Tutorial: Introduction to PyTorch and DDP

A beginner-friendly introduction to PyTorch, one of the most popular libraries for deep learning, and an overview of how to scale training with Distributed Data Parallel (DDP).

Core Concepts

PyTorch & Training

- **Tensor**: multi-dimensional array (like NumPy) that can live on **CPU/GPU**.
- **Model / Layer**: functions with **learnable weights**.
- **Forward pass**: compute predictions from inputs.
- **Loss**: measures how wrong the predictions are.
- **Gradient (autograd)**: “how much & in which direction” each weight should change to reduce the loss (computed **automatically** by PyTorch).
- **Optimizer**: updates weights using the gradients (e.g., **SGD**, **Adam**).
- **Batch / Mini-batch**: small set of samples processed together.
- **Epoch**: one full pass over the training dataset.

Core Concepts

Distributed / DDP

- **Process / Rank**: one **OS process** per GPU; its global id is the **rank**.
- **Local rank**: the GPU index inside the current node.
- **World size**: total number of processes (usually = total GPUs).
- **Backend**: comms engine (**NCCL** for GPUs, **Gloo** for CPU, **MPI** on HPC).
- **DistributedSampler**: shards the dataset so each rank sees a **different slice**.
- **All-reduce (gradient sync)**: averages gradients across all ranks → **identical weights** everywhere.
- **DDP = synchronous data parallel**: all ranks step **together** each batch.
- **Throughput (img/s)**: how many samples per second we process (our demo metric).
- **Rendezvous (MASTER_ADDR:PORT)**: how processes find each other when launched with **torchrun**.
- **Node rank**: the id of each machine in multi-node runs.

Why not just use your laptop/pc?

Training deep learning models on a personal laptop seems convenient at first.

But as your datasets grow and your models become more complex, the limitations quickly appear:

- **Not enough memory (RAM/VRAM)**
- **Slow CPU or GPU**
- **Long training times**
- **Overheating and hardware strain**

For realistic or large-scale projects, we need more compute power than a laptop or pc can offer.

It's not just raw speed — it's **time-to-feedback**. Faster feedback leads to better next iterations/epochs.

This is where distributed training comes in.

The Problem with Local Training

As your machine learning projects grow, you start working with larger datasets and more complex models.
Training locally on a single CPU or GPU becomes a bottleneck and leads to several issues:

- **Extremely slow training time**
- **Out-of-memory (OOM) errors**
- **Limited batch size**
- **Difficulty experimenting quickly**
- **Inefficient use of modern hardware resources**

These issues make it hard to iterate, experiment, or scale your work efficiently.

Takeaway: scaling out = faster feedback, not just raw speed.

What is Distributed Training?

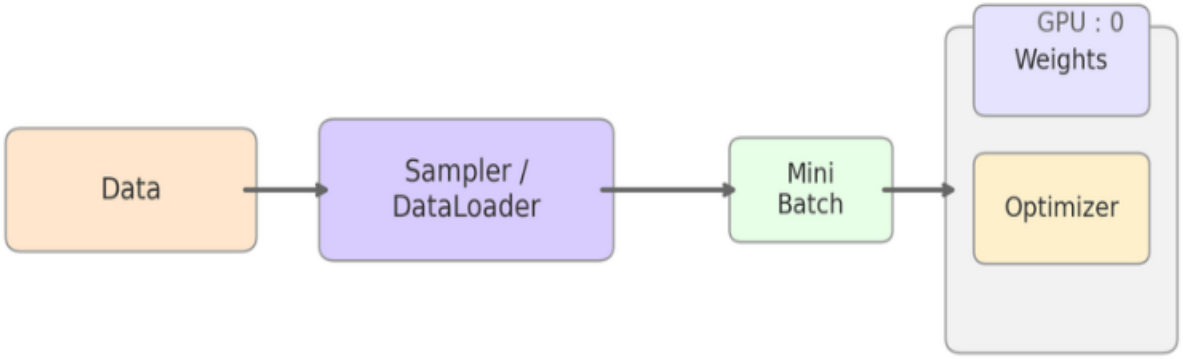
Distributed training is a method used to speed up model training by spreading the workload across multiple devices.
Instead of using just one GPU or machine, you use several working together.

