


Table of Contents

Getting Started with Fully Sharded Data Parallel(FSDP)

Author: [Hamid Shojanazeri](#), [Yanli Zhao](#), [Shen Li](#)

• NOTE

 View and edit this tutorial in [github](#).

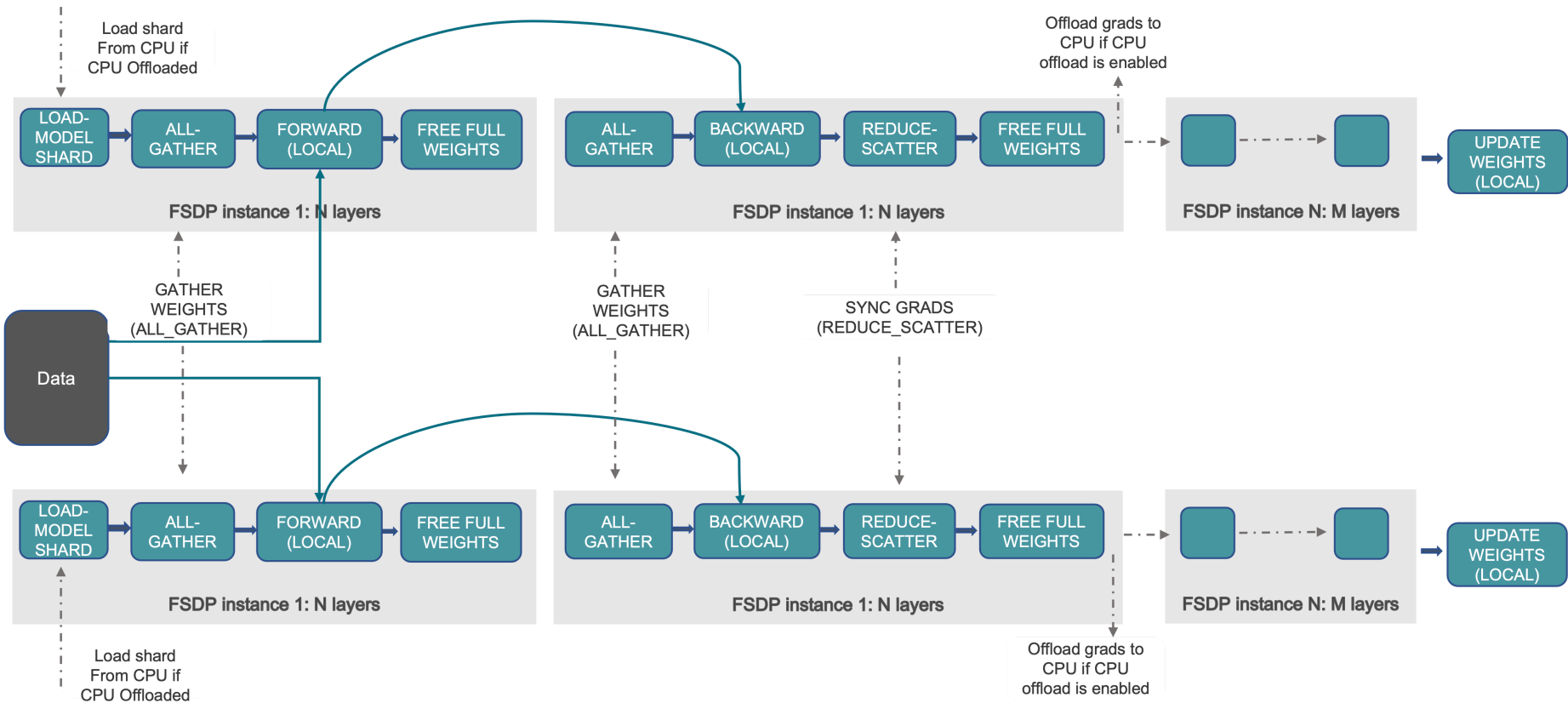
Training AI models at a large scale is a challenging task that requires a lot of compute power and resources. It also comes with considerable engineering complexity to handle the training of these very large models. [PyTorch FSDP](#), released in PyTorch 1.11 makes this easier.

In this tutorial, we show how to use [FSDP APIs](#), for simple MNIST models that can be extended to other larger models such as [HuggingFace BERT models](#), [GPT 3 models up to 1T parameters](#). The sample DDP MNIST code has been borrowed from [here](#).

