Open-source Infrastructure for model training

Philipp Krähenbühl, UT Austin

Checkpointing

`torch.utils.checkpoint.checkpoint`

- Wrap module.forward in checkpoint function
- Requires access to model
- Use `preserve_rng_state=True`
  - if module has randomness (i.e. Dropout)

- Documentation: https://pytorch.org/docs/stable/generated/torch.nn.DataParallel.html

DataParallel

```
model = torch.nn.DataParallel(model, device_ids=[0, 1, 2])
```

- Easy to use
- Just wrap your model
- Multi-thread training
    - torch will split batch across devices for model forwarding
    - backwards on devices are summed into the original module
- Be wary:
    - Access your model now through model.module
    - This can break existing checkpoint loading and model attribute access

- Documentation: https://pytorch.org/docs/stable/generated/torch.nn.DataParallel.html

DistributedDataParallel

```
import torch.distributed as dist

dist.init_process_group(backend='nccl')

# Get Rank - method 1
local_rank = int(os.environ[“LOCAL_RANK"])

# Get Rank - method 2
local_rank = dist.get_rank()

model = torch.nn.parallel.DistributedDataParallel(model,
device_ids=[local_rank])
```

- bit more involved
  - initialize distributed setup
  - wrap model
  - call torchrun

```
torchrun --nproc_per_node=4 train.py
```

- Multi-process training
  - synchronizes gradients across processes on backward

- Documentation: https://pytorch.org/tutorials/intermediate/ddp_tutorial.html

DistributedDataParallel

```
torchrun --nnodes=4 --nproc-per-node=4
    --rdzv-id=$JOB_ID
    --rdzv-backend=c10d
    --rdzv-endpoint=$MASTER_ADDR:29400
     train.py
```

- supports multiple nodes training
    - fine for 2 nodes
    - better methods exist for more nodes (imo)
- Be wary:
    - perform saving and logging on main process only

- Documentation: https://pytorch.org/docs/stable/elastic/run.html

Fully Sharded Data Parallel (FSDP)

- Used for large model training
  - DP/DDP stores exact copy of model parameters, gradients, and optimizer states.
  - We don't want that for large model
  - FSDP splits ("shards") a model into N pieces (N = # data-parallel workers. Usually # devices.)
  - Gather model weights across workers a layer at a time

Fully Sharded Data Parallel (FSDP)

