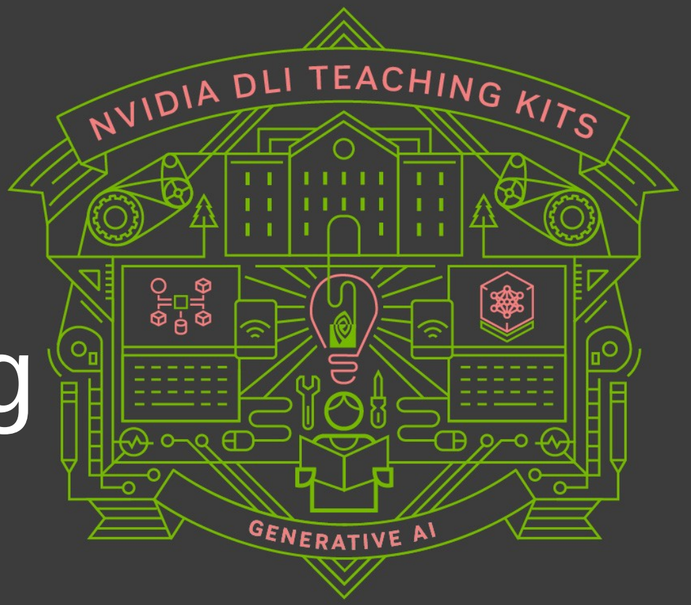
Lecture 9.2 - Challenges and Libraries for Distributed Training

Generative AI Teaching Kit



This lecture

- Common Challenges in Distributed Training
- Distributed Training Libraries and Frameworks
- Precision and Quantization in Distributed Training
- Best Practices and Approaches

Common Challenges in Distributed Training

A Recap on distributed computing in GenAI

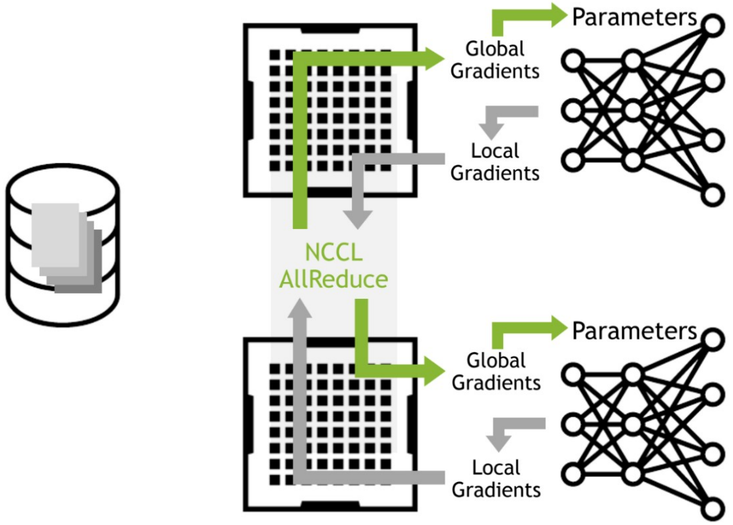
In the previous lesson we covered training in a distributed style:

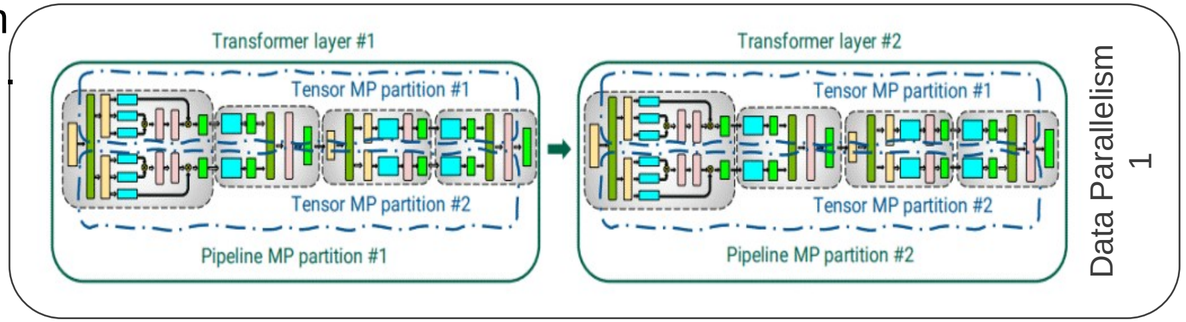
- Data parallelism enabled faster training with duplicate model copies and a distributed training dataset.

- Model parallelism split large models across multiple GPUs to allow for truly gigantic model training.

To perform these parallelism algorithms, we leverage distributed computing.

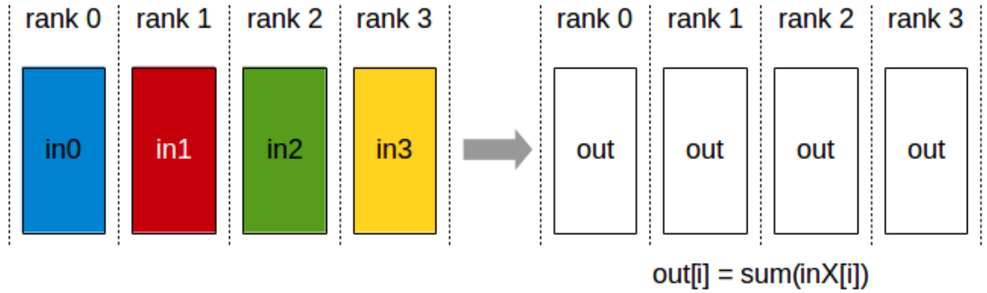
In this lesson we will dive deeper into some of the challenges in distributed computing and how to overcome and manage them .



