PyTorch
Distributed Training

Lecture 15 for 14-763/18-763

Guannan Qu

Oct 28, 2024

Recall: GPU

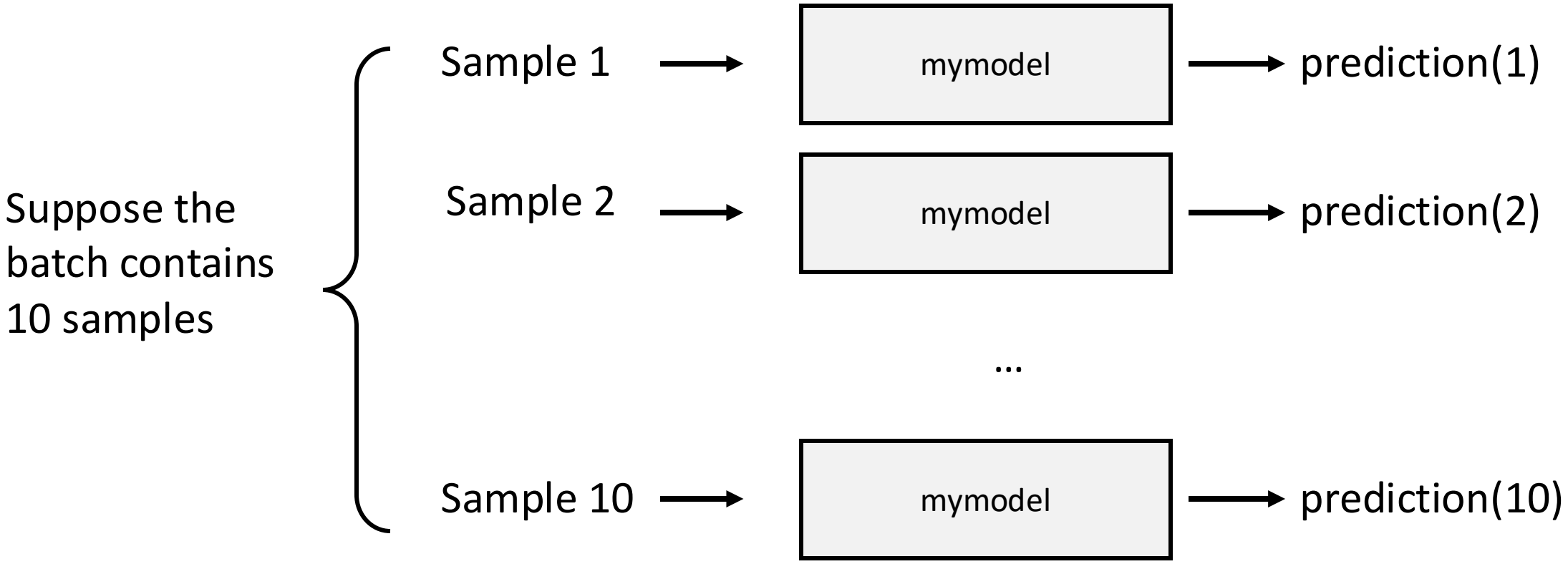
- Does GPU parallelize within a batch or across different batches. The answer is WITHIN THE BATCH.

```
for batch_id, (x_batch, y_batch) in enumerate(train_dataloader):
    start_time = time.time()
    # data to device
    x_batch = x_batch.to(device)
    y_batch = y_batch.to(device)

    # pass input data to get the prediction outputs by the current model
    prediction = mynn(x_batch)

    # compare prediction and the actual output label and compute the loss
    loss = nn.functional.cross_entropy(prediction,y_batch)

    # compute the gradient
    optimizer.zero_grad()
    loss.backward()

    # update parameters
    optimizer.step()
```

Batch comes after batch in a for loop, so parallelization is not across the batches

Parallelization happens inside a batch

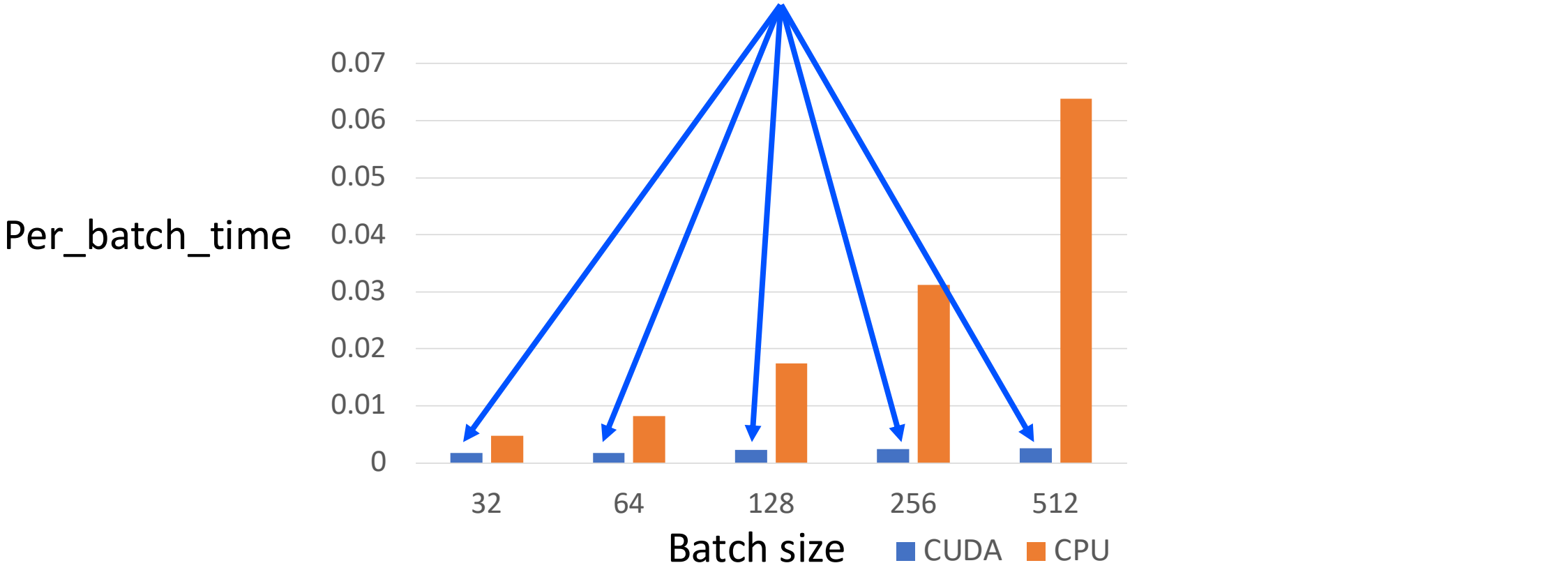
Recall: GPU



The prediction for the 10 samples are computed in parallel!