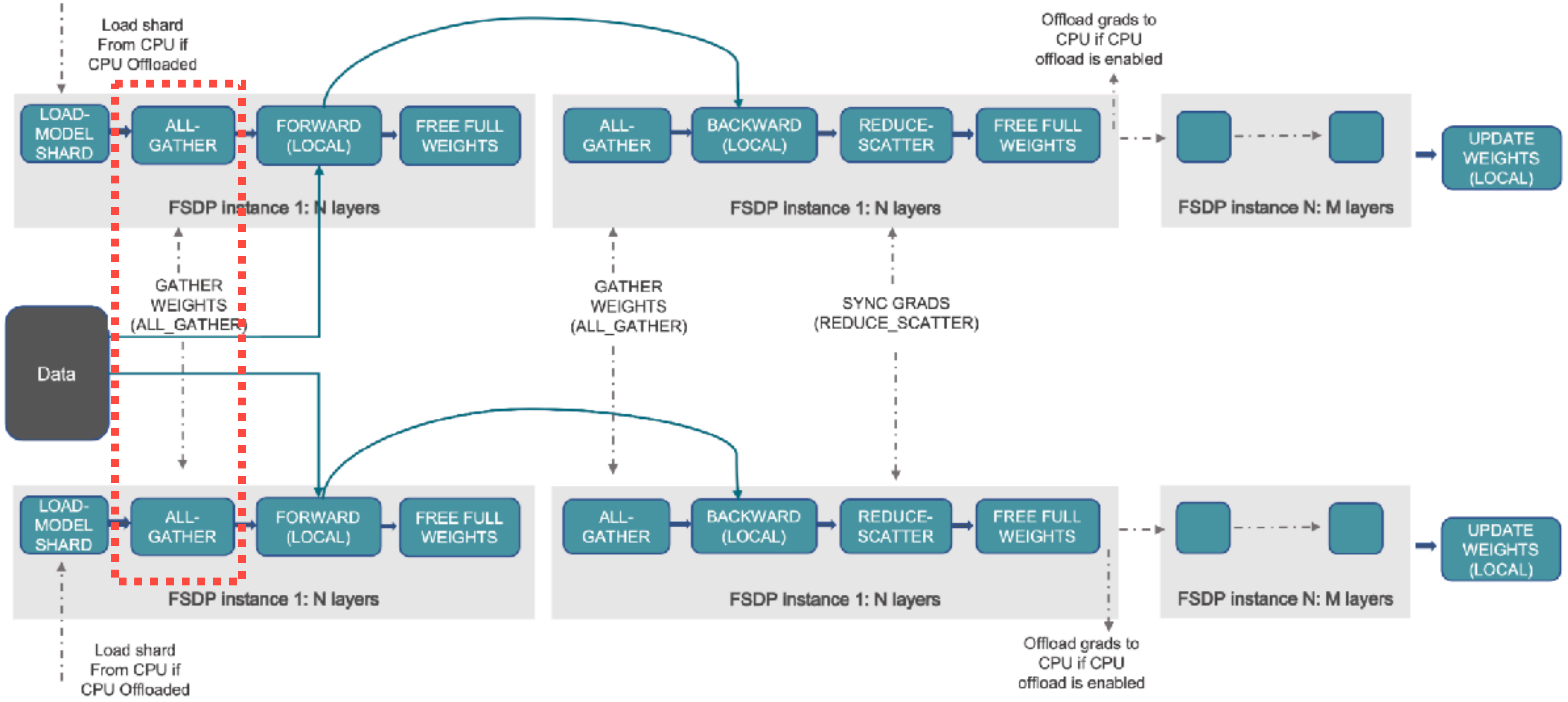

Figure 1. FSDP workflow

1. At every layer at a time, each workers gathers parameters to construct a full layer.

Image: https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/

Fully Sharded Data Parallel (FSDP)