Key ideas:

- **Replicate the same model** on each GPU (or node)
- **Shard the dataset** (DistributedSampler) — each GPU sees a different slice
- After each batch, **synchronize gradients** (NCCL all-reduce)
- **Training time is significantly reduced**

This approach is essential when working with large-scale models and data.

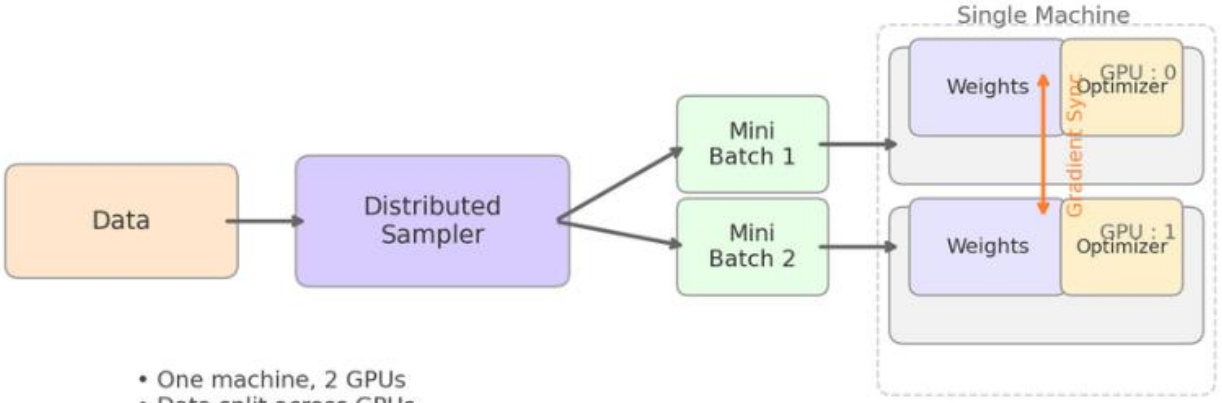
Non-Distributed (Single GPU)



- One device
- Sequential batches
- No gradient sync required

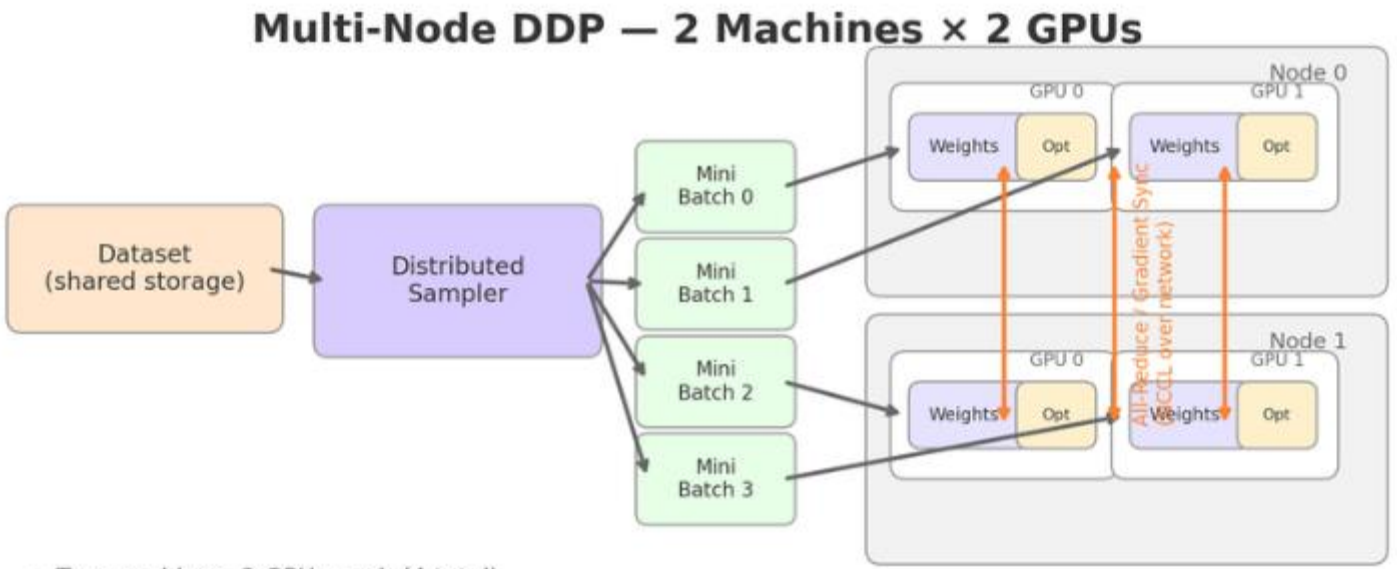
Single-Machine DDP (2 GPUs)

