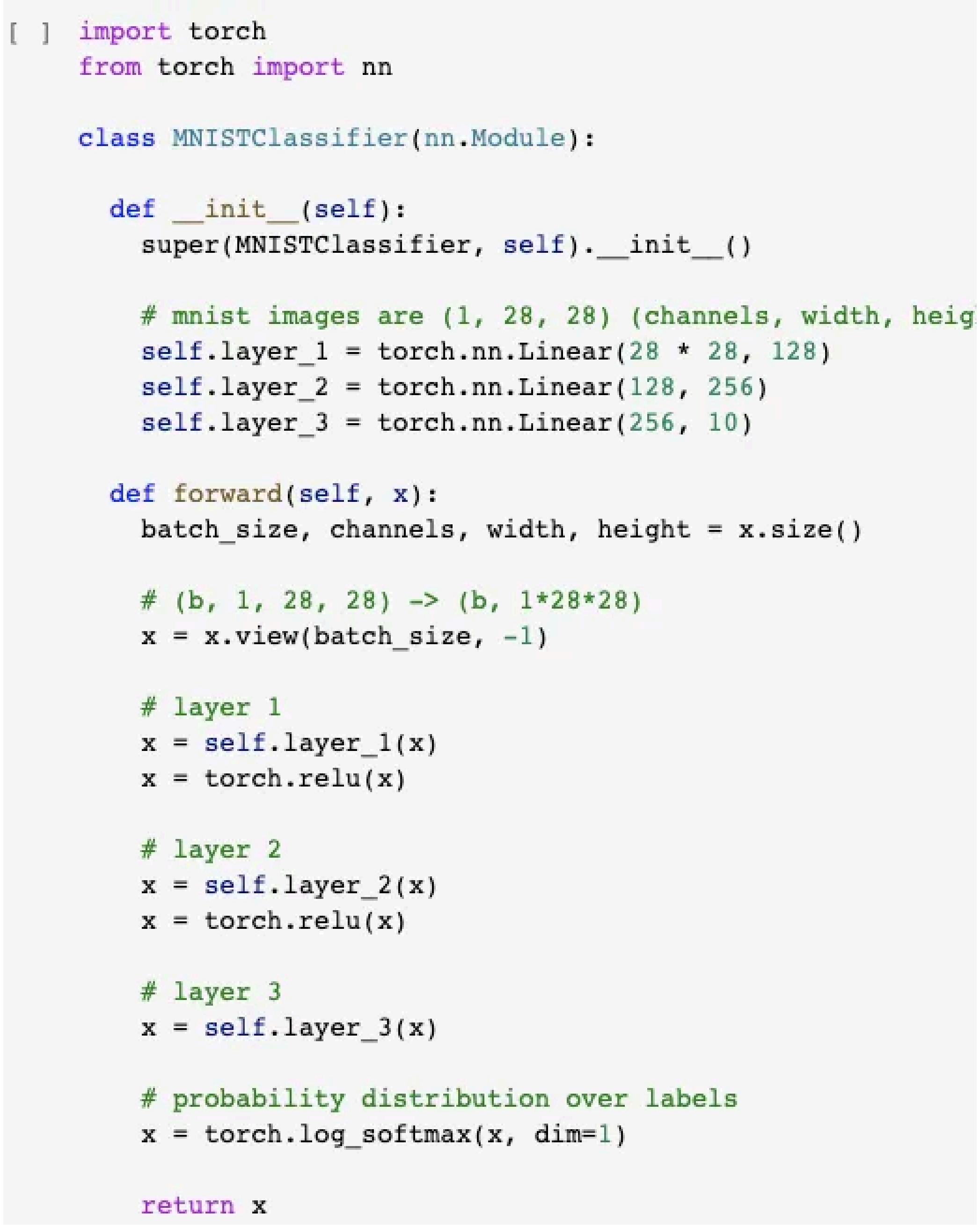
In a research project, we normally want to identify the following key components:

- the model(s)
- the data
- the loss
- the optimizer(s)

The Model

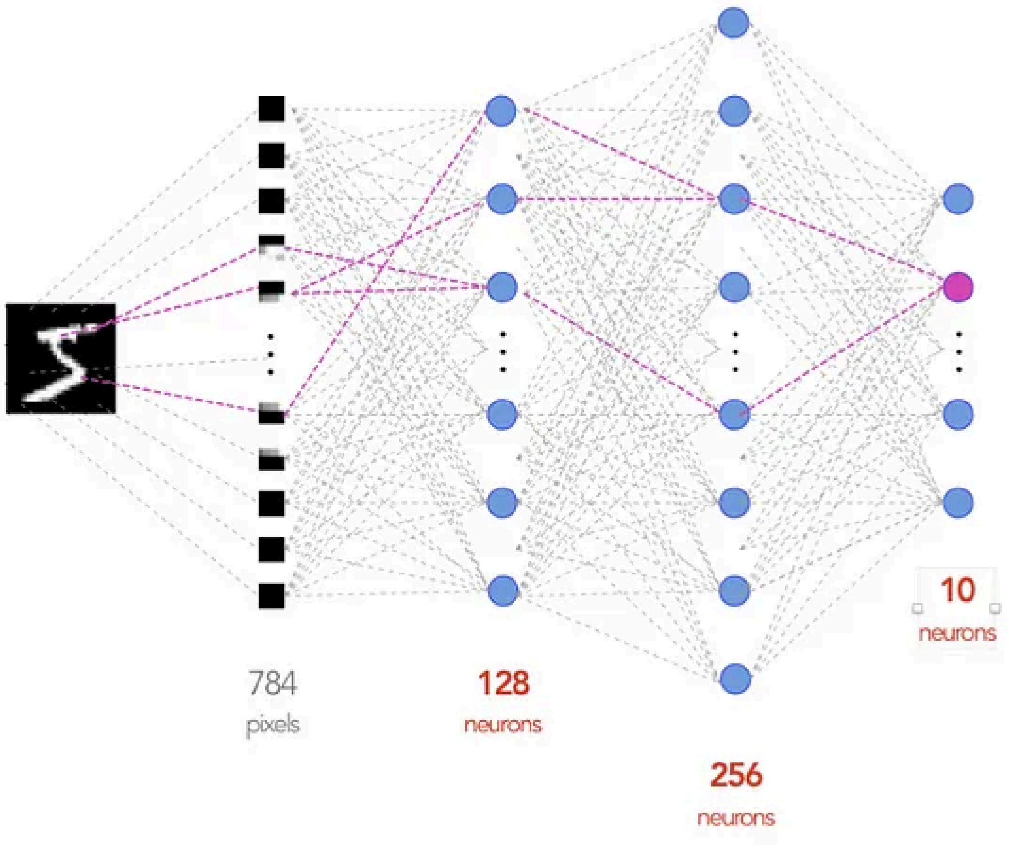
Let's design a 3-layer fully-connected neural network that takes as input an image that is 28x28 and outputs a probability distribution over 10 possible labels.