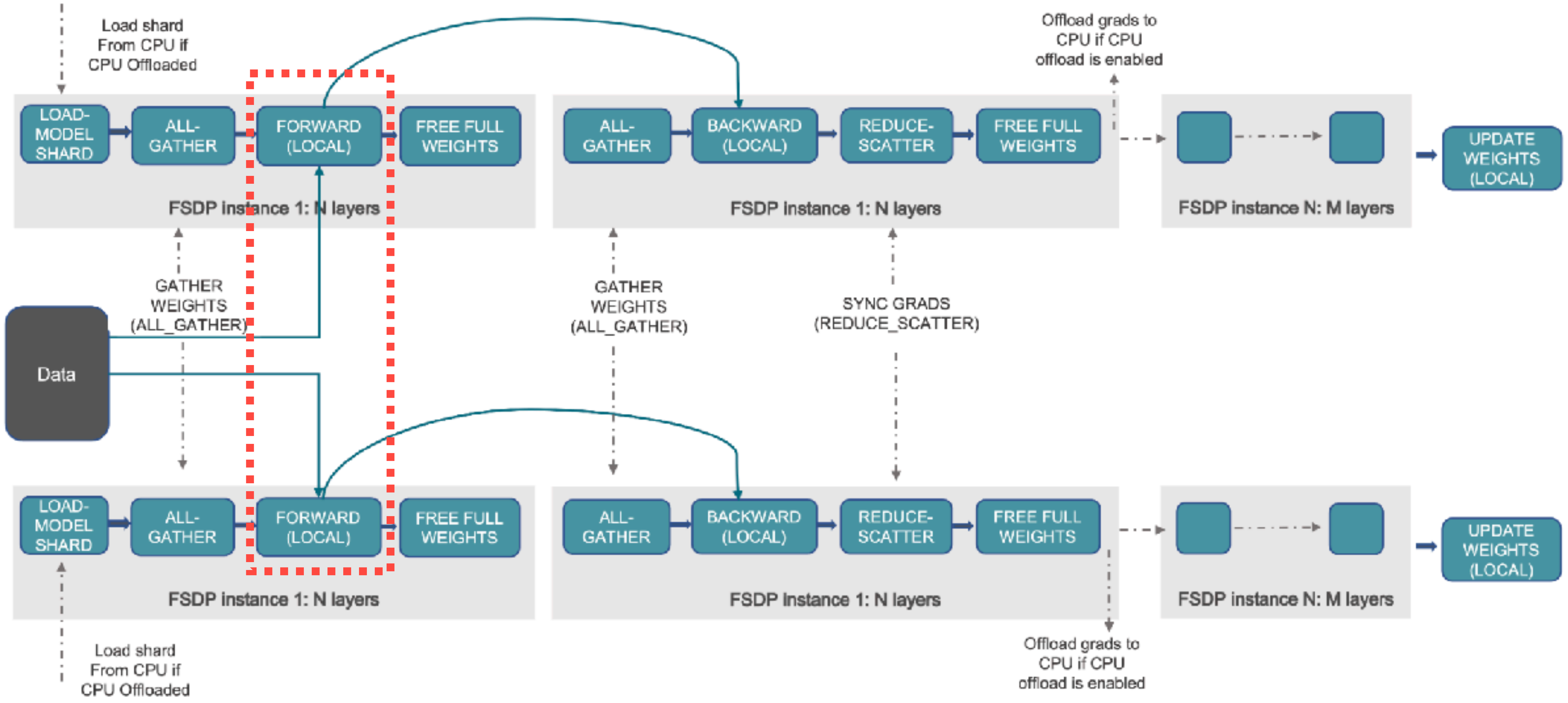

Figure 1. FSDP workflow

2. Forward each data (per-workers) to get full intermediate output.

Image: https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/

Fully Sharded Data Parallel (FSDP)

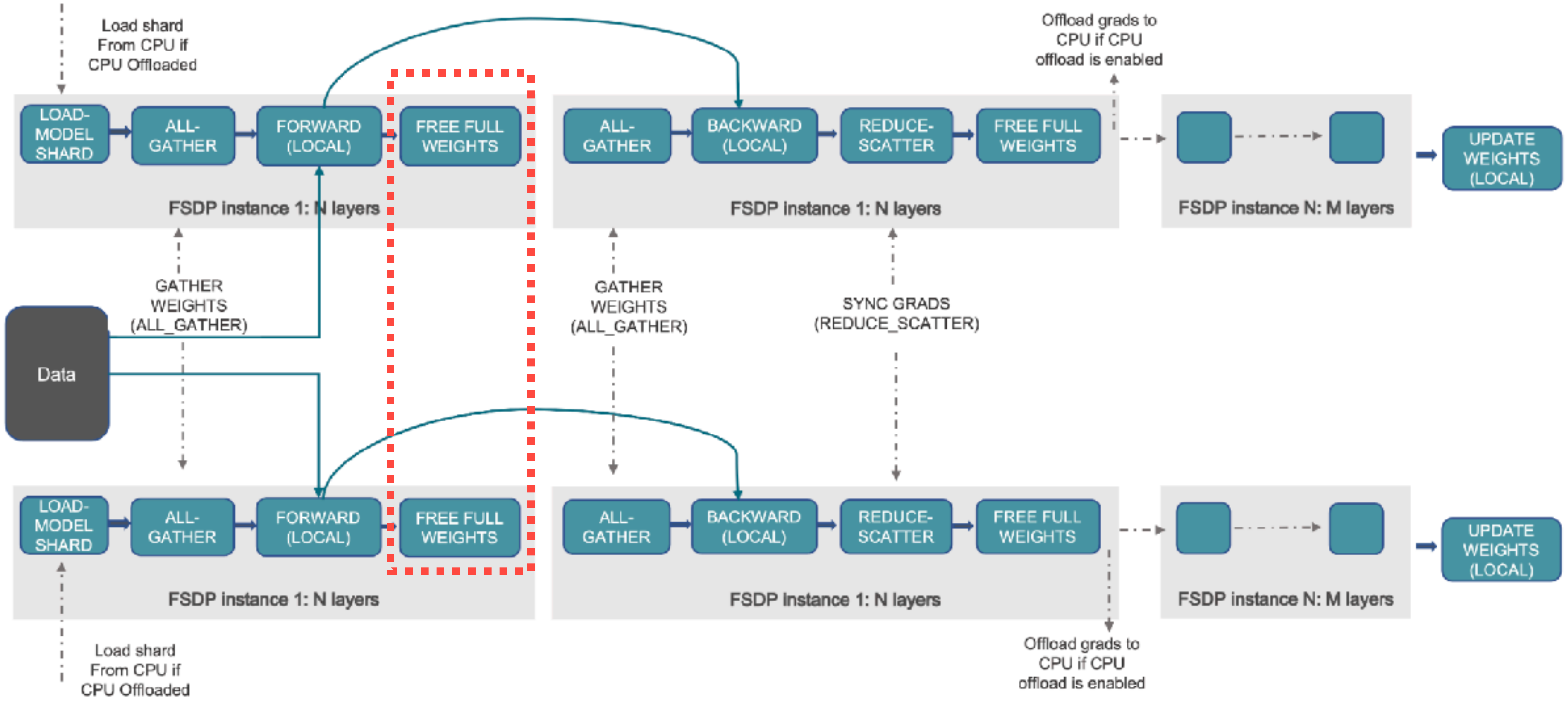

Figure 1. FSDP workflow

3. Discard parameters from other workers.

Image: https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/

Fully Sharded Data Parallel (FSDP)

