Lightning

How to make cleaner deep learning codebase

Problems with pure PyTorch

- Lots of boilerplate code
- Hard to scale (multiple GPUs, switch to TPU, etc.)
- Hard coded logging (parameters, figure, etc.)
- Little reproducibility
- No common structure

Introducing Pytorch Lightning

- Only define what you need
- Change the type and number of accelerators (CPU, GPU, TPU) in one line
- Fully Flexible
- Automatic basic logging

LightningModule

- Only define what you need
  - Training step
  - Validation step
  - Optimizers
- Access to multiple "hooks"
  - On epoch end
  - On validation start
  - Etc.
- Can be used as a torch nn.Module

```python
# define the LightningModule
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.
        # it is independent of forward
        x, _ = batch
        x = x.view(x.size(0), -1)
        z = self.encoder(x)
        x_hat = self.decoder(z)
        loss = nn.functional.mse_loss(x_hat, x)
        # Logging to TensorBoard (if installed) by default
        self.log("train_loss", loss)
        return loss

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

Training

```
model = MyLightningModule()

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

```
# enable grads
torch.set_grad_enabled(True)

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

    # train step
    loss = training_step(batch)

    # clear gradients
    optimizer.zero_grad()

    # backward
    loss.backward()

    # update parameters
    optimizer.step()

    losses.append(loss)
```

```
# CPU accelerator
trainer = Trainer(accelerator="cpu")

# Training with GPU Accelerator using 2 GPUs
trainer = Trainer(devices=2, accelerator="gpu")

# Training with TPU Accelerator using 8 tpu cores
trainer = Trainer(devices=8, accelerator="tpu")

# Training with GPU Accelerator using the DistributedDataParallel strategy
trainer = Trainer(devices=4, accelerator="gpu", strategy="ddp")
```

```
from lightning.pytorch import Trainer, seed_everything

seed_everything(42, workers=True)
# sets seeds for numpy, torch and python.random.
model = Model()
trainer = Trainer(deterministic=True)
```

LightningDataModule

```python
# regular PyTorch
test_data = MNIST(my_path, train=False, download=True)
predict_data = MNIST(my_path, train=False, download=True)
train_data = MNIST(my_path, train=True, download=True)
train_data, val_data = random_split(train_data, [55000, 5000])

train_loader = DataLoader(train_data, batch_size=32)
val_loader = DataLoader(val_data, batch_size=32)
test_loader = DataLoader(test_data, batch_size=32)
predict_loader = DataLoader(predict_data, batch_size=32)
```

```python
class MNISTDataModule(L.LightningDataModule):
    def __init__(self, data_dir: str = "path/to/dir", batch_size: int = 32):
        super().__init__()
        self.data_dir = data_dir
        self.batch_size = batch_size

    def setup(self, stage: str):
        self.mnist_test = MNIST(self.data_dir, train=False)
        self.mnist_predict = MNIST(self.data_dir, train=False)
        mnist_full = MNIST(self.data_dir, train=True)
        self.mnist_train, self.mnist_val = random_split(
            mnist_full, [55000, 5000], generator=torch.Generator().manual_seed(42)
        )

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

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

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

    def predict_dataloader(self):
        return DataLoader(self.mnist_predict, batch_size=self.batch_size)

    def teardown(self, stage: str):
        # Used to clean-up when the run is finished
        ...
```

LightningDataModule

```python
class MNISTDataModule(L.LightningDataModule):
    def __init__(self, data_dir: str = "path/to/dir", batch_size: int = 32):
        super().__init__()
        self.data_dir = data_dir
        self.batch_size = batch_size

    def setup(self, stage: str):
        self.mnist_test = MNIST(self.data_dir, train=False)
        self.mnist_predict = MNIST(self.data_dir, train=False)
        mnist_full = MNIST(self.data_dir, train=True)
        self.mnist_train, self.mnist_val = random_split(
            mnist_full, [55000, 5000], generator=torch.Generator().manual_seed(42)
        )

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

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

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

    def predict_dataloader(self):
        return DataLoader(self.mnist_predict, batch_size=self.batch_size)

    def teardown(self, stage: str):
        # Used to clean-up when the run is finished
        ...
```

```python
mnist = MNISTDataModule(my_path)
model = LitClassifier()

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

Very easy to reuse across projects

Logging

- Easy logging interface
- Chose backend in the trainer
- Unified API in training : self.log("my_metric", x)
- Ability to access original logger for more functionality :
  - experiment = self.logger.experiment
- Automatic progress bar support
- Automatic logging of hyperparameters
- Support for :
  - Tensorboard
  - Comet
  - CSV
  - MLFlow
  - Neptune
  - Wandb

```
from lightning.pytorch import loggers as pl_loggers

tb_logger = pl_loggers.TensorBoardLogger(save_dir="logs/")
trainer = Trainer(logger=tb_logger)
```

Composability

- Define common logic across possible tasks
- Use inheritance to implement differences
- Doable for both Modules AND Datamodules

Easy Checkpointing

- Lightning automatically save checkpoints
- Checkpointing is very easy compared to basic torch
- Allow to resume training by saving gradients, epochs, etc.

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

Conclusion

- Highly recommend to use a framework like Lightning
- Other possibilities exist like Ignite
- Make it very easy to reuse code between projects and inside a projects
- Make projects more readable
- Make projects easily reproducible

USE IT