OpenBioML PyTorch Lightning workshop (1/2)

Adrian Wälchli, Research Engineer
Luca Antiga, CTO

OpenBioML PyTorch Lightning workshop

Session 1: Thu 23 Feb, 3pm ET

Session 2: Thu 2 Mar, 3pm ET

https://harvard.zoom.us/j/97375262666

OpenBioML PyTorch Lightning workshop

Session 1:

- Intro to PyTorch Lightning + Fabric
- Hands-on: raw PyTorch -> Fabric -> PyTorch Lightning Trainer
- A look into OpenFold

By the end of Session 1 you will know how to build a model with Lightning and train it.

Distributed. On a SLURM cluster.

OpenBioML PyTorch Lightning workshop

Session 2:

- Intro to core distributed concepts
- Hands-on: how to debug and optimize performance, single node and distributed, running benchmarks
- More on OpenFold
- Quick look at the Lightning platform

By the end of Session 2 you will know how to make sure you are setting up your training correctly and verify you are leveraging your hardware the best.

Join us here

discord	discord.gg/MWAEvnC5fU
forums	lightning.ai/forums
twitter	@LightningAI

https://linktr.ee/lightningai

Intro: PyTorch Lightning and Lightning Fabric

PyTorch Lightning in the wild

Stable diffusion

https://github.com/Stability-AI/stablediffusion
https://github.com/CompVis/stable-diffusion

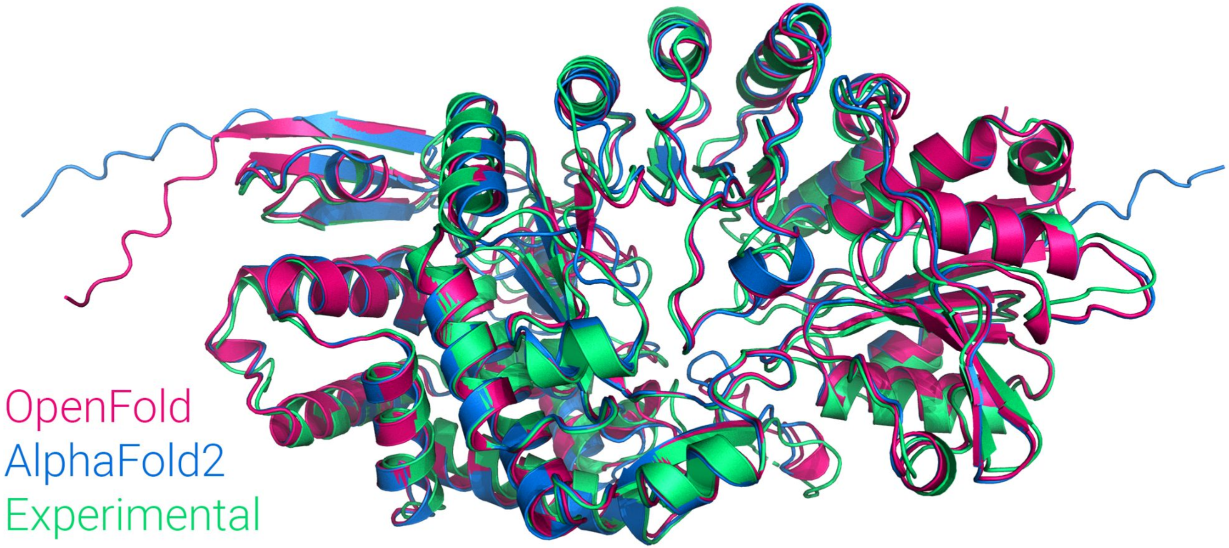
PyTorch Lightning in the wild

Instruct Pix2Pix

https://github.com/timothybrooks/instruct-pix2pix

PyTorch Lightning in the wild

OpenFold



https://github.com/aqlaboratory/openfold
https://www.biorxiv.org/content/10.1101/2022.11.20.517210v1.full.pdf

PyTorch Lightning in the wild

NeMo Megatron (framework)

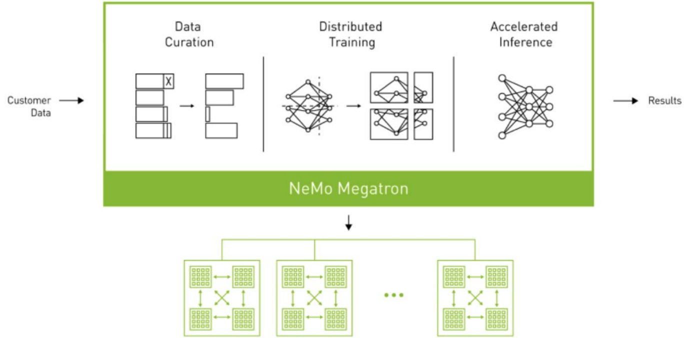
NVIDIA
NEMO

part of NeMo
https://nvidia.github.io/NeMo/

Acoustic
Decoders
Language
BERT
Speech Synthesis
Vision Encoder

NVIDIA NeMo Megatron is an end-to-end framework for training and deploying LLMs with billions and trillions of parameters.

Download Now



https://developer.nvidia.com/nemo/megatron

PyTorch Lightning

You do the science, we do the engineering

33 million+
DOWNLOADS

780+
CONTRIBUTORS

pip install lightning | 21,512

13,000+
PROJECTS USING LIGHTNING

6,000+
SLACK MEMBERS

10,000+ ORGANIZATIONS BUILD WITH LIGHTNING

amazon · NVIDIA · Meta · Microsoft · habana · MARS

https://lightning.ai

PyTorch Lightning

Organized PyTorch

LightningModule

Trainer

```
import lightning as L
from torch import n, optim

encoder = nn.Sequential(nn.Linear(28 * 28, 64), nn.ReLU(), nn.Linear(64, 3))
decoder = nn.Sequential(nn.Linear(3, 64), n.ReLU() , nn. Linear (64, 28 * 28))

class LitAutoEncoder(L.LightningModule):
    def _init__(self, encoder, decoder):
        super()._init__()
        self.encoder = encoder
        self.decoder = decoder

    def training_step(self, batch, batch_idx):
        X, y = 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)
        self.log("train_loss", loss) return loss

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

autoencoder = LitAutoEncoder(encoder, decoder)
dataset = MIST(os.getcwd(), download=True, transform=ToTensor())

train_loader = utils.data.DataLoader (dataset)

trainer = L.Trainer(limit_train_batches=100, max_epochs=1)
trainer.fit(model=autoencoder, train_dataloaders=train_loader)
```

