Scaling to Multiple GPUs with DistributedDataParallel (DDP)

The DistributedDataParallel (DDP) package in PyTorch is a deep learning framework to accelerate training routines. In this lab, you will learn about the principles underlying DDP. Though our focus will be on distributing the training of a classification network across multiple GPUs (specifically H100s), the general concepts presented here are applicable for any model architecture and application.

Lab Outline

The progression of this lab is as follows:

- A high level introduction to DDP. General parallel computing concepts necessary to understand a distributed training framework like DDP are also presented.
- An overview and initial run of the existing code base, which is a classification model using PyTorch and the Fashion-MNIST dataset, that is built to run on only a single GPU.
- A multistep refactor of the existing code base so that it uses DDP to run distributed across this environment's available GPUs, introducing DDP terminology and practices throughout.
- A final run of the refactored and distributed code base, with discussion of its speed up.

This lab draws heavily on content provided in the official DDP tutorials.

Learning Objectives

By the time you complete this lab, you will be able to:

- Discuss what DDP is, how it works, and why it is an effective tool for distributed training.
- Use DDP to refactor or build deep learning models that train distributed across multiple GPUs.

Introduction to DDP

Model parallelism and data parallelism are popular strategies to train neural networks in a distributed manner. As their names suggest, while model parallelism partitions the layers of a given architecture across multiple GPUs, data parallelism batches the data into smaller, disjoint subsets that are simultaneously processed by the available GPUs. While it is possible to combine the two strategies for highly efficient training, it is common practice to begin with data parallelism and, if needed (as in the case of large language models like GPT-3 with billions of parameters), incorporate model parallelism as well.

The focus of this notebook will be on implementing data distributed training. The software package applied for such techniques is often dictated by the programming platform (e.g., Keras, PyTorch, MXNet, etc.). Because more and more researchers are using PyTorch, PyTorch's native DDP library is arising as a popular choice for data distributed training. With this in mind, DDP is leveraged in this notebook to teach the principles underlying data parallelism for accelerated neural network training. Learning to use other data parallelism tools like Horovod or DeepSpeed will be greatly facilitated with knowledge of DDP.

DDP Terminology

The following key terms will be important to understand DDP:

- A process or a worker (interchangeable terms) refers to an instance of the Python program. Each process controls a single, unique GPU. The number of available GPUs for distributed learning dictates the number of processes that are launched by DDP.
- A node or a host (another set of interchangeable terms) is a physical computer along with all of its hardware components (e.g., CPUs, GPUs, RAM, etc.). Each node is assigned a unique ID.
- World size is the total number of processes participating in training.
- The global rank is the unique ID assigned to each process. The process with global rank 0 is referred to as the main process.
- The local rank is the unique ID assigned to each process within a node.

Using these terms, we can review an example configuration with two nodes having two GPUs each (please refer to figure below). In this case, if we are leveraging all the available GPUs, the world size will be 4, and there will be four independent processes (one for each GPU). Each process is assigned a global rank ranging from 0 to 3, and each node/host has an ID of 0 or 1. The processes within each node have a local rank of either 0 or 1.

Data Parallelism with DDP

In data parallel training, the model architecture will be replicated across the available GPUs, with each GPU being fed a unique slice of the data. Training (i.e., forward propagation and backward propagation) is synchronously performed in a manner such that all model replicas are parametrized by the same weights. When it comes to setting up the model on each GPU (using `torch.nn.parallel.DistributedDataParallel`), an important point to note is that weights are initialized only on the main process (the rank 0 process), and are subsequently broadcasted to all other ranks. This is done to ensure all GPUs have the same copy of the model parameters before training commences. DDP addresses this synchronization internally. Following initialization, to send non-overlapping subsets of data to each GPU, we apply `torch.utils.data.distributed.DistributedSampler`. This allows each GPU to independently perform a forward pass operation, yielding a loss value. The calculated gradients are then broadcasted (via an all-reduce call), which provides an averaged gradient for each

parameter in our model. A backward pass using the averaged gradients ensures that model parameters continue to be synchronized across all GPUs after a training step is complete. Communication and synchronization between processes is handled by the `torch.distributed.init_process_group` and the `torch.nn.parallel.DistributedDataParallel` methods.