Single-Machine — DDP with 2 GPUs



- One machine, 2 GPUs
- Data split across GPUs
- Gradients synchronized via NCCL (DDP)

Multi-Node DDP (2 machines × 2 GPUs)



How processes talk to each other in DDP

- **NCCL** — GPU-optimized, best choice for NVIDIA multi-GPU / multi-node
- **Gloo** — CPU-first (simple setup; fine for CPU training or quick tests)
- **MPI** — integrates with existing MPI stacks on HPC clusters
- **MPS** — Apple Silicon GPUs (M1/M2+) It is not backend, its a device.

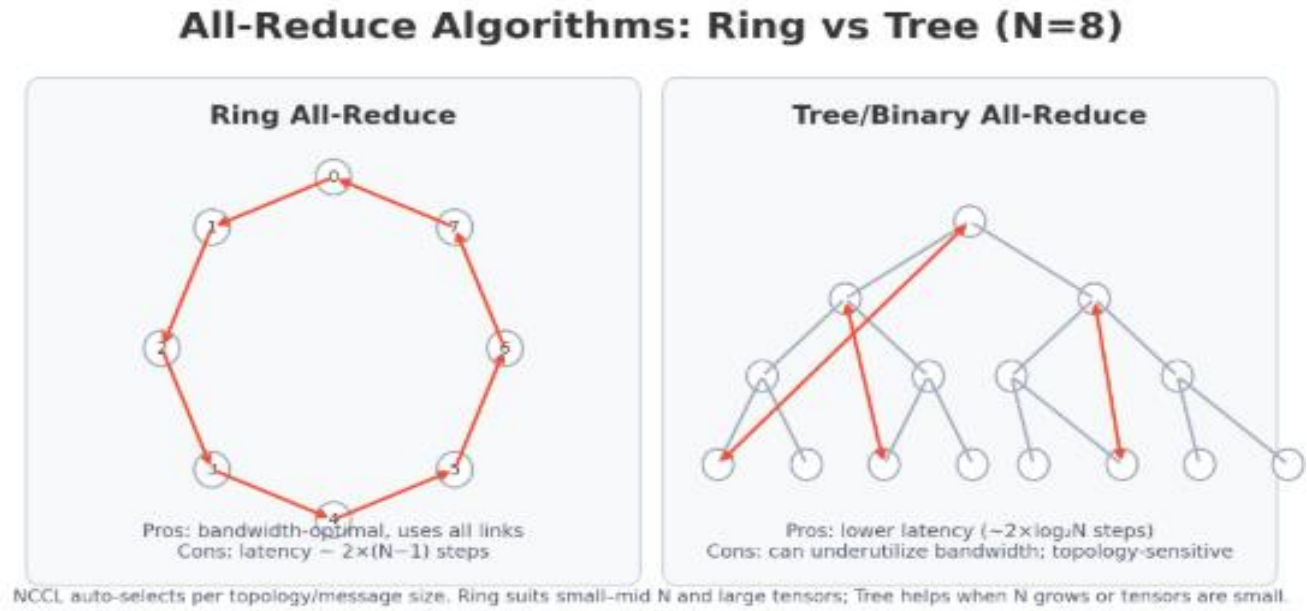
How to pick:

- Training on **NVIDIA GPUs** → use **NCCL**
- **CPU-only** or quick local tests → use **Gloo**
- On an **MPI-managed HPC** cluster → use **MPI**
- On **Mac (M1/M2)** → use device **`"mps"`**

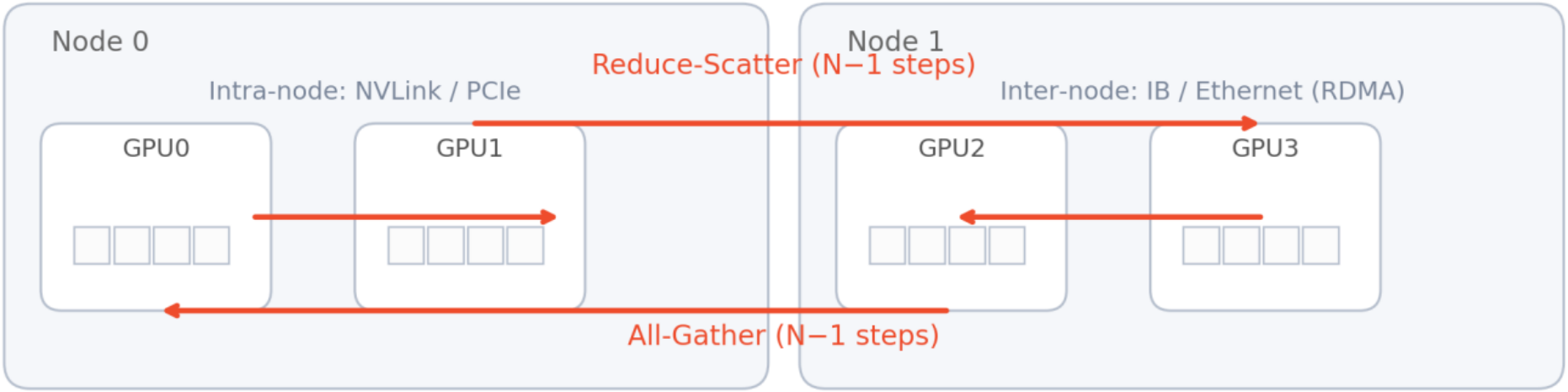
```
import torch, torch.distributed as dist
device = ("cuda" if torch.cuda.is_available()
          else "mps" if torch.backends.mps.is_available()
          else "cpu")
dist.init_process_group(backend="nccl")  # ή "gloo", "mpi"
```

Why Nccl Ring? (vs Tree/Binary)

- **Ring** = neighbors exchange chunks in a loop.**Reduce-Scatter** then **All-Gather**. *Bandwidth-optimal*, uses all links but Latency ~ **2×(N−1)** steps, worse while N (GPUs) increase.
- **Tree/Binary:** lower latency ≈ **$2\times\log_2 N$** steps. Better for **small tensors** or **large N**; may **underutilize bandwidth**



NCCL Ring All-Reduce (PyTorch DDP)



- Each GPU holds a different chunk of the gradient tensor.
- Reduce-Scatter: GPUs rotate chunks around the ring, summing as they go.
- All-Gather: the reduced chunks are broadcast so every GPU gets the full sum.
- Total cost ~ 2 × (N−1) steps; ring maximizes link bandwidth.

What is PyTorch?

PyTorch is an open-source machine learning framework developed by Meta (formerly Facebook).
It is widely used in both academia and industry for building deep learning models.

Why PyTorch?

