Pytorch with Lightning⚡

Why ⚡

- **Full flexibility**
  - Try any ideas using raw PyTorch without the boilerplate.
- **Reproducible + Readable**
  - Decoupled research and engineering code enable reproducibility and better readability.
- **Simple multi-GPU training**
  - Use multiple GPUs/TPUs/HPUs etc... without code changes.
- **Built-in testing**

Torch Model

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 10)
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        return self.fc2(x)
```

Torch Training Loop

```
for epoch in range(EPOCHS):
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data

        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        ...
```

Ok… Someone told me LR scheduler helps training

```
scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gar
for epoch in range(EPOCHS):
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer.zero_grad()

        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        scheduler.step()
```

Perhaps I should clip my gradients

```
scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7,
for epoch in range(EPOCHS):
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer.zero_grad()

        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(),
        optimizer.step()
        scheduler.step()
```

Let’s speed up the training and accumulate gradients

```
scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7,
for epoch in range(EPOCHS):
    loss = 0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer.zero_grad()

        outputs = net(inputs)
        loss += criterion(outputs, labels)  # Compute loss
        if (i+1) % accumulation_steps == 0:
            loss = loss / accumulation_steps
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters
            optimizer.step()
            scheduler.step()
```

Oh.. I found this github repo with something interesting

```
from suspicious_not_tested_github_repo import EarlyStopping

es = EarlyStopping(no_documentation_whatsoever=True)
```

Ups, I forgot to log my metrics and losses

```
scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7,
writer = SummaryWriter()

for epoch in range(EPOCHS):
    loss = 0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer.zero_grad()

        outputs = net(inputs)
        loss += criterion(outputs, labels)  # Compute loss

        if phase = "training":
            if (i+1) % accumulation_steps == 0:
                loss = loss / accumulation_steps
```

Wait my training loop isn't working…

Where is the bug

```
for epoch in range(EPOCHS):
    loss = 0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        #optimizer.zero_grad()

        outputs = net(inputs)
        loss += criterion(outputs, labels)  # Compute loss

        if phase = "training":
            if (i+1) % accumulation_steps == 0:
                loss = loss / accumulation_steps
                loss.backward()
                torch.nn.utils.clip_grad_norm_(model.parame
                optimizer.step()
```

Nice we got a GPU, lets update the code!

```
device = torch.device("cuda:0" if torch.cuda.is_available(
model = model.to(device)

for epoch in range(EPOCHS):
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)

        optimizer.zero_grad()

        outputs = net(inputs)
        loss += criterion(outputs, labels)  # Compute loss
        ...
```

But wait… What if I want to use more than 1 GPU? 😫

Lets simplify it with ⚡

The main principle

You do the cool staff

Lightning takes care of the boilerplate

```python
for epoch in range(EPOCHS):
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        ...
```

```python
class CustomClassifier(pl.LightningModule):

    def __init__(self):
        super().__init__()
        self.criterion = nn.CrossEntropyLoss()
        self.fc1 = nn.Linear(784, 10)

    def forward(self, x):
        return self.fc1(x)

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

        loss = self.criterion(logits, y)
```

How about data?

```
class CustomDatamodule(pl.LightningDataModule):
    def __init__(self, batch_size):
        super().__init__()
        self.batch_size = batch_size

    def setup(self, stage=None):
        self.train_set = StandardTorchDataset(train=True)

        self.val_set = StandardTorchDataset(train=False)

    def train_dataloader(self):
        return DataLoader(self.train_set,
                batch_size=self.batch_size,
                shuffle=True, num_workers=4)
```

We have it all, lets train!

```
model = CustomClassifier()
dm = CustomDatamodule(batch_size=21)

trainer = pl.Trainer()
trainer.fit(model, dm, epochs=37)
```

Can we finally use multiple GPUs?
Or even a TPU?

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

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

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

