Introduction to Pytorch

Jupyter file with notes : ch02/Pyare-ch02/Apendix A_Introduction_to_Pytorch.ipynb

Training with multiple GPUs

```
import torch.multiprocessing as mp
from torch.utils.data.distributed import DistributedSampler
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.distributed import init_process_group, destroy_process_group
```

```
import torch.multiprocessing as mp
```

PyTorch's `multiprocessing` submodule contains functions such as `multiprocessing .spawn`, which we will use to spawn multiple processes and apply a function to multiple inputs in parallel. We will use it to spawn one training process per GPU. If we spawn multiple processes for training, we will need a way to **divide the dataset among these different processes**. For this, we will use the `DistributedSampler`.

`init_process_group` and `destroy_process_group` are used to initialize and quit the distributed training mods. The `init_process_group` function should be called at the **beginning of the training script** to initialize a process group for each process in the distributed setup, and `destroy_process_group` should be called at the **end of the training script** to destroy a given process group and release its resources.

```
import os
import torch
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader

import torch.multiprocessing as mp
from torch.utils.data.distributed import DistributedSampler
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.distributed import init_process_group, destroy_process_group

def ddp_setup(rank, world_size):
    os.environ['MASTER_ADDR'] = 'localhost' # 1 Address of the main node
    os.environ['MASTER_PORT'] = '12355' # 2. Any free port on the machine

    init_process_group(
        backend='nccl', #3 nccl stands for NVIDIA Collective Communication Library.
        rank=rank, # #4 rank refers to the index of the GPU we want to use.
        world_size=world_size, #5 world_size is the number of GPUs to use.
    )
```

```
    torch.cuda.set_device(rank) # #6 Sets the current GPU device on which tensors will be a

class ToyDataset(Dataset):
    def __init__(self,X,y):
        self.features = X
        self.labels = y

    def __getitem__(self,index): #1. Instructions for retrieving exactly one data record and the

        one_x = self.features[index] # 1.
        one_y = self.labels[index] # 1.
        return one_x, one_y

    def __len__(self):
        return self.labels.shape[0] #2 Instructions for returning the total length of the dataset


class NeuralNetwork(torch.nn.Module):
    def __init__(self, num_inputs, num_outputs):
        ''' 1. Coding the number of inputs and outputs as variables
        allows us to reuse the same code for datasets with different
        numbers of features and classes
        '''
        super().__init__()

        self.layers = torch.nn.Sequential(
            # first hidden layer
            torch.nn.Linear(num_inputs, 30), #2. The Linear layer takes the number of input and
            torch.nn.ReLU(),            #3. Nonlinear activation functions are placed between the h

            # second hidden layer
            torch.nn.Linear(30,20),   #4. The number of output nodes of one hidden layer has to
            torch.nn.ReLU(),


            # the output layer
            torch.nn.Linear(20, num_outputs),

        )

    def forward (self, x):
        logits = self.layers(x)

        return logits  # 5. The outputs of the last layer are called logits.
```

```
def prepare_dataset():
    X_train = torch.tensor([
        [-1.2, 3.1],
        [-0.9, 2.9],
        [-0.5, 2.6],
        [2.3, -1.1],
        [2.7, -1.5]
    ])
    y_train = torch.tensor([0, 0, 0, 1, 1])

    X_test = torch.tensor([
        [-0.8, 2.8],
        [2.6, -1.6],
    ])
    y_test = torch.tensor([0, 1])

    # Uncomment these lines to increase the dataset size to run this script on up to 8 GPUs:
    # factor = 4
    # X_train = torch.cat([X_train + torch.randn_like(X_train) * 0.1 for _ in range(factor)])
    # y_train = y_train.repeat(factor)
    # X_test = torch.cat([X_test + torch.randn_like(X_test) * 0.1 for _ in range(factor)])
    # y_test = y_test.repeat(factor)

    train_ds = ToyDataset(X_train, y_train)
    test_ds = ToyDataset(X_test, y_test)

    train_loader = DataLoader(