Recall: Effect of batch size

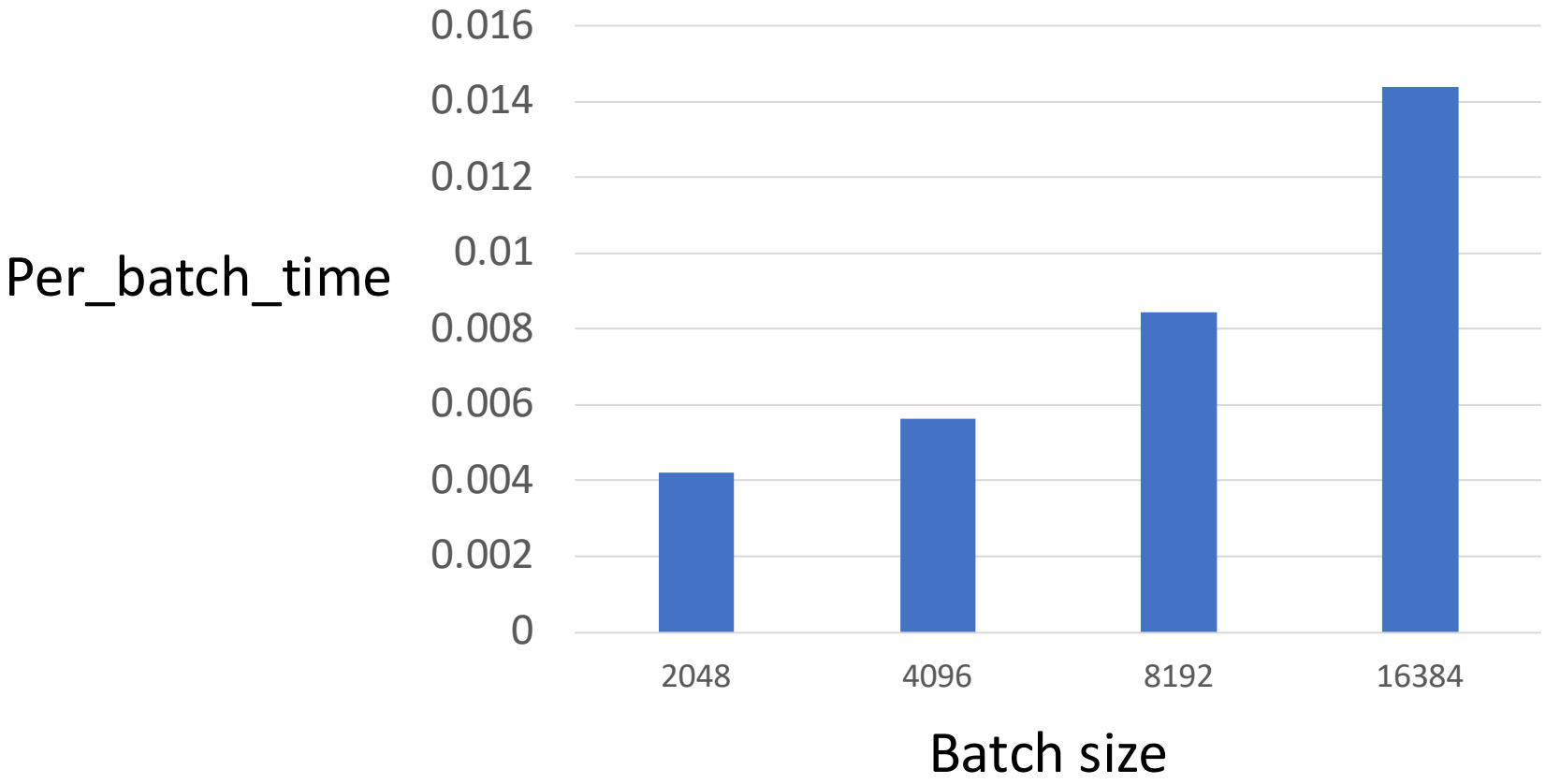
GPU computing time almost does not increase with batch_size thanks to parallelization!



Test environment: Google CoLab

Recall: Effect of batch size

When the batch_size is too large, GPU per batch time DOES INCREASE with batch size!



Test environment: Google CoLab

Up next: distributed training

Big Data

- NSL KDD has about 125k data points, whereas each data has a 113-dim feature
- Our model has roughly 4 hidden layers

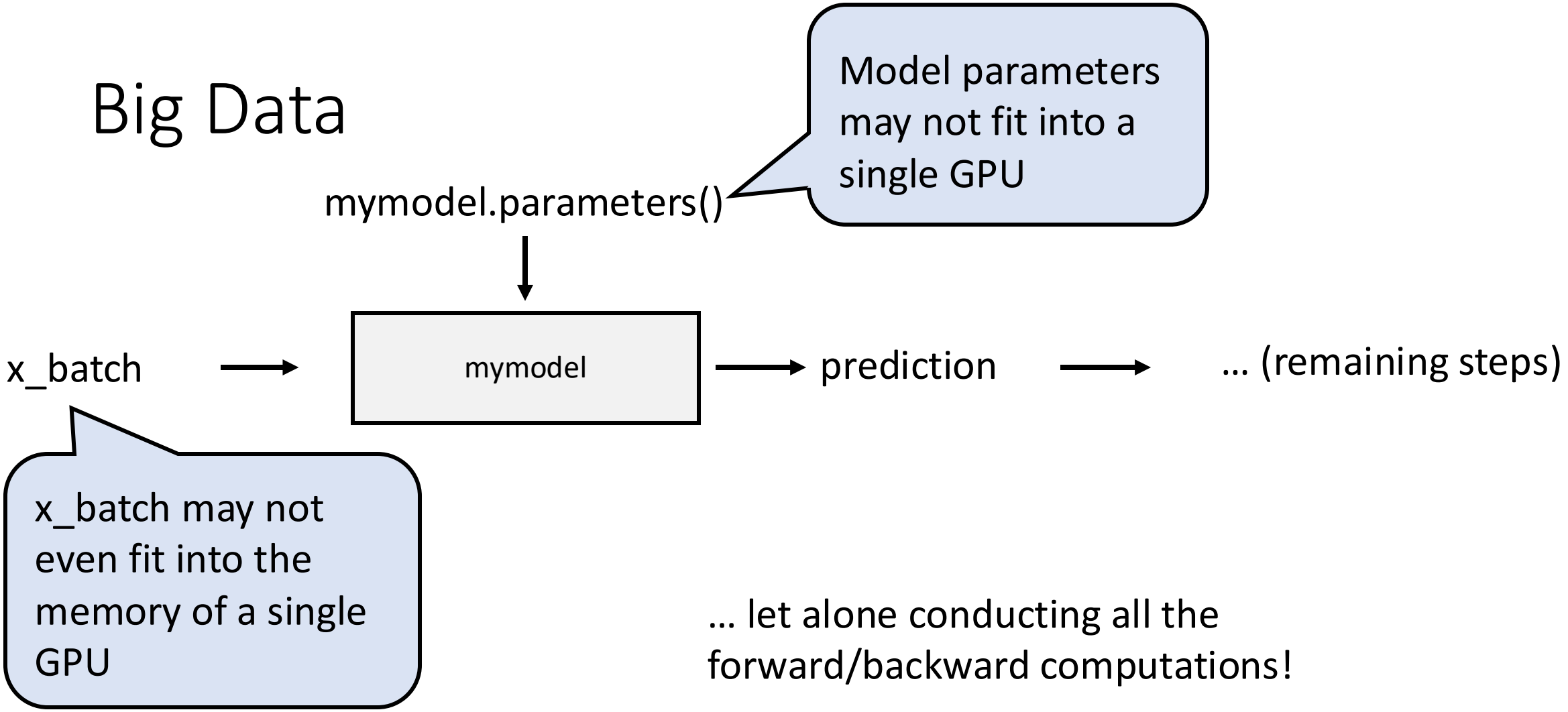
In today's era, both the data and the model size can be huge.