```
trainer = pl.Trainer(tpu_cores=8)
trainer.fit(model, dm)
```

Lets see a full example

Package to interact with the Operating System

Pytorch package

Pytorch submodule with the Neural Network layers

Pytorch submodule for tensor manipulation

Torchvision submodule with dataset transformations

Torchvision submodule with predefined datasets

Class used to load a dataset

⚡

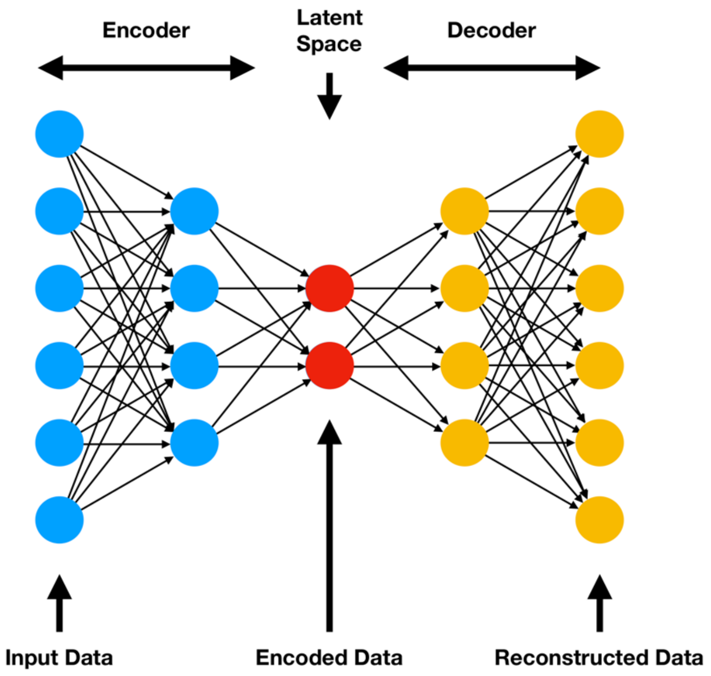
```python
import os
import torch
from torch import nn
import torch.nn.functional as F
from torchvision import transforms
from torchvision.datasets import MNIST
from torch.utils.data import DataLoader
import lightning as L
```

How Lightning is organised

A LightningModule will help us organise our code into 6 sections:

- Initialisation
- Train Loop
- Validation Loop
- Test Loop
- Prediction Loop
- Optimizers and LR Schedulers

Encoder Decoder in Pytorch




```python
class Encoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.l1 = nn.Sequential(nn.Linear(28 * 28, 64), nn.ReLU(), nn.Linear(64, 3))

    def forward(self, x):
        return self.l1(x)


class Decoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.l1 = nn.Sequential(nn.Linear(3, 64), nn.ReLU(), nn.Linear(64, 28 * 28))

    def forward(self, x):
        return self.l1(x)
```

Lets wrap it up with ⚡

For this example we only need the training step and the optimizers

```python
class LitAutoEncoder(L.LightningModule):
    def __init__(self, encoder, decoder):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder

    def training_step(self, batch, batch_idx):
        # training_step defines the train loop.
        x, y = batch
        x = x.view(x.size(0), -1)
        z = self.encoder(x)
        x_hat = self.decoder(z)
        loss = F.mse_loss(x_hat, x)
        return loss

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

Lets define the dataset and the training

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

```
# model
autoencoder = LitAutoEncoder(Encoder(), Decoder())

# train model
trainer = L.Trainer()
trainer.fit(model=autoencoder, train_dataloaders=train_loader)
```

What is happening under the hood

```
autoencoder = LitAutoEncoder(Encoder(), Decoder())
optimizer = autoencoder.configure_optimizers()

for batch_idx, batch in enumerate(train_loader):
    loss = autoencoder.training_step(batch, batch_idx)

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

Validation and Testing

Imports

```
import torch.utils.data as data
from torchvision import datasets
import torchvision.transforms as transforms

# Load data sets
transform = transforms.ToTensor()
train_set = datasets.MNIST(root="MNIST", download=True, train=True, transform=transform)
test_set = datasets.MNIST(root="MNIST", download=True, train=False, transform=transform)
```

test_step function

```
class LitAutoEncoder(L.LightningModule):
    def training_step(self, batch, batch_idx):
        ...

    def test_step(self, batch, batch_idx):
        # this is the test loop
        x, y = batch
        x = x.view(x.size(0), -1)
        z = self.encoder(x)
        x_hat = self.decoder(z)
        test_loss = F.mse_loss(x_hat, x)
        self.log("test_loss", test_loss)
```

How to test the model?

```
from torch.utils.data import DataLoader

# initialize the Trainer
trainer = Trainer()

# test the model
trainer.test(model, dataloaders=DataLoader(test_set))
```

Validation and Testing

validation_step function

```
class LitAutoEncoder(L.LightningModule):
    def training_step(self, batch, batch_idx):
        ...

    def validation_step(self, batch, batch_idx):
        # this is the validation loop
        x, y = batch
        x = x.view(x.size(0), -1)
        z = self.encoder(x)
        x_hat = self.decoder(z)
        val_loss = F.mse_loss(x_hat, x)
        self.log("val_loss", val_loss)
```

How to use the validation slipt?

```
from torch.utils.data import DataLoader

train_loader = DataLoader(train_set)
valid_loader = DataLoader(valid_set)
model = LitAutoEncoder(...)

# train with both splits
trainer = L.Trainer()
trainer.fit(model, train_loader, valid_loader)
```

Saving and Loading Checkpoints

Defining the root directory for the checkpoints

```
# saves checkpoints to 'some/path/' at every epoch end
trainer = Trainer(default_root_dir="some/path/")
```

How to load from a checkpoint

```
model = MyLightningModule.load_from_checkpoint("/path/to/checkpoint.ckpt")

# disable randomness, dropout, etc...
model.eval()

# predict with the model
y_hat = model(x)
```