Multi-GPU Training in PyTorch with Code (Part 3): Distributed Data Parallel

Anthony Peng · Follow
Published in Polo Club of Data Science | Georgia Tech
12 min read · Jul 7, 2023

Listen Share More

We discussed single-GPU training in Part 1 and multi-GPU training with DP in Part 2. In Part 2, we found DP is incompatible with GPUs w/o NVLinks. In this article, we will bypass that issue by leveraging multi-GPU training with Distributed Data Parallel (DDP).

Part 1. Single GPU Example — Training ResNet34 on CIFAR10

Part2. Data Parallel — Training code & issue between DP and NVLink

> ***Part3. Distributed Data Parallel (this article)*** *— Training code & Analysis*

Part4. Torchrun — Fault tolerance

Multi-GPU Distributed Data Parallel

1. What is Distributed Data Parallel (DDP)?
   DDP enables data parallel training in PyTorch. Data parallelism is a way to process multiple data batches across multiple devices simultaneously to achieve better performance. In PyTorch, the DistributedSampler ensures each device gets a non-overlapping input batch.

2. What's the difference between DP and DDP?
   TL;DR — Use DDP instead of DP
   This table in the official tutorial provides a nice comparison between DP and DDP.

3. How is DDP implemented in PyTorch?
   This less than 4min video provides a quick overview of the underlying logic of DDP. If you want more details, please check the internal design of DDP.

DDP is more intrusive into your code than DP, so we need to modify multiple parts of the single-GPU example in Part 1.

DDP initialization. Rank is the unique ID of your GPU, and world_size is the total processes, which is the number of GPUs since each process controls one GPU. The init_process_group currently supports three types of backends: gloo, nccl, and mpi. The nccl is required if we want to build with CUDA.

```
import os
import torch.multiprocessing as mp
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.utils.data.distributed import (
    DistributedSampler,
)  # Distribute data across multiple gpus
from torch.distributed import init_process_group, destroy_process_group


# Each process control a single gpu
def ddp_setup(rank: int, world_size: int):
    os.environ["MASTER_ADDR"] = "localhost"
    os.environ["MASTER_PORT"] = "54321"  # select any idle port on your machine

    init_process_group(backend="nccl", rank=rank, world_size=world_size)
```

Dataloader. Dataloader determines how we load the data into batches during training, evaluation, and testing time. In DDP, the DistributedSampler ensures each device gets a non-overlapping input batch.

```
        bs: int,
    ) -> Tuple[DataLoader, DataLoader, DistributedSampler]:
        sampler_train = DistributedSampler(trainset)
        trainloader = DataLoader(
            trainset, batch_size=bs, shuffle=False, sampler=sampler_train, num_work
        )
        testloader = DataLoader(
            testset,
            batch_size=bs,
            shuffle=False,
            sampler=DistributedSampler(testset, shuffle=False),
            num_workers=8,
        )

        return trainloader, testloader, sampler_train
```

TrainerDDP. As TrainerSingle in Part 1 and TrainerDP in Part 2, TrainerDDP takes care of the training and testing process. TrainerDDP inherits TrainerSingle so that we can better visualize what's changed from the single-GPU example.

In the Dataloader, we set shuffle=False due to the DistributedSampler.

> PyTorch DataLoader
> ***sampler*** *(Sampler or Iterable, optional) — defines the strategy to draw samples from the dataset. Can be any* `Iterable` *with* `__len__` *implemented. If specified,* `shuffle` *must not be specified.*

To shuffle the dataset while using the DistributedSampler, we have to call the DistributedSampler.set_epoch method. We will explore whether this *set_epoch* shuffles intra- or cross- GPU in the Experiment section.