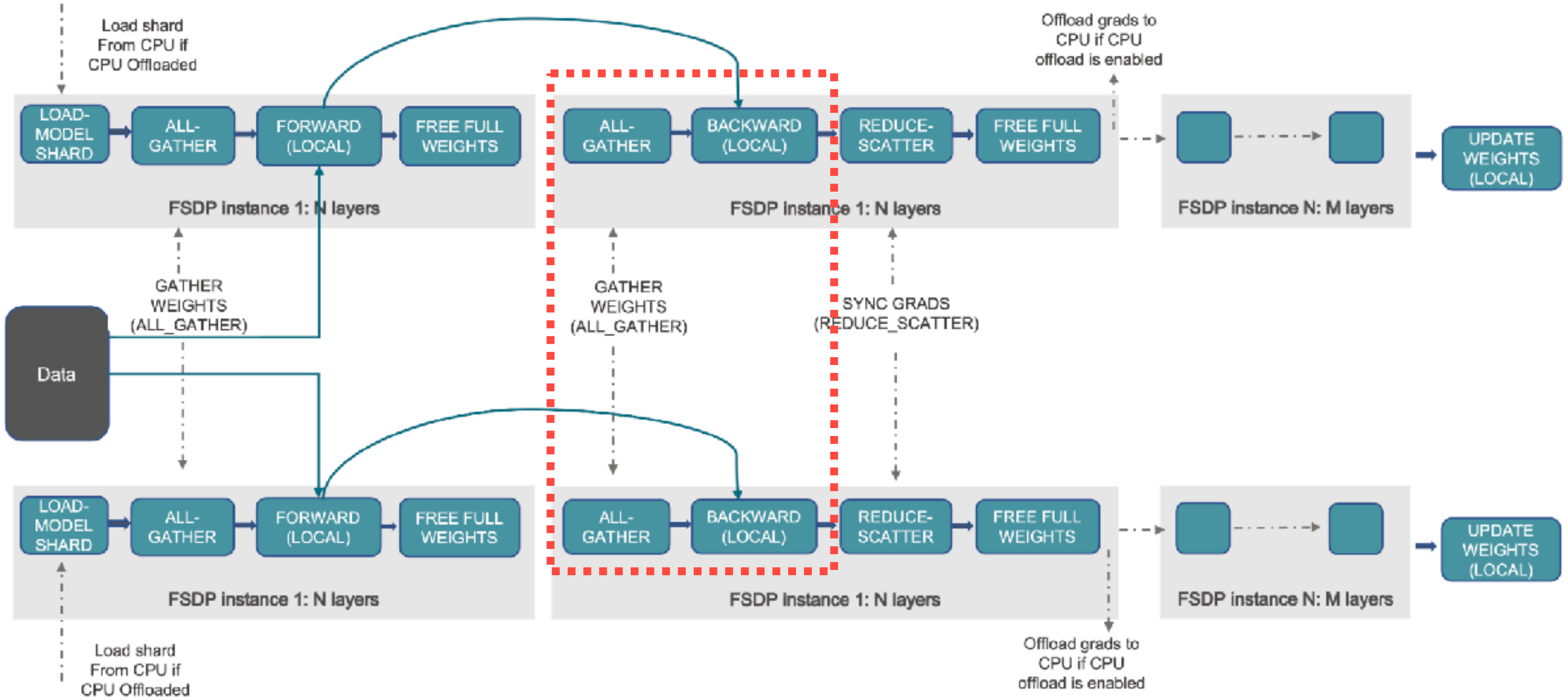

Figure 1. FSDP workflow

4. Similar to (1)-(2), gather and backward.

Image: https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/

Fully Sharded Data Parallel (FSDP)