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Sizes, architectures, and learning hyper-parameters of the GPT-3 models

Dataset	**Quantity (tokens)**
Common Crawl (filtered)	410 billion
WebText2	19 billion
Books1	12 billion
Books2	55 billion
Wikipedia	3 billion

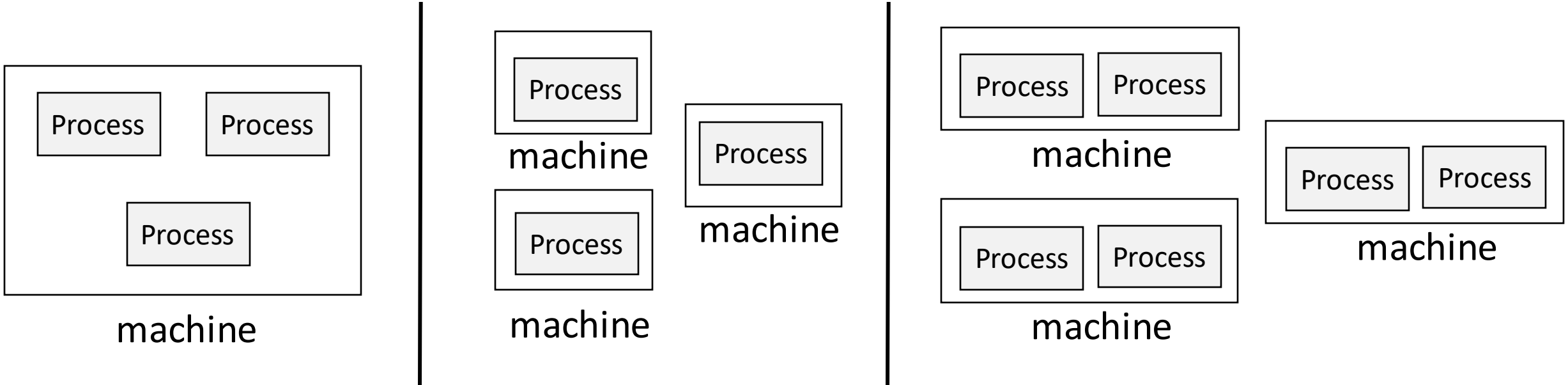
Big Data



… let alone conducting all the forward/backward computations!

Parallelism

- Parallelism means distribute the computation across different “nodes”.
- Each node is a process that runs in parallel, either on the same machine or across different machine, or a mix of both