PyTorch Lightning

Accelerators

GPU, TPU, HPU, IPU, MPS

Strategies

DDP, FSDP, DeepSpeed, Colossal AI

Precision

Callbacks

```
# train on 4 GPUs
trainer = Trainer(
     devices=4,
     accelerator="gpu"
)

# train 1B+ parameter models with Deepspeed/fsdp
trainer = Trainer(
    devices=4,
    accelerator="gpu",
    strategy="deepspeed_stage_2",
    precision=16
)

# 20+ helpful flags for rapid idea iteration
trainer = Trainer(
    max_epochs=10,
    min_epochs=5,
    overfit_batches=1
)

# access the latest state of the art techniques
trainer = Trainer(callbacks=[StochasticWeightAveraging(...)])
```

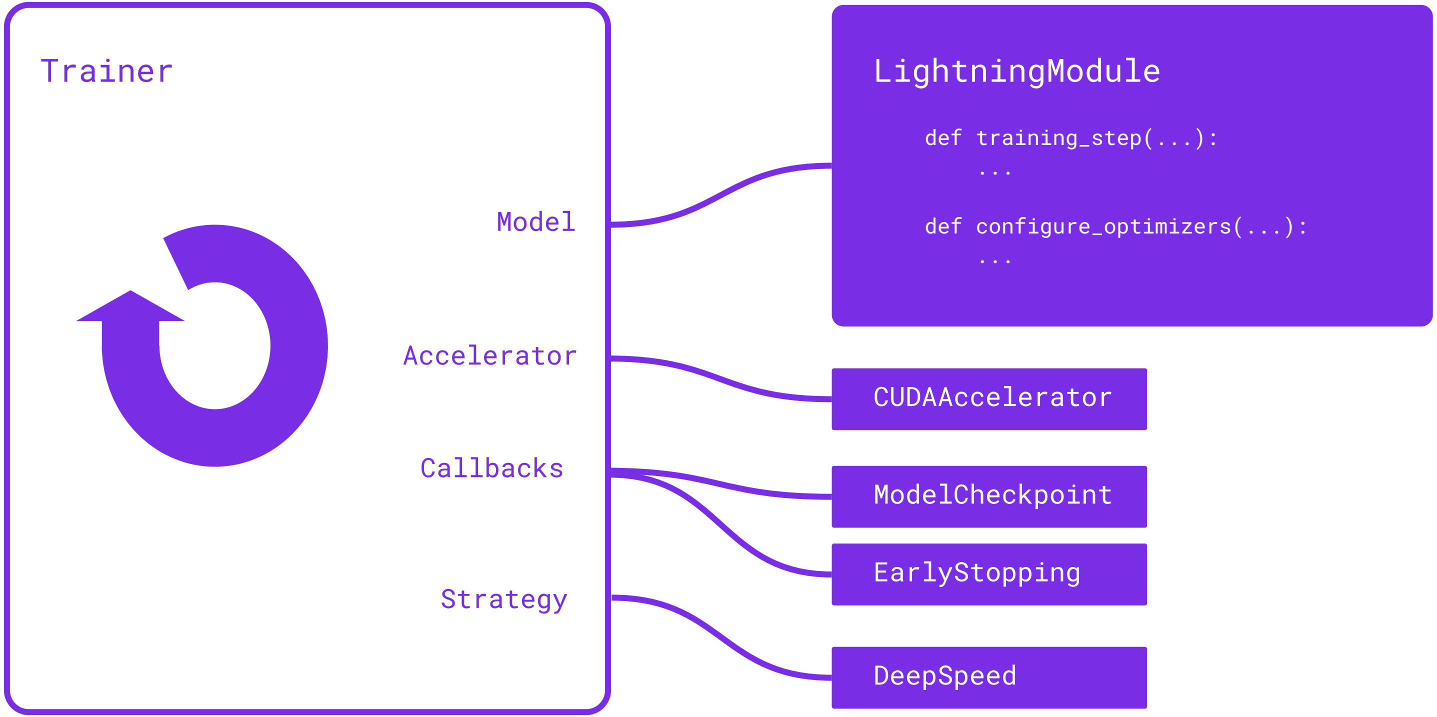
PyTorch Lightning

The anatomy of `.fit()`

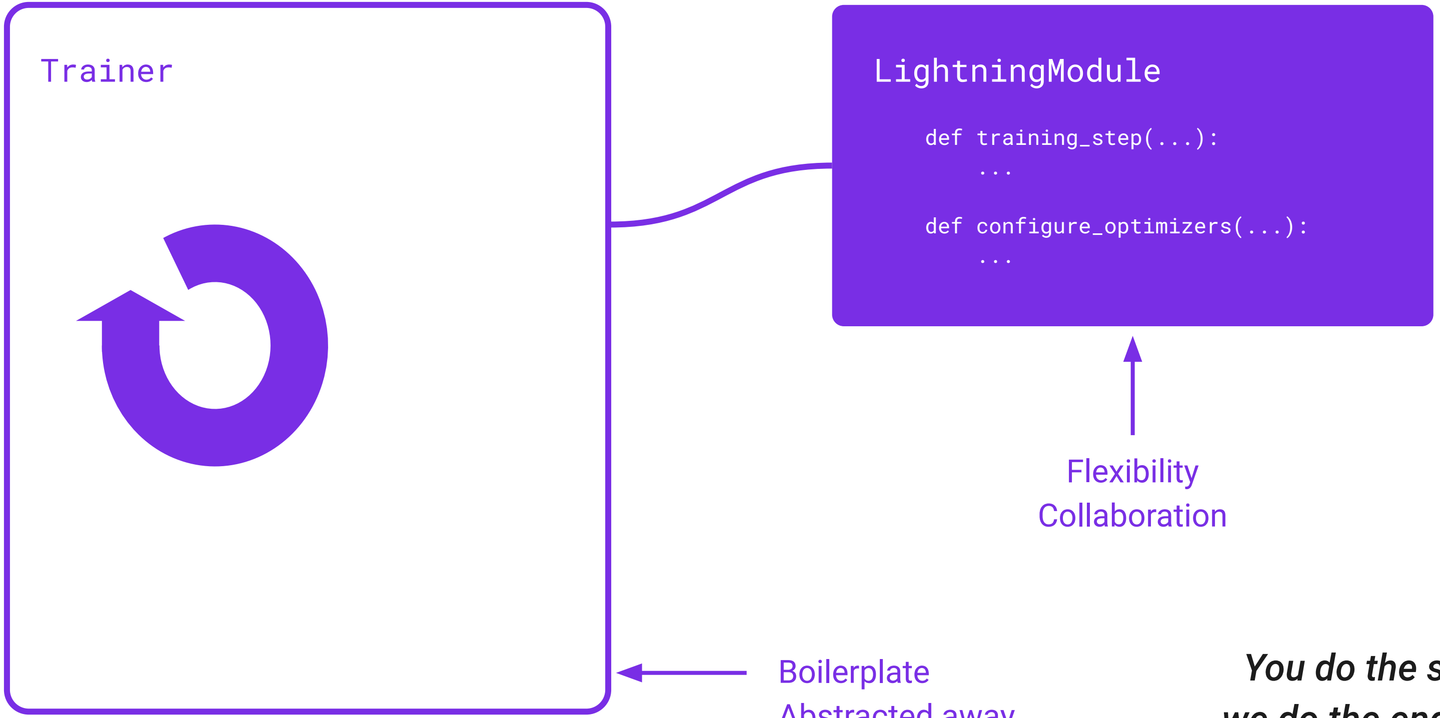
```
def fit(self):
    if global_rank == 0:
        # prepare data is called on GLOBAL ZERO only
        prepare_data()

    configure_callbacks()

    with parallel(devices):
        # devices can be GPUs, TPUs,
        train_on_device(model)

def train_on_device(model):
    # called PER DEVICE
    setup("fit")
    configure_optimizers()
    on_fit_start()

    # sanity check runs here
    on_train_start()

    for epoch in epochs:
        fit_loop()

    on_train_end()
    on_fit_end()
    teardown("fit")
```

```
def fit_loop():
    on_train_epoch_start()

    for batch in train_dataloader():
        on_train_batch_start()

        on_before_batch_transfer()
        transfer_batch_to_device()
        on_after_batch_transfer()

        training_step()

        on_before_zero_grad()
        optimizer_zero_grad()

        on_before_backward()
        backward()
        on_after_backward()

        on_before_optimizer_step()
        configure_gradient_clipping()
        optimizer_step()

        on_train_batch_end()

        if should_check_val:
            val_loop()

    # end training epoch
    training_epoch_end()

    on_train_epoch_end()
```