        dataset=train_ds,
        batch_size=2,
        shuffle=False,  # NEW: False because of DistributedSampler below #7 Distributed-Sam
        pin_memory=True, #8 Enables faster memory transfer when training on GPU
        drop_last=True, #9 Splits the dataset into distinct, non-overlapping subsets for each pr
        # NEW: chunk batches across GPUs without overlapping samples:
        sampler=DistributedSampler(train_ds)  # NEW #10 Distributed-Sampler takes care of th
    )
    test_loader = DataLoader(
        dataset=test_ds,
        batch_size=2,
        shuffle=False,
    )
    return train_loader, test_loader

# def prepare_dataset():
#         # insert the dataset preparation code here
```

```
#       train_loader = DataLoader(
#           dataset = train_ds,
#           batch_size= 2,
#           shuffle= False, #7 Distributed-Sampler takes care of the shuffling now.
#           pin_memory= True, #8 Enables faster memory transfer when training on GPU
#           drop_last= True, #9 Splits the dataset into distinct, non-overlapping subsets for ea


#           sampler= DistributedSampler(train_ds)
#       )
#       return train_loader, test_loader



# NEW: wrapper
def main(rank, world_size, num_epochs):

    ddp_setup(rank, world_size)  # NEW: initialize process groups

    train_loader, test_loader = prepare_dataset()
    model = NeuralNetwork(num_inputs=2, num_outputs=2)
    model.to(rank)
    optimizer = torch.optim.SGD(model.parameters(), lr=0.5)

    model = DDP(model, device_ids=[rank])  # NEW: wrap model with DDP
    # the core model is now accessible as model.module

    for epoch in range(num_epochs):
        # NEW: Set sampler to ensure each epoch has a different shuffle order
        train_loader.sampler.set_epoch(epoch)

        model.train()
        for features, labels in train_loader:

            features, labels = features.to(rank), labels.to(rank)  # New: use rank
            logits = model(features)
            loss = F.cross_entropy(logits, labels)  # Loss function

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            # LOGGING
            print(f"[GPU{rank}] Epoch: {epoch+1:03d}/{num_epochs:03d}"
                f" | Batchsize {labels.shape[0]:03d}"
```

```
                f" | Train/Val Loss: {loss:.2f}")

    model.eval()

    try:
        train_acc = compute_accuracy(model, train_loader, device=rank)
        print(f"[GPU{rank}] Training accuracy", train_acc)
        test_acc = compute_accuracy(model, test_loader, device=rank)
        print(f"[GPU{rank}] Test accuracy", test_acc)

    ####################################################
    # NEW (not in the book):
    except ZeroDivisionError as e:
        raise ZeroDivisionError(
            f"{e}\n\nThis script is designed for 2 GPUs. You can run it as:\n"
            "CUDA_VISIBLE_DEVICES=0,1 python DDP-script.py\n"
            f"Or, to run it on {torch.cuda.device_count()} GPUs, uncomment the code on lines 10
        )
    ####################################################

    destroy_process_group()  # NEW: cleanly exit distributed mode


def compute_accuracy(model, dataloader, device):
    model = model.eval()
    correct = 0.0
    total_examples = 0

    for idx, (features, labels) in enumerate(dataloader):
        features, labels = features.to(device), labels.to(device)

        with torch.no_grad():
            logits = model(features)
        predictions = torch.argmax(logits, dim=1)
        compare = labels == predictions
        correct += torch.sum(compare)
        total_examples += len(compare)
    return (correct / total_examples).item()


if __name__ == "__main__":
    # This script may not work for GPUs > 2 due to the small dataset
    # Run `CUDA_VISIBLE_DEVICES=0,1 python DDP-script.py` if you have GPUs > 2
    print("PyTorch version:", torch.__version__)
    print("CUDA available:", torch.cuda.is_available())
```

```
    print("Number of GPUs available:", torch.cuda.device_count())
    torch.manual_seed(123)

    # NEW: spawn new processes
    # note that spawn will automatically pass the rank
    num_epochs = 3
    world_size = torch.cuda.device_count()
    mp.spawn(main, args=(world_size, num_epochs), nprocs=world_size)
    # nprocs=world_size spawns one process per GPU
```