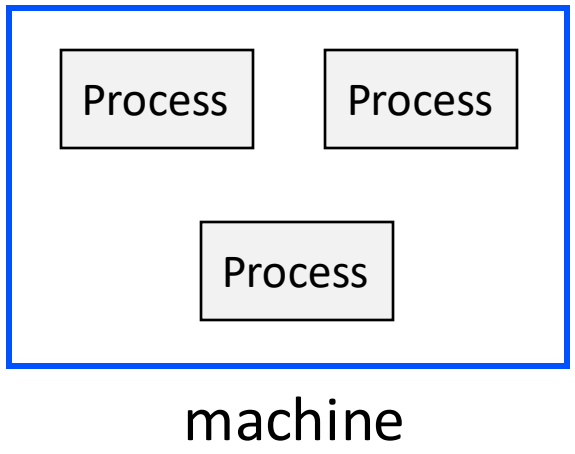
Parallelism

In the context of deep learning, each “process” is run on a GPU

Common in lab setting

e.g. A workstation with 8 GPUs

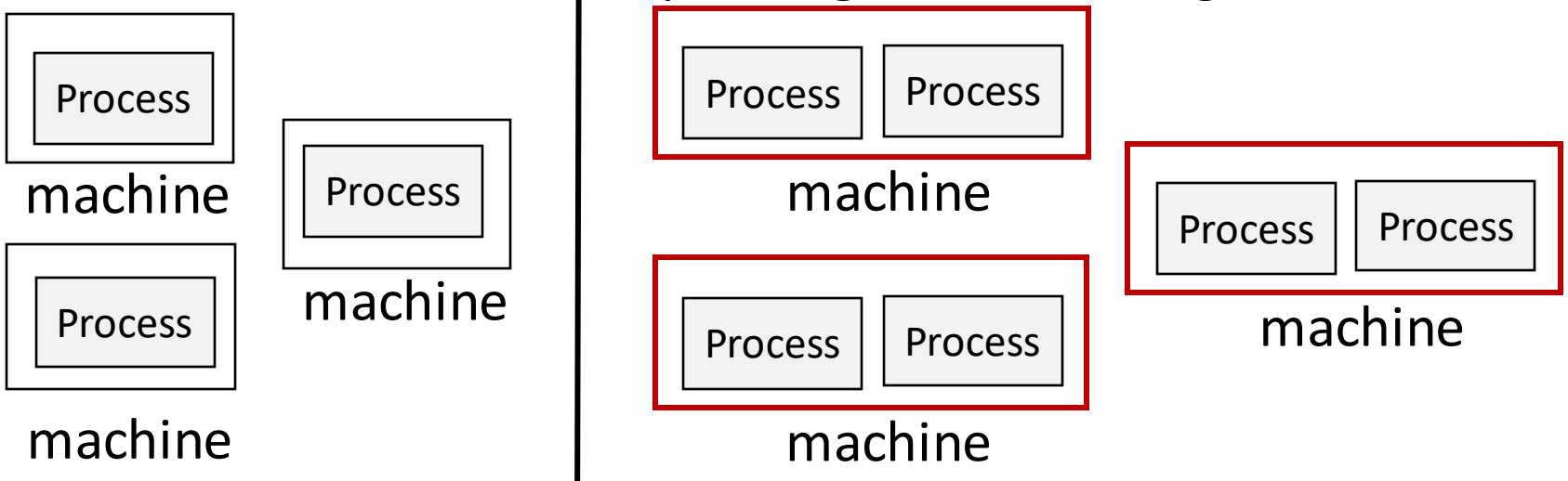
- Communication between GPUs are fast



Common in cluster setting

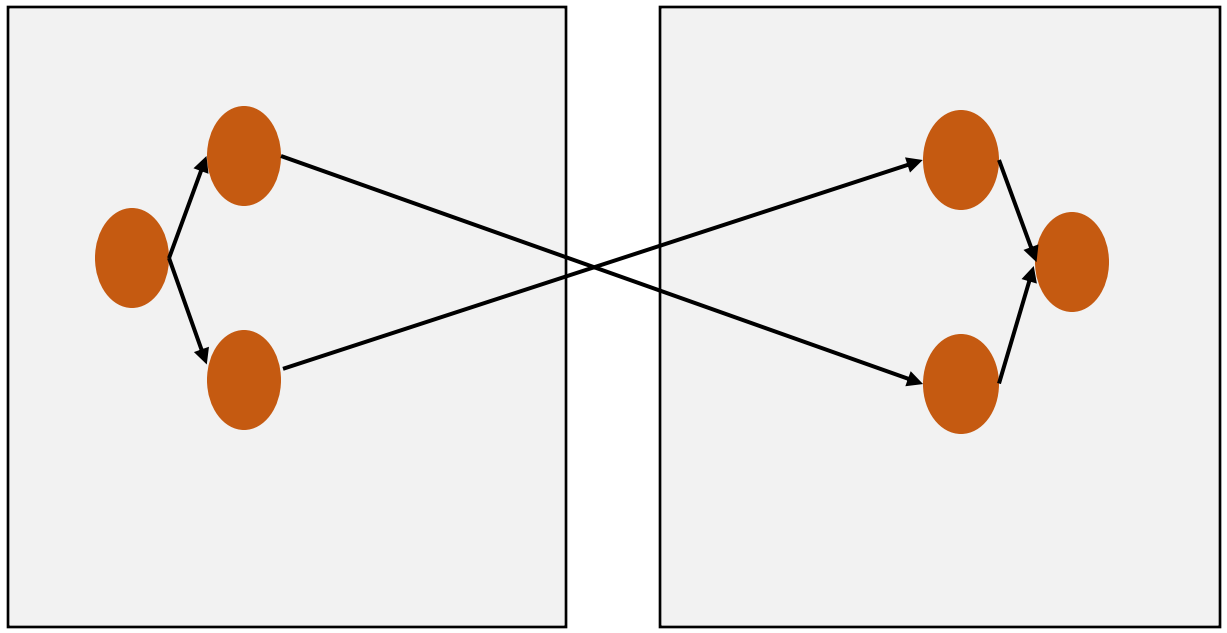
e.g. A company has a cluster with many machines each with multiple GPUs

- Communication among machines depending on networking



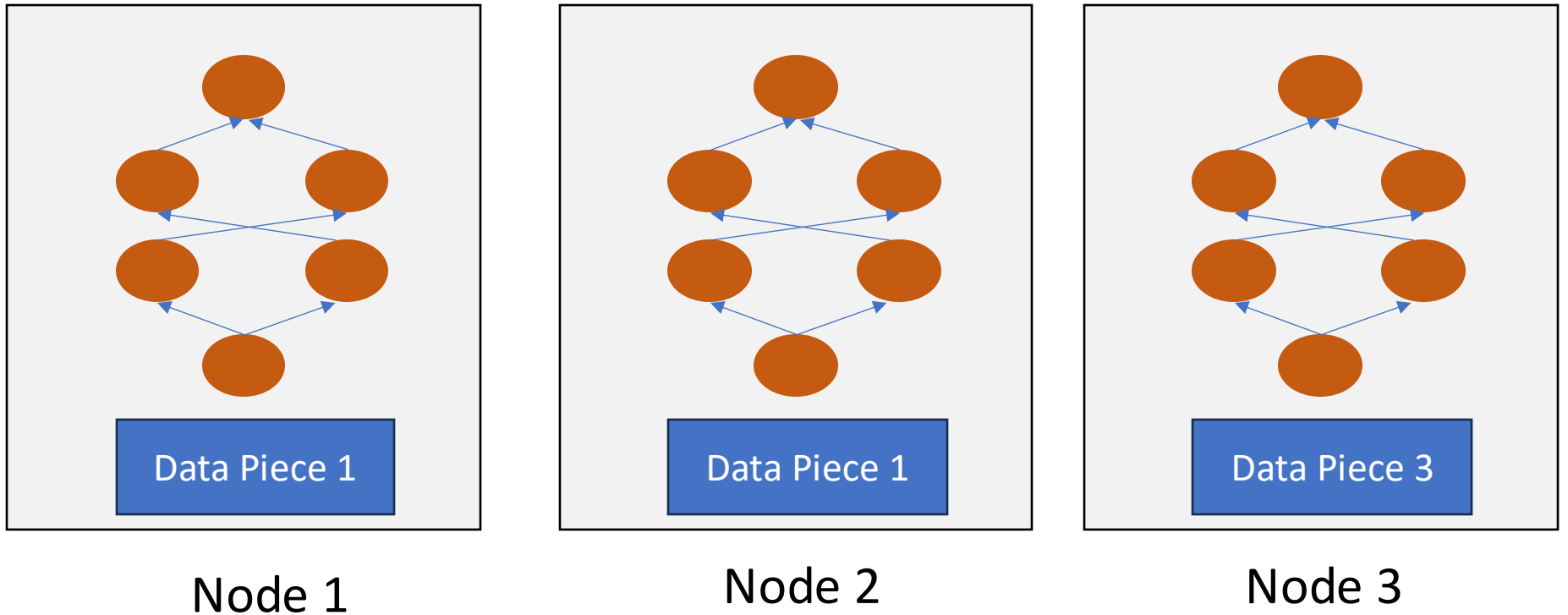
Parallelism

- **Model Parallelism**: different nodes are responsible for the computations in different parts of a single neural network - for example, each layer in the neural network may be assigned to a different node.