PyTorch Lightning



PyTorch Lightning



You do the science,
we do the engineering

pytorch-lightning.readthedocs.io

Get Started

- Lightning in 15 minutes
- Installation

Level Up

- Basic skills
- Intermediate skills
- Advanced skills
- Expert skills

Core API

- LightningModule
- Trainer
- Fabric (Beta)

API Reference

- accelerators
- callbacks
- cli
- core
- loggers
- profiler
- trainer
- strategies

v: stable

Docs > Train 1 trillion+ parameter models

Edit on GitHub Shortcuts

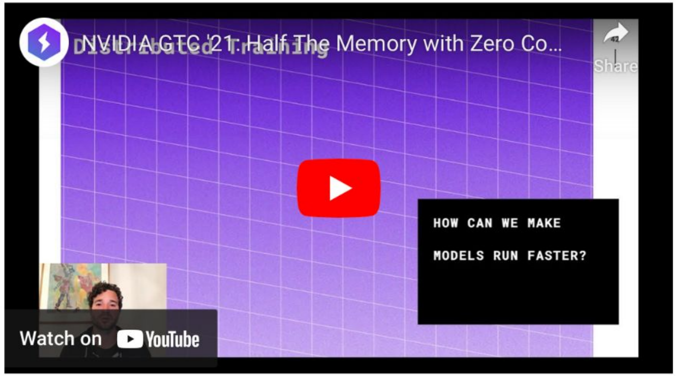
TRAIN 1 TRILLION+ PARAMETER MODELS

When training large models, fitting larger batch sizes, or trying to increase throughput using multi-GPU compute, Lightning provides advanced optimized distributed training strategies to support these cases and offer substantial improvements in memory usage.

Note that some of the extreme memory saving configurations will affect the speed of training. This Speed/Memory trade-off in most cases can be adjusted.

Some of these memory-efficient strategies rely on offloading onto other forms of memory, such as CPU RAM or NVMe. This means you can even see memory benefits on a **single GPU**, using a strategy such as DeepSpeed ZeRO Stage 3 Offload.

Check out this amazing video explaining model parallelism and how it works behind the scenes:



Choosing an Advanced Distributed GPU Strategy

Lightning AI

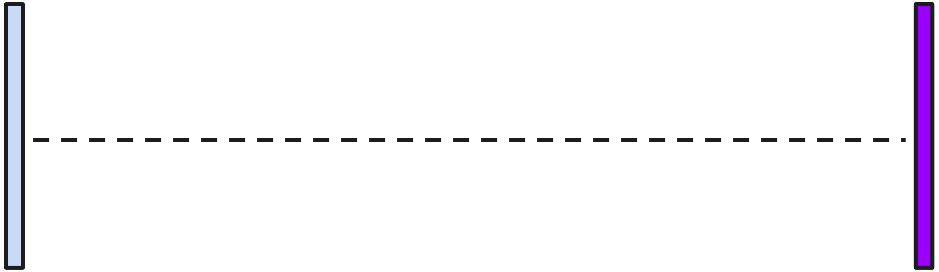
Training in the age of foundation models

Sometimes the science *is* the engineering

Large models, long projects

Often very little iteration on the models, but need of full control on the training process