First, let's define the model in PyTorch



```
[ ]  import torch
     from torch import nn

     class MNISTClassifier(nn.Module):

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

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

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

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

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

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

         # layer 3
         x = self.layer_3(x)

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

         return x
```

This model defines the computational graph to take as input an MNIST image and convert it to a probability distribution over 10 classes for digits 0–9.



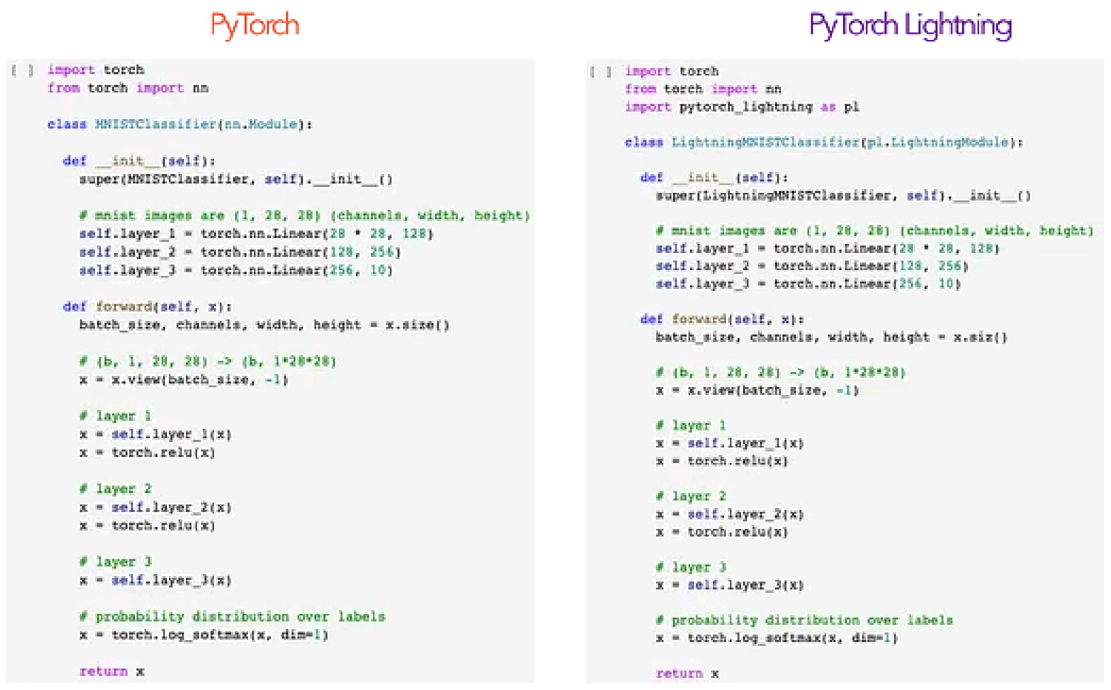
3-layer network (illustration by: William Falcon)

To convert this model to PyTorch Lightning we simply replace the *nn.Module* with the *pl.LightningModule*

```
class MNISTClassifier(nn.Module): # PyTorch
class MNISTClassifier(pl.LightningModule): # Pytorch Lightning
```

The new PyTorch Lightning class is EXACTLY the same as the PyTorch, except that the LightningModule provides **a structure** for the research code.

> *Lightning provides structure to PyTorch code*



```
import torch
from torch import nn

class MNISTClassifier(nn.Module):

  def __init__(self):
    super(MNISTClassifier, 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)

  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
```

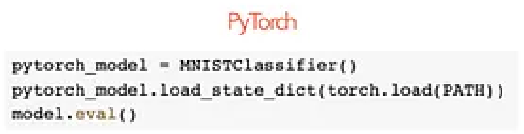
```
import torch
from torch import nn
import pytorch_lightning as pl

class LightningMNISTClassifier(pl.LightningModule):

  def __init__(self):
    super(LightningMNISTClassifier, 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)

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

    # (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
```

See? The code is EXACTLY the same for both!

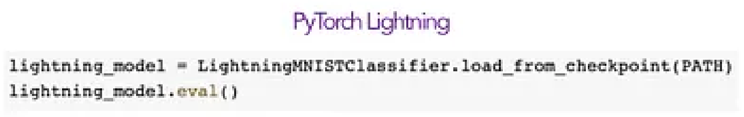
This means you can use a LightningModule exactly as you would a PyTorch module such **as prediction**

```
pytorch_model = MNISTClassifier()
lightning_model = LightningMNISTClassifier()

x = torch.Tensor(32, 1, 28, 28)

pt_out = pytorch_model(x)
pl_out = lightning_model(x)
```

Or use it as a pretrained model



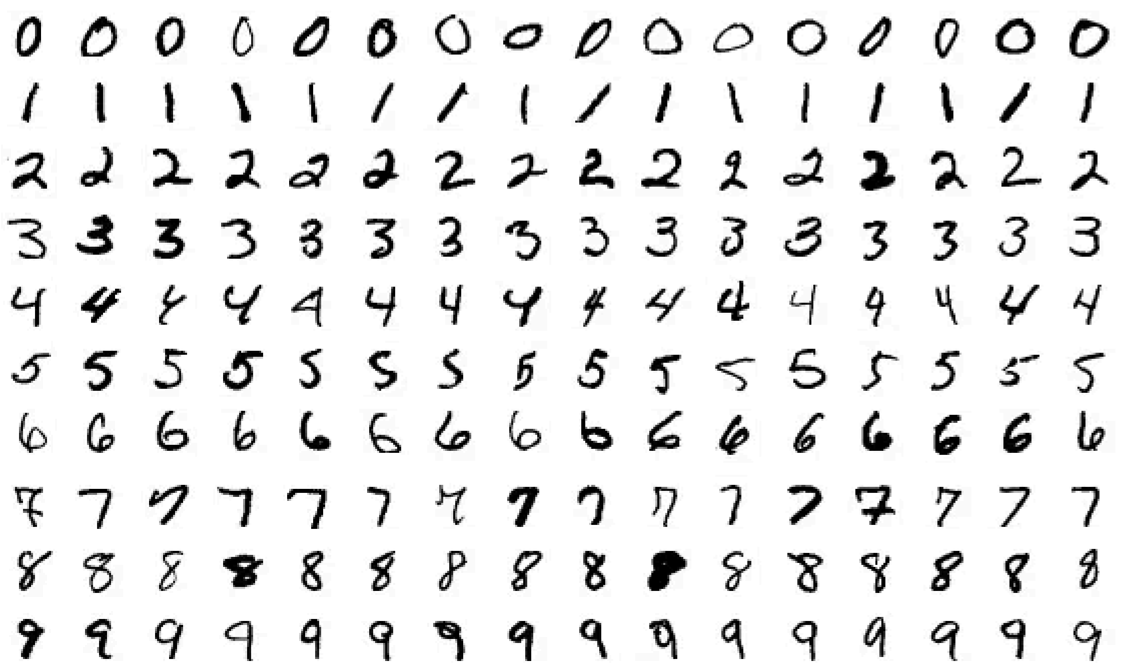
```
pytorch_model = MNISTClassifier()
pytorch_model.load_state_dict(torch.load(PATH))
model.eval()
```



```
lightning_model = LightningMNISTClassifier.load_from_checkpoint(PATH)
lightning_model.eval()
```

The Data

For this tutorial we’re using MNIST.