- **Pythonic and intuitive — feels like working with NumPy**
- **Dynamic computation graphs (eager execution)**
- **Built-in support for GPUs and automatic differentiation**
- **Large and active community with rich ecosystem**
- **Ideal for research and production**

PyTorch gives you full control and flexibility when designing, training, and deploying AI models.

Installing and Using PyTorch

You can explore and install PyTorch from the official website:
https://pytorch.org

Once installed, you can:

- **Create tensors (multi-dimensional arrays)**
- **Perform GPU-accelerated operations**
- **Build and train neural networks**
- **Use built-in tools for autograd (automatic differentiation)**

PyTorch allows you to prototype quickly and scale when needed — all using standard Python.

A Simple Code Example

- With NumPy(CPU only):

```python
import numpy as np

x = np.array([1.0, 2.0])
y = x * 2
print(y)
```

With PyTorch(CPU or GPU):

```python
import torch

x = torch.tensor([1.0, 2.0], device='cuda')  # or 'cpu'
y = x * 2
print(y)
```

Key differences:

- **PyTorch supports GPUs (`device='cuda'`)**
- **You can switch between CPU and GPU easily**
- **PyTorch integrates seamlessly with deep learning models**
- **This flexibility is why PyTorch is so powerful for AI workloads**

A more complicated example

Plain NumPy (manual gradients):

```python
import numpy as np  # Linear regression y ≈ w*x + b

x = np.array([1., 2., 3., 4.])
y = np.array([2., 4., 6., 8.])

w, b, lr = 0.0, 0.0, 0.1

for _ in range(100):
    yhat = w * x + b
    loss = ((yhat - y) ** 2).mean()
    # manual gradients (dMSE/dw, dMSE/db)
    dw = 2 * ((yhat - y) * x).mean()
    db = 2 * (yhat - y).mean()
    w -= lr * dw
    b -= lr * db
```

PyTorch (autograd + optimizer):

```python
import torch  # Linear regression y ≈ w*x + b

x = torch.tensor([1., 2., 3., 4.])
y = torch.tensor([2., 4., 6., 8.])

w = torch.randn(1, requires_grad=True)
b = torch.zeros(1, requires_grad=True)
opt = torch.optim.SGD([w, b], lr=0.1)

for _ in range(100):
    opt.zero_grad()
    yhat = w * x + b
    loss = ((yhat - y) ** 2).mean()
    loss.backward()    # autograd builds the graph & applies the chain rule
    opt.step()         # updates w and b using their .grad
```

What is DDP (Distributed Data Parallel)?

DDP (DistributedDataParallel) is a module in PyTorch that allows you to train your model across multiple GPUs or machines efficiently.

How it works:

- **The model is replicated on each GPU**
- **The dataset is split among the GPUs**
- **Each GPU does forward and backward pass independently**
- **Gradients are synchronized automatically after every batch**
- **All model copies are kept in sync**

DDP helps you scale training with **minimal code changes**, offering both performance and stability.

How DDP Works

DistributedDataParallel works by distributing the training process across multiple GPUs (or nodes), while keeping all model replicas in sync.

At each training step:

1) Each GPU gets a **copy of the model**
2) A **subset of the dataset** is sent to each GPU
3) Each GPU performs:
    - **Forward pass(**compute predictions from inputs.**)**
    - **Loss calculation(**measures how wrong the predictions are.**)**
    - **Backward pass (gradient computation)**
4) **Gradients are synchronized** across all GPUs (all-reduce (NCCL))
5) All models are **updated consistently**

This process repeats after every batch, ensuring all devices work together as one.

When Should You Use DDP?

DDP is most useful when your model or dataset is **too large** or **too slow** to train on a single machine.

Use DDP when:

- You're working with **large-scale datasets** (e.g. images, video, audio)
- Your model is **complex and deep** (e.g. transformers, CNNs)
- Training on one GPU takes **too long**
- You're using **multiple GPUs** or a **cluster/cloud environment**
- You need to **reduce training time significantly**

With DDP, training time can go from hours to minutes depending on the hardware.