A binary choice until today



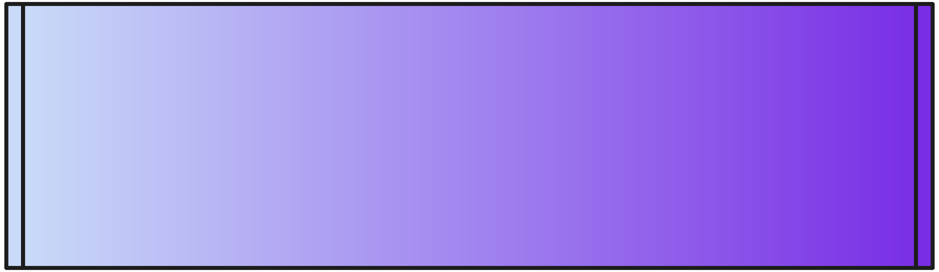
Raw PyTorch

Full control
More boilerplate

PyTorch Lightning

Fully managed
Less boilerplate

Introducing Fabric



Raw PyTorch | Fabric | PyTorch Lightning

Full control
More boilerplate

Fully managed
Less boilerplate

Fabric

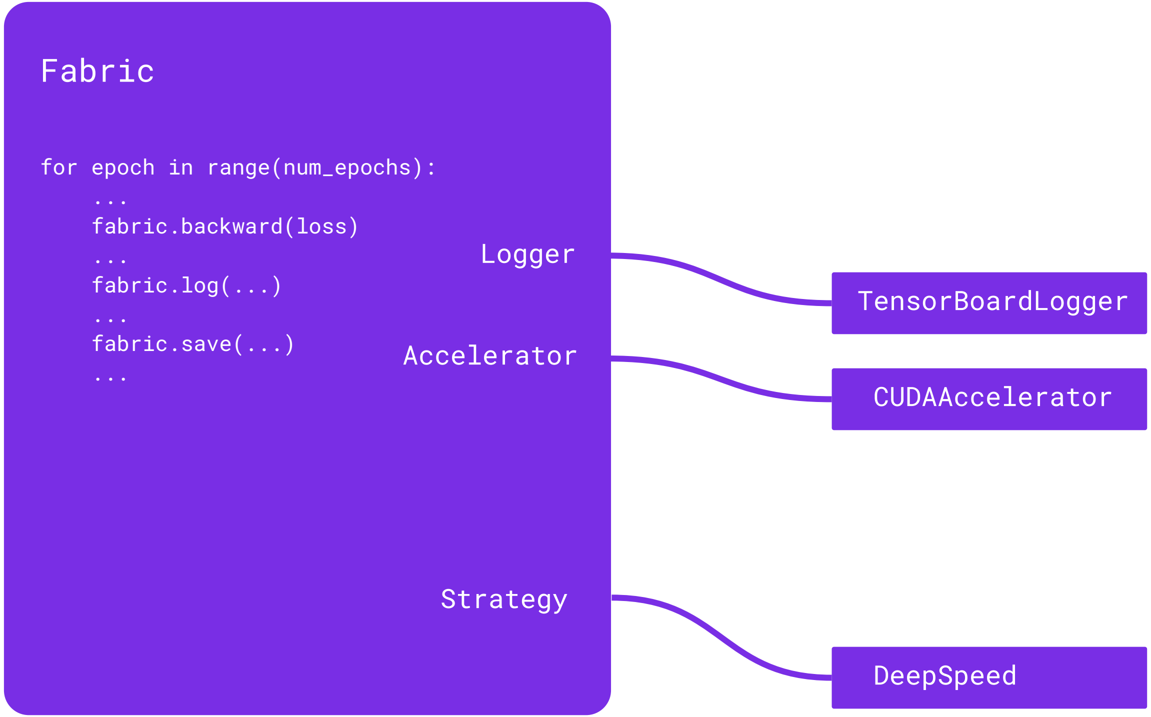
3 Your code

1 Fabric object

2 Setup model, optimizer, dataloader

```
import lightning as L

def train(fabric, model, optimizer, dataloader):
    # Training loop
    model.train()
    for epoch in range(num_epochs):
        for i, batch in enumerate(dataloader):
            ...

def main():
    # (Optional) Parse command line options
    args = parse_args()

    # Configure Fabric
    fabric = L.Fabric(...)

    # Instantiate objects
    model = ...
    optimizer = ...
    train dataloader = ...

    # Set up objects
    model, optimizer = fabric.setup(model, optimizer)
    train_dataloader = fabric.setup_dataloaders(train_dataloader)

    # Run training loop
    train(fabric, model, optimizer, train_dataloader)

if __name__ == "__main__":
    main()
```

Fabric


Fabric
for epoch in range(num_epochs):
...
fabric.backward(loss)
...
fabric.log(...)
...
fabric.save(...)
...
Logger
Accelerator
Strategy
TensorBoardLogger
CUDAAccelerator
DeepSpeed

Fabric

Accelerators

Strategies

Precision

```
fabric = Fabric( devices=4, accelerator="gpu", strategy="ddp", precision=16)
```

Fabric

Loggers

```
from lightning.fabric import Fabric
from lightning.fabric.loggers import CSVLogger,
TensorBoardLogger

tb_logger = TensorBoardLogger(root_dir="logs/tensorboard")
csv_logger = CSVLogger(root_dir="logs/csv")

# Add multiple loggers in a list
fabric = Fabric(loggers=[tb_logger, cs_logger])

# Calling .log() or .log_dict() always logs to all loggers
simultaneously
fabric.log("some_value", value)
```

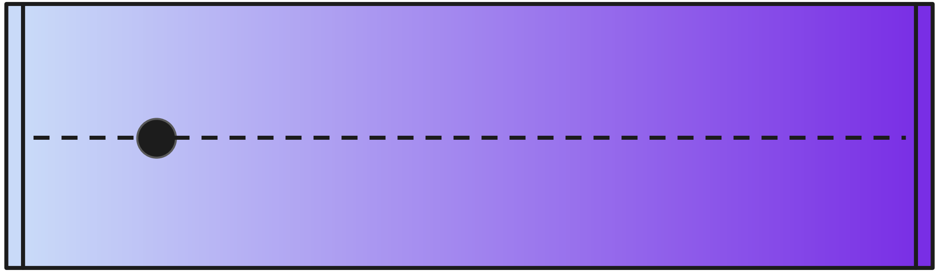
Fabric

Checkpoints

```
# Define the state of your program/loop
state = {"model1": model1, "model2": model2, "optimizer": optimizer,
"iteration": iteration, "hparams": ...}

# Save a checkpoint
fabric.save("path/to/checkpoint ckpt", state)

# Load a checkpoint
fabric.load("path/to/checkpoint.ckpt", state)

# Restore part of a checkpoint
state = {"model1": model1}
remainder = fabric.load("path/to/checkpoint.ckpt", state)
```

Fabric

“Raw Fabric”