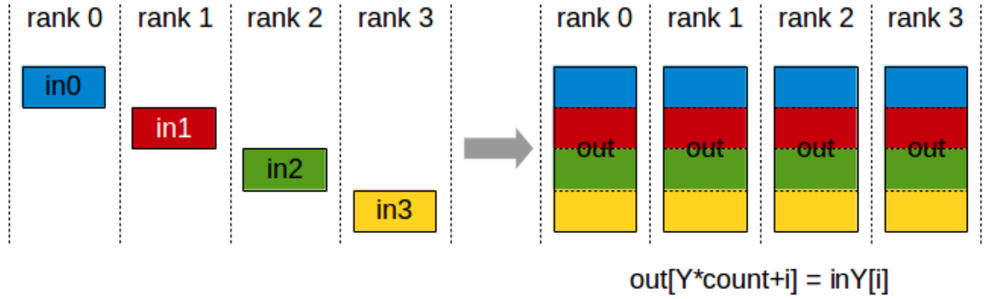


Important communications during training

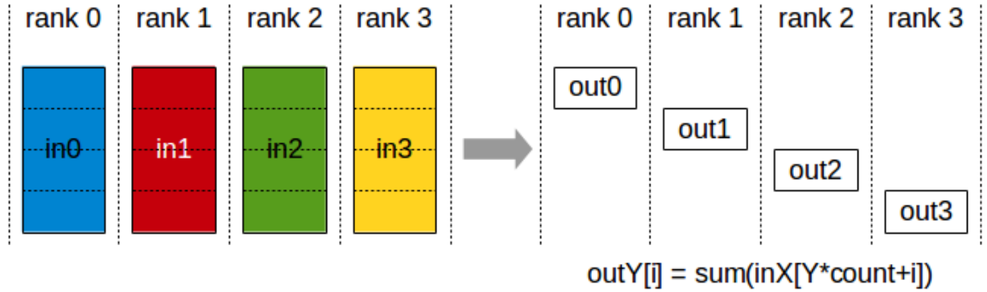
All-Reduce
A collective communication process that sums gradients from all GPUs and then distributes the result back to each GPU. It ensures that each GPU has the same updated gradients, which allows for synchronized parameter updates across the entire model.



All-Gather
A collective communication process that collects values from all GPUs and then distributes the result back to each GPU. It ensures that each GPU has the same collection of individual values.



Reduce Scatter
A collective communication process that performs the same operation as the Reduce operation, except the result is scattered in equal blocks among ranks, each rank getting a chunk of data based on its rank index. This is very important for FSDP.

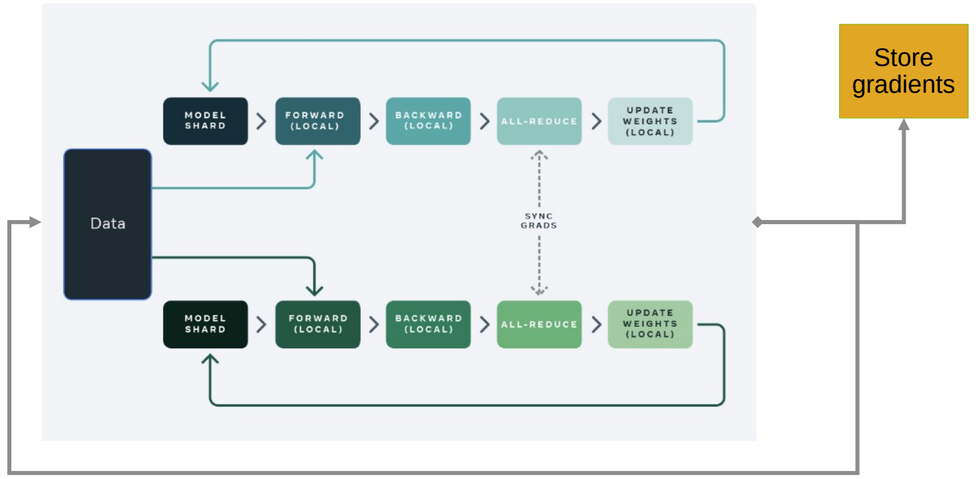


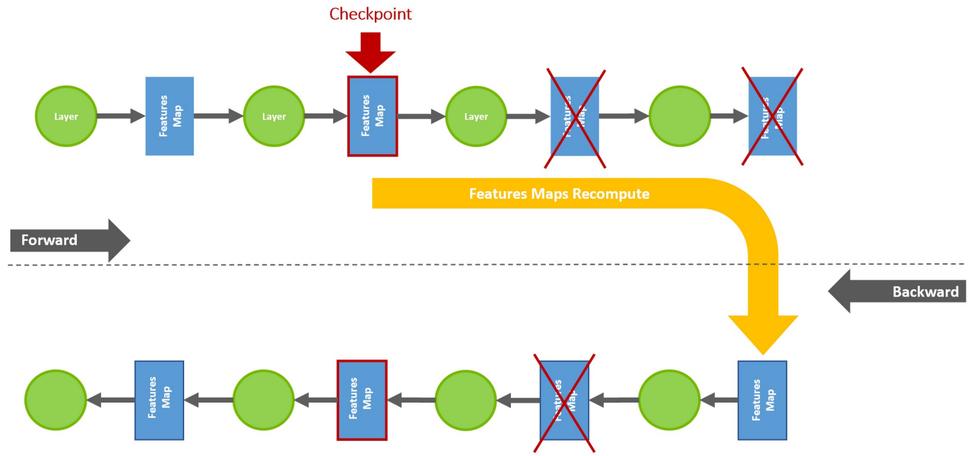
Memory vs. Computation Trade-off

In an attempt to alleviate the memory pressure of computing all of the mini-batch sent to the GPU and performing a gradient update, we can instead pass multiple micro batches and accumulate these gradients as these micro batches are performed at each time.

This Gradient Accumulation approach trades throughput for memory efficiency which can often be worthwhile when critical effective mini batch sizes need to be met.

Activation checkpointing is another approach to minimize memory overhead. In this method large, fast to compute, activations are freed up once they are calculated and only recalculated on the back pass for updating weights.