Source: Wikipedia

Let’s generate three splits of MNIST, a training, validation and test split.

This again, is the same code in PyTorch as it is in Lightning.

The dataset is added to the Dataloader which handles the loading, shuffling and batching of the dataset.

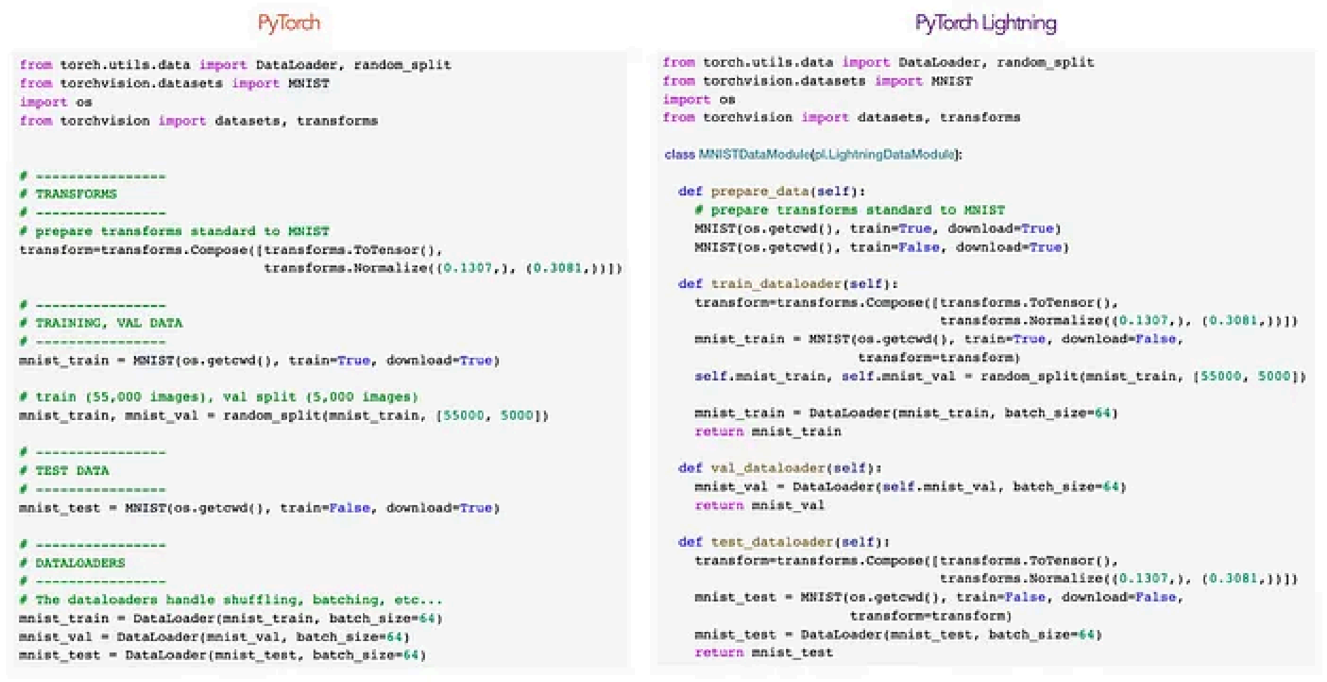
In short, data preparation has 4 steps:

1. Download images

2. Image transforms (these are highly subjective).

3. Generate training, validation and test dataset splits.

4. Wrap each dataset split in a DataLoader



```python
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,))])

# ----------------
# TRAINING, VAL DATA
# ----------------
mnist_train = MNIST(os.getcwd(), train=True, download=True)

# train (55,000 images), val split (5,000 images)
mnist_train, mnist_val = random_split(mnist_train, [55000, 5000])

# ----------------
# TEST DATA
# ----------------
mnist_test = MNIST(os.getcwd(), train=False, download=True)

# ----------------
# DATALOADERS
# ----------------
# The dataloaders handle shuffling, batching, etc...
mnist_train = DataLoader(mnist_train, batch_size=64)
mnist_val = DataLoader(mnist_val, batch_size=64)
mnist_test = DataLoader(mnist_test, batch_size=64)
```

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

class MNISTDataModule(pl.LightningDataModule):

  def prepare_data(self):
    # prepare transforms standard to MNIST
    MNIST(os.getcwd(), train=True, download=True)
    MNIST(os.getcwd(), train=False, download=True)

  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)
    self.mnist_train, self.mnist_val = random_split(mnist_train, [55000, 5000])

    mnist_train = DataLoader(mnist_train, batch_size=64)
    return mnist_train

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

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

Again, the code is **exactly** the same except that we've organized the PyTorch code into 4 functions:

prepare_data

This function handles downloads and any data processing. This function makes sure that when you use multiple GPUs you don't download multiple datasets or apply double manipulations to the data.

This is because each GPU will execute the same PyTorch thereby causing duplication. ALL of the code in Lightning makes sure the critical parts are called from **ONLY** one GPU.

train_dataloader, val_dataloader, test_dataloader

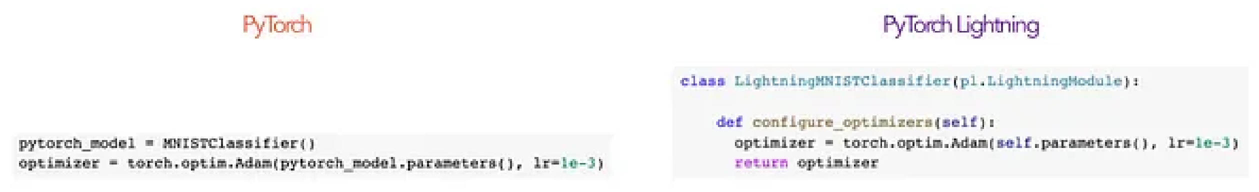
Each of these is responsible for returning the appropriate data split. Lightning structures it this way so that it is VERY clear HOW the data are being manipulated. If you ever read random github code written in PyTorch it's nearly impossible to see how they manipulate their data.

Lightning even allows multiple dataloaders for testing or validating.

This code is organized under what we call a DataModule. Although this is 100% optional and lightning can use DataLoaders directly, a DataModule makes your data reusable and easy to share.

The Optimizer

Now we choose how we're going to do the optimization. We'll use Adam instead of SGD because it is a good default in most DL research.