How FSDP works

In [DistributedDataParallel](#), (DDP) training, each process/ worker owns a replica of the model and processes a batch of data, finally it uses all-reduce to sum up gradients over different workers. In DDP the model weights and optimizer states are replicated across all workers. FSDP is a type of data parallelism that shards model parameters, optimizer states and gradients across DDP ranks.

When training with FSDP, the GPU memory footprint is smaller than when training with DDP across all workers. This makes the training of some very large models feasible by allowing larger models or batch sizes to fit on device. This comes with the cost of increased communication volume. The communication overhead is reduced by internal optimizations like overlapping communication and computation.



FSDP Workflow

At a high level FSDP works as follow:

In constructor

- Shard model parameters and each rank only keeps its own shard

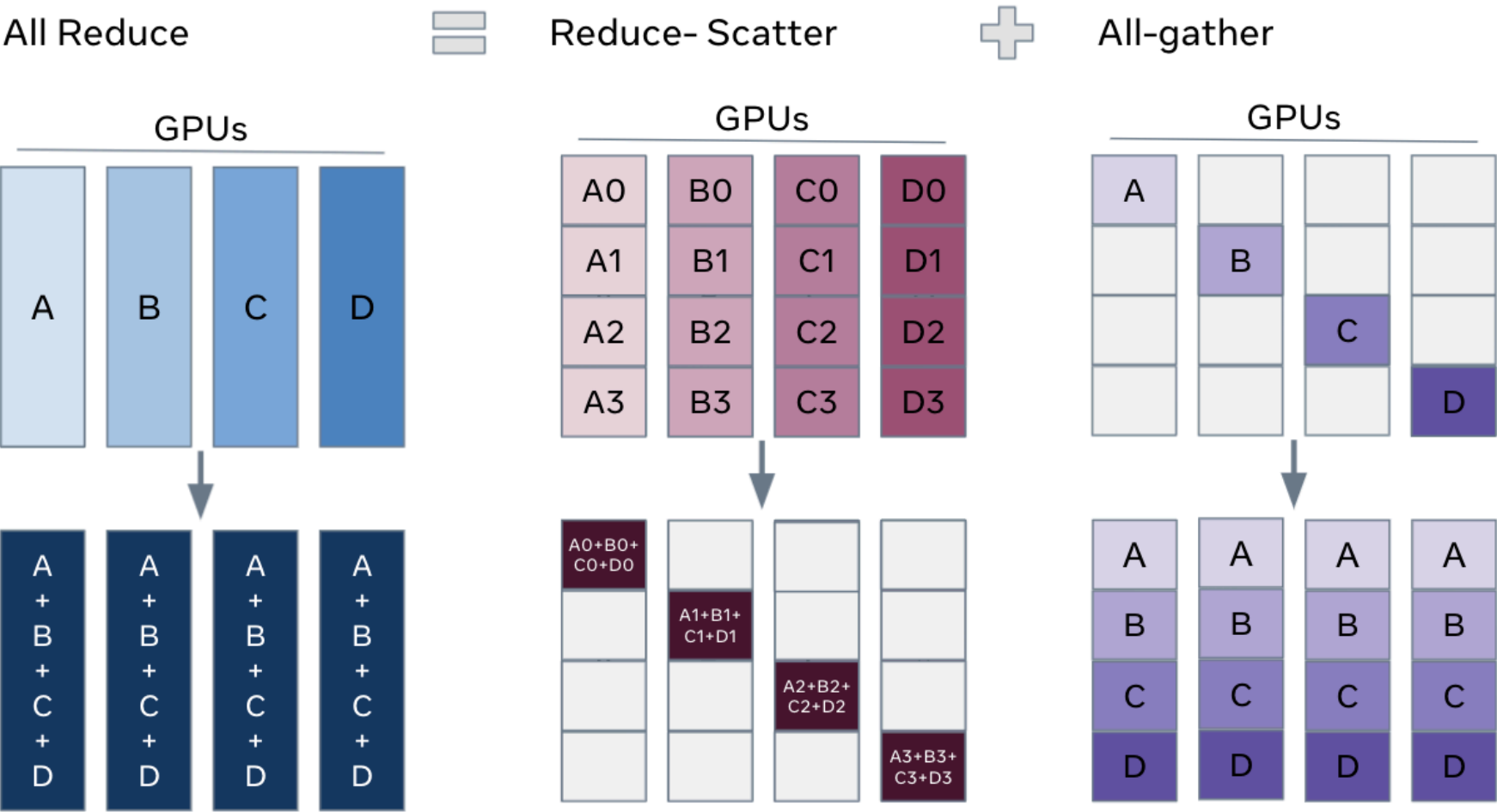
In forward path

- Run all_gather to collect all shards from all ranks to recover the full parameter in this FSDP unit
- Run forward computation
- Discard parameter shards it has just collected

In backward path

- Run all_gather to collect all shards from all ranks to recover the full parameter in this FSDP unit
- Run backward computation
- Run reduce_scatter to sync gradients
- Discard parameters.