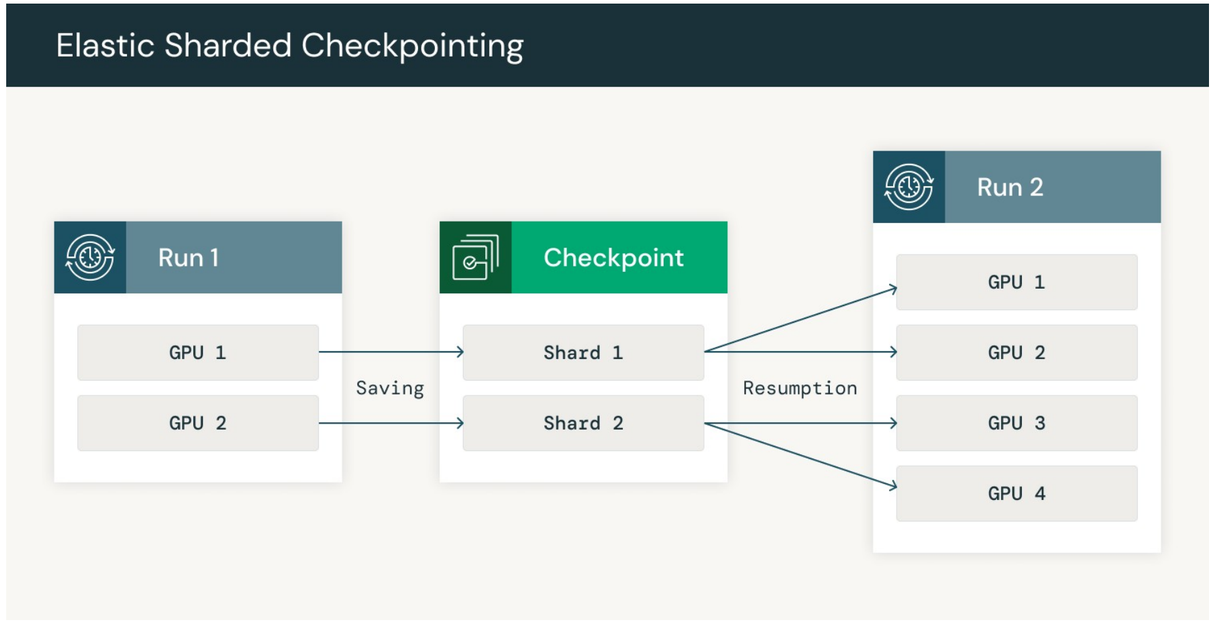


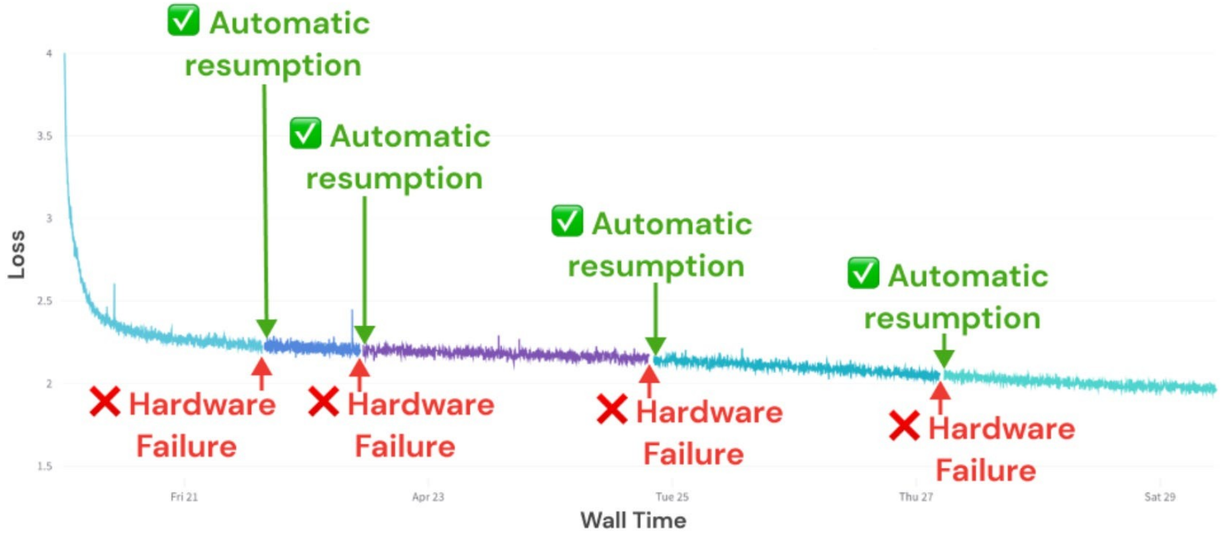


DARTMOUTH ENGINEERING | NVIDIA

Elastic Training

- Fault tolerance is crucial for ensuring that LLMs can be trained reliably over extended periods, especially in distributed environments where node failures are common.
- When a failure occurs, the system can resume from the last saved state rather than starting over.
- To ensure robustness to failures, checkpointing often and saving and loading checkpoints in the most performant way possible to minimize downtime.