As you work through this course, continue to review the above terminologies/workflows, especially the fact that DDP will be sending your single program to be executed in parallel by multiple processes handling disjoint subsets of a training batch. Keeping these concepts in mind will support your intuition and understanding about why we do what we do with DDP, even though you will only be making edits to a single program.

Overview of Existing Model Files

On the left-hand side of this lab environment, you will see a file directory with this notebook, a couple of Python files, and a `solutions` directory.

The file `fashion_mnist.py` contains the PyTorch model that does not have any DDP code while `solutions/fashion_mnist_solution.py` has all the DDP features added. In this tutorial, we will guide you to transform `fashion_mnist.py` into `solutions/fashion_mnist_solution.py` step-by-step. As you complete the exercises in this task, you can, if needed, compare your code with the `solutions/fashion_mnist_after_step_N.py` files that correspond to the step you are working on.

Baseline: train the model

Before we go into the modifications required to scale our WideResNet model, please make sure you can train the single GPU version of the model. This is the same as where we left off at the end of Lab 1, except that now we are using the full dataset. We'll just run a few epochs with a relatively large batch size. This will take a bit longer than before, so feel free to read ahead while this is training. Take note of how long the training took when it is done.

In [ ]:
```
!python fashion_mnist.py --epochs 5 --batch-size 512
```

Modify the training script

Our current call to fashion_mnist.py entails the optional specification of several parameters like batch size, number of epochs, and learning rate. In moving to distributed training, it will be important to specify as inputs the number of nodes (along with their IDs) and the number of GPUs per node. As a result, our call to the function (and correspondingly its inner workings) will significantly change. It will be helpful to keep this in mind as we gradually add distributed functionality to the code. At some point (but outside the scope of this workshop) if we would

like to execute training on 4 nodes with 8 GPUs each, we will need 4 terminals (one on each node). On the node designated to have ID 0, the following will be our function call:

```
python fashion_mnist.py --node-id 0 --num-gpus 8 --num-nodes 4
```

Then, on the other nodes:

```
python fashion_mnist.py --node-id i --num-gpus 8 --num-nodes 4
```

for i ∈ {1,2,3}. In other words, we will run the script on each node, telling it to launch num_gpus processes (for a total of 32 processes across nodes in this example) that all sync with one other before training begins. As you will observe below, part of this synchronization is enabled by the `init_process_group()` command mentioned above, which servers as a blocker for each worker. That is, it ensures that each worker has successfully executed the command before proceeding to other operations.

We are going to start making modifications to the training script with an eye towards the goal of invoking the `fashion_mnist.py` function as just described for distributed training. Before we do, let's make a copy of it on disk -- that way, if you make a mistake and want to back up to the beginning, you have a reference copy to refer to.

In [ ]:
```
!cp fashion_mnist.py fashion_mnist_original.py
```

Double-click `fashion_mnist.py` in the left pane to open it in the editor.

0. Import DDP

We will begin by including the following import statement: `import torch.distributed as dist`. Note that this is simply an abbreviation for convenience. We do not need to import any new packages since the DDP library is included in the standard PyTorch installation. Complete this simple step in the `TODO Step 0` section of `fashion_mnist.py`.

1. Add input arguments for the number of nodes, number of GPUs per node, and a node ID

We will now expand the arguments accepted by our program to include three key parameters: number of nodes (in this notebook, we are limited to one node, but we will be building the distributed code for the general case where we have access to more than one node), the number of GPUs in our system, and an ID indicating our host (note that, as explained above, this node ID is different from the local rank and the global rank associated with each process). Implement these arguments at `TODO Step 1` in `fashion_mnist.py`.