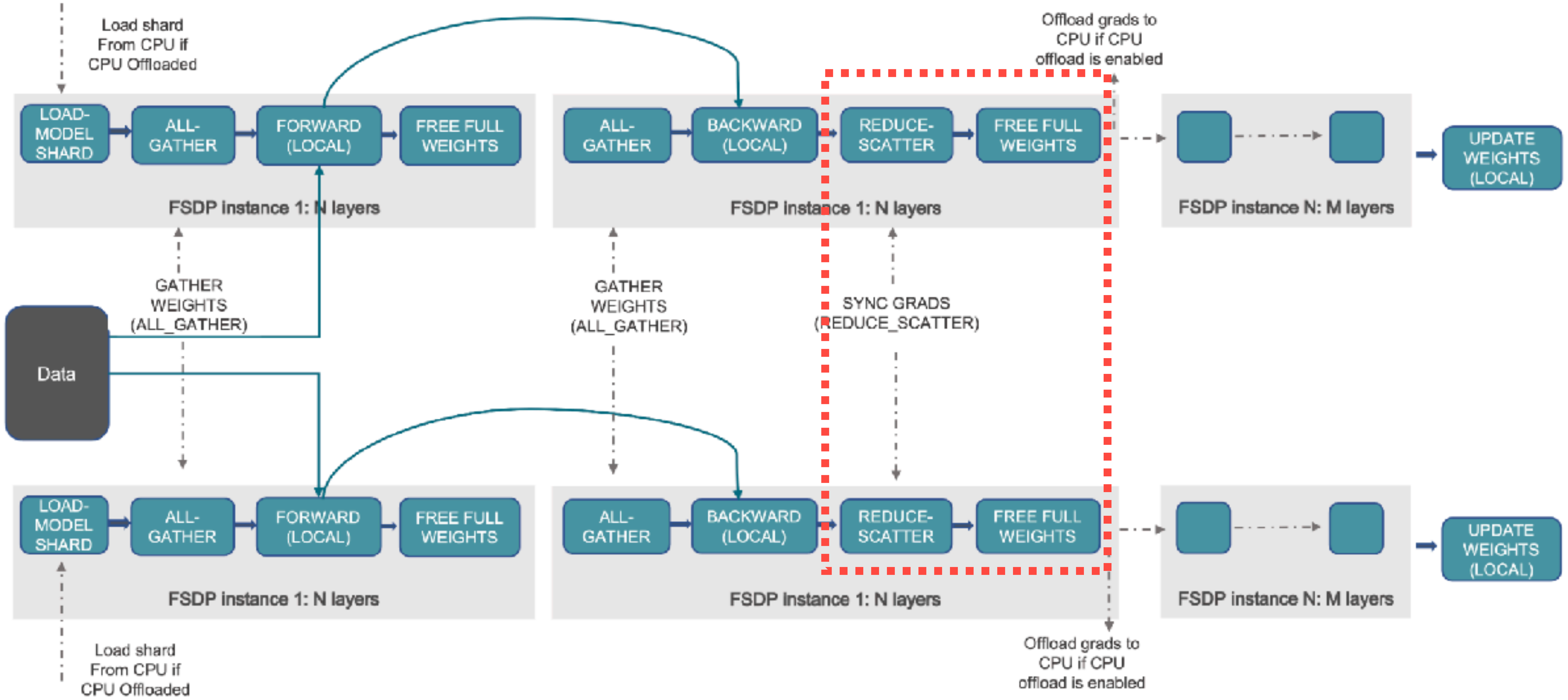

Figure 1. FSDP workflow

5. Scatter each "local" gradient across workers, and discard parameters.

Image: https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/

Fully Sharded Data Parallel (FSDP)