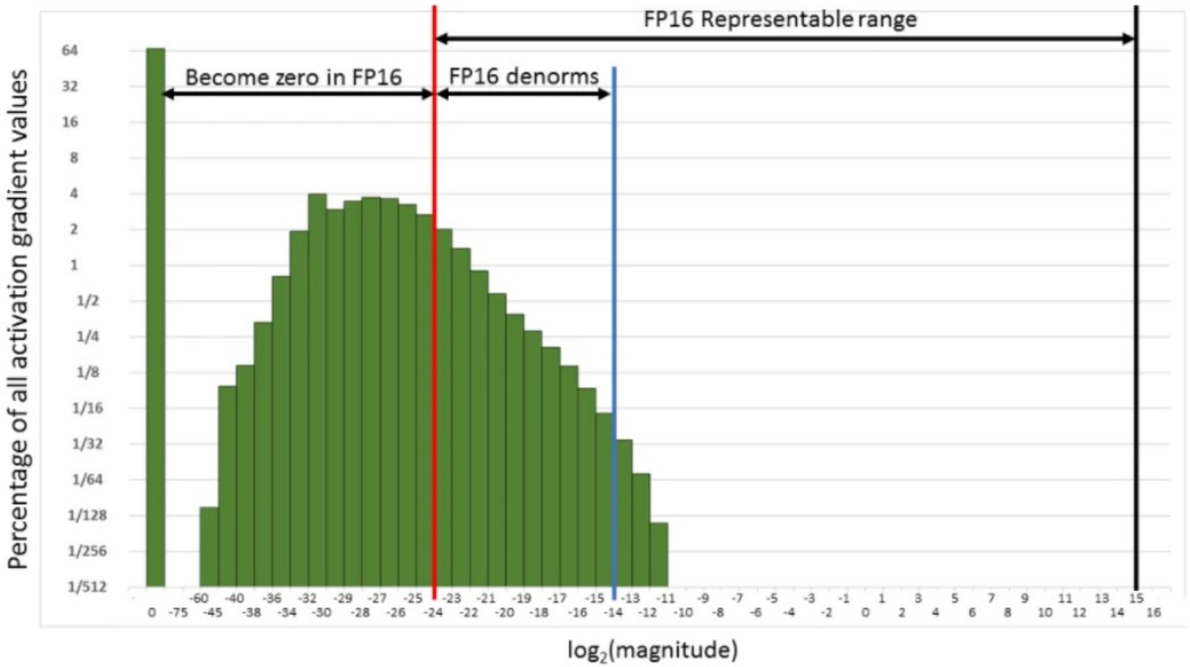


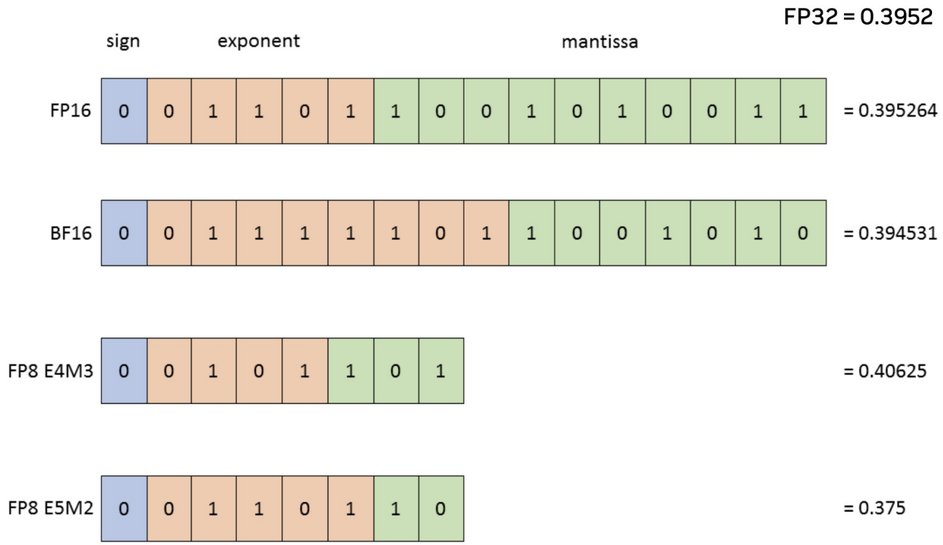
Precision and Quantization in Distributed Training

Number Precision

Optimizing precision is crucial in distributed setups to balance memory efficiency and computational accuracy.

Precision is the amount of memory allocated to storing a single number





Mixed-Precision Training

Mixed Precision Training

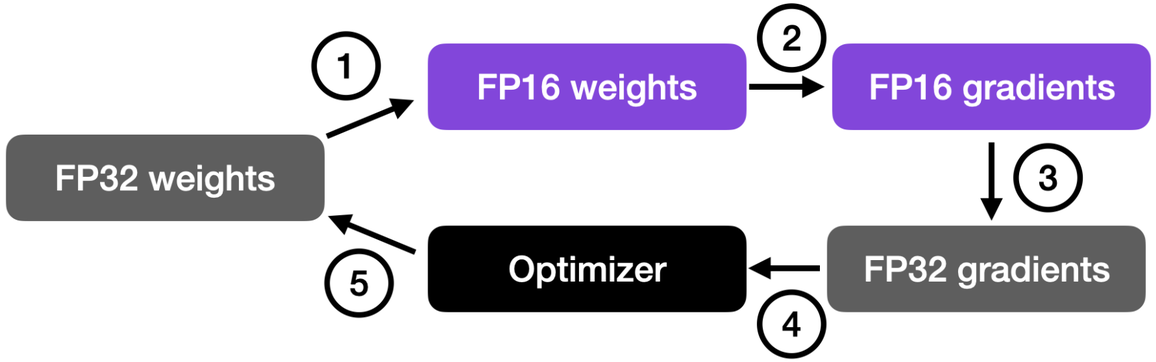
Definition: Combines 16-bit floating-point (FP16) operations with 32-bit (FP32) for certain computations.

Advantages:

- Reduces memory usage and speeds up training.
- Supported by frameworks like PyTorch (torch.cuda.amp) and TensorFlow.

Challenges:

- Numerical instability (e.g., gradients vanishing or exploding).
- Requires careful tuning of loss scaling.



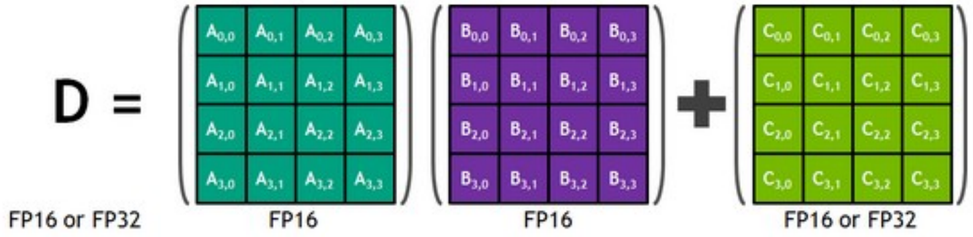
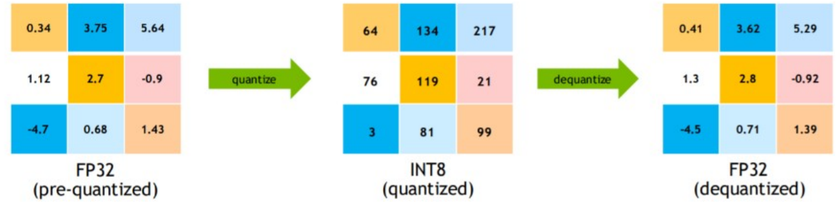


Figure 1: Tensor Core 4x4x4 matrix multiply and accumulate.

Quantization in training

Quantization reduces the bit width of weights and activations (e.g., using INT8 instead of FP32). This can be extremely useful in distributed training and inference as the same model can be communicated with lower overhead.



Input Datatype	Accumulation Datatype	Math Throughput	Bandwidth Reduction
FP32	FP32	1x	1x
FP16	FP16	8x	2x
INT8	INT32	16x	4x
INT4	INT32	32x	8x
INT1	INT32	128x	32x

Advantages:

- Saves memory and increases inference speed.
- Common in inference rather than training (due to accuracy degradation during training).

Disadvantages:

- Loss of accuracy
- Potential instability in training
- Complexities in implementation:
  - Post-training Quantization
  - Quantization-aware training

Post-training quantization (PTQ)	Quantization-aware training (QAT)
Start with a pre-trained model and evaluate it on a calibration dataset.	Start with a pre-trained model and introduce quantization ops at various layers
Calibration data is used to calibrate the model. It can be a subset of training data.	Finetune it for a small number of epochs.
Calculate dynamic ranges of weights and activations in the network to compute quantization parameters (q-params).	Simulates the quantization process that occurs during inference.
Quantize the network using q-params and run inference	The goal is to learn the q-params which can help to reduce the accuracy drop between the quantized model and pre-trained model.

Key Distributed Training Libraries and Frameworks

Why use frameworks?

Much of the software architecture in deep learning is heavily reused time and again. From layer definitions and loss functions to automatic differentiation libraries, writing code from scratch is a poor use of time and resources.

Deep learning frameworks have now reached a level of maturity that they can be used reliably in both research and commercial development.

These frameworks are also able to run across giant supercomputer clusters over many months contiguously.