Parallelism

- **Data Parallelism (Today's focus)**: Different processes have a complete copy of the model; each process simply gets a different portion of the data, and results from each are somehow combined.



Revisit Loss Function

Prediction of current model on $x(i)$

$$loss(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left(prediction(x(i)) - y(i)\right)^2$$

- Suppose the data is divided into K pieces, each of size M with $N = K * M$
- Each node has access to one piece.

$\underbrace{1,2,\dots,M}_{\text{Piece 1}}$ $\underbrace{M+1, M+2, \dots, 2M}_{\text{Piece 2}}$ … $\underbrace{(K-1)M+1, \dots, KM}_{\text{Piece K}}$

$$loss(\theta) = \frac{1}{K}\sum_{j=1}^{K}\frac{1}{M}\sum_{i \in Piece_j}\left(prediction\left(x(i)\right) - y(i)\right)^2 = \frac{1}{K}\sum_{j=1}^{K} loss_j(\theta)$$

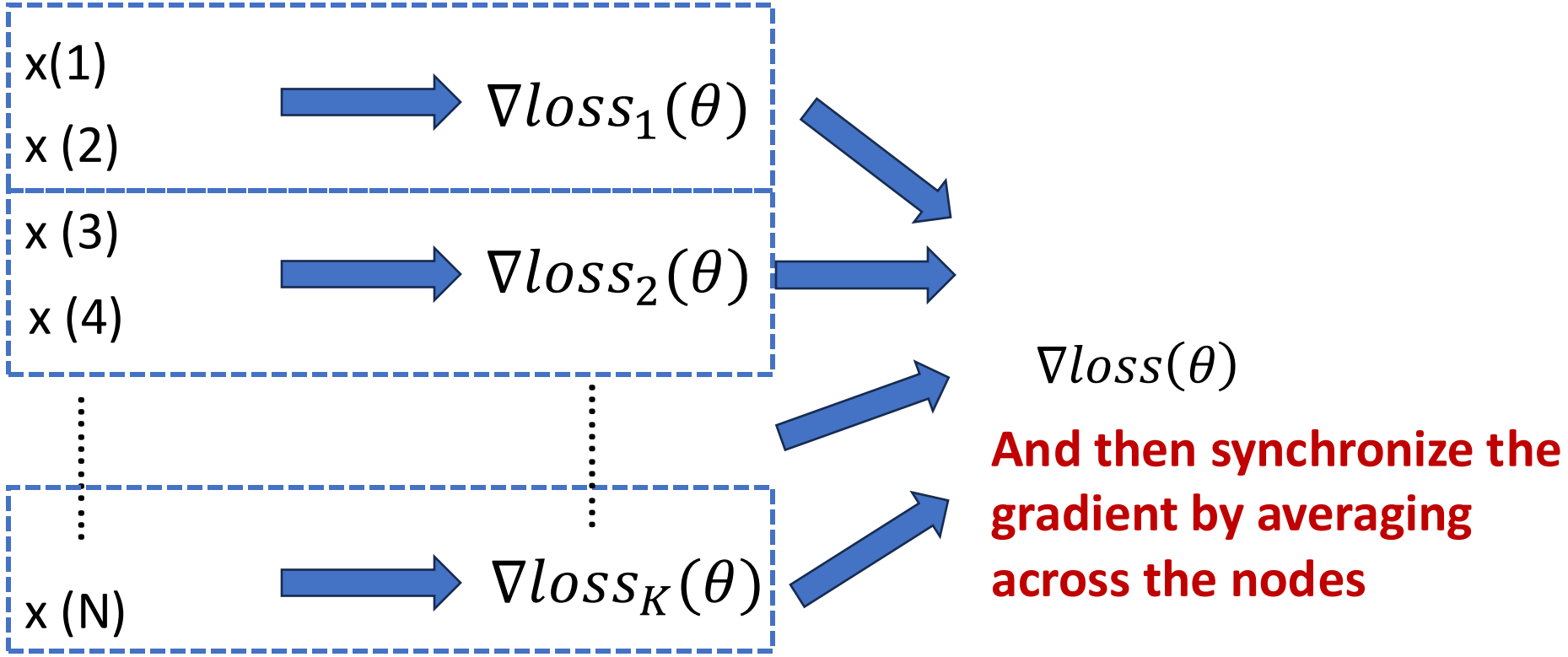
Revisit Loss Function

The gradient of the loss using the whole dataset equals the average of the gradients using the smaller pieces

$$\nabla loss(\theta) = \frac{1}{K}\sum_{j=1}^{K} \nabla loss_j(\theta)$$

Revisit Loss Function

Each node computes its gradient using its respective piece of data



And then synchronize the gradient by averaging across the nodes

$$\nabla loss(\theta) = \frac{1}{K}\sum_{j=1}^{K} \nabla loss_j(\theta)$$

Distributed Data Parallel

torch.nn.parallel.DistributedDataParallel is the Pytorch implementation of Data Parallelism