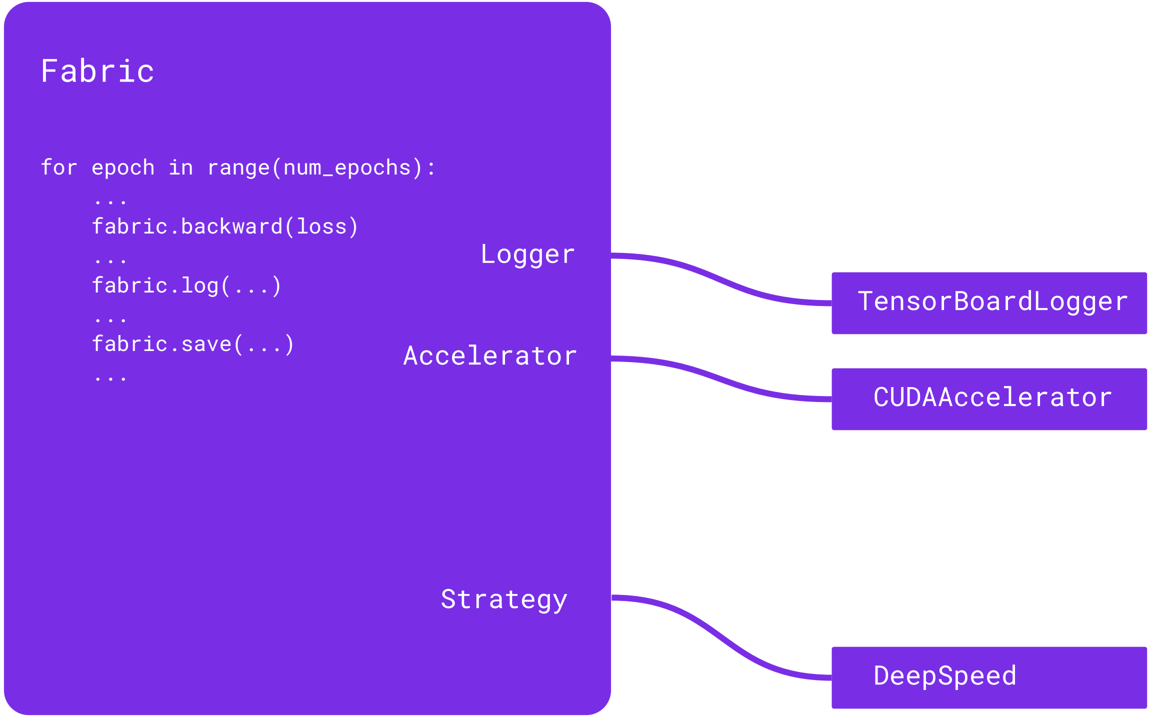


Raw PyTorch | Fabric | PyTorch Lightning

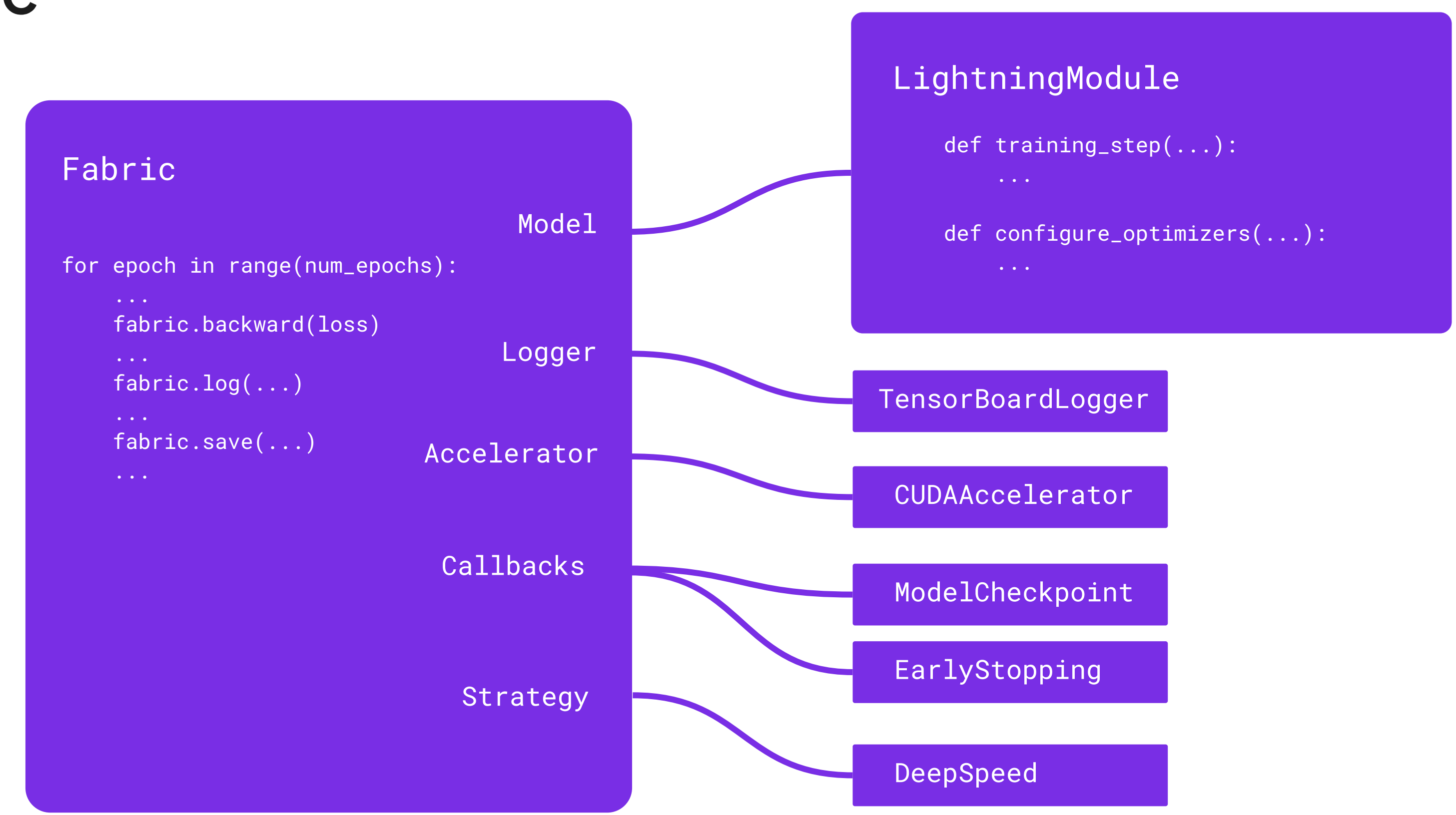
Full control
More boilerplate

Fully managed
Less boilerplate

Fabric


Fabric
for epoch in range(num_epochs):
...
fabric.backward(loss)
...
fabric.log(...)
...
fabric.save(...)
...
Logger
Accelerator
Strategy
TensorBoardLogger
CUDAAccelerator
DeepSpeed

Fabric


Fabric
Model
for epoch in range(num_epochs):
...
fabric.backward(loss)
...
fabric.log(...)
...
fabric.save(...)
...
Logger
Accelerator
Callbacks
Strategy
LightningModule
def training_step(...):
...
def configure_optimizers(...):
...
TensorBoardLogger
CUDAAccelerator
ModelCheckpoint
EarlyStopping
DeepSpeed

Fabric

💜 LightningModule

```python
import lightning as L

class LitModel(L.LightningModule):
    def __init__(self):
        super().__init__()
        self.model = ...

    def training_step(self, batch, batch_idx):
        # Main forward, loss computation, and metrics goes here
        x, y = batch
        y_hat = self.model(x)
        loss = self.loss_fn(y, y_hat)
        acc = self.accuracy(y, y_hat)
        ...
        return loss

    def configure_optimizers(self):
        # Return one or several optimizers
        return torch.optim.Adam(self.parameters() , ...)

    def train_dataloader(self):
        # Return your dataloader for training
        return DataLoader(...)

    def on_train_start(self):
        # Do something at the beginning of training
        ...

    def any_hook_you_like(self, *args, **kwargs):
        ...
```

