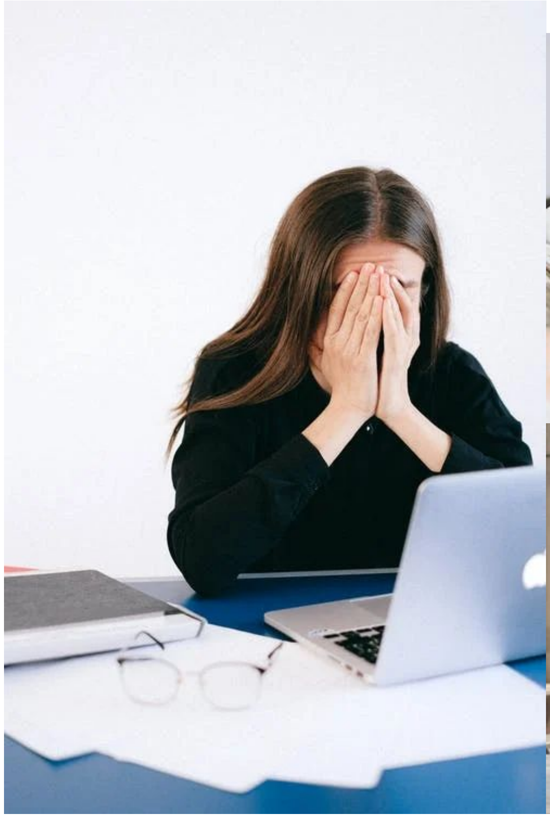
Frustrated?

Does Training take too long?

Does switching from CPU to GPU cause you to have a mild panic attack?

Are you lost in your own code?!



Frustrated?

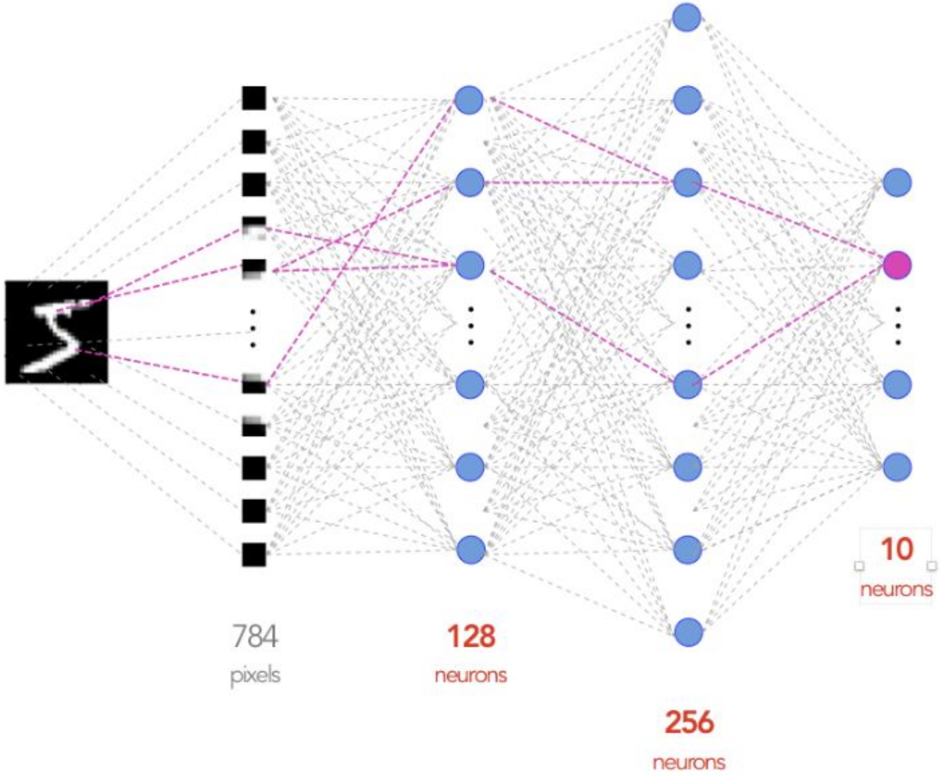
Does Training take too long?

Does switching from CPU to GPU cause you to have a mild panic attack?

Are you lost in your own code?!

You need PyTorch Lightning!

You may ask yoursel how would I implement a 3-layer fully connected neural network to perform classification over MNIST using PyTorch?