What DDP is NOT for

While DDP is powerful for distributed training, it is not designed for other types of parallel processing.

DDP is NOT meant for:

- Model inference (use TorchServe or ONNX instead)
- Serving models in production
- Parallel data processing (use Spark, Dask, or Ray)
- General-purpose multiprocessing

DDP is specialized for **training** deep learning models, not for deploying or running them after training.

Demo setup

- **Dataset:** Coco 118,287 images
- **Model:** ResNet-18 ~11.7M params (pretrained), I only change the FC → 80 classes
- **Loss / Optimizer:** CrossEntropyLoss/ Adam
- 2 machines with 2 x NVIDIA GeForce RTX 2080 Ti (11264MiB) each
- **Task (for simplicity):** detection → single-label classification (the 1st category of the image)
- **DDP:** DistributedSampler, wrap the model with DDP, launch with torchrun
- **Throughput (img/s)**
- **Batch size**= 256

How to run

Single machine 1 GPU:

```
torchrun --standalone
--nnodes=1
--nproc_per_node=1
demo_coco_resnet18.py
--coco_image_path /mnt/vol0/poldaf/train2017/
--annotation_path /mnt/vol0/poldaf/annotations/instances_train2017.json
--num_epochs 2
```

multiple machines:

```
torchrun
  --nnodes="$WORLD_SIZE"
  --node_rank="$NODE_RANK"
  --nproc_per_node="$GPUS_PER_NODE"
  --rdzv_id=123
  --rdzv_backend=static
  --rdzv_endpoint="$MASTER_ADDR:$MASTER_PORT"
  demo_coco_resnet18.py
    --coco_image_path /mnt/vol0/poldaf/train2017/
    --annotation_path /mnt/vol0/poldaf/annotations/instances_train2017.json
    --num_epochs 2
```

1gb/s Ethernet vs 56gb/s InfiniBand

1gb/s Ethernet:

```
[GPU 0] Epoch 1 | Step 906 | Loss: 1.0805
[GPU 1] Epoch 1 | Step 911 | Loss: 1.0290
[GPU 0] Epoch 1 | Step 911 | Loss: 1.1354
[GPU 1] Epoch 1 | Step 916 | Loss: 1.3558
[GPU 0] Epoch 1 | Step 916 | Loss: 1.3187
[GPU 1] Epoch 1 | Step 921 | Loss: 1.1987
[GPU 0] Epoch 1 | Step 921 | Loss: 1.4499
[GPU 1] Epoch 1 complete | Avg loss: 1.8165
[GPU 0] Cleanup complete. Total wall time: 4h 8m 59.79s (14939.786s)
[GPU 1] Cleanup complete. Total wall time: 4h 8m 59.62s (14939.617s)
```

56gb/s InfiniBand:

```
[GPU 1] Epoch 1 | Step 911 | Loss: 1.0339
[GPU 0] Epoch 1 | Step 911 | Loss: 1.1168
[GPU 0] Epoch 1 | Step 916 | Loss: 1.3348
[GPU 1] Epoch 1 | Step 916 | Loss: 1.3758
[GPU 0] Epoch 1 | Step 921 | Loss: 1.4388
[GPU 1] Epoch 1 | Step 921 | Loss: 1.2611
[GPU 1] Epoch 1 complete | Avg loss: 1.7999
[GPU 0] Cleanup complete. Total wall time: 19m 24.99s (1164.994s)
[GPU 1] Cleanup complete. Total wall time: 19m 24.77s (1164.770s)
```

Conclusion & Questions

PyTorch provides a flexible and powerful framework for building deep learning models.
When your workloads grow, **DistributedDataParallel (DDP)** helps you scale training across multiple GPUs or machines — with minimal changes to your code.

Key takeaways:

- PyTorch is easy to start with and scales well
- DDP is the go-to method for distributed training
- Use it when working with large datasets or complex models
- Not suitable for inference or general parallel computing

Thank you for your attention!