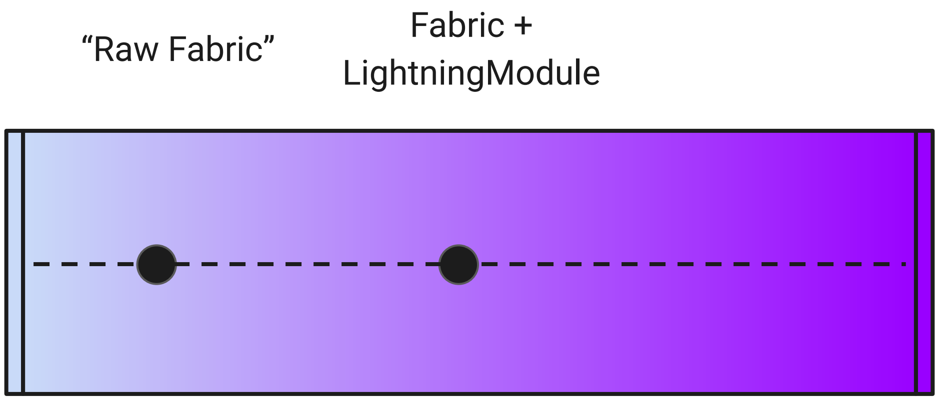
Fabric

💜 LightningModule

Training code is model-independent

```python
import lightning as L

fabric = L.Fabric(...)

# Instantiate the LightningModule
model = LitModel()

# Get the optimizer(s) from the LightningModule
optimizer = model.configure optimizers()

# Get the training data loader from the LightningModule
train_dataloader = model.train_dataloader()

# Set up objects
model, optimizer = fabric.setup(model, optimizer)
train_dataloader = fabric.setup_dataloaders(train_dataloader)

# Call the hooks at the right time
model.on_train_start()

model.train()
for epoch in range(num_epochs) :
    for i, batch in enumerate(dataloader):
        optimizer.zero_grad()
        loss = model.training_step(batch, i)
        fabric.backward(loss)
        optimizer.step()

        # Control when hooks are called
        if condition:
            model.any_hook_you_like()
```

Fabric



Raw PyTorch   Fabric   PyTorch Lightning

Full control
More boilerplate

Fully managed
Less boilerplate

Fabric

💜 Callbacks

```python
from lightning. fabric import Fabric

# The code of a callback can live anywhere, away from the training loop
from my_callbacks import MyCallback

# Add one or several callbacks:
fabric = Fabric(callbacks=[MyCallback()])

for iteration, batch in enumerate(train_dataloader):
    ...
    fabric.backward(loss)
    optimizer.step()

    # Let a callback add some arbitrary processing at the appropriate
place
    # Give the callback access to some varibles
    fabric.call("on_train_batch_end", loss=loss, output=...)
```


Lightning AI

Fabric

💜 Callbacks

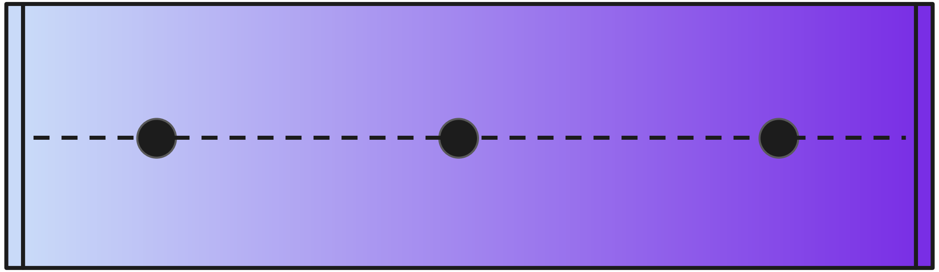
```python
# Add multiple callback implementations in a list
callback1 = LearningRateMonitor()
callback2 = Profiler()

fabric = Fabric(callbacks=[callback1, callback2])

# Let Fabric call the implementations (if they exist)
fabric.call("any_callback_method", arg1=..., arg2=...)

# fabric.call is the same as doing this
callback1.any_callback_method(arg1=..., arg2=...)
callback2.any_callback_method(arg1=..., arg2=...)
```

Fabric

“Raw Fabric”

Fabric +
LightningModule

Callbacks
Build-your-own-trainer



Raw PyTorch

Fabric

PyTorch Lightning

Full control
More boilerplate

Fully managed
Less boilerplate

Fabric


Raw PyTorch
Fabric (5-line change) Accelerators
Fabric + Distributed strategies
Fabric + Managed logging and checkpointing
Fabric + LightningModule
Fabric + Build your own trainer
PyTorch Lightning Trainer
Raw PyTorch
Full control, more boilerplate
Lightning Fabric
PyTorch Lightning
Fully managed, no boilerplate

Communication

boilerplate

```python
    # Build the data-parallel groups.
    global _DATA_PARALLEL_GROUP
    global _DATA_PARALLEL_GLOBAL_RANKS
    assert _DATA_PARALLEL_GROUP is None, 'data parallel group is already initialized'
    all_data_parallel_group_ranks = []
    for i in range(pipeline_model_parallel_size):
        start_rank = i * num_pipeline_model_parallel_groups
        end_rank = (i + 1) * num_pipeline_model_parallel_groups
        for j in range(tensor_model_parallel_size):
            ranks = range(start_rank + j, end_rank, tensor_model_parallel_size)
            all_data_parallel_group_ranks.append(list(ranks))
            group = torch.distributed.new_group(ranks)
            if rank in ranks:
                _DATA_PARALLEL_GROUP = group
                _DATA_PARALLEL_GLOBAL_RANKS = ranks

    # Build the model-parallel groups.
    global _MODEL_PARALLEL_GROUP
    assert _MODEL_PARALLEL_GROUP is None, 'model parallel group is already initialized'
    for i in range(data_parallel_size):
        ranks = [data_parallel_group_ranks[i]
                 for data_parallel_group_ranks in all_data_parallel_group_ranks]
        group = torch.distributed.new_group(ranks)
        if rank in ranks:
            _MODEL_PARALLEL_GROUP = group

    # Build the tensor model-parallel groups.
    global _TENSOR_MODEL_PARALLEL_GROUP
    assert _TENSOR_MODEL_PARALLEL_GROUP is None, \
        'tensor model parallel group is already initialized'
    for i in range(num_tensor_model_parallel_groups):
        ranks = range(i * tensor_model_parallel_size,
                      (i + 1) * tensor_model_parallel_size)
        group = torch.distributed.new_group(ranks)
        if rank in ranks:
            _TENSOR_MODEL_PARALLEL_GROUP = group

    # Build the pipeline model-parallel groups and embedding groups
    # (first and last rank in each pipeline model-parallel group).
    global _PIPELINE_MODEL_PARALLEL_GROUP
```