> *In distributed mode, calling the* `set_epoch()` *method at the beginning of each epoch* ***before*** *creating the* `DataLoader` *iterator is necessary to make shuffling work properly across multiple epochs. Otherwise, the same ordering will be always used.*

We use the torchmetrics to compute the classification accuracy due to its support of distributed scenarios. We will manually compute the accuracy and verify its correctness in the Experiment section.

```
            f"[GPU{self.gpu_id}] Test Acc: {100 * self.valid_acc.compute().item
        )
```

Main function. We need an extra line of code that sets up the distributed scenario and another line of code that cleans up all processes after training.

```
def main_ddp(
    rank: int,
    world_size: int,
    final_model_path: str,
):
    ddp_setup(rank, world_size)  # initialize ddp

    const = prepare_const()
    train_dataset, test_dataset = cifar_dataset(const["data_root"])
    train_dataloader, test_dataloader, train_sampler = cifar_dataloader_ddp(
        train_dataset, test_dataset, const["batch_size"]
    )
    model = cifar_model()
    trainer = TrainerDDP(
        gpu_id=rank,
        model=model,
        trainloader=train_dataloader,
        testloader=test_dataloader,
        sampler_train=train_sampler,
    )
    trainer.train(const["total_epochs"])
    trainer.test(final_model_path)

    destroy_process_group()  # clean up
```

Experiments

```
if __name__ == "__main__":
    world_size = torch.cuda.device_count()
    final_model_path = Path("./trained_models/CIFAR10_ddp_epoch14.pt")
    mp.spawn(
        main_ddp,
```

```
        args=(world_size, final_model_path),
        nprocs=world_size,
    )  # nprocs - total number of processes - # gpus
```

1. GPUs w/ & w/o NVLinks. Recall that GPU 6&7 have the NVLink, but GPU 5&6 doesn't. The training converges to the same level of loss and accuracy, which bypasses the issue we encountered back in DP. Same observations also apply to GPU 5&6&7 and 4&5&6&7.

```
----------------------------------------------------------------------------------------------
[GPU0] Epoch  4 | Batchsize: 128 | Steps: 196 | LR: 0.0100 | Loss: 0.9888 | Acc
----------------------------------------------------------------------------------------------
[GPU1] Epoch  5 | Batchsize: 128 | Steps: 196 | LR: 0.0100 | Loss: 0.7641 | Acc
----------------------------------------------------------------------------------------------
[GPU0] Epoch  5 | Batchsize: 128 | Steps: 196 | LR: 0.0100 | Loss: 0.7484 | Acc
----------------------------------------------------------------------------------------------
[GPU1] Epoch  6 | Batchsize: 128 | Steps: 196 | LR: 0.0100 | Loss: 0.6775 | Acc
----------------------------------------------------------------------------------------------
[GPU0] Epoch  6 | Batchsize: 128 | Steps: 196 | LR: 0.0100 | Loss: 0.6746 | Acc
----------------------------------------------------------------------------------------------
[GPU1] Epoch  7 | Batchsize: 128 | Steps: 196 | LR: 0.0100 | Loss: 0.6188 | Acc
----------------------------------------------------------------------------------------------
[GPU0] Epoch  7 | Batchsize: 128 | Steps: 196 | LR: 0.0100 | Loss: 0.6156 | Acc
----------------------------------------------------------------------------------------------
[GPU1] Epoch  8 | Batchsize: 128 | Steps: 196 | LR: 0.0100 | Loss: 0.5622 | Acc
----------------------------------------------------------------------------------------------
[GPU0] Epoch  8 | Batchsize: 128 | Steps: 196 | LR: 0.0100 | Loss: 0.5509 | Acc
----------------------------------------------------------------------------------------------
[GPU1] Epoch  9 | Batchsize: 128 | Steps: 196 | LR: 0.0010 | Loss: 0.4815 | Acc
----------------------------------------------------------------------------------------------
[GPU0] Epoch  9 | Batchsize: 128 | Steps: 196 | LR: 0.0010 | Loss: 0.4905 | Acc
----------------------------------------------------------------------------------------------
[GPU1] Epoch 10 | Batchsize: 128 | Steps: 196 | LR: 0.0010 | Loss: 0.3508 | Acc
----------------------------------------------------------------------------------------------
[GPU0] Epoch 10 | Batchsize: 128 | Steps: 196 | LR: 0.0010 | Loss: 0.3537 | Acc
----------------------------------------------------------------------------------------------
[GPU1] Epoch 11 | Batchsize: 128 | Steps: 196 | LR: 0.0010 | Loss: 0.3093 | Acc
----------------------------------------------------------------------------------------------
[GPU0] Epoch 11 | Batchsize: 128 | Steps: 196 | LR: 0.0010 | Loss: 0.3141 | Acc
----------------------------------------------------------------------------------------------
[GPU1] Epoch 12 | Batchsize: 128 | Steps: 196 | LR: 0.0010 | Loss: 0.2853 | Acc
----------------------------------------------------------------------------------------------
[GPU0] Epoch 12 | Batchsize: 128 | Steps: 196 | LR: 0.0010 | Loss: 0.2949 | Acc
----------------------------------------------------------------------------------------------
[GPU0] Epoch 13 | Batchsize: 128 | Steps: 196 | LR: 0.0010 | Loss: 0.2731 | Acc
----------------------------------------------------------------------------------------------
[GPU1] Epoch 13 | Batchsize: 128 | Steps: 196 | LR: 0.0010 | Loss: 0.2651 | Acc
----------------------------------------------------------------------------------------------
[GPU1] Epoch 14 | Batchsize: 128 | Steps: 196 | LR: 0.0001 | Loss: 0.2477 | Acc
----------------------------------------------------------------------------------------------
[GPU0] Epoch 14 | Batchsize: 128 | Steps: 196 | LR: 0.0001 | Loss: 0.2567 | Acc
[GPU0] Test Acc: 70.1200%
[GPU1] Test Acc: 70.1200%
```