TensorFlow

NEMO

deepspeed

PyTorch Distributed

PyTorch Distributed is a highly flexible and scalable framework for distributed deep learning. It provides tools to parallelize training across multiple GPUs and nodes using different strategies:

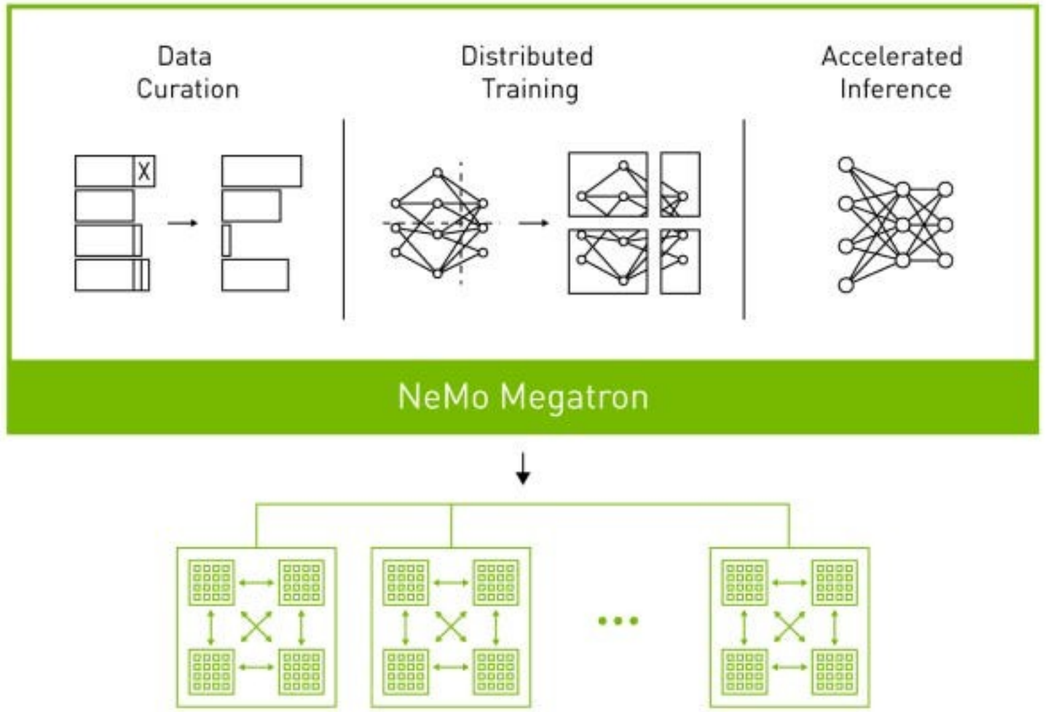
- **Backend Support**: Offers multiple backends (NCCL, Gloo, MPI) for inter-device communication.
- **Distributed Data Parallel (DDP)**: Efficient data parallelism with gradient synchronization.
- **RPC Framework**: Supports model and task parallelism via remote procedure calls.
- **Elastic Training**: Enables dynamic resource scaling (e.g., fault tolerance and auto-scaling in the cloud).
- **Ease of Use**: Seamless integration with PyTorch APIs, allowing users to adapt existing models for distributed training with minimal code changes.

```python
import torch
import torch.nn as nn
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel

# Initialize process group
dist.init_process_group(backend="nccl")

# Wrap model
model = nn.Sequential(...)
ddp_model = DistributedDataParallel(model, device_ids=[local_rank])

# Distributed sampler
dist_sampler = DistributedSampler(dataset)
dataloader = DataLoader(dataset, batch_size=32, sampler=dist_sampler)

for batch in dataloader:
   loss = ddp_model(batch).sum()
   loss.backward()
   optimizer.step() # Gradients aggregated
```

NVIDIAs Nemo/Megatron

NVIDIA NeMo is a powerful framework for building, training, and fine-tuning large AI models, particularly in Generative AI.

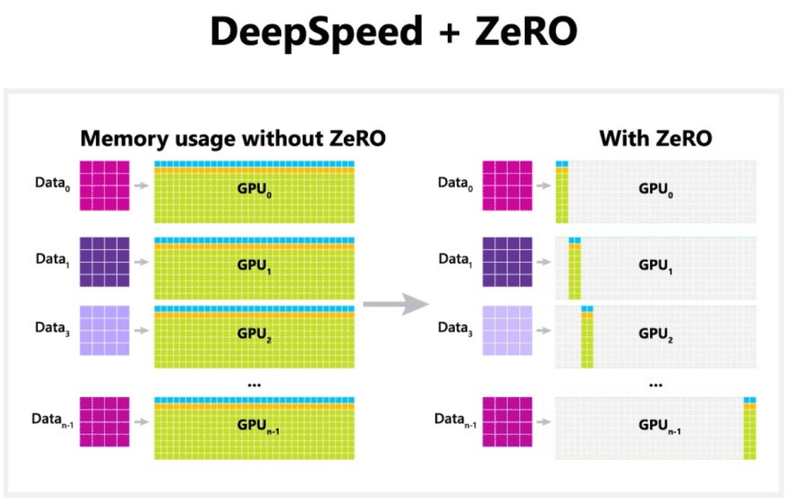
- **Prebuilt Models**: Provides pretrained models for ASR, NLP, and TTS that can be fine-tuned on custom datasets.
- **Modular Design**: Components are built as reusable modules, enabling flexibility and customization.
- **Scalability**: Designed to work seamlessly with NVIDIA's GPU infrastructure and supports multi-GPU/multi-node training with **PyTorch Lightning** or **NVIDIA Megatron** for large models.
- **Optimizations**: Integrates NVIDIA libraries (e.g., Apex for mixed precision, TensorRT for inference) for maximum performance.
- **Ease of Use**: User-friendly APIs, Jupyter Notebook tutorials, and out-of-the-box pipelines.