One way to view FSDP's sharding is to decompose the DDP gradient all-reduce into reduce-scatter and all-gather. Specifically, during the backward pass, FSDP reduces and scatters gradients, ensuring that each rank possesses a shard of the gradients. Then it updates the corresponding shard of the parameters in the optimizer step. Finally, in the subsequent forward pass, it performs an all-gather operation to collect and combine the updated parameter shards.



FSDP Allreduce

How to use FSDP

Here we use a toy model to run training on the MNIST dataset for demonstration purposes. The APIs and logic can be applied to training larger models as well.

Setup

1.1 Install PyTorch along with Torchvision

See the [Get Started guide](#) for information on installation.

We add the following code snippets to a python script “FSDP_mnist.py”.

1.2 Import necessary packages

• NOTE

This tutorial is intended for PyTorch versions 1.12 and later. If you are using an earlier version, replace all instances of `size_based_auto_wrap_policy` with `default_auto_wrap_policy`.

```
# Based on: https://github.com/pytorch/examples/blob/master/mnist/main.py
import os
import argparse
import functools
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms

from torch.optim.lr_scheduler import StepLR

import torch.distributed as dist
import torch.multiprocessing as mp
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.utils.data.distributed import DistributedSampler
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP
from torch.distributed.fsdp.fully_sharded_data_parallel import (
    CPUOffload,
    BackwardPrefetch,
)
from torch.distributed.fsdp.wrap import (
    size_based_auto_wrap_policy,
    enable_wrap,
    wrap,
)
```

1.3 Distributed training setup. As we mentioned FSDP is a type of data parallelism which requires a distributed training environment, so here we use two helper functions to initialize the processes for distributed training and clean up.

```
def setup(rank, world_size):
    os.environ['MASTER_ADDR'] = 'localhost'
    os.environ['MASTER_PORT'] = '12355'

    # initialize the process group
    dist.init_process_group("nccl", rank=rank, world_size=world_size)

def cleanup():
    dist.destroy_process_group()
```

2.1 Define our toy model for handwritten digit classification.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):

        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output
```

2.2 Define a train function

```
def train(args, model, rank, world_size, train_loader, optimizer, epoch, sampler=None):
    model.train()
    ddp_loss = torch.zeros(2).to(rank)
    if sampler:
        sampler.set_epoch(epoch)
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(rank), target.to(rank)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target, reduction='sum')
        loss.backward()
        optimizer.step()
        ddp_loss[0] += loss.item()
        ddp_loss[1] += len(data)

    dist.all_reduce(ddp_loss, op=dist.ReduceOp.SUM)
    if rank == 0:
        print('Train Epoch: {} \tLoss: {:.6f}'.format(epoch, ddp_loss[0] / ddp_loss[1]))
```

2.3 Define a validation function

```
def test(model, rank, world_size, test_loader):
    model.eval()
    correct = 0
    ddp_loss = torch.zeros(3).to(rank)
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(rank), target.to(rank)
            output = model(data)
            ddp_loss[0] += F.nll_loss(output, target, reduction='sum').item() # sum up batch loss
            pred = output.argmax(dim=1, keepdim=True) # get the index of the max log-probability
            ddp_loss[1] += pred.eq(target.view_as(pred)).sum().item()
            ddp_loss[2] += len(data)

    dist.all_reduce(ddp_loss, op=dist.ReduceOp.SUM)

    if rank == 0:
        test_loss = ddp_loss[0] / ddp_loss[2]
        print('Test set: Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
            test_loss, int(ddp_loss[1]), int(ddp_loss[2]),
            100. * ddp_loss[1] / ddp_loss[2]))
```

2.4 Define a distributed train function that wraps the model in FSDP

Note: to save the FSDP model, we need to call the state_dict on each rank then on Rank 0 save the overall states.