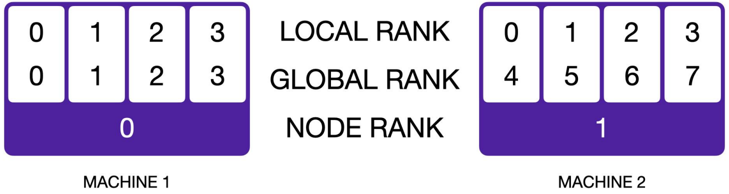
from Megatron-LM

forked in BLOOM, GPT-NeoX

https://github.com/NVIDIA/Megatron-LM

Fabric communication API

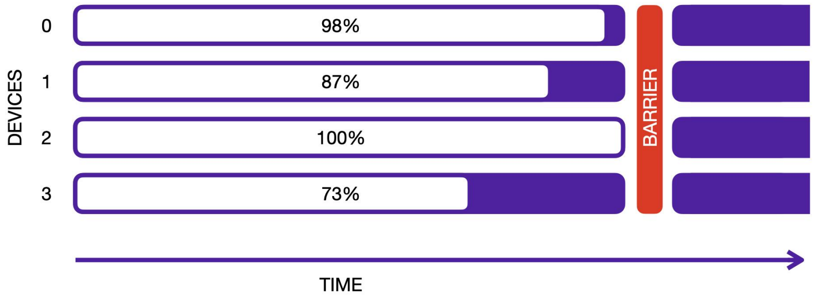
```python
from lightning.fabric import Fabric

# Devices and num nodes determine how many processes there are
fabric = Fabric(devices=2, num_nodes=3)
fabric.launch()
```




```python
# The total number of processes running across all devices and nodes
fabric.world_ size  # 2 * 3 = 6

# The global index of the current process across all devices and nodes
fabric.global_rank  # -> {0, 1, 2, 3, 4, 5}

# The index of the current process among the processes running on the
local node
fabric.local_rank   # -> {0, 1}

# The index of the current node
fabric.node_rank    # -> {0, 1, 2}

# Do something only on rank 0
if fabric.global_rank == 0:
    ...
```

Fabric communication API

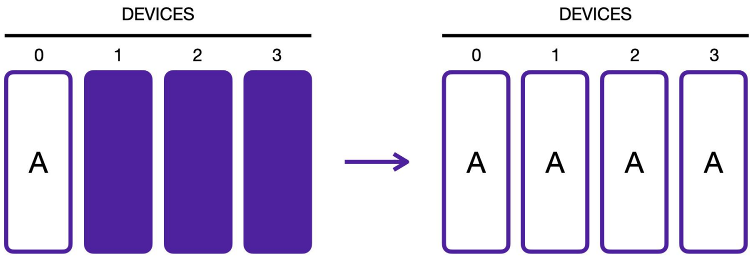
Barrier




```python
if fabric.global_rank == 0:
    print("Downloading dataset. This can take a while ...")
    download_dataset()

# All other processes wait here until rank 0 is done with downloading:
fabric.barrier()

# After everyone reached the barrier, they can access the downloaded files:
load_dataset()
```

Fabric communication API

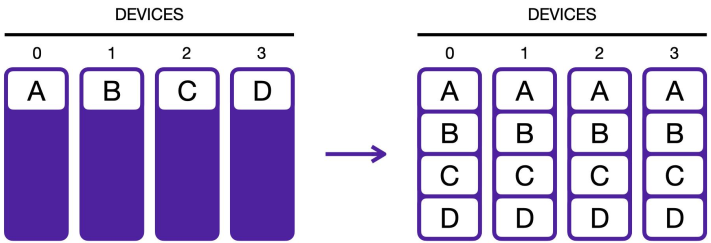
Broadcast




```python
fabric = Fabric(devices=4, accelerator="gpu")
fabric.launch()

# Data is different on each process
learning_rate = torch.rand(1)
print("Before broadcast:", learning_rate)

# Transfer the tensor from one process to all the others
learning_rate = fabric.broadcast(learning_rate)
print("After broadcast:", learning rate)
```

Fabric communication API

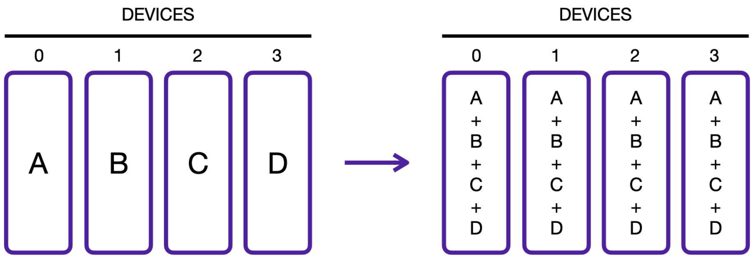
Gather




```
fabric = Fabric(devices=4, accelerator="gpu")
fabric.launch()

# Data is different in each process
result = torch.tensor(10 * fabric.global_rank)

# Every process gathers the tensors from all other processes
# and stacks the result:
result = fabric.all_gather(data)
print("Result of all-gather:", result)  # tensor([0, 10, 20, 30])
```

Fabric communication API

Reduce

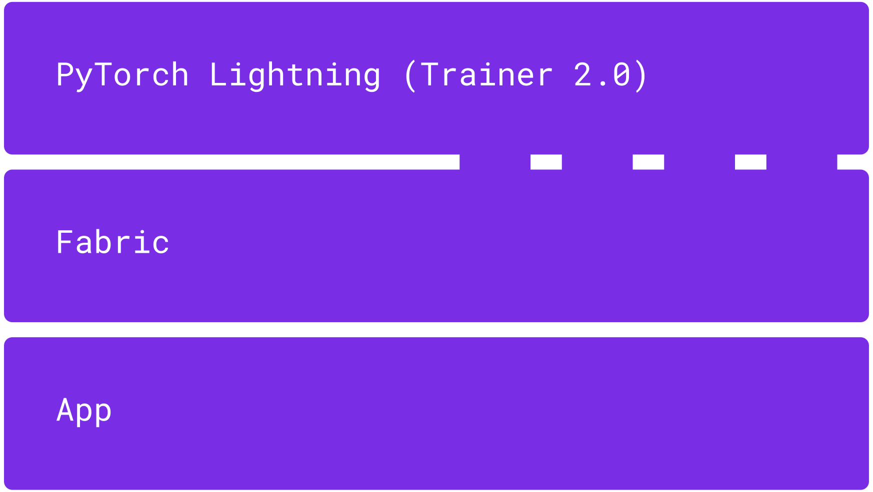



```python
fabric = Fabric(devices=4, accelerator="gpu")
fabric.launch()

# Data is different in each process
data = torch.tensor(10 * fabric.global_rank)

# Sum the tensors from every process
result = fabric.all_reduce(data, reduce_op="sum")

# sum (0 + 10 + 20 + 30) = tensor(60)
print("Result of all-reduce:", result)
```

Lightning 2.0

Coming mid-March 2023

Lightning 2.0



* PyTorch Lightning 1.9.x will be supported long-term

Lightning 2.0

Trainer 2.0: tighter, fewer abstractions, fewer dependencies

Super-stable API for Trainer and LightningModule

More opt-in, less opt-out

Full control with Fabric

Full PyTorch 2.0 support

PyTorch Lightning (Trainer 2.0)

Fabric

App

Demo: From PyTorch to Fabric to Trainer

github.com/Lightning-AI/open-bio-ml-workshop

Data Loading

"If Lightning can iterate over it, it can load the data."

```
for data in dataloader:
    ...
```

Data Loading

- PyTorch DataLoader
- TorchData (DataLoader2)
- WebDataset
- NVIDIA DALI
- ...