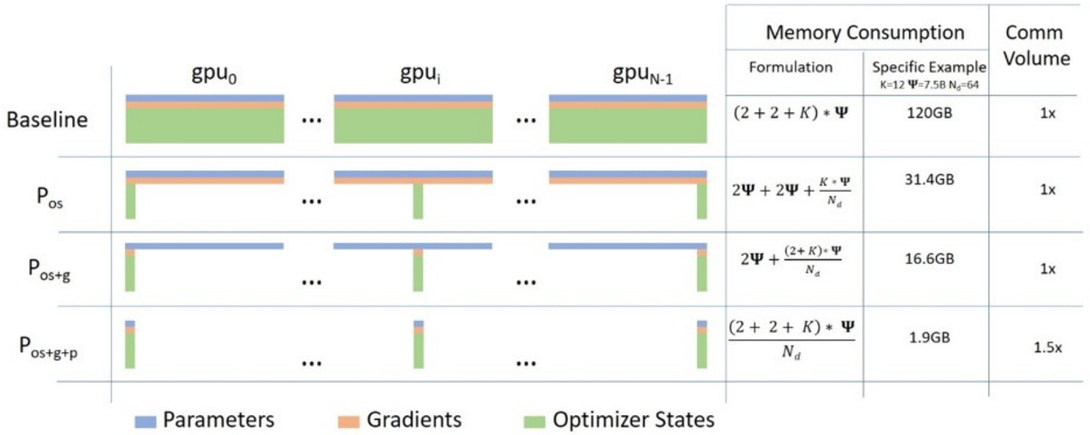


DeepSpeed

DeepSpeed is an open-source deep learning library by Microsoft designed for efficient distributed training and inference of large models. Key features include:

- **ZeRO Optimization**: Minimizes memory use for massive models by sharding weights, gradients, and optimizer states.
- **3D Parallelism**: Combines data, pipeline, and model parallelism for scalability.
- **Increased Efficiency**: Supports mixed precision training with **DeepSpeed-Inference** for fast, low-latency model deployment.
- **Ease of Use**: Works seamlessly with PyTorch and simplifies scaling to hundreds of GPUs.



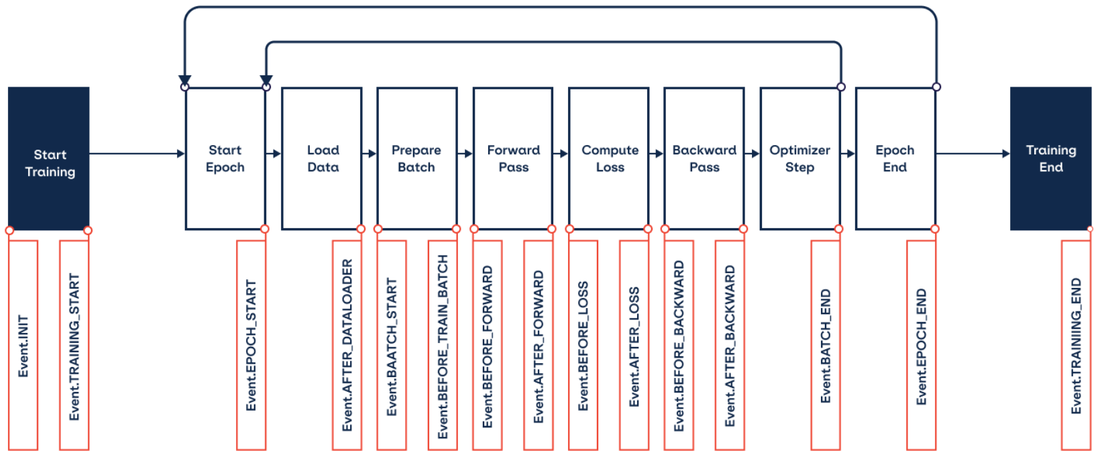


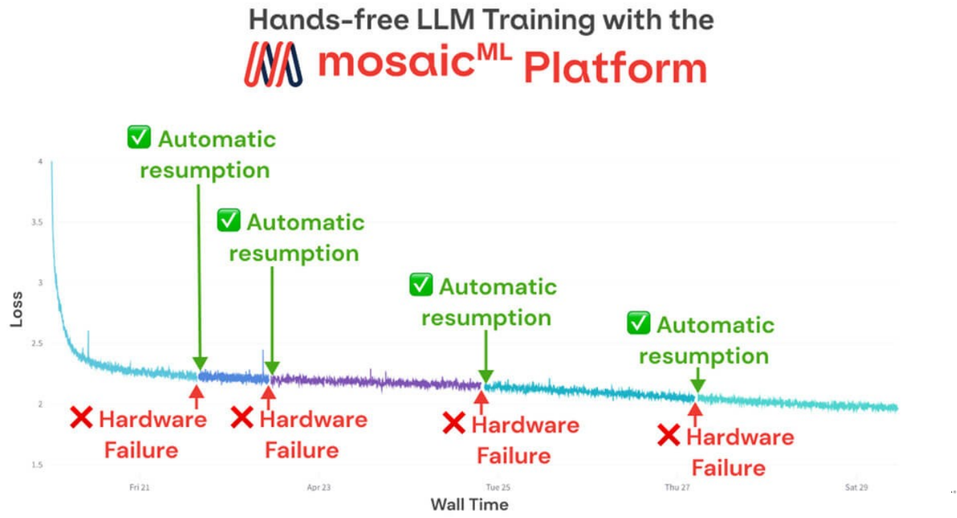
DARTMOUTH ENGINEERING | NVIDIA

MosaicML Composer

MosaicML Composer is a flexible deep learning training library that optimizes and accelerates model training workflows. Key features include:

- **Speed-Up Techniques**: Includes cutting-edge algorithms like selective backpropagation, gradient skipping, and efficient data augmentation to reduce training time.
- **Seamless Integration**: Built on PyTorch, it integrates easily with existing codebases and supports popular frameworks.
- **Scale with Ease**: Optimized for single-node and multi-node setups, supporting both data and model parallelism.
- **Customizable**: Modular design lets you plug in and experiment with new training algorithms.





composer

Hugging Face Accelerate

Hugging Face Accelerate simplifies distributed training and inference for machine learning models. Key features include:

- **Ease of Use**: Minimal code changes to enable multi-GPU, TPU, or multi-node training.
- **Flexibility**: Works with PyTorch and supports dynamic scaling for diverse hardware setups.
- **Zero Boilerplate**: Handles device placement, mixed precision, and gradient accumulation automatically.
- **Community Integration**: Seamlessly integrates with Hugging Face libraries like Transformers and Datasets.


```
+ from accelerate import Accelerator
+ accelerator = Accelerator()

+ model, optimizer, training_dataloader, scheduler = accelerator.prepare(
+     model, optimizer, training_dataloader, scheduler
+ )

  for batch in training_dataloader:
      optimizer.zero_grad()
      inputs, targets = batch
      inputs = inputs.to(device)
      targets = targets.to(device)
      outputs = model(inputs)
      loss = loss_function(outputs, targets)
+     accelerator.backward(loss)
      optimizer.step()
      scheduler.step()
```

Deciding what to choose

Choose your library based on project size, ease of use, and hardware compatibility. For small projects, prioritize simplicity; for large-scale models, focus on advanced optimization and scalability. Match the tool to your expertise and performance needs.