We have a `__name__ == "main"` clause at the bottom containing code executed when we run the code as a Python script instead of importing it as a module. This code first prints the number of available GPUs using torch.cuda.device_count(), sets a random seed for reproducibility, and then spawns new processes using PyTorch's multiprocessesing.spawn function. Here, the spawn function launches one process per GPU setting nproces=world_size, where the world size is the number of available GPUs. This spawn function launches the code in the main function we define in the same script with some additional arguments provided via args. Note that the main function has a rank argument that we don't include in the `mp.spawn()` call. That's because the rank, which refers to the process ID we use as the GPU ID, is already passed automatically.

We transfer the model and data to the target device via .to(rank), which we use to refer to the GPU device ID. Also, we wrap the model via DDP, which enables the synchronization of the gradients between the different GPUs during training. After the training finishes and we evaluate the models, we use destroy_process_group() to cleanly exit the distributed training and free up the allocated resources.

Earlier I mentioned that each GPU will receive a different subsample of the training data. To ensure this, we set sampler=DistributedSampler(train_ds) in the training loader. The last function to discuss is ddp_setup. It sets the main node's address and port to allow for communication between the different processes, initializes the process group with the NCCL backend (designed for GPU-to-GPU communication), and sets the rank (process identifier) and world size (total number of processes). Finally, it specifies the GPU device corresponding to the current model training process rank.

Selecting available GPUs on a multi-GPU machine

If you wish to restrict the number of GPUs used for training on a multi-GPU machine, the simplest way is to use the CUDA_VISIBLE_DEVICES environment variable. To illustrate this, suppose your machine has multiple GPUs, and you only want to use one GPU—for example, the GPU with index 0. Instead of python some_script.py, you can run the following code from the terminal:

```
CUDA_VISIBLE_DEVICES=0 python some_script.py
```

Or, if your machine has four GPUs and you only want to use the first and third GPU, you can use

```
CUDA_VISIBLE_DEVICES=0,2 python some_script.py
```

Setting CUDA_VISIBLE_DEVICES in this way is a simple and effective way to manage GPU allocation without modifying your PyTorch scripts. Let's now run this code and see how it works in practice by launching the code as a script from the terminal:

```
python /Vaibhav/pyare/LLMs/DistributedDataParallel_DDP_script.py
```

```
(.venv) vaibhav@anusandhan:/Vaibhav/pyare/LLMs$ nvidia-smi
Tue Jul  1 08:45:34 2025
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.183.01             Driver Version: 535.183.01   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA A100 80GB PCIe          Off | 00000000:98:00.0 Off |                    0 |
| N/A   45C    P0             66W / 300W |  48914MiB / 81920MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+
|   1  NVIDIA A100 80GB PCIe          Off | 00000000:B1:00.0 Off |                    0 |
| N/A   72C    P0             330W / 300W |  75795MiB / 81920MiB |    100%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|    0   N/A  N/A      2547      G   /usr/lib/xorg/Xorg                            4MiB |
|    0   N/A  N/A   1079612      C   python                                      768MiB |
|    0   N/A  N/A   1214460      C   python                                     6064MiB |
|    0   N/A  N/A   3061911      C   python                                    42040MiB |
|    0   N/A  N/A   3566901      G   /usr/lib/xorg/Xorg                            4MiB |
|    1   N/A  N/A      2547      G   /usr/lib/xorg/Xorg                            4MiB |
|    1   N/A  N/A   2585305      C   ...miniconda3/envs/clldm/bin/python3.9    75764MiB |
|    1   N/A  N/A   3566901      G   /usr/lib/xorg/Xorg                            4MiB |
+---------------------------------------------------------------------------------------+
```

As currently i don’t have 2nd Gpu available i will GPU=0,

```
CUDA_VISIBLE_DEVICES=0 python /Vaibhav/pyare/LLMs/DistributedDataParallel_DDP_scrip
```

```
(.venv) vaibhav@anusandhan:/Vaibhav/pyare/LLMs$ CUDA_VISIBLE_DEVICES=0 python /V
PyTorch version: 2.7.1+cu126
CUDA available: True
Number of GPUs available: 1
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.60
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.22
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.05
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.05
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.02
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.01
[GPU0] Training accuracy 1.0
[GPU0] Test accuracy 1.0
```

```
(.venv) vaibhav@anusandhan:/Vaibhav/pyare/LLMs$ CUDA_VISIBLE_DEVICES=0,1 python /
PyTorch version: 2.7.1+cu126
CUDA available: True
Number of GPUs available: 2
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.84
[GPU1] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.80
[GPU1] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.28
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.30
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.08
[GPU1] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.06
[GPU0] Training accuracy 1.0
[GPU0] Test accuracy 1.0
[GPU1] Training accuracy 1.0
[GPU1] Test accuracy 1.0
```

we can see that some batches are processed on the first GPU (GPU0) and others on the second (GPU1). However, we see duplicated output lines when printing the training and test accuracies. Each process (in other words, each GPU) prints the test accuracy independently. Since DDP replicates the model onto each GPU and each process runs independently, if you have a print statement inside your testing loop, each process will execute it, leading to repeated output lines. If this bothers you, you can fix it using the rank of each process to control your print statements:

```
if rank==0: # only print the first process
        print("Test Accuracy",accuracy)
```

For additional details follow :
https://docs.pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html

> 👉🏽 If you prefer a more straightforward way to use multiple GPUs in PyTorch, you can consider add-on APIs like the open-source Fabric library. Check blog "Accelerating PyTorch Model Training: Using Mixed-Precision and Fully Sharded Data Parallelism" (https://mng.bz/jXle).