2. Disable shuffling. We disable shuffling by commenting out *self.sampler_train.set_epoch(epoch)* in the *TrainerDDP.train* method. We omit the

output and present the training result below. In general, training w/ shuffling leads to higher accuracy.

```
DDP - w/o shuffling
      - Acc:  74.99%, 72.94%, 72.34%, [73.86% (gpu5+6)], [72.33% (gpu4+5+6+7)]
      - Loss: 0.0150, 0.0060, 0.0094, [0.0105 (gpu5+6)], [0.0106 (gpu4+5+6+7)]
DDP - w/ shuffling
      - Acc:  76.04%, 74.25%, 74.72%, [75.63% (gpu5+6)], [76.12% (gpu4+5+6+7)]
      - Loss: 0.0026, 0.0115, 0.0115, [0.0105 (gpu5+6)], [0.0075 (gpu4+5+6+7)]
```

3. Intra- or Cross- GPU shuffling in DistributedSampler. The Dataset is a list of integers.

```
    world_size = torch.cuda.device_count()
    mp.spawn(main_ddp_shuffle, args=(world_size,), nprocs=world_size)
```

Output. Clearly, the shuffling is cross-GPU since GPU0 has new data in epoch 1 that was possessed by GPU1 in epoch 0. The DistributedSampler code also echos the cross-GPU shuffling.

```
$ CUDA_VISIBLE_DEVICES=5,6 python main.py
[GPU1] Epoch 0 | Step 0 | Data [0, 3, 5, 6]
[GPU1] Epoch 1 | Step 0 | Data [4, 6, 3, 0]
[GPU0] Epoch 0 | Step 0 | Data [4, 7, 2, 1]
[GPU0] Epoch 1 | Step 0 | Data [5, 2, 7, 1]
```

4. Torchmetrics. We manually compute the classification accuracy on each GPU. The code modification in TrainerDDP.test is shown below.

```
class TrainerDDP(TrainerSingle):
    def test(self, final_model_path: str):
        self.model.load_state_dict(torch.load(final_model_path, map_location="c
        self.model.eval()

        total_count = 0
        correct_count = 0

        with torch.no_grad():
            for src, tgt in self.testloader:
                src = src.to(self.gpu_id)
                tgt = tgt.to(self.gpu_id)
                out = self.model(src)
                self.valid_acc.update(out, tgt)

                total_count += len(tgt)
                correct_count += torch.sum(torch.argmax(out, dim=1) == tgt)

        print(
            f"[GPU{self.gpu_id}] Test Acc: {100 * self.valid_acc.compute().item
        )
```