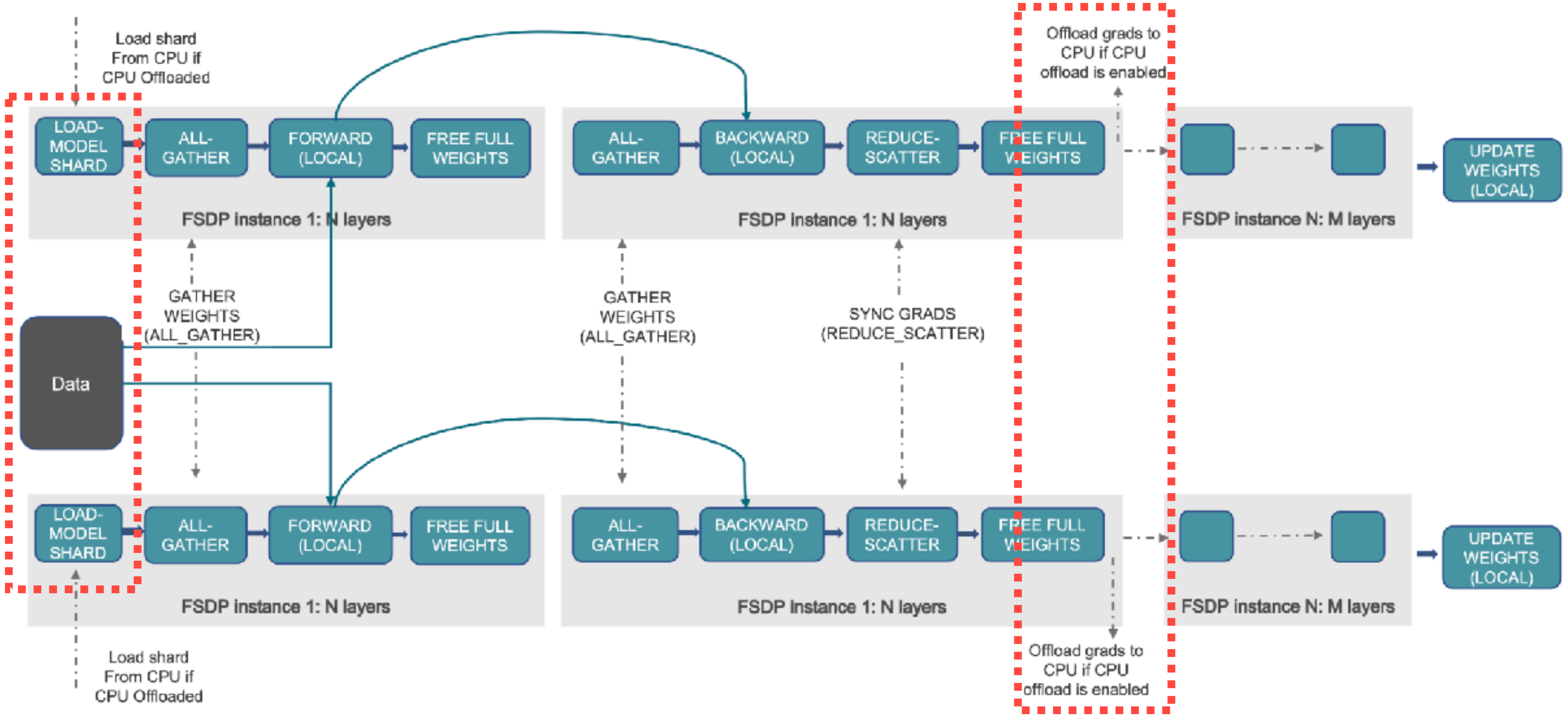

Figure 1. FSDP workflow

6. (Optional) Load/ offload sharded model parameters from CPU. (Reduce mem., increase time)

Image: https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/

Fully Sharded Data Parallel (FSDP)

- Simply wrap your model with *FullyShardedDataParallel*
- Doc: https://pytorch.org/docs/stable/fsdp.html
- Be wary:
  - Support is experimental (frequent interface changes)
  - Inter-device communication can become a bottleneck

```python
from torch.distributed.fsdp import (
   FullyShardedDataParallel,
   CPUOffload,
)
from torch.distributed.fsdp.wrap import (
   default_auto_wrap_policy,
)
import torch.nn as nn

class model(nn.Module):
   def __init__(self):
       super().__init__()
       self.layer1 = nn.Linear(8, 4)
       self.layer2 = nn.Linear(4, 16)
       self.layer3 = nn.Linear(16, 4)

model = DistributedDataParallel(model())
fsdp_model = FullyShardedDataParallel(
   model(),
   fsdp_auto_wrap_policy=default_auto_wrap_policy,
   cpu_offload=CPUOffload(offload_params=True),
)
```

DeepSpeed

- Used for large model training
  - DP/DDP stores exact copy of model parameters, gradients, and optimizer states.
  - Zero redundancy: Optimizer State Partitioning
  - DeepSpeed provides a lightweight wrapper of ZeRO plus mixed precision, gradient accumulation, and checkpoints…

DeepSpeed

Initializing DeepSpeed Models

- Training is accomplished using the DeepSpeed engine

- `deepspeed.initialize` ensures that DDP and/or mixed precision training setups are done appropriately under the hood

```
model_engine, optimizer, _, _ = deepspeed.initialize(
    args=cmd_args,
    model=model,
    model_parameters=model.parameters())
```