The Standard Project Approach

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

# ----------------
# MODEL
# ----------------
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.sizes()

    # (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

# --
# DATA
# ----------------
transform=transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))])
mnist_train = MNIST(os.getcwd(), train=True, download=True, transform=transform)
mnist_test = MNIST(os.getcwd(), train=False, download=True, transform=transform)

# train (55,000 images), val split (5,000 images)
mnist_train, mnist_val = random_split(mnist_train, [55000, 5000])
mnist_test = MNIST(os.getcwd(), train=False, download=True)

# 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)

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

# ----------------
# LOSS
# ----------------
def cross_entropy_loss(logits, labels):
  return F.nll_loss(logits, labels)

# ----------------
# 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())
```

The Model

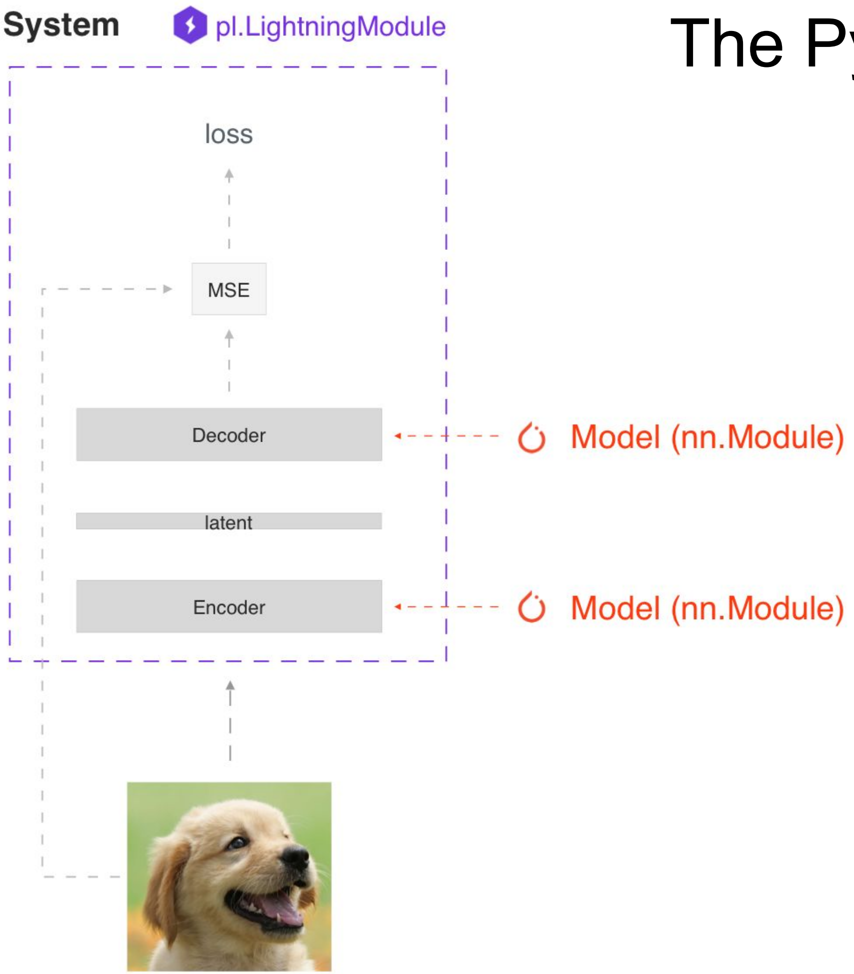
Data

An Optimizer

The Loss

Training & Validation

Can we be more clear?



The PyTorch Lightning Approach

Define the system not the model.

- remove boilerplate code by decoupling research portion
- increase readability
- increase reproducibility
- scalable to any hardware!
- separates training from inference

PyTorch

```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)
```

PyTorch Lightning

```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
```

prevents double manipulations to the data across all GPUs

Notice how we wrap the dataset train, validation, and test split in separate DataLoader member functions in a separate DataModule class.

PyTorch

```python
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
```

PyTorch Lightning

```python
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
```

The Model: same code!

PyTorch

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

PyTorch Lightning

```
class LightningMNISTClassifier(pl.LightningModule):

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

...and the Optimizer and Loss are the same too but are now member functions of the Model class!

PyTorch

```
from torch.nn import functional as F

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

PyTorch Lightning

```
from torch.nn import functional as F

class LightningMNISTClassifier(pl.LightningModule):

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

What about training?

In Pytorch if you...

- Go from CPU to using multiple GPUs
- Add gradient clipping
- Add early stopping
- Add checkpointing
- Use TPUs
- Use 16 bit precision

You will have to change your code and it can quickly start getting complex and difficult to understand.

STOP **You need PyTorch Lightning!**

PyTorch

```python
# ---------------------
# 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())
```

PyTorch Lightning

```python
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)

  (automatically reduced across epochs)
```

Now that’s **readable**!

To run training use **Trainer**:

PyTorch

```
# ----------------
# 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())
```

PyTorch Lightning

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

Look, only two lines, wow!

Notice that the code

- doesn’t loop over epochs or datasets
- doesn’t set the model to evaluation or training
- doesn’t enable or disable gradients!

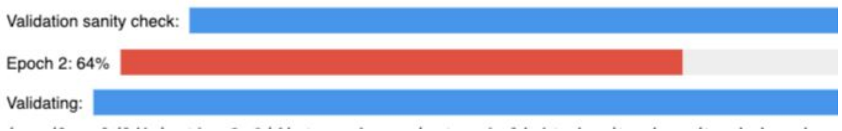
The **Trainer** also has flags such as

- auto_scale_batch_size
- auto_lr_find
- fast_dev_run

...to name a few...

