Bonus: Organizing PyTorch

LightningModule and the Trainer use the underlying tech as Fabric, but offer more structure and features

```
pip install lightning

from lightning import LightningModule

from lightning import Trainer
```

Formerly:

```
from pytorch_lightning import LightningModule
from pytorch_lightning import Trainer
```

Now:

```
from lightning import LightningModule
from lightning import Trainer
```

Lightning Trainer

```
trainer = Trainer(
    max_epochs=NUM_EPOCHS,
    accelerator="auto",  # Uses GPUs, TPUs etc. if available
    devices="auto",  # Uses all available GPUs/TPUs if applicable
    logger=logger,
    deterministic=True,
    ...
)

trainer.fit(model=lightning_model, ...)
```

The Trainer trains the Lightning model (which wraps the PyTorch model)

LightningModule

```python
# Regular PyTorch Module
class PyTorchNet(torch.nn.Module):
    def __init__(self, ...):
        super().__init__()
        # Initialize layers ...

    def forward(self, x):
        # ...
        return logits

```

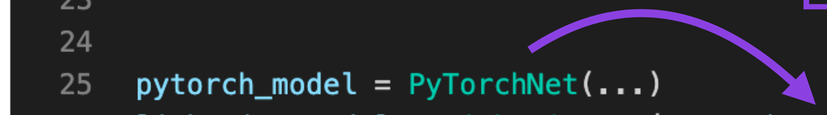
```python
# LightningModule that receives a PyTorch model as input
class LightningNet(LightningModule):
    def __init__(self, model, ...):
        super().__init__()
        self.model = model

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

    def training_step(self, batch, batch_idx):
        self.log(...)
        return loss  # this is passed to the optimizer for training

    def validation_step(self, batch, batch_idx):
        self.log(...)

    def test_step(self, batch, batch_idx):
        self.log(...)

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)
        return optimizer


pytorch_model = PyTorchNet(...)
lightning_model = LightningNet(pytorch_model, ...)
```

LightningModule wraps a regular PyTorch model

```python
# Regular PyTorch Module
class PyTorchNet(torch.nn.Module):
    def __init__(self, ...):
        super().__init__()
        # Initialize layers ...

    def forward(self, x):
        # ...
        return logits

```

```python
# LightningModule that receives a PyTorch model as input
class LightningNet(LightningModule):
    def __init__(self, model, ...):
        super().__init__()
        self.model = model

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

    def training_step(self, batch, batch_idx):
        self.log(...)
        return loss  # this is passed to the optimizer for training

    def validation_step(self, batch, batch_idx):
        self.log(...)

    def test_step(self, batch, batch_idx):
        self.log(...)

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

```python
# LightningModule that receives a PyTorch model as input
class LightningNet(LightningModule):
    def __init__(self, model, ...):
        super().__init__()
        self.model = model

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

    def training_step(self, batch, batch_idx):
        self.log(...)
        return loss  # this is passed to the optimizer for training

    def validation_step(self, batch, batch_idx):
        self.log(...)

    def test_step(self, batch, batch_idx):
        self.log(...)

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)
        return optimizer


pytorch_model = PyTorchNet(...)
lightning_model = LightningNet(pytorch_model, ...)
```



LightningModule wraps a regular PyTorch model

Lightning Trainer

```
trainer = Trainer(
    max_epochs=NUM_EPOCHS,
    accelerator="auto",  # Uses GPUs, TPUs etc. if available
    devices="auto",  # Uses all available GPUs/TPUs if applicable
    logger=logger,
    deterministic=True,
    ...
)

trainer.fit(model=lightning_model, ...)
```

The Trainer trains the Lightning model (which wraps the PyTorch model)

Optional: DataModules

A DataModule is a shareable, reusable class that encapsulates data (pre)processing and data loading logic

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

    def prepare_data(self):
        # download
        MNIST(self.data_dir, train=True, download=True)
        MNIST(self.data_dir, train=False, download=True)

    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])

    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
        ...
```

Dataset download logic goes here
(avoids duplication in multi-GPU training settings)

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

    def prepare_data(self):
        # download
        MNIST(self.data_dir, train=True, download=True)
        MNIST(self.data_dir, train=False, download=True)

    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])

    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
        ...
```

Steps that can/need to be executed on each GPU go here

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

    def prepare_data(self):
        # download
        MNIST(self.data_dir, train=True, download=True)
        MNIST(self.data_dir, train=False, download=True)

    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])

    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
        ...
```

Setup data loaders as usual

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

    def prepare_data(self):
        # download
        MNIST(self.data_dir, train=True, download=True)
        MNIST(self.data_dir, train=False, download=True)

    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])

    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
        ...
```

These are used by ...

trainer.fit()

trainer.fit()
trainer.validate()

trainer.test()

trainer.predict()

Once we defined the DataModule, the usage is straight-forward:

```
dm = MNISTDataModule()
model = Model()

trainer.fit(model, datamodule=dm)
trainer.test(datamodule=dm)
trainer.validate(datamodule=dm)
trainer.predict(datamodule=dm)
```

Adding Functionality With Custom Callbacks

What is a callback?

A self-contained program/function
to extend existing functionality

It hooks into your code

What can you do with callbacks?

- Update learning rates
- Visualize gradients
- Send emails during training (or when it finished)
- …
- You are only limited by your imagination

Looking at a pre-built checkpointing callback

```
from lightning.pytorch.callbacks import ModelCheckpoint

lightning_model = LightningModel(pytorch_model, learning_rate=...)

checkpoint_callback = ModelCheckpoint(
        save_top_k=1, mode="max", monitor="valid_acc"
)  # save top 1 model



trainer = L.Trainer(
    max_epochs=NUM_EPOCHS,
    callbacks=[callbacks],
    ...
)
```

Let’s build our custom callback

```
from lightning.pytorch.callbacks import Callback
import time


class CustomTimingCallback(Callback):
    def on_train_start(self, trainer, lightning_module):
        self.start = time.time()
        print("Training is starting.")

    def on_train_end(self, trainer, lightning_module):
        self.end = time.time()
        total_minutes = (self.end - self.start) / 60
        print(f"Training has finished. It took {total_minutes} min.")
```

Let’s build our custom callback

```
from lightning.pytorch.callbacks import Callback
import time


class CustomTimingCallback(Callback):
    def on_train_start(self, trainer, lightning_module):
        self.start = time.time()
        print("Training is starting.")

    def on_train_end(self, trainer, lightning_module):
        self.end = time.time()
        total_minutes = (self.end - self.start) / 60
        print(f"Training has finished. It took {total_minutes} min.")


callbacks=[CustomTimingCallback()]

trainer = L.Trainer(
    max_epochs=5,
    callbacks=callbacks,
)

trainer.fit(model=lightning_model, datamodule=dm)
```

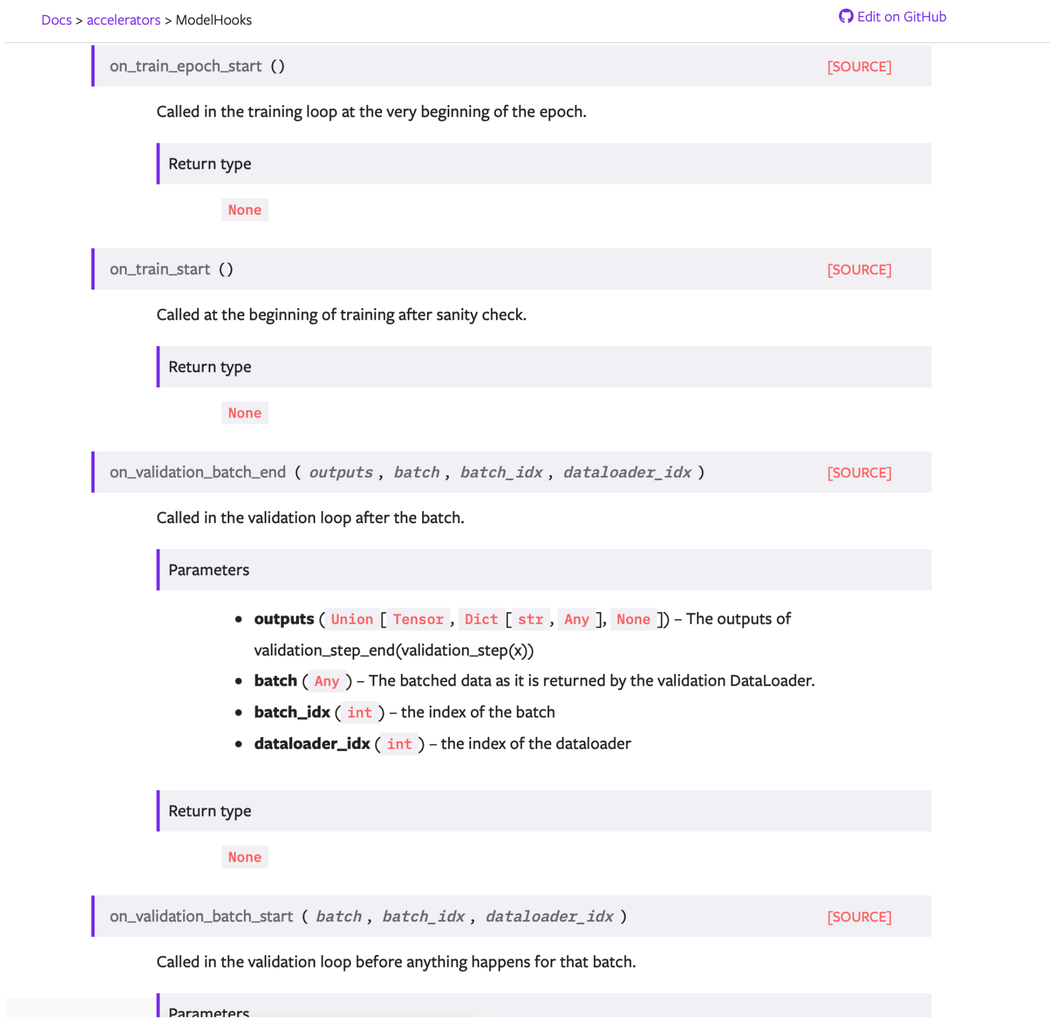
Let’s build our custom callback