```
def fsdp_main(rank, world_size, args):
    setup(rank, world_size)

    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])

    dataset1 = datasets.MNIST('../data', train=True, download=True,
                               transform=transform)
    dataset2 = datasets.MNIST('../data', train=False,
                               transform=transform)

    sampler1 = DistributedSampler(dataset1, rank=rank, num_replicas=world_size, shuffle=True)
    sampler2 = DistributedSampler(dataset2, rank=rank, num_replicas=world_size)

    train_kwargs = {'batch_size': args.batch_size, 'sampler': sampler1}
    test_kwargs = {'batch_size': args.test_batch_size, 'sampler': sampler2}
    cuda_kwargs = {'num_workers': 2,
                   'pin_memory': True,
                   'shuffle': False}
    train_kwargs.update(cuda_kwargs)
    test_kwargs.update(cuda_kwargs)

    train_loader = torch.utils.data.DataLoader(dataset1,**train_kwargs)
    test_loader = torch.utils.data.DataLoader(dataset2, **test_kwargs)
    my_auto_wrap_policy = functools.partial(
        size_based_auto_wrap_policy, min_num_params=100
    )
    torch.cuda.set_device(rank)

    init_start_event = torch.cuda.Event(enable_timing=True)
    init_end_event = torch.cuda.Event(enable_timing=True)

    model = Net().to(rank)

    model = FSDP(model)

    optimizer = optim.Adadelta(model.parameters(), lr=args.lr)

    scheduler = StepLR(optimizer, step_size=1, gamma=args.gamma)
    init_start_event.record()
    for epoch in range(1, args.epochs + 1):
        train(args, model, rank, world_size, train_loader, optimizer, epoch, sampler=sampler1)
        test(model, rank, world_size, test_loader)
        scheduler.step()

    init_end_event.record()

    if rank == 0:
        print(f"CUDA event elapsed time: {init_start_event.elapsed_time(init_end_event) / 1000}sec")
        print(f"{model}")

    if args.save_model:
        # use a barrier to make sure training is done on all ranks
        dist.barrier()
        states = model.state_dict()
        if rank == 0:
            torch.save(states, "mnist_cnn.pt")

    cleanup()
```

2.5 Finally, parse the arguments and set the main function

```
if __name__ == '__main__':
    # Training settings
    parser = argparse.ArgumentParser(description='PyTorch MNIST Example')
    parser.add_argument('--batch-size', type=int, default=64, metavar='N',
                        help='input batch size for training (default: 64)')
    parser.add_argument('--test-batch-size', type=int, default=1000, metavar='N',
                        help='input batch size for testing (default: 1000)')
    parser.add_argument('--epochs', type=int, default=10, metavar='N',
                        help='number of epochs to train (default: 14)')
    parser.add_argument('--lr', type=float, default=1.0, metavar='LR',
                        help='learning rate (default: 1.0)')
    parser.add_argument('--gamma', type=float, default=0.7, metavar='M',
                        help='Learning rate step gamma (default: 0.7)')
    parser.add_argument('--no-cuda', action='store_true', default=False,
                        help='disables CUDA training')
    parser.add_argument('--seed', type=int, default=1, metavar='S',
                        help='random seed (default: 1)')
    parser.add_argument('--save-model', action='store_true', default=False,
                        help='For Saving the current Model')
    args = parser.parse_args()

    torch.manual_seed(args.seed)