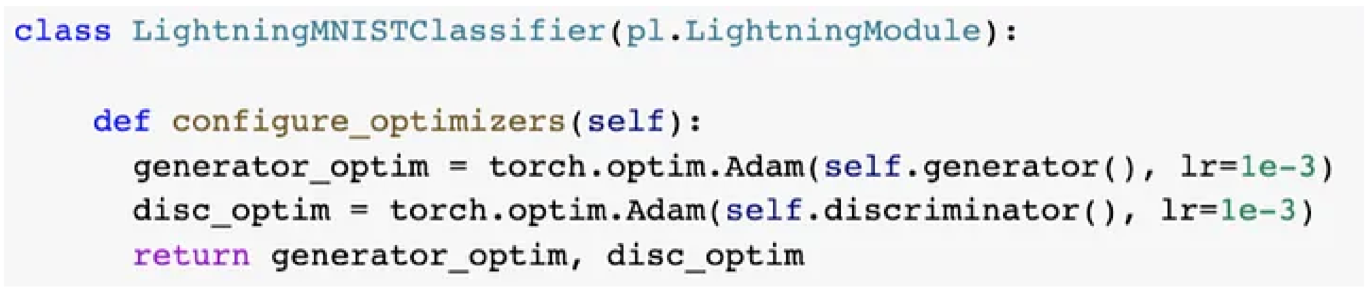


```
pytorch_model = MNISTClassifier()
optimizer = torch.optim.Adam(pytorch_model.parameters(), lr=1e-3)
```

```
class LightningMNISTClassifier(pl.LightningModule):

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

Again, this is **exactly** the same in both except it is organized into the configure optimizers function.

Lightning is **extremely extensible.** For instance, if you wanted to use multiple optimizers (ie: a GAN), you could just return both here.

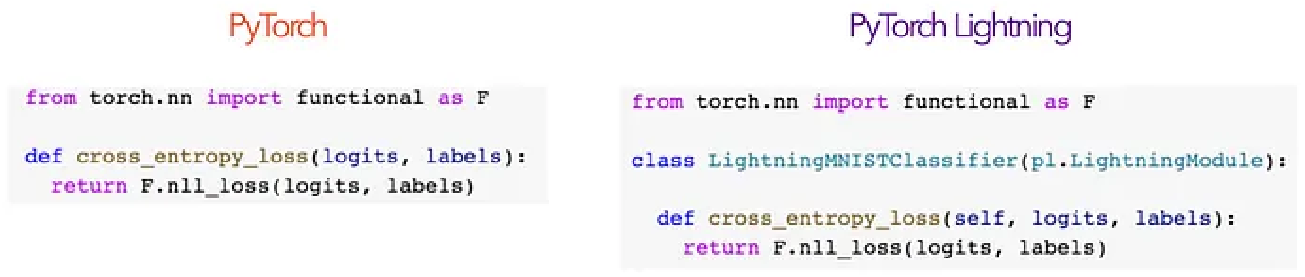


```
class LightningMNISTClassifier(pl.LightningModule):

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

You'll also notice that in Lightning we pass in ***self.parameters()*** and not a model because the LightningModule IS the model.

The Loss

For n-way classification we want to compute the cross-entropy loss. Cross-entropy is the same as NegativeLogLikelihood(log_softmax) which we'll use instead.



```python
from torch.nn import functional as F

def cross_entropy_loss(logits, labels):
  return F.nll_loss(logits, labels)
```

```python
from torch.nn import functional as F

class LightningMNISTClassifier(pl.LightningModule):

  def cross_entropy_loss(self, logits, labels):
    return F.nll_loss(logits, labels)
```

Again… code is exactly the same!

Training and Validation Loop

We assembled all the key ingredients needed for training:

1. The model (3-layer NN)
2. The dataset (MNIST)
3. An optimizer
4. A loss

Now we implement a full training routine which does the following:

- Iterates for many epochs (an epoch is a full pass through the dataset **D**)

$$D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$$

in math

```
num_epochs = 100
for epoch in range(num_epochs):
```

in code

- Each epoch iterates the dataset in small chunks called batches **b**

$$b \in D$$

in math

```
for batch in dataloader:
```

in code

- We perform a forward pass

$$\hat{y} = f(x)$$

in math

```
logits = model(x)
```

the code

- Compute the loss

$$L = -\sum_{i}^{C} y_i \log(\hat{y}_i)$$

in math

```
loss = cross_entropy_loss(logits, y)
```

in code

- Perform a backward pass to calculate all the gradients for each weight

$$\nabla w_i = \frac{\partial L}{\partial w_i} \qquad \forall \quad w_i$$

in math

```
loss.backward()
```

in code

- Apply the gradients to each weight

$$w_i = w_i + \alpha \nabla w_i$$

in math

```
optimizer.step()
```

in code

In both PyTorch and Lightning the pseudocode looks like this

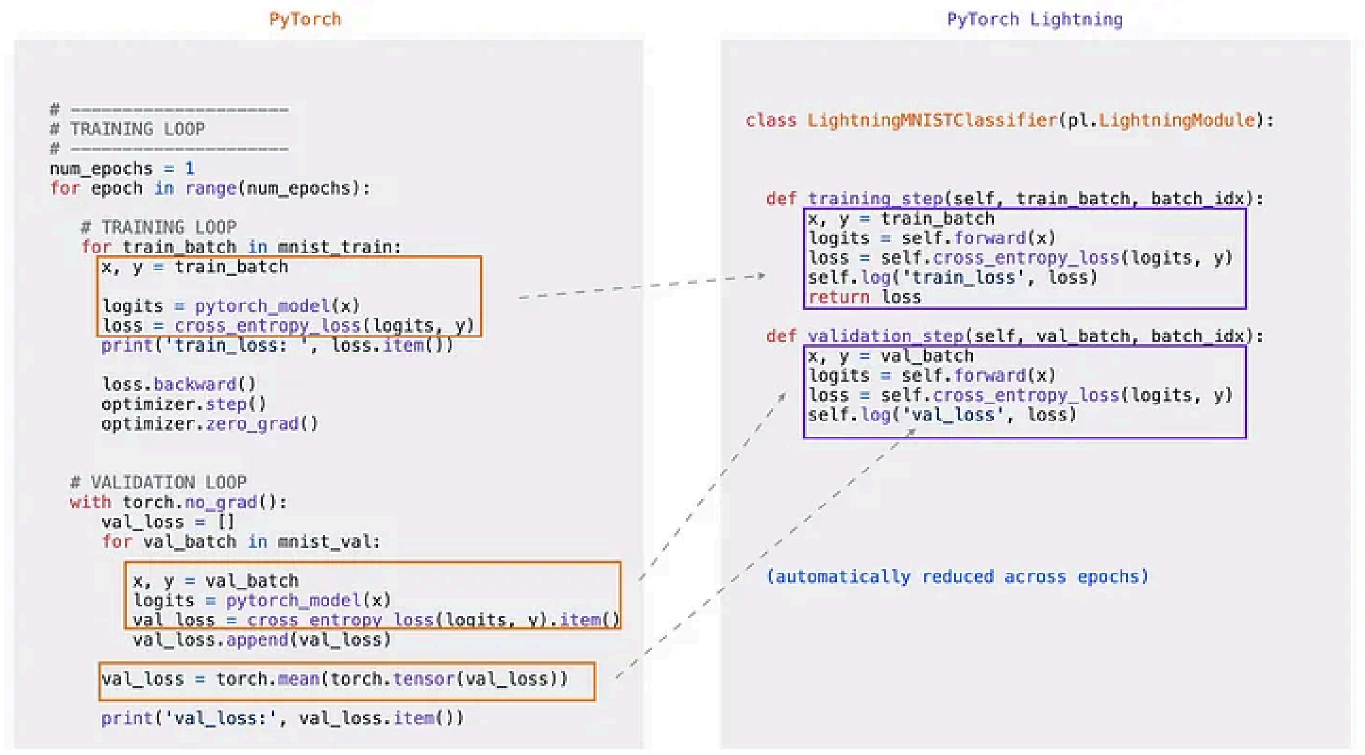
```
num_epochs = 100
for epoch in range(num_epochs):                # (1)
  for batch in dataloader:                     # (2)
    x, y = batch

    logits = model(x)                          # (3)
    loss = cross_entropy_loss(logits, y)       # (4)

    loss.backward()                            # (5)
    optimizer.step()                           # (6)
```

This is where lightning differs though. In PyTorch, you write the for loop yourself which means you have to remember to call the correct things in the right order — this leaves a lot of room for bugs.