1. **Project Scale**
   - **Small to Medium**: Use **Hugging Face Accelerate** for simplicity and fast scaling.
   - **Large-Scale**: Choose **DeepSpeed** or **NVIDIA NeMo** for advanced optimization and scalability.

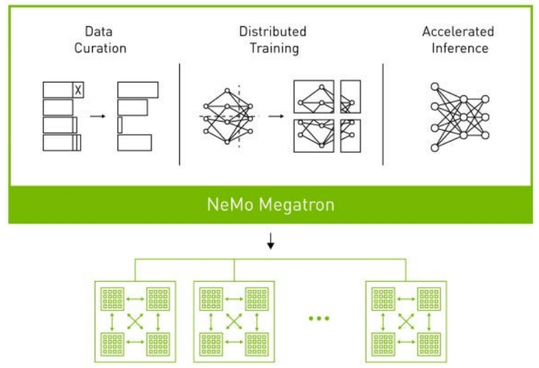
2. **Ease of Use vs Customization**
   - **Easy to Start**: **Hugging Face Accelerate** for minimal setup.
   - **Advanced Control**: **DeepSpeed**, **PyTorch Distributed**, or **MosaicML Composer** for custom workflows.

3. **Performance Optimization**
   - **Memory Efficiency**: **DeepSpeed (ZeRO)** for massive models.
   - **Speed and Cost**: **MosaicML Composer** for innovative training techniques.

4. **Hardware & Applications**
   - **NVIDIA Ecosystem**: **NVIDIA NeMo** for speech, NLP, and multimodal tasks.
   - **Multi-GPU/Node**: **DeepSpeed** or **PyTorch Distributed** for scaling.

Accelerate

composer



DARTMOUTH ENGINEERING | NVIDIA

Best Practices and Solutions

Utilizing Mixed Precision

Traditionally, deep neural networks are trained using the IEEE single-precision format. Mixed precision training allows you to use half precision (FP16) for most operations while maintaining the accuracy of single precision (FP32). This approach combines both single- and half-precision representations, enhancing performance without compromising model quality.

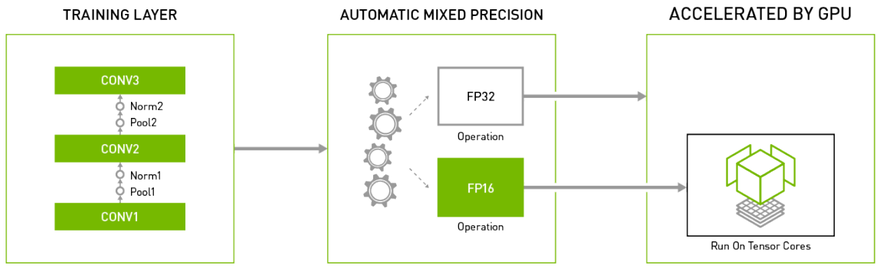
```
import torch
# Creates once at the beginning of training
scaler = torch.cuda.amp.GradScaler()

for data, label in data_iter:
   optimizer.zero_grad()
   # Casts operations to mixed precision
   with torch.amp.autocast(device_type="cuda", dtype=torch.float16):
      loss = model(data)

   # Scales the loss, and calls backward()
   # to create scaled gradients
   scaler.scale(loss).backward()

   # Unscales gradients and calls
   # or skips optimizer.step()
   scaler.step(optimizer)

   # Updates the scale for next iteration
   scaler.update()
```

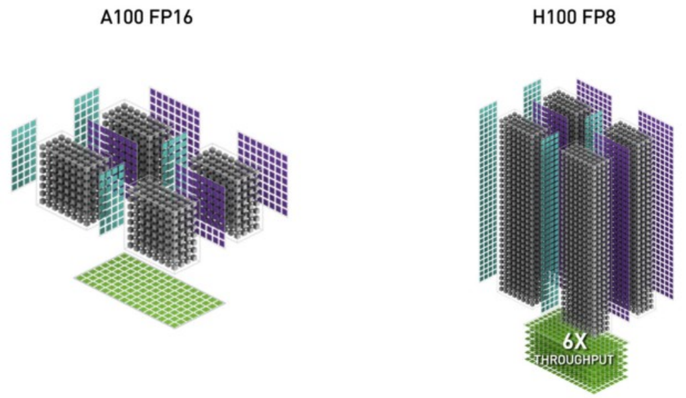
Benefits of Mixed Precision Training:

- **Faster Computation**: Speeds up math-heavy operations (e.g., linear and convolution layers) by utilizing Tensor Cores.
- **Improved Memory Efficiency**: Reduces memory usage by accessing half the bytes compared to single precision, enabling larger models or minibatches.
- **Optimized Hardware Utilization**: Takes full advantage of the specialized hardware capabilities in modern GPUs.

How to Enable Mixed Precision Training:

- **Adopt Half Precision**: Modify the model to use the half-precision (FP16) data type for suitable operations.
- **Use Loss Scaling**: Apply loss scaling to prevent small gradient values from vanishing.





Mixed precision is supported on NVIDIA Ampere, Volta, and Turing GPUs, making it easier for researchers and engineers to adopt this performance-enhancing technique.

Good Monitoring Tools

Why Monitor Training?

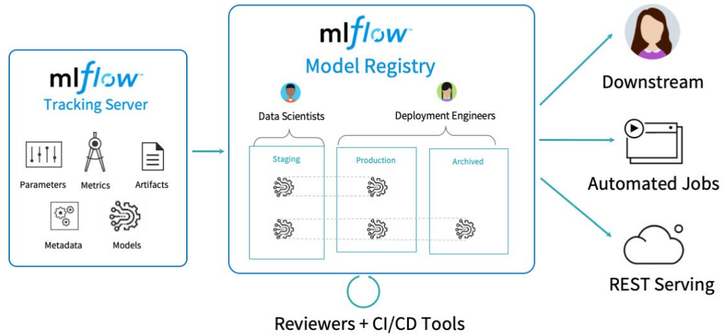
Gain insights into model performance, training stability, and resource usage. Detect issues like overfitting, vanishing gradients, or data pipeline bottlenecks early.

Popular Monitoring Tools:

- **MLflow**:
  - Track experiments, log metrics, and version models.
  - Supports integration with various frameworks and deployment pipelines.
- **TensorBoard**:
  - Visualize metrics, model graphs, and hyperparameters in real-time.
  - Ideal for PyTorch and TensorFlow users.
- **Weights & Biases (WandB)**:
  - Collaborate with teammates via shared dashboards.
  - Log training runs, system metrics, and hyperparameter sweeps seamlessly.

Best Practices:

- Automate logging with framework-specific integrations.
- Use alert systems for anomalous behavior (e.g., sudden metric drops).
- Combine multiple tools if needed (e.g., MLflow for model tracking and W&B for visualization).