    WORLD_SIZE = torch.cuda.device_count()
    mp.spawn(fsdp_main,
            args=(WORLD_SIZE, args),
            nprocs=WORLD_SIZE,
            join=True)
```

We have recorded cuda events to measure the time of FSDP model specifics. The CUDA event time was 110.85 seconds.

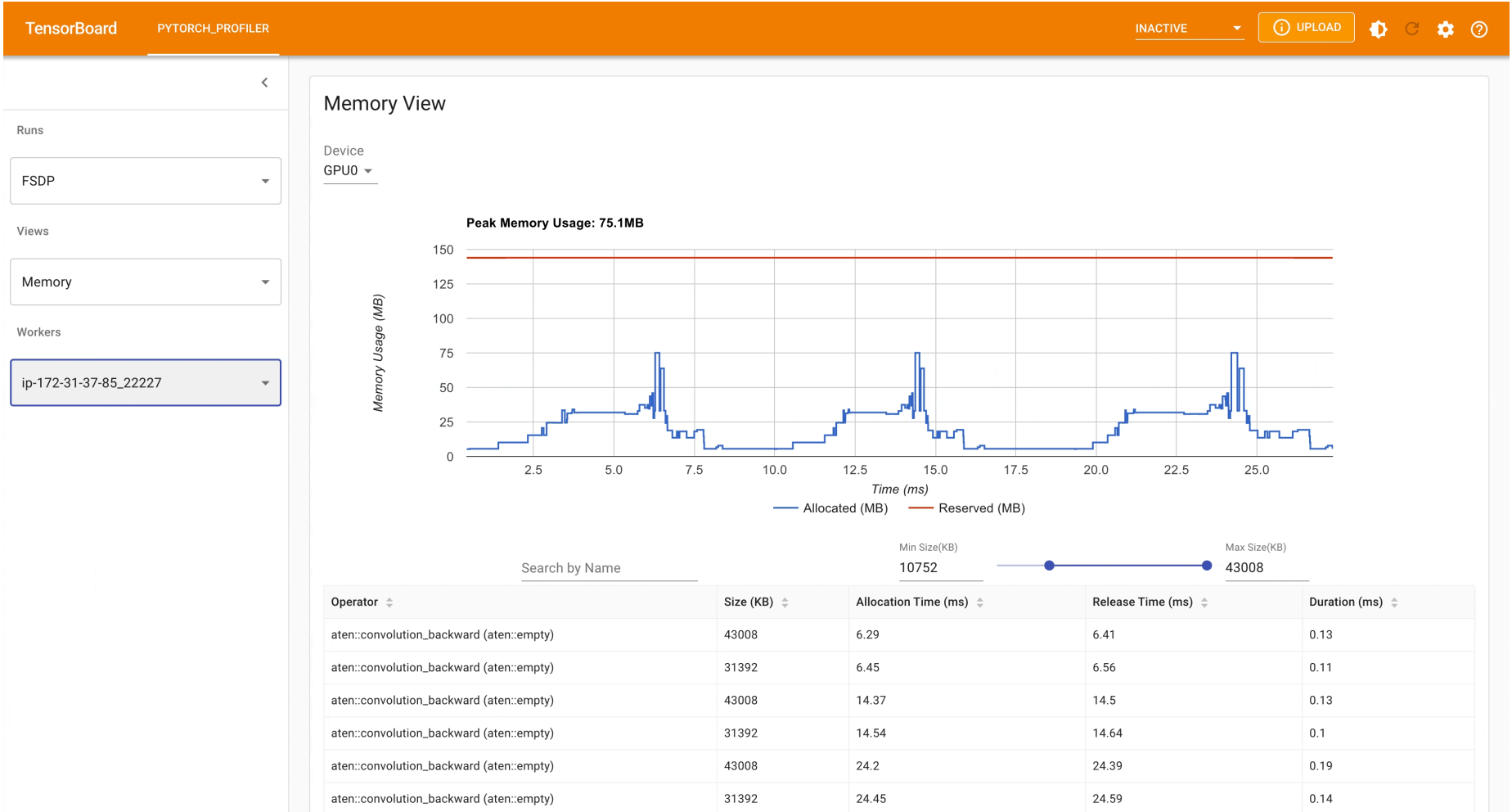
```
python FSDP_mnist.py

CUDA event elapsed time on training loop 40.67462890625sec
```

Wrapping the model with FSDP, the model will look as follows, we can see the model has been wrapped in one FSDP unit. Alternatively, we will look at adding the fsdp_auto_wrap_policy next and will discuss the differences.