```python
from lightning.pytorch.callbacks import Callback
import time


class CustomTimingCallback(Callback):
    def on_train_start(self, trainer, lightning_module):
        self.start = time.time()
        print("Training is starting.")

    def on_train_end(self, trainer, lightning_module):
        self.end = time.time()
        total_minutes = (self.end - self.start) / 60
        print(f"Training has finished. It took {total_minutes} min.")


callbacks=[CustomTimingCallback()]

trainer = L.Trainer(
    max_epochs=5,
    callbacks=callbacks,
)

trainer.fit(model=lightning_model, datamodule=dm)
```

```
GPU available: True (mps), used: False
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs

  | Name      | Type       | Params
-----------------------------------------
0 | model     | PyTorchMLP | 40.8 K
1 | train_acc | Accuracy   | 0
2 | val_acc   | Accuracy   | 0
3 | test_acc  | Accuracy   | 0
-----------------------------------------
40.8 K    Trainable params
0         Non-trainable params
40.8 K    Total params
0.163     Total estimated model params size (MB)
Training is starting.
Epoch 4: 100% ████████████ 938/938 [00:03<00:00, 235.17it/s, loss=0.155, v_num=1, val_loss=0.171, val_acc=0.953, train_acc=0.954]
`Trainer.fit` stopped: `max_epochs=5` reached.
Training has finished. It took 0.33937496741612755 min.
```

Callback hooks

There are many, many hooks …



Docs > accelerators > ModelHooks

Edit on GitHub

on_train_epoch_start () [SOURCE]

Called in the training loop at the very beginning of the epoch.

Return type

None

on_train_start () [SOURCE]

Called at the beginning of training after sanity check.

Return type

None

on_validation_batch_end (*outputs* , *batch* , *batch_idx* , *dataloader_idx*) [SOURCE]

Called in the validation loop after the batch.

Parameters

- **outputs** (Union [Tensor , Dict [str , Any], None]) – The outputs of validation_step_end(validation_step(x))
- **batch** (Any) – The batched data as it is returned by the validation DataLoader.
- **batch_idx** (int) – the index of the batch
- **dataloader_idx** (int) – the index of the dataloader

Return type

None

on_validation_batch_start (*batch* , *batch_idx* , *dataloader_idx*) [SOURCE]

Called in the validation loop before anything happens for that batch.

Parameters

https://lightning.ai/docs/pytorch/stable/api/lightning.pytorch.core.hooks.ModelHooks.html

Exercise:

Develop a custom callback to track the training & validation accuracy differences

Hint: Use

```
lightning_module.train_acc.compute()
lightning_module.val_acc.compute()
```

```
from lightning.pytorch.callbacks import Callback
```

```
L.pytorch.seed_everything(123)



lightning_model = LightningModel(model=pytorch_model, learning_rate=5e-5)

trainer = L.Trainer(
    max_epochs=20,
    callbacks=[CustomCallback()],
    accelerator="cpu",
    devices=1,
)
```

```
for i, diff in enumerate(train_val_diff):
    print(f"Epoch {i:03d}: Train-Validation accuracy difference: {diff*100:.2f}%", )
```

```
Epoch 000: Train-Validation accuracy difference: -14.06%
Epoch 001: Train-Validation accuracy difference: -24.90%
Epoch 002: Train-Validation accuracy difference: -3.58%
Epoch 003: Train-Validation accuracy difference: -1.35%
Epoch 004: Train-Validation accuracy difference: -0.60%
Epoch 005: Train-Validation accuracy difference: -0.27%
Epoch 006: Train-Validation accuracy difference: -0.28%
Epoch 007: Train-Validation accuracy difference: -0.11%
Epoch 008: Train-Validation accuracy difference: 0.06%
Epoch 009: Train-Validation accuracy difference: 0.05%
Epoch 010: Train-Validation accuracy difference: 0.29%
Epoch 011: Train-Validation accuracy difference: 0.24%
Epoch 012: Train-Validation accuracy difference: 0.19%
Epoch 013: Train-Validation accuracy difference: 0.30%
Epoch 014: Train-Validation accuracy difference: 0.37%
Epoch 015: Train-Validation accuracy difference: 0.26%
Epoch 016: Train-Validation accuracy difference: 0.24%
Epoch 017: Train-Validation accuracy difference: 0.41%
Epoch 018: Train-Validation accuracy difference: 0.34%
Epoch 019: Train-Validation accuracy difference: 0.10%
Epoch 020: Train-Validation accuracy difference: 0.25%
```