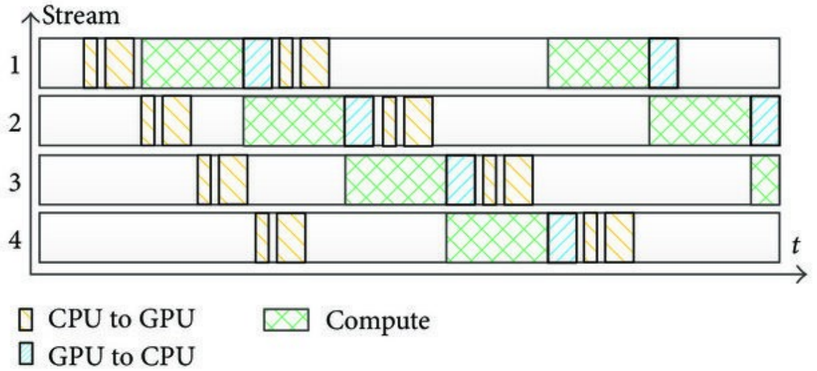


DARTMOUTH ENGINEERING | NVIDIA

Overlapping Computation and Communication

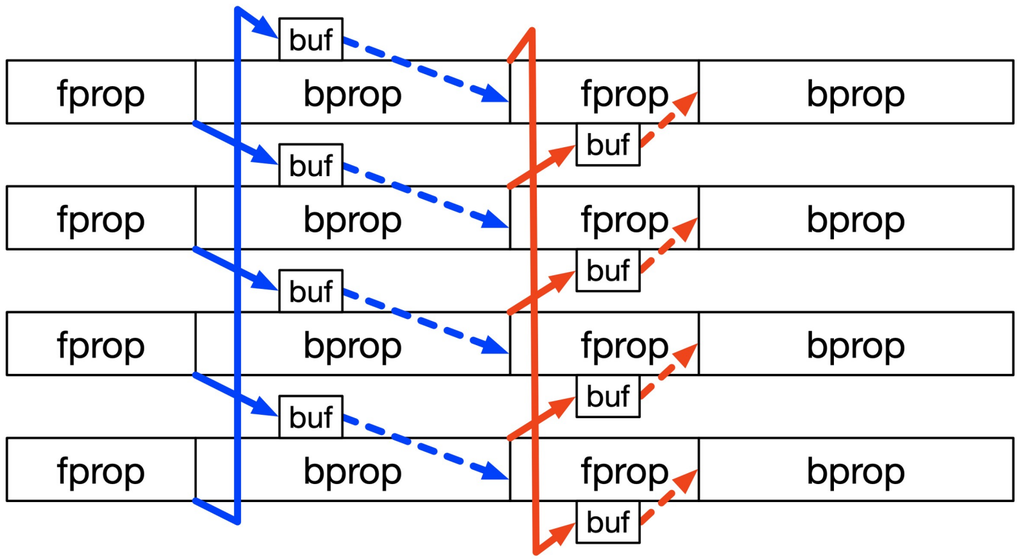
Efficient Data Loading:

- Minimize GPU idle time by optimizing data pipelines.
- Use **prefetching**, **asynchronous loading**, and **multiprocessing** to load data in parallel.
- Tools: PyTorch's DataLoader, tf.data, and caching mechanisms like **TFRecord** or **WebDataset**.



Overlapping Communication with Computation:

- Hide data transfer latency by overlapping GPU computation with CPU data preparation and communication.
- Use **non-blocking operations** (e.g., torch.cuda.streams) for seamless execution.
- For distributed training, leverage frameworks like **NCCL** to overlap gradient synchronization with forward and backward passes.

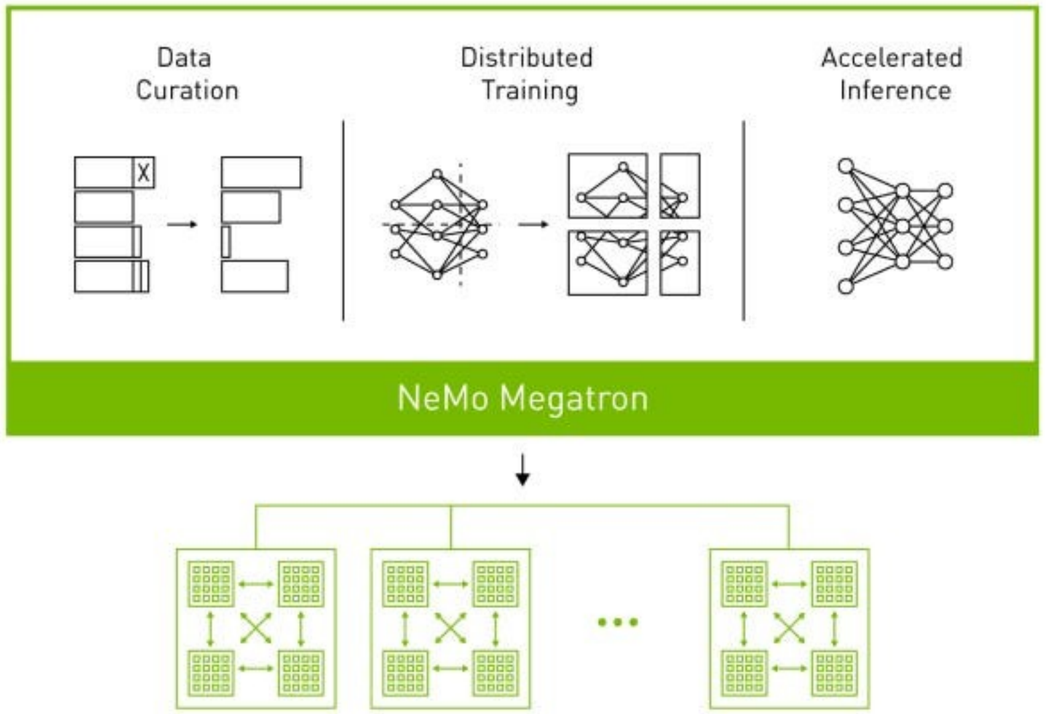


Wrap Up

Challenges and Libraries for Distributed Training

- Today we dove deeper into the practical implications of distributed training
- We saw different approaches to managing tradeoffs common in distributed training
- Common distributed training libraries were introduced with guidance on which framework to use based on the scale of development and maturity.
- Several best practices, including monitoring, mixed precision, and communication/computation management were also discussed.

In the next class we'll change our focus into the challenges of distributed inference!



Thank you!