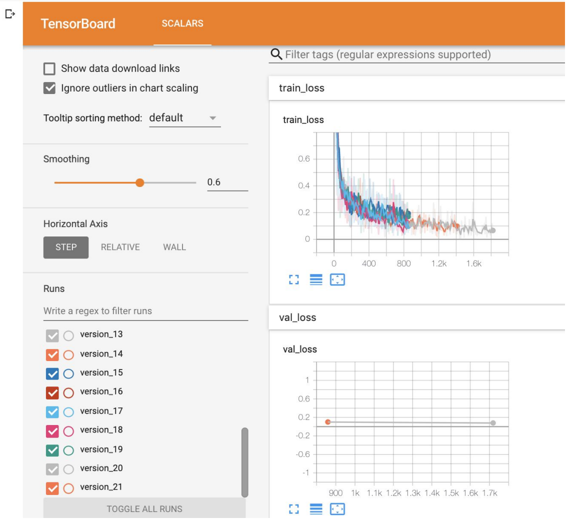
But wait there's more...

- Colorful progress bars
- weights summary
- tensorboard logs
- training on CPUs, GPUs, or TPUs without changing your code
- 16 bit precision training to speed things up
- a profiler
- extensibility with hooks



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

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

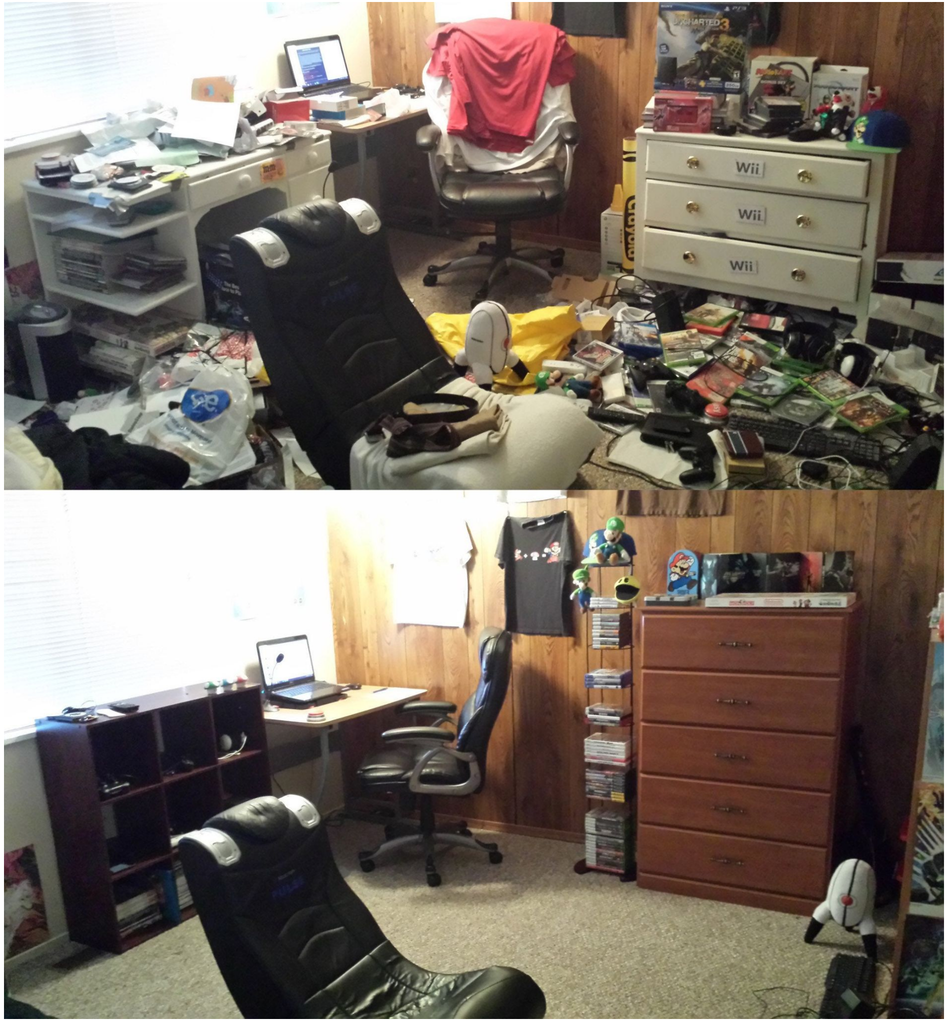


And **Callbacks**

A callback is non-essential code that is not related to the research portion:

```
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')
```

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



And just like that you cleaned your code into:

1. DataLoader code (DataModule)
2. Research code (LightningModule)
3. Engineering code (Trainer)
4. Non-research code (Callbacks)

**Terms and Conditions May Apply:
PyTorch Lightning is good for rudimentary off-the-shelf things like boilerplate code for training but may hide details specific to more complex models.. PyTorch Lightning is not recommended for prototyping more complex, application-specific cases.