Even if your model is simple, it won’t be once you start doing more advanced things like using multiple GPUs, gradient clipping, early stopping, checkpointing, TPU training, 16-bit precision, etc… Your code complexity will quickly explode.

> *Even if your model is simple, it won’t be once you start doing more advanced things*

Here’s are the validation and training loop for both PyTorch and Lightning



```
# ----------------------
# TRAINING LOOP
# ----------------------
num_epochs = 1
for epoch in range(num_epochs):

   # TRAINING LOOP
   for train_batch in mnist_train:
      x, y = train_batch

      logits = pytorch_model(x)
      loss = cross_entropy_loss(logits, y)
      print('train_loss: ', loss.item())

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


  # VALIDATION LOOP
  with torch.no_grad():
      val_loss = []
      for val_batch in mnist_val:

         x, y = val_batch
         logits = pytorch_model(x)
         val_loss = cross_entropy_loss(logits, y).item()
         val_loss.append(val_loss)

      val_loss = torch.mean(torch.tensor(val_loss))

      print('val_loss:', val_loss.item())
```

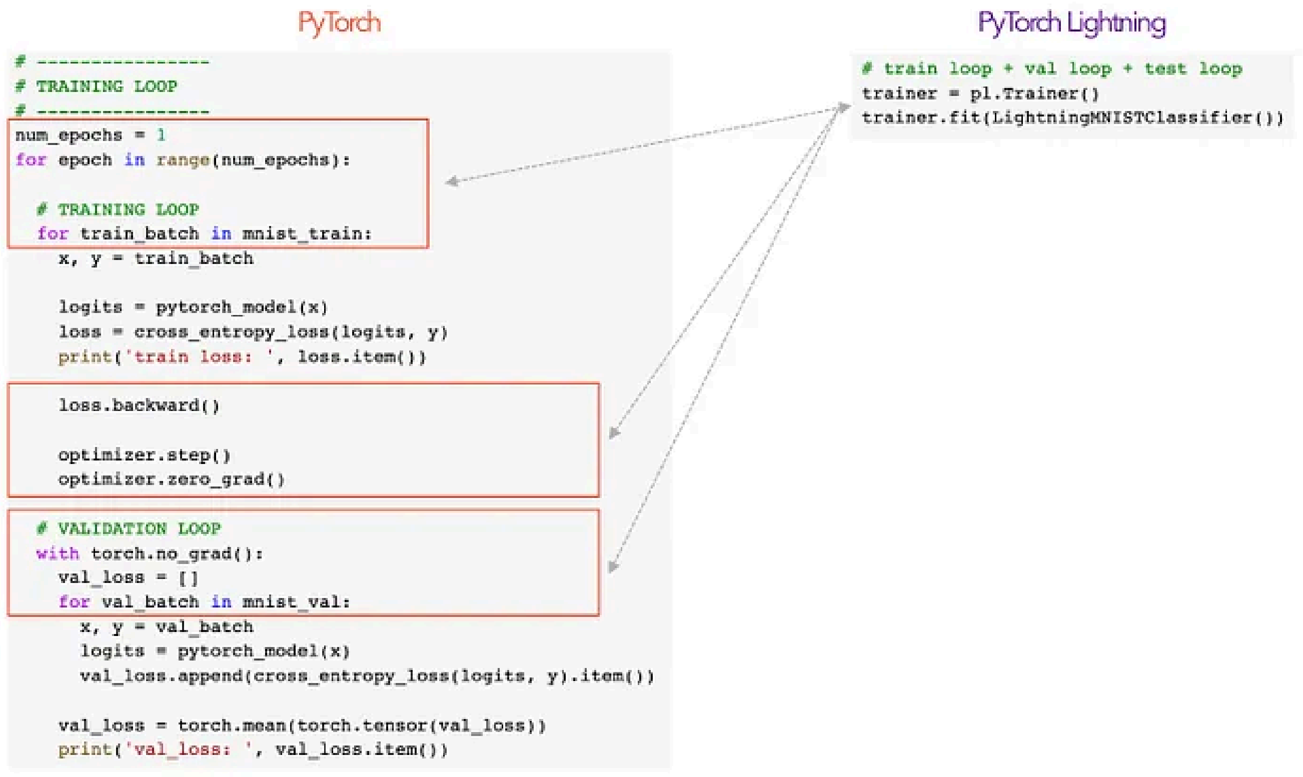
```
class LightningMNISTClassifier(pl.LightningModule):


  def training_step(self, train_batch, batch_idx):
      x, y = train_batch
      logits = self.forward(x)
      loss = self.cross_entropy_loss(logits, y)
      self.log('train_loss', loss)
      return loss

  def validation_step(self, val_batch, batch_idx):
      x, y = val_batch
      logits = self.forward(x)
      loss = self.cross_entropy_loss(logits, y)
      self.log('val_loss', loss)
```

This is the beauty of lightning. It abstracts the boilerplate (the stuff not in boxes) but **leaves everything else unchanged.** This means you are STILL writing PyTorch except your code has been structured nicely.

This increases readability which helps with reproducibility!

The Lightning Trainer

The trainer is how we abstract the boilerplate code.



```
# ----------------
# TRAINING LOOP
# ----------------
num_epochs = 1
for epoch in range(num_epochs):

  # TRAINING LOOP
  for train_batch in mnist_train:
    x, y = train_batch

    logits = pytorch_model(x)
    loss = cross_entropy_loss(logits, y)
    print('train loss: ', loss.item())

    loss.backward()

    optimizer.step()
    optimizer.zero_grad()

  # VALIDATION LOOP
  with torch.no_grad():
    val_loss = []
    for val_batch in mnist_val:
      x, y = val_batch
      logits = pytorch_model(x)
      val_loss.append(cross_entropy_loss(logits, y).item())

    val_loss = torch.mean(torch.tensor(val_loss))
    print('val_loss: ', val_loss.item())
```

```
# train loop + val loop + test loop
trainer = pl.Trainer()
trainer.fit(LightningMNISTClassifier())
```

Again, this is possible because ALL you had to do was organize your PyTorch code into a LightningModule

Full Training Loop for PyTorch

The full MNIST example written in PyTorch is as follows:

Full Training loop in Lightning

The lightning version is EXACTLY the same except:

- The core ingredients have been organized by the LightningModule
- The training/validation loop code has been abstracted by the Trainer

This version does not use the DataModule, but instead keeps the dataloaders defined freely.