To identify the number of GPUs in our single node, execute the following command:

In [ ]:
```
!nvidia-smi
```

The "!" prefix means that we execute the above in the terminal; now let's do this in actual terminal. Open a new launcher (File > New Launcher in the menu bar), select the "Terminal"

option, execute the `nvidia-smi` command there, and verify it provides the same output. Notice that there is a "GPU-Util" column, which measures the GPU's utilization. It tells you what fraction of the last second the GPU was in use. We can thus easily monitor GPU activity by regularly checking this output. One way to do that is using the Linux utility watch): `watch nvidia-smi` will set up a loop that refreshes the `nvidia-smi` output every 2 seconds. Make sure you run that in the separate terminal window, not here in the notebook, because the notebook can only run one process at a time. You can type Ctrl+C in the terminal to end the loop later.

Exercise: set up nvidia-smi to regularly monitor the GPU activity in a terminal as above, and here in the notebook, start a training process below. Then, switch back to the terminal and watch the GPU activity. Can you verify that only one GPU is used? Does it match the GPU ID you asked for in the training script (the GPU ID is prescribed in the following line of `fashion_mnist.py`: `device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu"`)? Also, keep an eye out for other utilization metrics like power consumption and memory usage.

In [ ]:
```
!python fashion_mnist.py --epochs 1 --batch-size 512
```

2. Determine the world size, and specify the node associated with the main process (global rank 0 process)

Having prescribed the number of nodes, number of GPUs per node, and the node ID, compute the world size at `TODO Step 2` in `fashion_mnist.py`. Also, in this same location, identify the IP address of the node associated with the main process. Because we have a single host in this notebook, we can set this IP address as simply `localhost`. In addition, designate a free port number (from 1024 to 49151) for the node associated with the main process. Specifying the IP address and a port number of the node containing the global rank 0 process is critical to enable communication between all the nodes.

3. Move our current training code into a `worker` method

When training our model using DDP, we will use torch's multiprocessing feature to spawn an independent process (executing a series of identical training steps) for each GPU. To enable this, we will first move our training logic into a `worker` method. Shift the training code at `TODO Step 3` in `fashion_mnist.py` to a new method called `worker`. Our goal is to have each process perform the training operations implemented in the `worker` function. The method to launch a predefined number of processes will be covered later in the notebook.

4. Compute global rank for each process and initialize process synchronization

Once processes are launched and the `worker` function within each process is invoked, we need to ensure proper initialization and synchronization with all other processes with the

`dist.init_process_group` method. This function accepts three parameters:

- A backend multiprocessing platform (we will be using the NVIDIA Collective Communications Library (NCCL))
- World size (calculated in step 2)
- Global rank of each process

The global rank of each process can be determined with the ID of the node, the number of GPUs per node, and the local rank. At `TODO Step 4` in `fashion_mnist.py`, compute the global rank of each process and apply the `dist.init_process_group` method.

5. Download dataset only once per host

The next several steps will focus on minimizing redundancy and facilitating communication between processes. Within a node, for efficiency purposes, we want only one process to download the dataset. Modify the `TODO Step 5` in `fashion_mnist.py` to reflect this change. Also include a call to `dist.barrier()` after the download step to ensure all other processes in a given node wait for the dataset to completely download.

6. Wrap training and validation data with DistributedSampler, and enable data shuffling

As discussed earlier, each process/GPU will be receiving a unique slice of the data for forward propagation. This can be accomplished with `torch.utils.data.distributed.DistributedSampler`. For each epoch, this function splits the dataset into a number of segments determined by the world size. Each GPU is then assigned a unique segment to process with the prescribed batch size. Refer to `TODO Step 6` in `fashion_mnist.py` to implement this function.