Data Loading

Trainer accepts training-, validation-, and test data iterables

Performs sanity checks for potential user error: Shuffling, num workers, uneven lengths (DDP), …

```
trainer = Trainer(...)

train_data = DataLoader (..., batch_size=16, num_workers=4)
val_data = DataLoader(..., batch_size=8, num_workers=2)
test_data = DataLoader(..., batch_size=8, num_workers=2)

trainer.fit(model, train_data, val_data)
trainer.test(model, test_data)
```

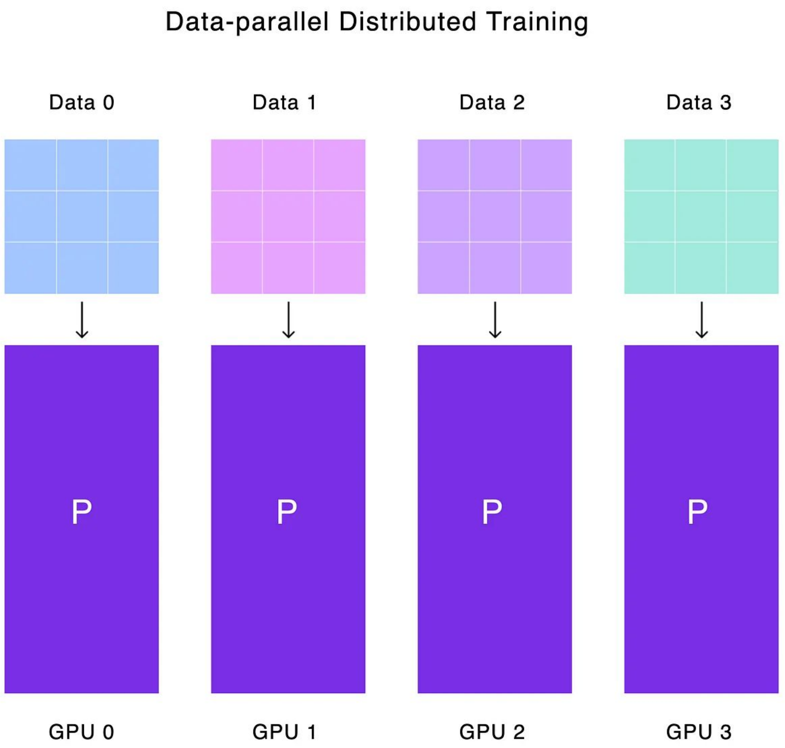
Data Loading

LightningModule organizes data-loading code (optional)

Hooks get called by Trainer at the right time

```
import lightning as L

class LitModel(L.LightningModule):
    def __init__(self):
        super().__init__()
        ...

    def training_step(self, batch, batch_idx):
        ...

    def train_dataloader(self):
        return DataLoader( ...)

    def val_dataloader(self):
        return DataLoader(...)

    def test_dataloader(self):
        return DataLoader(...)
```

Distributed Data Loading




```python
from torch.utils. data import DistributedSampler, DataLoader

if args.distributed:
    train_sampler = DistributedSampler(train_dataset)
    val_sampler = DistributedSampler(val_dataset, shuffle=False,
drop_last=True)
else:
    train_sampler = None
    val_sampler = None

train_loader = DataLoader(
    train_dataset, batch_size=args.batch_size, shuffle=(train_sampler is
None),
    num_workers=args.workers, pin_memory=True, sampler=train_sampler
)

val_loader = DataLoader(
    val_dataset, batch_size=args.batch_size, shuffle=False,
    num_workers=args.workers, pin_memory=True, sampler=val_sampler
)
```

Warning: Boilerplate code!

```python
for epoch in range(args.start_epoch, args.epochs):
    if args.distributed:
        train_sampler.set_epoch(epoch)
```

Distributed Data Loading

Fabric

```python
fabric = Fabric(...)

train data = DataLoader(..., batch_size=16, num_workers=4)
val_data = DataLoader(..., batch_size=8, num_workers=2)
test_data = DataLoader(..., batch_size=8, num_workers=2)

train_data, val_data = fabric.setup_dataloaders(train_data, val_data)

for batch in train_data:
    ....

test_data = fabric.setup_dataloaders(test_data)
...
```

Trainer

```python
trainer = Trainer(...)

train_data = DataLoader(..., batch_size=16, num_workers=4)
val_data = DataLoader(..., batch_size=8, num_workers=2)
test_data = DataLoader(..., batch_size=8, num_workers=2)

trainer.fit(model, train_data, val_data)
trainer.test(model, test_data)
```

Multi-node: SLURM

Step 1: Set devices and number of nodes

```
# Fabric
fabric = Fabric(accelerator="gpu", devices=8, num_nodes=4)
fabric.launch()
...

# OR

# Trainer
trainer = Trainer(accelerator="gpu", devices=8, num_nodes=4)
trainer.fit(...)
```

Multi-node: SLURM

Step 1: Set devices and number of nodes

Step 2: Create SLURM job submission script

[Template in our docs!]

submit.sh

```
# submit.sh

#SBATCH --nodes=4
#SBATCH --ntasks-per-node=8
#SBATCH --gres=gpu:8          # Request N GPUs per machine
#SBATCH --time=0-02:00:00

# Optional: Activate environment
source venv/bin/activate

# Run your training script
srun python train.py
```

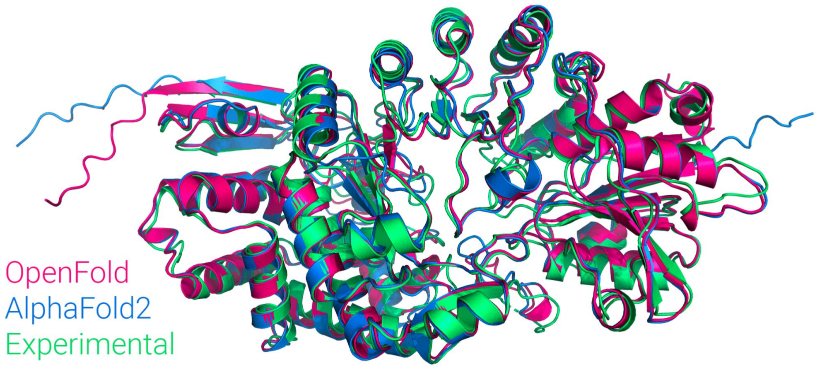
Multi-node: SLURM

Step 1: Set devices and number of nodes

Step 2: Create SLURM job submission script

Step 3: Submit the job

```
> sbatch submit.sh
```



Walkthrough: OpenFold – AlphaFold 2 with Lightning

Reach out!

discord	discord.gg/MWAEvnC5fU
forums	lightning.ai/forums
twitter	@LightningAI

https://linktr.ee/lightningai

Thanks

Adrian Wälchli
Akihiro Nitta
Carlos Mocholí
Eden Afek
Ethan Harris
Jirka Borovec
Justus Schock
Thomas Chaton
William Falcon

\+

💜 community

https://linktr.ee/lightningai

Lightning LEAGUE

Thanks