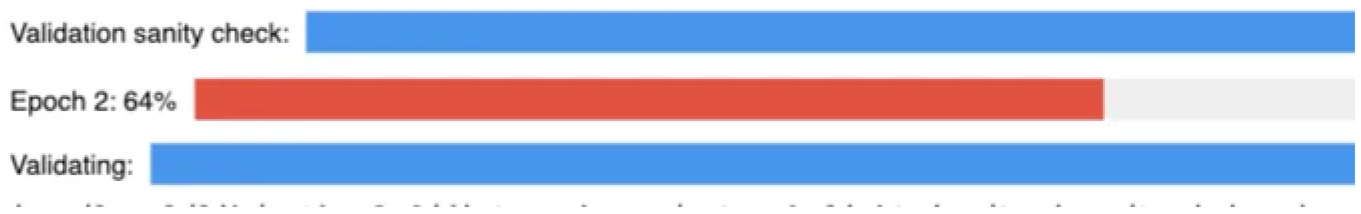
And here is the same code but the data has been grouped under the DataModule and made more reusable.

Highlights

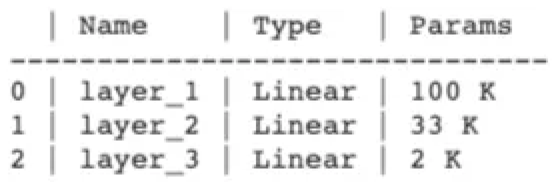
Let’s call out a few key points

1. Without Lightning, the PyTorch code is allowed to be in arbitrary parts. With Lightning, this is structured.
2. It is the same exact code for both except that it's structured in Lightning. (worth saying twice lol).
3. As the project grows in complexity, your code won't because Lightning abstracts out most of it.
4. You retain the flexibility of PyTorch because you have full control over the key points in training. For instance, you could have an arbitrarily complex training_step such as a seq2seq

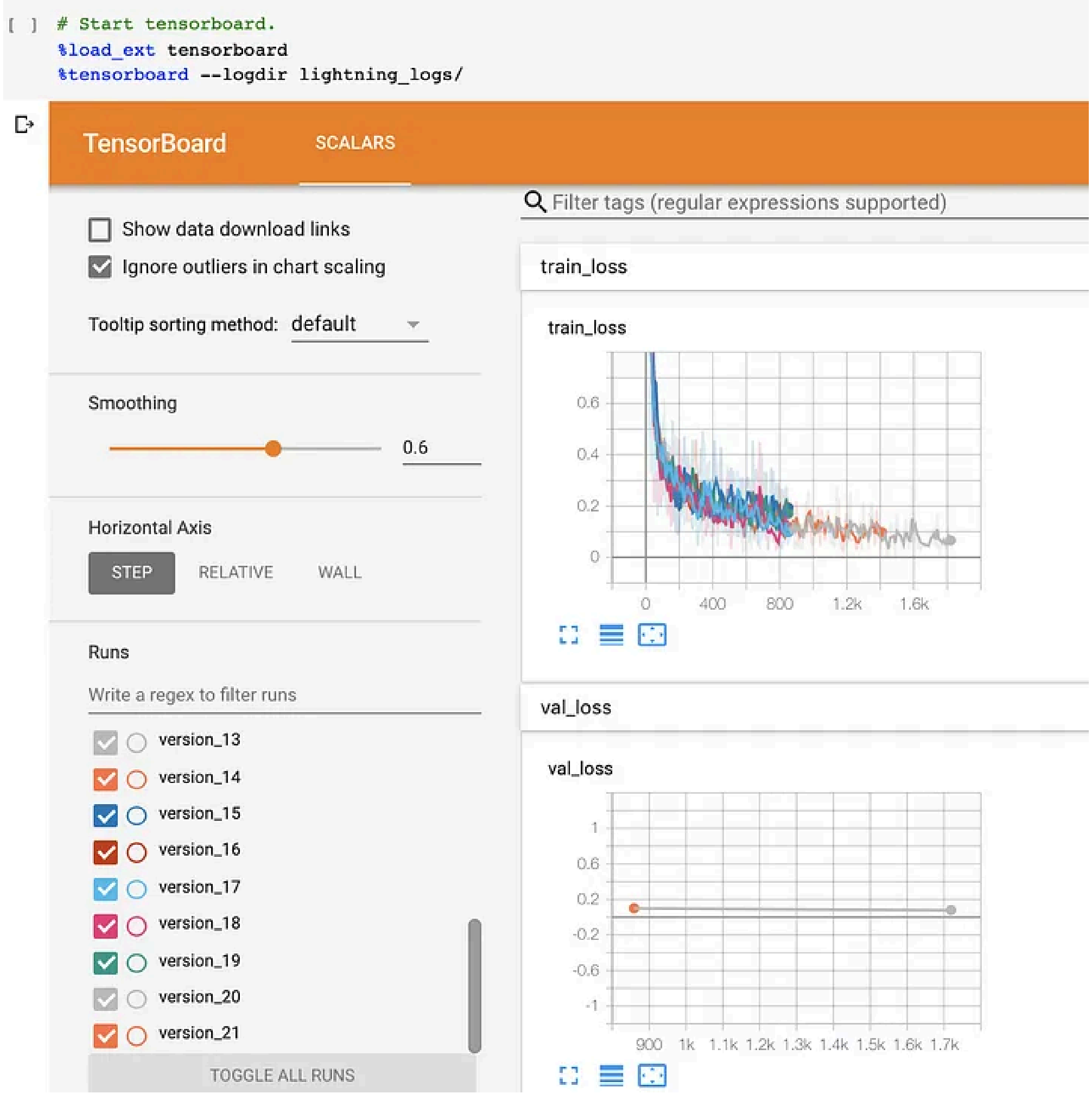
5. In Lightning you got a bunch of freebies such as a sick progress bar



you also got a beautiful weights summary


```
  | Name    | Type   | Params
------------------------------
0 | layer_1 | Linear | 100 K
1 | layer_2 | Linear | 33 K
2 | layer_3 | Linear | 2 K
```

tensorboard logs (yup! you had to nothing to get this)



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

TensorBoard SCALARS

Filter tags (regular expressions supported)

☐ Show data download links
☑ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0.6

Horizontal Axis [✓ STEP] RELATIVE WALL

Runs

Write a regex to filter runs

- ☑ version_13
- ☑ version_14
- ☑ version_15
- ☑ version_16
- ☑ version_17
- ☑ version_18
- ☑ version_19
- ☑ version_20
- ☑ version_21

TOGGLE ALL RUNS

and free checkpointing, and early stopping.

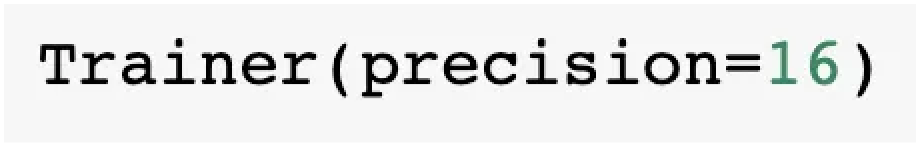
All for free!