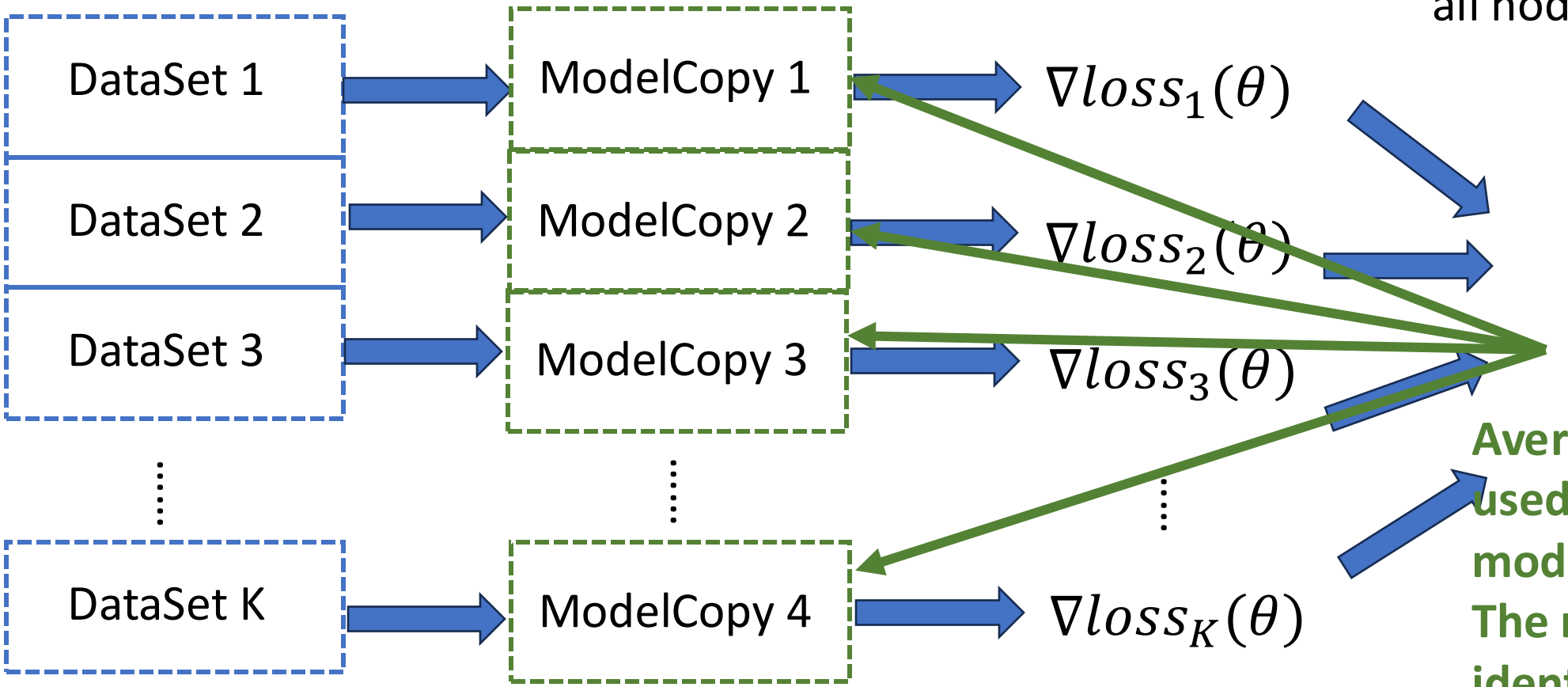
- Concepture idea: each node uses its own data to conduct forward-backward
- BUT synchronizes and calculates the AVERAGE gradient before conducting gradient descent!

Different nodes store **DIFFERENT** datasets

Different nodes store **IDENTICAL** COPY of model

Gradient computed by different nodes are **DIFFERENT**

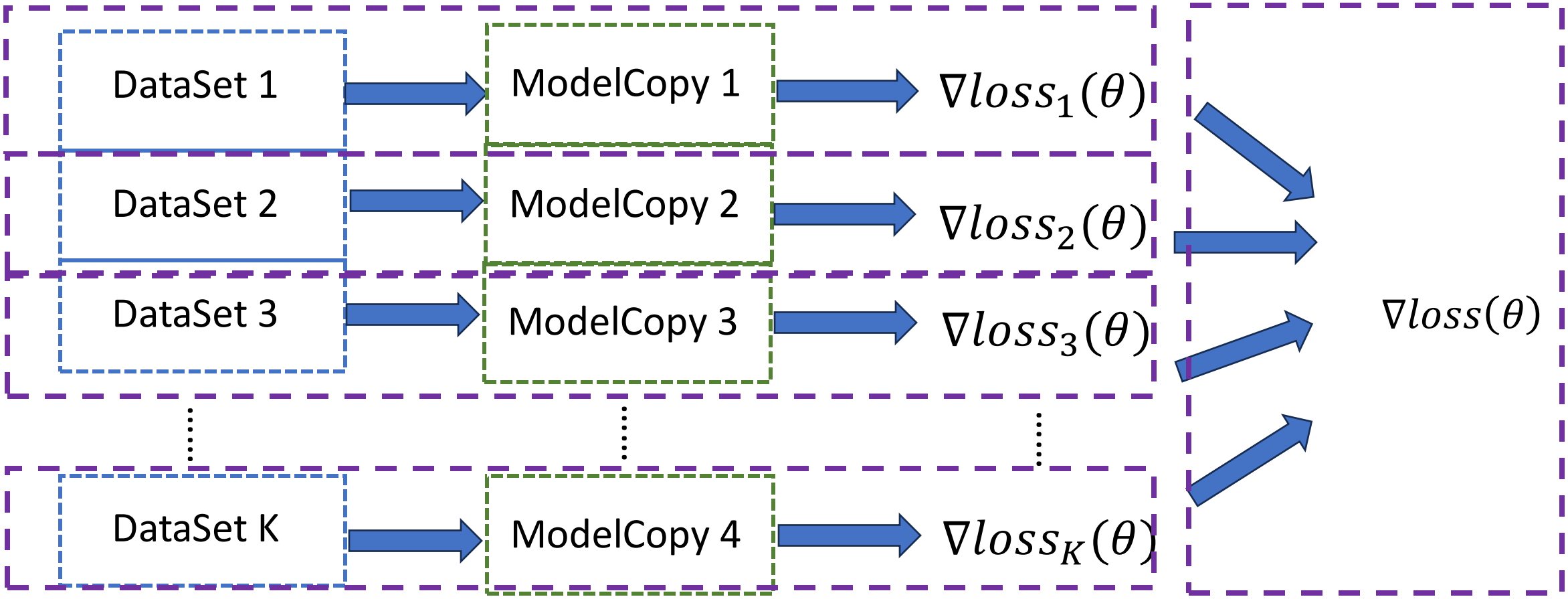
Different nodes communicate to compute average gradient, **IDENTICAL** for all nodes



Average gradient is used to update all models.
The models remain identical across nodes!

Programming Model:
Each process loads its own data, conducts the forward/backward as usual.

PyTorch's DistributedDataParallel Handles the communication between processes and calculation of average gradient



Outline of Programming Model

We are going to code a program that will be run by multiple nodes in parallel

- Initialize Process Group, where we need to tell PyTorch the following:
  - How many nodes are there in total (world_size)
  - What is the ID of this the current node, also known as “node rank”, an integer value starting from 0
  - How to communicate with the lead node (node with rank 0)
- Prepare data: this node only has access to a piece of data
- Prepare model: create model with DistributedDataParallel
- Training loops

Initialize Process Group

```
def train(args):
    # Setup the communication with other processes
    rank = args.nr
    print(f"hello! Node # {rank} is being initialized! Awaiting all nodes to join. ")
    os.environ['MASTER_ADDR'] = args.master_addr
    os.environ['MASTER_PORT'] = args.master_port
    dist.init_process_group(
        backend='gloo',
        init_method='env://',
        world_size=args.world_size,
        rank=rank
    )
    print(f"hello! Node # {rank} is running!")
```

Getting what is the rank of this node

Setting the address and port of the node with rank 0#

Initialize torch.distributed

- Backend means the communication mechanism between nodes. Use “gloo” for CPU training, “nccl” for GPU training
- init_method: where to find the configuration (address and port). “env://” means it will find the set up in OS environment variables
- World_size: how many nodes in total
- Rank: rank of this current node running this program