Output. The training code is commented out for testing only. In the first test, (3521 + 3491) / 10,000 = 70.12%, which is exactly what torchmetrics have computed. In the second example, we computed the accuracy on four GPUs. (1751 + 1730 + 1740 + 1791) / 10,000 = 70.12%.

```
$ CUDA_VISIBLE_DEVICES=5,6 python main.py
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
[GPU0] Test Acc: 70.1200% | My Acc: 3491 / 5000 = 69.8200
[GPU1] Test Acc: 70.1200% | My Acc: 3521 / 5000 = 70.4200

$ CUDA_VISIBLE_DEVICES=4,5,6,7 python main.py
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
[GPU2] Test Acc: 70.1200% | My Acc: 1751 / 2500 = 70.0400
[GPU1] Test Acc: 70.1200% | My Acc: 1730 / 2500 = 69.2000
[GPU0] Test Acc: 70.1200% | My Acc: 1740 / 2500 = 69.6000
[GPU3] Test Acc: 70.1200% | My Acc: 1791 / 2500 = 71.6400
```

5. Uneven inputs. Let's compute the accuracy on three GPUs. Since 10,000 test samples are not evenly divided by 3, the DistributedSampler will pad the dataset with more samples until it can be evenly distributed to all GPUs. This design is to avoid code hanging in the ring all-reduce. The below results on three GPUs show that each GPU has 3334 samples. However, 3334 * 3 = 10,002 > 10,000, which is the result of padding. The accuracy is also slightly different than 70.12% due to the two extra correctly classified data samples — (-2 + 2317 + 2332 + 2365) / 10,000 = 70.12%.

```
$ CUDA_VISIBLE_DEVICES=4,5,6 python main.py
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
```

```
[GPU1] Test Acc: 70.1260% | My Acc: 2317 / 3334 = 69.4961
[GPU2] Test Acc: 70.1260% | My Acc: 2332 / 3334 = 69.9460
[GPU0] Test Acc: 70.1260% | My Acc: 2365 / 3334 = 70.9358
```

There are two ways to mitigate the excessive padding, 1) always perform testing on a single GPU, and 2) Customize a DistributedSampler to prevent padding. Below shows the code of the second solution.

```
class MyDistributedSampler(DistributedSampler):
    def __iter__(self) -> Iterator:
        indices = list(range(len(self.dataset)))
        indices = indices[self.rank : len(self.dataset) : self.num_replicas]
        return iter(indices)
```

We also replace the sampler in the Dataloader.

```python
    const = prepare_const()
    train_dataset, test_dataset = cifar_dataset(const["data_root"])
    train_dataloader, test_dataloader, train_sampler = cifar_dataloader_ddp_une
        train_dataset, test_dataset, const["batch_size"]
    )
    model = cifar_model()
    trainer = TrainerDDP(
        gpu_id=rank,
        model=model,
        trainloader=train_dataloader,
        testloader=test_dataloader,
        sampler_train=train_sampler,
    )
    # trainer.train(const["total_epochs"])
    trainer.test(final_model_path)

    destroy_process_group()  # clean up

if __name__ == "__main__":
    world_size = torch.cuda.device_count()
    final_model_path = Path("./trained_models/CIFAR10_ddp_epoch14.pt")
    mp.spawn(
        main_ddp_uneven,
        args=(world_size, final_model_path),
        nprocs=world_size,
    )
```

Output. We skip the training and test on the same trained model weights. A total of 3333 + 3333 + 3334 = 10,000 samples are distributed to three GPUs, and we have 70.12% back.

```
$ CUDA_VISIBLE_DEVICES=4,5,6 python main.py
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
[GPU1] Test Acc: 70.1200% | My Acc: 2316 / 3333 = 69.4869
[GPU2] Test Acc: 70.1200% | My Acc: 2331 / 3333 = 69.9370
[GPU0] Test Acc: 70.1200% | My Acc: 2365 / 3334 = 70.9358
```

Deep Learning | Gpu | Machine Learning | Pytorch