Additional Features

But Lightning is known best for out of the box goodies such as TPU training etc…

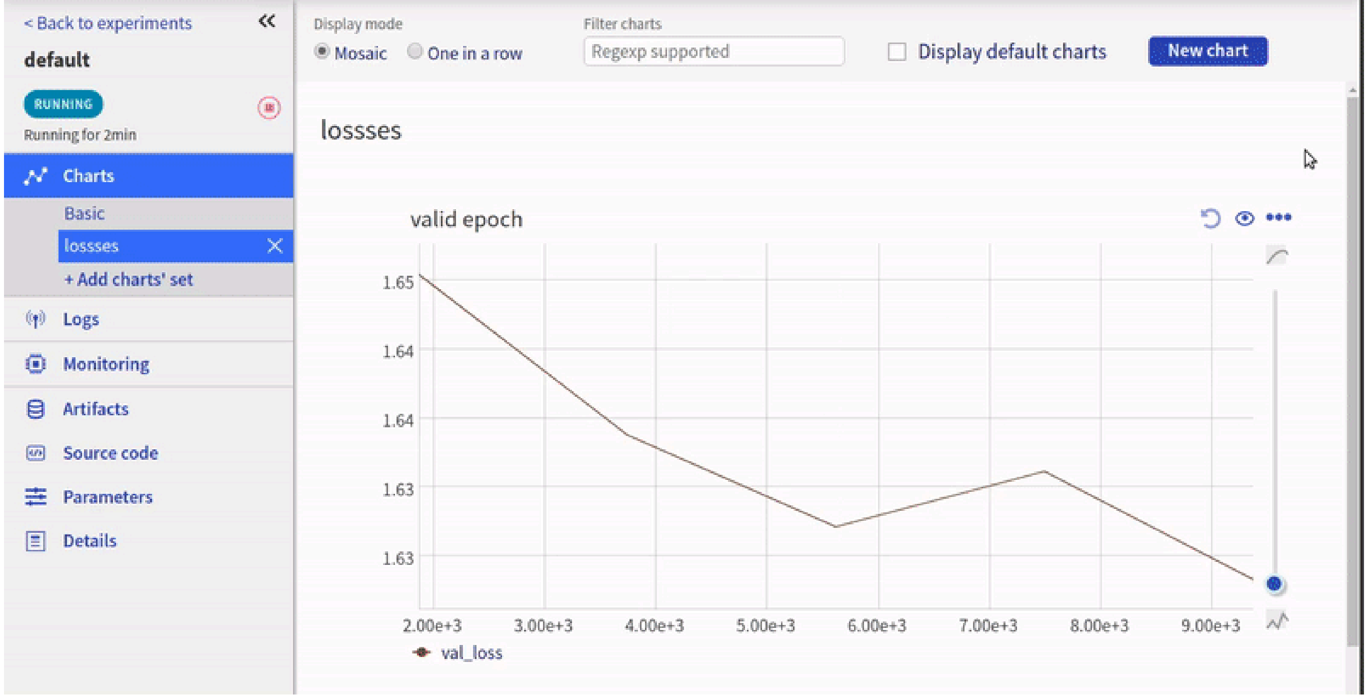
In Lightning, you can train your model on CPUs, GPUs, Multiple GPUs, or TPUs without changing a single line of your PyTorch code.

You can also do 16-bit precision training

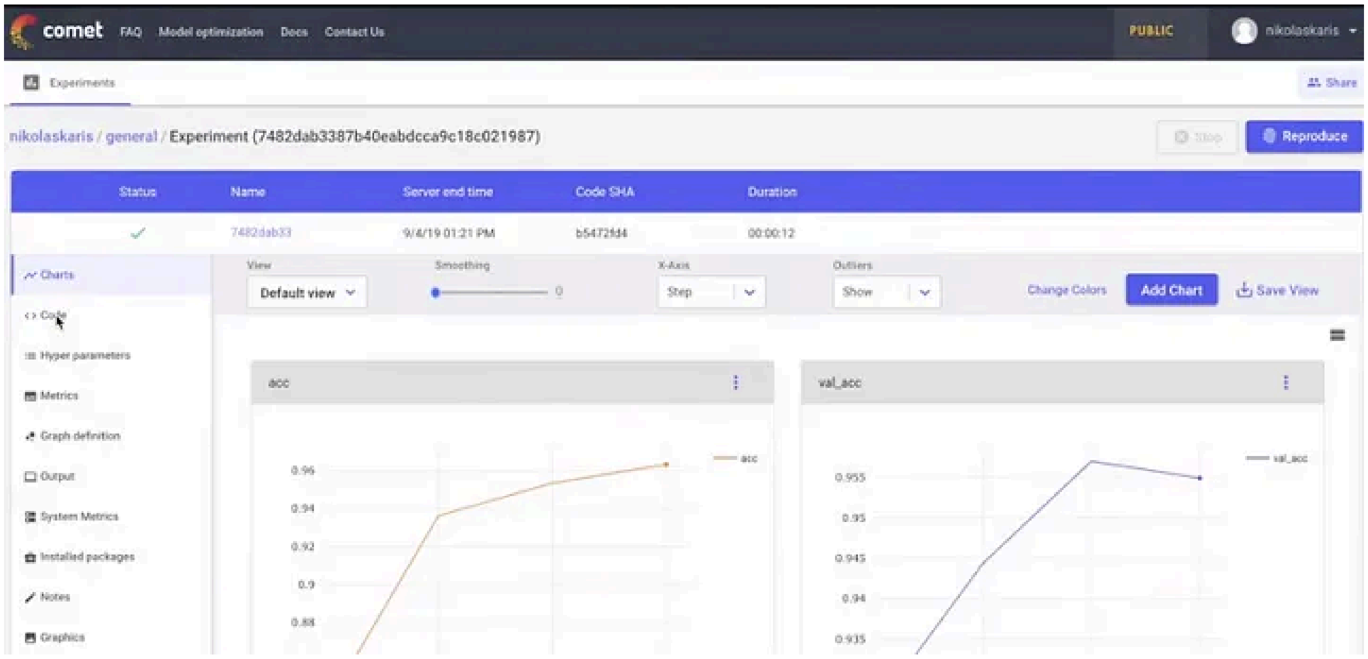


```
Trainer(precision=16)
```

Log using 5 other alternatives to Tensorboard



Logging with Neptune.AI (credits: Neptune.ai)



Status	Name	Server end time	Code SHA	Duration
✓	7482dab33	9/4/19 01:21 PM	b5472fd4	00:00:12

Logging with Comet.ml

We even have a built in profiler that can tell you where the bottlenecks are in your training.

```
trainer = Trainer(..., profiler=True)
```

Setting this flag on gives you this output

```
Profiler Report

Action                  |  Mean duration (s)    |  Total time (s)
-----------------------------------------------------------------
on_epoch_start          |  5.993e-06            |  5.993e-06
get_train_batch         |  0.0087412            |  16.398
on_batch_start          |  5.0865e-06           |  0.0095372
model_forward           |  0.0017818            |  3.3408
model_backward          |  0.0018283            |  3.4282
on_after_backward       |  4.2862e-06           |  0.0080366
optimizer_step          |  0.0011072            |  2.0759
on_batch_end            |  4.5202e-06           |  0.0084753
on_epoch_end            |  3.919e-06            |  3.919e-06
on_train_end            |  5.449e-06            |  5.449e-06
```

Or a more advanced output if you want

```
profiler = AdvancedProfiler()
trainer = Trainer(..., profiler=profiler)
```

```
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__)
```

We can also train on multiple GPUs at once without you doing any work (you still have to submit a SLURM job)

```
Trainer(gpus=8, distributed_backend='ddp')
```

And there are about 40 other features it supports which you can read about in the documentation.