```
FullyShardedDataParallel(
  (_fsdp_wrapped_module): FlattenParamsWrapper(
    (_fpw_module): Net(
      (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1))
      (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
      (dropout1): Dropout(p=0.25, inplace=False)
      (dropout2): Dropout(p=0.5, inplace=False)
      (fc1): Linear(in_features=9216, out_features=128, bias=True)
      (fc2): Linear(in_features=128, out_features=10, bias=True)
    )
  )
)
```

The following is the peak memory usage from FSDP MNIST training on g4dn.12.xlarge AWS EC2 instance with 4 GPUs captured from PyTorch Profiler.



FSDP Peak Memory Usage

Applying `fsdp_auto_wrap_policy` in FSDP otherwise, FSDP will put the entire model in one FSDP unit, which will reduce computation efficiency and memory efficiency. The way it works is that, suppose your model contains 100 Linear layers. If you do `FSDP(model)`, there will only be one FSDP unit which wraps the entire model. In that case, the allgather would collect the full parameters for all 100 linear layers, and hence won't save CUDA memory for parameter sharding. Also, there is only one blocking allgather call for the all 100 linear layers, there will not be communication and computation overlapping between layers.

To avoid that, you can pass in an `fsdp_auto_wrap_policy`, which will seal the current FSDP unit and start a new one automatically when the specified condition is met (e.g., size limit). In that way you will have multiple FSDP units, and only one FSDP unit needs to collect full parameters at a time. E.g., suppose you have 5 FSDP units, and each wraps 20 linear layers. Then, in the forward, the 1st FSDP unit will allgather parameters for the first 20 linear layers, do computation, discard the parameters and then move on to the next 20 linear layers. So, at any point in time, each rank only materializes parameters/grads for 20 linear layers instead of 100.

To do so in 2.4 we define the `auto_wrap_policy` and pass it to FSDP wrapper, in the following example, `my_auto_wrap_policy` defines that a layer could be wrapped or sharded by FSDP if the number of parameters in this layer is larger than 100. If the number of parameters in this layer is smaller than 100, it will be wrapped with other small layers together by FSDP. Finding an optimal auto wrap policy is challenging, PyTorch will add auto tuning for this config in the future. Without an auto tuning tool, it is good to profile your workflow using different auto wrap policies experimentally and find the optimal one.

```
my_auto_wrap_policy = functools.partial(
    size_based_auto_wrap_policy, min_num_params=20000
)
torch.cuda.set_device(rank)
model = Net().to(rank)

model = FSDP(model,
    fsdp_auto_wrap_policy=my_auto_wrap_policy)
```

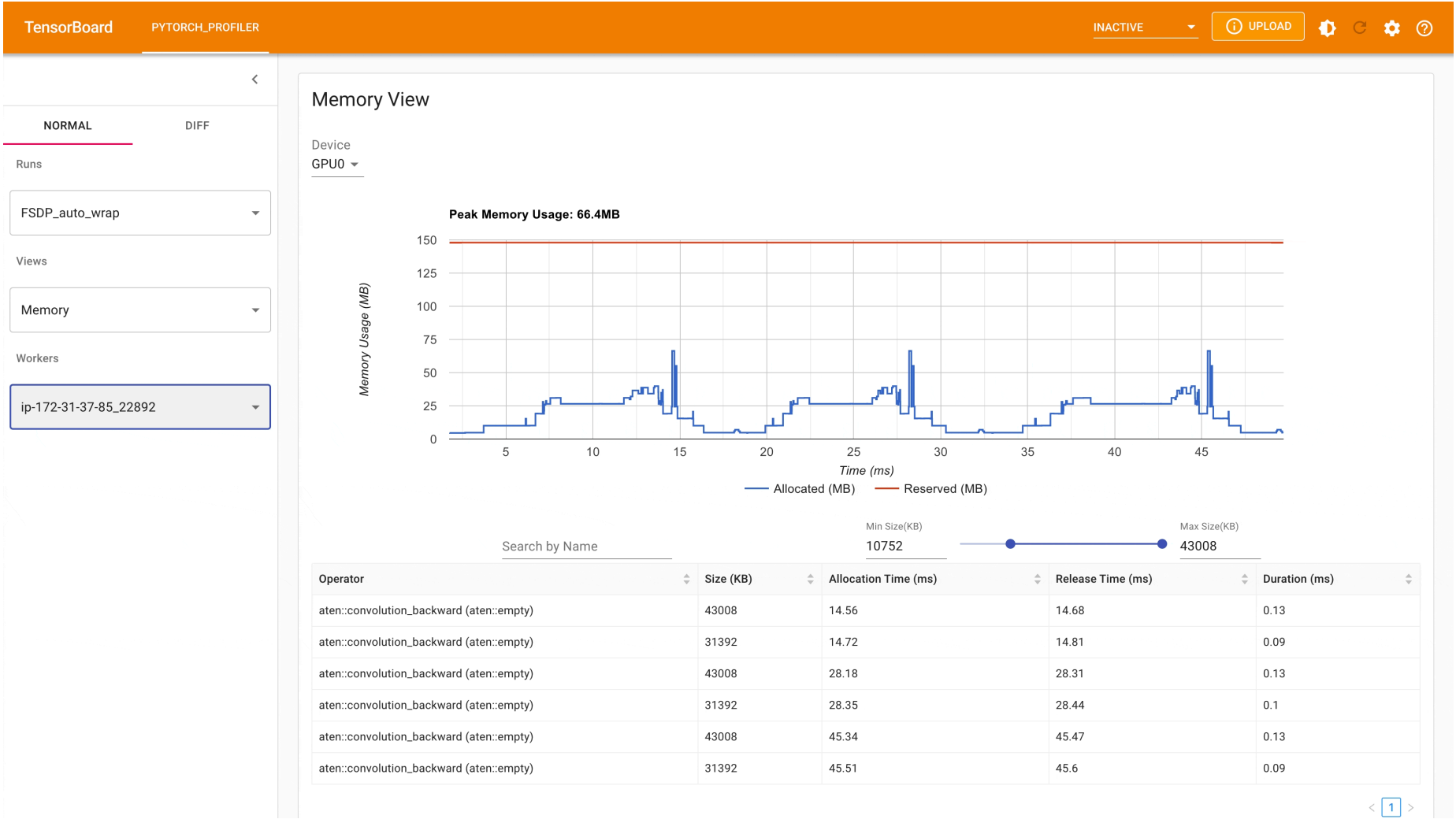
Applying the `fsdp_auto_wrap_policy`, the model would be as follows:

```
FullyShardedDataParallel(
  (_fsdp_wrapped_module): FlattenParamsWrapper(
    (_fpw_module): Net(
      (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1))
      (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
      (dropout1): Dropout(p=0.25, inplace=False)
      (dropout2): Dropout(p=0.5, inplace=False)
      (fc1): FullyShardedDataParallel(
        (_fsdp_wrapped_module): FlattenParamsWrapper(
          (_fpw_module): Linear(in_features=9216, out_features=128, bias=True)
        )
      )
      (fc2): Linear(in_features=128, out_features=10, bias=True)
    )
  )
)
```

```
python FSDP_mnist.py

CUDA event elapsed time on training loop 41.89130859375sec
```

The following is the peak memory usage from FSDP with `auto_wrap` policy of MNIST training on a `g4dn.12.xlarge` AWS EC2 instance with 4 GPUs captured from PyTorch Profiler. It can be observed that the peak memory usage on each device is smaller compared to FSDP without auto wrap policy applied, from ~75 MB to 66 MB.



FSDP Peak Memory Usage using Auto_wrap policy

CPU Off-loading: In case the model is very large that even with FSDP wouldn’t fit into GPUs, then CPU offload can be helpful here.

Currently, only parameter and gradient CPU offload is supported. It can be enabled via passing in `cpu_offload=CPUOffload(offload_params=True)`.

Note that this currently implicitly enables gradient offloading to CPU in order for params and grads to be on the same device to work with the optimizer. This API is subject to change. The default is `None` in which case there will be no offloading.

Using this feature may slow down the training considerably, due to frequent copying of tensors from host to device, but it could help improve memory efficiency and train larger scale models.

In 2.4 we just add it to the FSDP wrapper