For each epoch, `torch.utils.data.distributed.DistributedSampler` by default shuffles the data prior to evenly dividing it into the number of segments determined by the world size. We, however, have to manually change the shuffling pattern (i.e., the random seed that is used to drive the shuffling) at the beginning of each epoch to avoid always using the same ordering. Refer to `TODO Step 6.5` within the epoch loop to make shuffling work properly.

Optional: synchronize batchnorm statistics across all processes

The default behavior of batchnorm layers in a distributed setting is to separately compute batch statistics for each process. Stated differently, for a given GPU, the mean and standard deviation for normalization are determined with only the slices of data passed to the device. Therefore, in any given training step, batch statistics do not capture all the batches processed in parallel by each GPU. This is not a significant issue when the batch size per device is large enough to obtain good statistics. However, as explored in the MegDet object detection paper, convergence and performance are negatively impacted if the batch size per device is very small. In such cases, it is important to synchronize batchnorm statistics across processes. To do this, we

simply need to convert regular batchnorm layers to `torch.nn.SyncBatchNorm` by using `torch.nn.SyncBatchNorm.convert_sync_batchnorm`. The `TODO Optional` section of `fashion_mnist.py` provides instructions for this change.

It is important to note that synchronization of batchnorm statistics will slow down training, given the additional communication overhead that is added in broadcasting batch statistics across processes. It is recommended to use this feature when model training is affected by a small batch size. This step is presented in this notebook for completeness, though it is not necessary to effectively train the WideResNet model in a distributed setting.

7. Pin process to the appropriate GPU and wrap model with DistributedDataParallel

In the single GPU setting, we defaulted to using the GPU with ID 0 as the device to store our model and our data. We now need to alter this logic to link each process a unique GPU. We will then call `nn.parallel.DistributedDataParallel` to produce a copy of the model on the designated GPU. Both these steps can be completed in the `TODO Step 7` of `fashion_mnist.py`. `nn.parallel.DistributedDataParallel` is the core function of the DDP library, enabling replication of the model across all processes and ensuring gradients for each parameter are averaged after a training step.

8. Modify computation for image throughput with a reduce call

The image throughput calculation for the single GPU case is now incorrect. The first step to properly calculate this metric will be to introduce a `dist.barrier()` call, which will ensure all processes have completed one training epoch. We will then perform a reduce operation to sum the image throughput across all the GPUs. Note that image throughput is now effectively the total number of images processed across GPUs divided by the duration of the slowest training epoch. Navigate to `TODO Step 8` of `fashion_mnist.py` to implement this functionality.

9. Average validation results among workers

Since we are not validating the full dataset on each GPU anymore, each process will have different validation results. To improve validation metric quality and reduce variance, we will average validation results among all workers. `TODO Step 9` of `fashion_mnist.py` focuses on combining validation metrics across all GPUs.

10. Spawn all processes with mp.spawn

Our code within the `worker` function has been modified to appropriately handle data, perform forward/back propagation, and compute training/validation metrics. We can now implement the method to launch each individual process. This is done in `TODO Step 10` of `fashion_mnist.py` with the `mp.spawn()` function call. Note that `mp.spawn()` should be

launched after bootstrapping of the overall program is complete (i.e., needs to be within `__main__`).

11. Quick evaluation

To make sure everything works as expected without any errors, call the function as below and observe the image throughput values that are being outputted. Note again that we are limited to a single node with four GPUs in this notebook. As mentioned earlier, to run this code on multiple nodes, the first step is to update the IP address (set as `localhost` in our example) of the main node with the global rank 0 process. Then, the same function call (after appropriately changing node ID and number of GPUs in node) needs to be replicated on a terminal window for each node.

In [ ]:
```
!python3 fashion_mnist.py --node-id 0 --num-gpus 4 --num-nodes 1  --epochs 3 --batch-s
```

Check your work

Congratulations! If you made it this far, your `fashion_mnist.py` should now be fully distributed. To verify, compare `fashion_mnist.py` to `solutions/fashion_mnist_solution.py`.