Extensibility With Hooks

You're probably wondering how it's possible for Lightning to do this for you and yet somehow make it so that you have **full control over everything?**

Unlike keras or other high-level frameworks lightning does not hide any of the necessary details. But if you do find the need to modify every aspect of training on your own, then you have two main options.

The first is extensibility by overriding hooks. Here's a non-exhaustive list:

- forward pass
- backward pass
- applying optimizers

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

- setting up distributed training

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

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

    except Exception as e:
        default_port = 12910

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

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

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

- setting up 16 bit

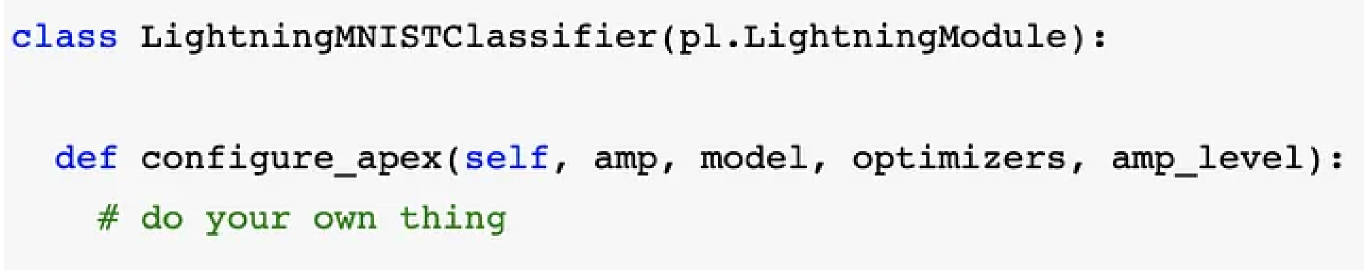
```
# Default implementation used by Trainer.
def configure_apex(self, amp, model, optimizers, amp_level):
    model, optimizers = amp.initialize(
        model, optimizers, opt_level=amp_level,
    )

    return model, optimizers
```

- how we do truncated back prop'

```
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
```

- ...
- anything you would need to configure

These overrides happen in the LightningModule

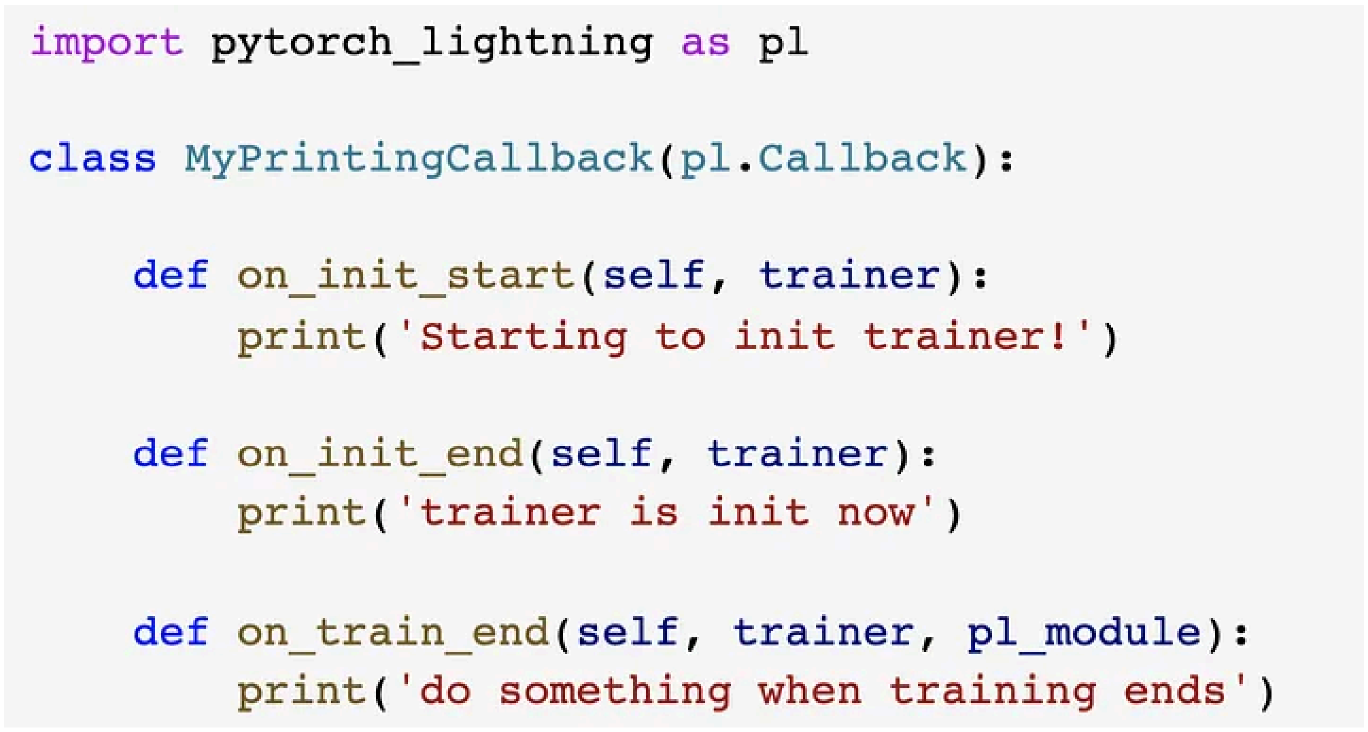


```
class LightningMNISTClassifier(pl.LightningModule):

  def configure_apex(self, amp, model, optimizers, amp_level):
    # do your own thing
```

Extensibility with Callbacks

A callback is a piece of code that you'd like to be executed at various parts of training. In Lightning callbacks are reserved for non-essential code such as logging or something not related to research code. This keeps the research code super clean and organized.

Let's say you wanted to print something or save something at various parts of training. Here's how the callback would look like



```
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')
```

PyTorch Lightning Callback

Now you pass this into the trainer and this code will be called at arbitrary times

```
trainer = pl.Trainer(num_tpu_cores=8, callbacks=[MyPrintingCallback()])
trainer.fit(model)
```

This paradigm keeps your research code organized into three different buckets

1. Research code (LightningModule) (this is the science).
2. Engineering code (Trainer)
3. Non-research related code (Callbacks)

How to start

Hopefully this guide showed you exactly how to get started. The easiest way to start is to run the colab notebook with the MNIST example here.

Or install Lightning

```
pip install pytorch-lightning
```

Or check out the Github page.