DeepSpeed

Training DeepSpeed Models

- Under the hood, DeepSpeed automatically handles:
  - Gradient Averaging
  - Loss Scaling (FP16+mixed precision training)
  - Learning Rate Scheduler

```
model_engine, optimizer, _, _ = deepspeed.initialize(
    args=cmd_args,
    model=model,
    model_parameters=model.parameters())

for step, batch in enumerate(data_loader):
    # forward() method
    loss = model_engine(batch)

    # runs backpropagation
    model_engine.backward(loss)

    # weight update
    model_engine.step()
```

DeepSpeed

Saving/loading DeepSpeed Models

- Saving and loading the training state is handled via the `save_checkpoint` and `load_checkpoint` API.

```
# load checkpoint
_, client_sd = model_engine.load_checkpoint(
    args.load_dir, args.ckpt_id)
step = client_sd['step']

# advance data loader to ckpt step
...

for step, batch in enumerate(data_loader):

    # forward, backward, and weight update,
    # shown in the previous slide

    # save checkpoint
    if step % args.save_interval:
        client_sd['step'] = step
        ckpt_id = loss.item()
        model_engine.save_checkpoint(
            args.save_dir,
            ckpt_id,
            client_sd = client_sd)
```

DeepSpeed

DeepSpeed Configuration

- DeepSpeed features are configured using a config JSON file that should be specified as `args.deepspeed_config`

```
{
    "train_batch_size": 8,
    "gradient_accumulation_steps": 1,
    "optimizer": {
        "type": "Adam",
        "params": {"lr": 0.00015}
    },
    "fp16": {
        "enabled": true
        ...
    },
    "amp": {
        "enabled": true,
        "opt_level": "O1",
        ...
    },
    "zero_optimization": {
        "stage": [0|1|2|3],
        "offload_param": {...},
        ...
    }
}
```

DeepSpeed

Launching DeepSpeed Training Jobs

- DeepSpeed installs the entry point `deepspeed` to launch distributed training
- `<client_entry.py>` is the entry script for your model
- `<client args>` is the argparse command line arguments
- `ds_config.json` is the configuration file for DeepSpeed

```
deepspeed --num_nodes=2 \
    <client_entry.py> \
    <client args> \
    --deepspeed \
    --deepspeed_config ds_config.json
```

PyTorch Lighting

- supports multiple nodes training
  - fine for 2 nodes
  - better methods exist for more nodes (imo)
- Be wary:
  - perform saving and logging on main process only

- Documentation: https://lightning.ai/docs/pytorch/stable/

PyTorch Lighting

- Automate a set of training infra:
  - distributed training (ddp, dp, fsdp…)
  - training and validation loop
  - checkpoint and logging
  - mixed precision
- Organize code:
  - clean separation of model, data and infra (loops, logging, etc.)
  - more friendly for new users (no need to build infra from scratch)
- Documentation: https://lightning.ai/docs/pytorch/stable/

PyTorch Lighting

- Step 1: Define a LightningModule

  - Set up model, training step, loss function

```python
# define any number of nn.Modules (or use your current ones)
encoder = nn.Sequential(nn.Linear(28 * 28, 64), nn.ReLU(), nn.Linear(64, 3))
decoder = nn.Sequential(nn.Linear(3, 64), nn.ReLU(), nn.Linear(64, 28 * 28))


# 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


# init the autoencoder
autoencoder = LitAutoEncoder(encoder, decoder)
```

- Documentation: https://lightning.ai/docs/pytorch/stable/

PyTorch Lighting

- Step 2: Define dataset

```
# setup data
dataset = MNIST(os.getcwd(), download=True, transform=ToTensor())
train_loader = utils.data.DataLoader(dataset)
```

- Step 3: Train the model

```
# train the model (hint: here are some helpful Trainer arguments for rapid idea iteration)
trainer = L.Trainer(limit_train_batches=100, max_epochs=1)
trainer.fit(model=autoencoder, train_dataloaders=train_loader)
```

- Documentation: https://lightning.ai/docs/pytorch/stable/

PyTorch Lighting

- Be wary:
    - debugging is more complex
    - less flexibility
    - potential infra overhead
    - big API changes over versions

- Documentation: https://lightning.ai/docs/pytorch/stable/