Prepare Training Data

```
# Loading a slice of the dataset
train_dataset = torchvision.datasets.MNIST(
    root='./data',
    train=True,
    transform=transforms.ToTensor(),
    download=True
)
train_sampler = torch.utils.data.distributed.DistributedSampler(
    train_dataset,
    num_replicas=args.world_size,
    rank=rank
)
```

Using the sampler to sample only a piece of the dataset

Prepare Model

```
# Create model and WRAP IT WITH DistributedDataParallel
model = ConvNet()
model.to(device)
model = nn.parallel.DistributedDataParallel(model)
```

Wrap the model with DistributedDataParallel

```
# Setup the loss, optimizer, loader
batch_size = 100
lr = 1e-4
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr)
train_loader = torch.utils.data.DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
    sampler=train_sampler)
```

Using the sampler to sample only a piece of the data

Training Loop

The training loop coding is as before

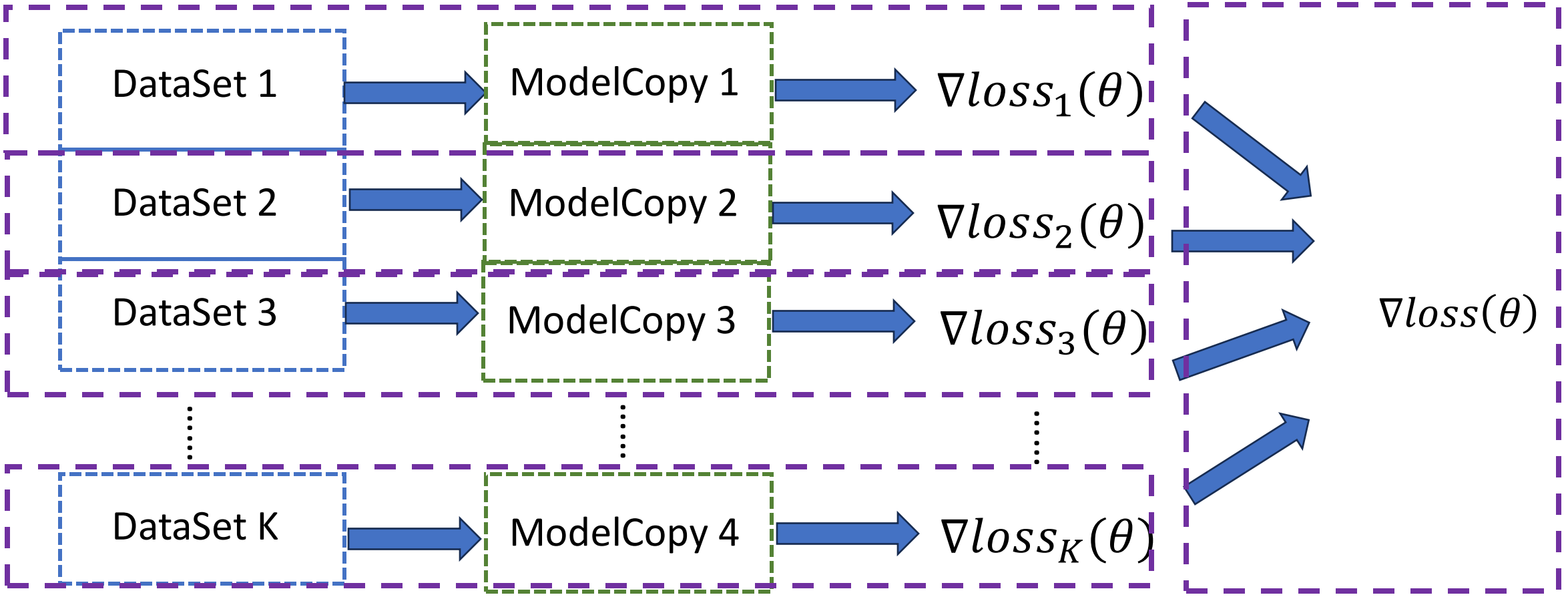
```
for epoch in range(args.epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)
        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

The forward computes the loss only using the piece of dataset of this node

However, PyTorch will automatically synchronizes with other nodes and calculate the average gradient

Programming Model:
Each process loads its own data, conducts the forward/backward/step as usual.

PyTorch's DistributedDataParallel Handles the communication between processes and calculation of average gradient



Summary So Far

We have written a train function that contains the following steps

def train(args)

- Initialize Process Group,
- Prepare data: this node only has access to a piece of data
- Prepare model: create model with DistributedDataParallel
- Training loops

We also need a main function that will configure the args properly for the train function

Main Function

```
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('-w', '--world_size', default=1,
                        type=int)
    parser.add_argument('-g', '--gpu', default=-1, type=int,
                        help='cuda device ID. -1 represents cpu')
    parser.add_argument('-nr', '--nr', default=0, type=int,
                        help='ranking within the nodes')
    parser.add_argument('--epochs', default=2, type=int,
                        help='number of total epochs to run')
    parser.add_argument('--master_addr', default="127.0.0.1", type=str,
                        help='address of master node (node with rank 0)')
    parser.add_argument('--master_port', default="8888", type=str,
                        help='port number of master node (node with rank 0)')

    args = parser.parse_args()
    print(args)
    train(args)
```

Parsing argument from user input

Run the train function with the args obtained

Simple Experiments

Open the terminal and run the following command :

```
(sparktest2) coolq@CoolQs-Air notebooks % python Lecture_16_distributed_data_parallel.py -w 2 -nr 0
--master_addr "127.0.0.1" --master_port 8888
```

It will show:

```
hello! Node # 0 is being initialized! Awaiting all nodes to join.
```

Open another terminal and run a similar command (nr changed from 0 to 1) :

```
(sparktest2) coolq@CoolQs-Air notebooks % python Lecture_16_distributed_data_parallel.py -w 2 -nr 1
--master_addr "127.0.0.1" --master_port 8888
```

Then both processes will start training!

Simple Experiments

notebooks — -zsh — 60×27

```
(sparktest2) coolq@CoolQs-Air notebooks % python Lecture_16_
distributed_data_parallel.py -w 2 -nr 0 --master_addr "127.0
.0.1" --master_port 8888
Namespace(world_size=2, gpu=-1, nr=0, epochs=2, master_addr=
'127.0.0.1', master_port='8888')
hello! Node # 0 is being initialized! Awaiting all nodes to
join.
hello! Node # 0 is running!
Epoch [1/2], Step [100/300], Loss: 2.2323
Epoch [1/2], Step [200/300], Loss: 2.1133
Epoch [1/2], Step [300/300], Loss: 2.0026
Epoch [2/2], Step [100/300], Loss: 1.8508
Epoch [2/2], Step [200/300], Loss: 1.7533
Epoch [2/2], Step [300/300], Loss: 1.7036
(sparktest2) coolq@CoolQs-Air notebooks %
```

notebooks — -zsh — 60×26