```
model = FSDP(model,
    fsdp_auto_wrap_policy=my_auto_wrap_policy,
    cpu_offload=CPUOffload(offload_params=True))
```

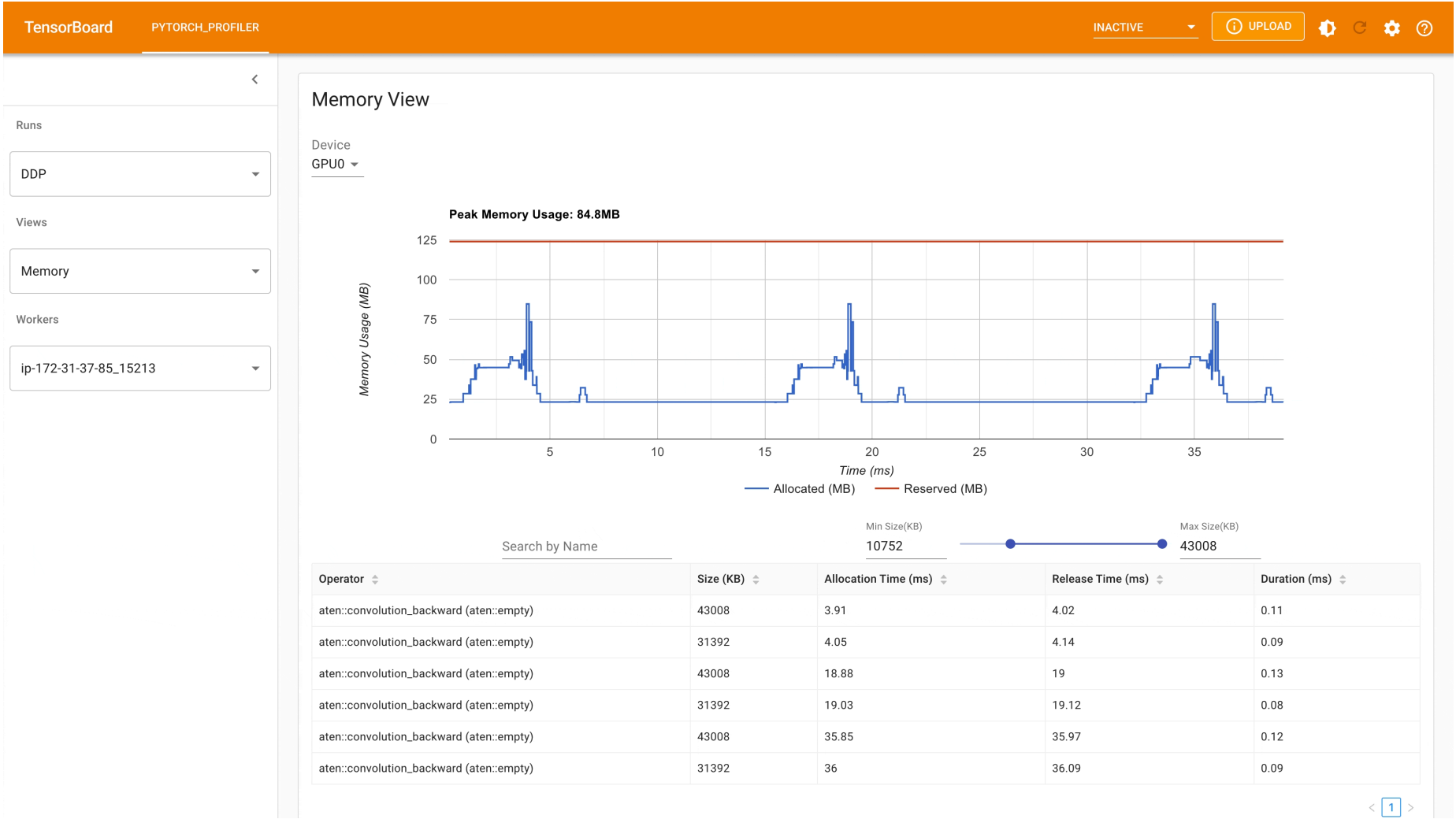
Compare it with DDP, if in 2.4 we just normally wrap the model in DPP, saving the changes in “DDP_mnist.py”.

```
model = Net().to(rank)
model = DDP(model)
```

```
python DDP_mnist.py

CUDA event elapsed time on training loop 39.77766015625sec
```

The following is the peak memory usage from DDP MNIST training on g4dn.12.xlarge AWS EC2 instance with 4 GPUs captured from PyTorch profiler.



DDP Peak Memory Usage using Auto_wrap policy

Considering the toy example and tiny MNIST model we defined here, we can observe the difference between peak memory usage of DDP and FSDP. In DDP each process holds a replica of the model, so the memory footprint is higher compared to FSDP which shards the model parameters, optimizer states and gradients over DDP ranks. The peak memory usage using FSDP with auto_wrap policy is the lowest followed by FSDP and DDP.

Also, looking at timings, considering the small model and running the training on a single machine, FSDP with and without auto_wrap policy performed almost as fast as DDP. This example does not represent most of the real applications, for detailed analysis and comparison between DDP and FSDP please refer to this [blog post](#) .

< Previous

Next >

Rate this Tutorial

☆☆☆☆☆

Docs

Access comprehensive developer documentation for PyTorch
[View Docs](#)

Tutorials

Get in-depth tutorials for beginners and advanced developers
[View Tutorials](#)

Resources

Find development resources and get your questions answered
[View Resources](#)

PyTorch

Get Started

Features

Ecosystem

Blog

Contributing

Resources

Tutorials

Docs

Discuss

Github Issues

Brand Guidelines

[Terms](#) | [Privacy](#)

© Copyright The Linux Foundation. The PyTorch Foundation is a project of The Linux Foundation. For web site terms of use, trademark policy and other policies applicable to The PyTorch Foundation please see www.linuxfoundation.org/policies/. The PyTorch Foundation supports the PyTorch open source project, which has been established as PyTorch Project a Series of LF Projects, LLC. For policies applicable to the PyTorch Project a Series of LF Projects, LLC, please see www.lfprojects.org/policies/.