```
(sparktest2) coolq@CoolQs-Air notebooks % python Lecture_16_
distributed_data_parallel.py -w 2 -nr 1 --master_addr "127.0
.0.1" --master_port 8888
Namespace(world_size=2, gpu=-1, nr=1, epochs=2, master_addr=
'127.0.0.1', master_port='8888')
hello! Node # 1 is being initialized! Awaiting all nodes to
join.
hello! Node # 1 is running!
Epoch [1/2], Step [100/300], Loss: 2.2321
Epoch [1/2], Step [200/300], Loss: 2.1098
Epoch [1/2], Step [300/300], Loss: 1.9686
Epoch [2/2], Step [100/300], Loss: 1.8606
Epoch [2/2], Step [200/300], Loss: 1.7610
Epoch [2/2], Step [300/300], Loss: 1.6502
(sparktest2) coolq@CoolQs-Air notebooks %
```

Use “Accelerate” package

accelerate is a package that helps you conduct distributed training more conveniently

- It is a wrapper of the torch DistributedDataParallel API
- You just write training loop as if you were doing centralized training
- with **4 more lines of code + some config**, accelerate will conduct the training distributedly

Accelerate is versatile. With minimal change in code, it supports various distributed setup:

- Single machine multiple-GPU
- Multiple machine multiple-GPU
- Integrates with other tools, e.g. DeepSpeed
- Supports running on AWS

Use “Accelerate” package

```
# Loading dataset
train_dataset = torchvision.datasets.MNIST(
    root='./data',
    train=True,
    transform=transforms.ToTensor(),
    download=True
)
# Create model
model = ConvNet()
# Setup the loss, optimizer, loader
epochs = 2
batch_size = 100
lr = 1e-4
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr)
train_loader = torch.utils.data.DataLoader(
    dataset=train_dataset,
   batch_size=batch_size)
total_step = len(train_loader)
```

Use “Accelerate” package

```
## use Accelerate to prepare distributed training
accelerator = Accelerator()
model, optimizer, train_loader, _ = accelerator.prepare(
    model, optimizer, train_loader, None)

for epoch in range(epochs):
    for i, (images, labels) in enumerate(train_loader):
        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)
        # Backward and optimize
        optimizer.zero_grad()
        # loss.backward() # Do not use loss.backward
        accelerator.backward(loss) # use accelerator to conduct backward
        optimizer.step()
```

Only 4 lines of code are changed!

Lab: use “accelerate” on two GPUs

run nvidia-smi to verify that we have two GPUs with ID as 0,1

```
(toolchain_finetuning) [gqu@node-gpu02 toolchain]$ nvidia-smi
Sun Sep 22 11:00:14 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.129.03             Driver Version: 535.129.03   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 H100 80GB HBM3          On  | 00000000:43:00.0 Off |                    0 |
| N/A   22C    P0              75W / 700W |      4MiB / 81559MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+
|   1  NVIDIA H100 80GB HBM3          On  | 00000000:61:00.0 Off |                    0 |
| N/A   23C    P0              75W / 700W |      4MiB / 81559MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

Lab: use “accelerate” on two GPUs

run “accelerate config” to setup the distributed setting

```
(toolchain_finetuning) [gqu@node-gpu02 toolchain]$ accelerate config
In which compute environment are you running?
This machine
Which type of machine are you using?
multi-GPU
How many different machines will you use (use more than 1 for multi-node training)? [1]: 1
Should distributed operations be checked while running for errors? This can avoid timeout issues but will be slower. [yes/NO]: NO
Do you wish to optimize your script with torch dynamo?[yes/NO]:NO
Do you want to use DeepSpeed? [yes/NO]: NO
Do you want to use FullyShardedDataParallel? [yes/NO]: NO
Do you want to use Megatron-LM ? [yes/NO]: NO
How many GPU(s) should be used for distributed training? [1]:2
What GPU(s) (by id) should be used for training on this machine as a comma-seperated list? [all]:0,1
Would you like to enable numa efficiency? (Currently only supported on NVIDIA hardware). [yes/NO]: NO
Do you wish to use mixed precision?
no
accelerate configuration saved at /home/gqu/.cache/huggingface/accelerate/default_config.yaml
```

Lab: use “accelerate” on two GPUs

After “accelerate config”, it will generate a config file

.cache > huggingface > accelerate > ! default_config.yaml

```yaml
compute_environment: LOCAL_MACHINE
debug: false
distributed_type: MULTI_GPU
downcast_bf16: 'no'
enable_cpu_affinity: false
gpu_ids: 0,1
machine_rank: 0
main_training_function: main
mixed_precision: 'no'
num_machines: 1
num_processes: 2
rdzv_backend: static
same_network: true
tpu_env: []
tpu_use_cluster: false
tpu_use_sudo: false
use_cpu: false
```

Lab: use “accelerate” on two GPUs

Lastly, run the python program. It will conduct training with the two GPUs

```
(toolchain_finetuning) [gqu@node-gpu02 toolchain]$ python lecture_16_distributed_data_parallel_with_accelerate.py
Epoch [1/2], Step [100/600], Loss: 2.1088
Epoch [1/2], Step [200/600], Loss: 2.0445
Epoch [1/2], Step [300/600], Loss: 1.9576
Epoch [1/2], Step [400/600], Loss: 1.9137
Epoch [1/2], Step [500/600], Loss: 1.7377
Epoch [1/2], Step [600/600], Loss: 1.6294
Epoch [2/2], Step [100/600], Loss: 1.3699
Epoch [2/2], Step [200/600], Loss: 1.4837
Epoch [2/2], Step [300/600], Loss: 1.4334
Epoch [2/2], Step [400/600], Loss: 1.4011
Epoch [2/2], Step [500/600], Loss: 1.2867
Epoch [2/2], Step [600/600], Loss: 1.2219
```

Additional Tools

- Accelerate is a wrapper of other distributed tools that allows you to conduct distributed training across various tools/setups without changing the code.
  - By default it will use the torch native DistributedDataParallel that conducts Data Parallelism, NOT Model Parallelism
- DeepSpeed (Microsoft) is another tool that supports model parallelism
  - DeepSpeed is integrated into accelerate
  - You can use DeepSpeed in accelerate by changing some config
- Other tools: Megatron-LM (Nvidia), FullyShardedDataParallel (torch)

Summary

- Today we have covered
  - Theory of Data Parallelism
  - Torch DistributedDataParallel API
  - “accelerate“ to make distributed training more convenient