# E0294: Systems for Machine Learning

# Assignment 3

**Nehal Shah**
SR - 24623
*nehalshah@iisc.ac.in*

---

**Problem 1:**Consider the LeNet-5 network proposed in https://ieeexplore.ieee.org/document/726791. Consider 100 examples from the MNIST dataset randomly. Divide the dataset into four parts (i.e., 25 examples in each part). Launch data parallel training on 4 CPU cores for 100 epochs.

**(a)** Use 'all reduce' to update the gradients after every epoch. **(5)**
**Solution:**code can be found in **1a.py** file.
In this implementation, we manually synchronize the gradients across all ranks using the **All-Reduce** technique. Each rank computes its local gradients, then sends them to rank 0, which aggregates (sums) the gradients from all ranks. Rank 0 then averages the gradients and sends them back to all ranks, ensuring each rank updates its model with the same set of gradients.

1. Split the dataset into 4 parts (25 examples each).
2. Perform forward and backward passes on each core for its part of the dataset.
3. Each core computes its local gradients.
   (a) **Compute gradients**: Each rank computes gradients based on its local data.
   (b) **Synchronize gradients**: Rank 0 collects gradients from all ranks, averages them, and sends them back to all ranks. averaged gradients.

   The gradient update rule for each parameter $w$ is as follows:

$$w = w - \eta \cdot \frac{1}{N} \sum_{i=1}^{N} \nabla L_i(w)$$

   Where:

   - $w$ is the model parameter,
   - $\eta$ is the learning rate,
   - $N$ is the number of ranks (workers),
   - $\nabla L_i(w)$ is the gradient computed by rank $i$ for the loss function $L$.

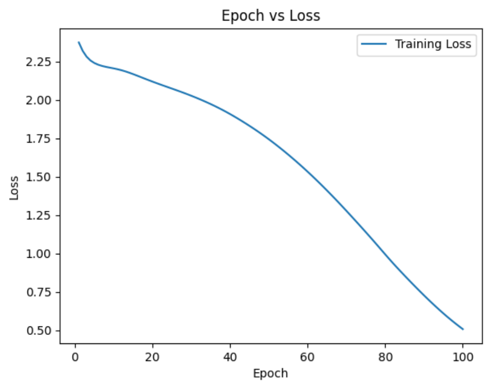
   This ensures that all ranks update their model consistently, allowing the training process to proceed in parallel across multiple workers.
4. Perform `all reduce` across all 4 cores to aggregate the gradients.
5. Update the model parameters based on the aggregated gradients.

.

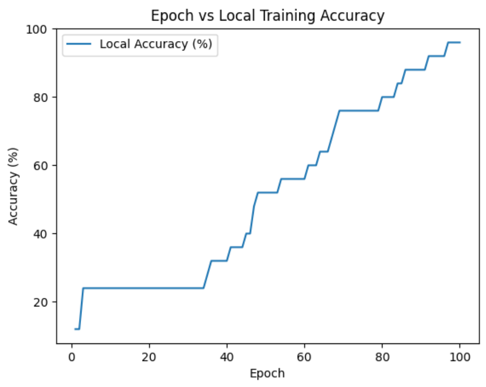
For each epoch:

- Each rank computes the gradients based on its subset of the data.
- The gradients from all ranks are then averaged using the "all-reduce" operation.
- The averaged gradients are used to update the model parameters across all ranks.

The final model parameters after each epoch will be the same across all ranks, ensuring that the model converges in parallel.



(a) Loss over Epochs



(b) Accuracy over Epochs

Figure 1: Training Loss and Accuracy over Epochs

**(b)** Use 'ring all reduce' to update the gradients after every epoch. **(5)**
**Solution:** code can be found in the **1b.py** file.
In this implementation, we manually synchronize the gradients across ranks using the **Ring All-Reduce** technique. Unlike the traditional **All-Reduce**, where gradients are aggregated and shared in one step, **Ring All-Reduce** uses a ring communication pattern. Each rank communicates with its neighboring rank, sharing and receiving gradients in a cyclic manner. This reduces the communication overhead compared to **All-Reduce**.

1. **Compute gradients**: Each rank computes gradients based on its local data.
2. **Synchronize gradients**: In a ring structure, each rank sends its gradients to the next rank and receives gradients from the previous rank.
3. **Aggregate gradients**: After receiving the gradients from its neighbor, each rank adds them to its local gradients.
4. **Average gradients**: After completing the communication steps, the gradients are averaged by dividing by the number of ranks.
5. **Update model**: Each rank updates its model using the averaged gradients.

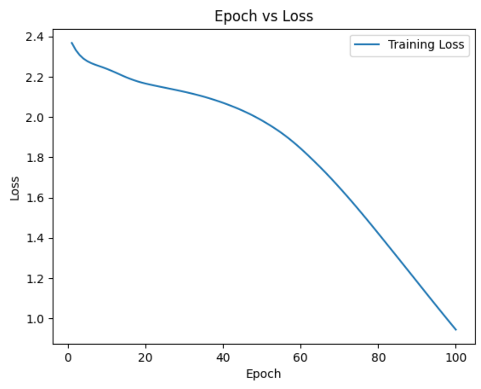
This ensures that all ranks have the same set of gradients and updates their models consistently, making it an efficient method for training in parallel.

For each epoch:

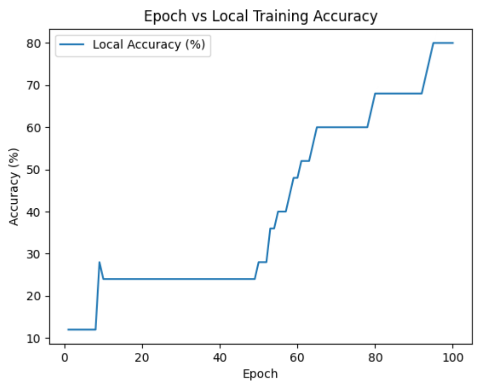
- Each rank computes its gradients based on its subset of data.

- The gradients are passed around the ring, with each rank updating its local gradients as the data circulates.
- The averaged gradients are then used to update the model parameters across all ranks.

The final model parameters after each epoch will be the same across all ranks, ensuring consistent updates while reducing communication costs.



(a) Loss over Epochs



(b) Accuracy over Epochs

Figure 2: Training Loss and Accuracy over Epochs for ring all reduce

**(c)** Clearly describe how you performed the above two exercises. Compare the accuracy of the trained model and the training time. **(15)**
**Solution:**
**Description of how we performed both the implementations:**

**1. All-Reduce for Gradient Synchronization**

The `All-Reduce` operation is a collective communication operation in distributed systems where each process (or rank) computes a local gradient and shares it with all other processes. This operation ensures that all processes have synchronized gradients for updating model parameters.

The steps for the All-Reduce operation are as follows:

- Each rank computes the gradients for its local model parameters after a forward and backward pass.
- Rank 0 collects gradients from all other ranks. It accumulates the gradients and averages them.
- The averaged gradient is then sent to all other ranks, ensuring that all ranks have the same gradient for updating the model parameters.
- This operation is typically performed for every parameter in the model, allowing the training to proceed in a synchronized manner across all ranks.

**2. Ring All-Reduce for Gradient Synchronization**

`Ring All-Reduce` is a specific implementation of the `All-Reduce` operation. Unlike the traditional All-Reduce, where data is sent to a central rank (usually rank 0), Ring All-Reduce follows a ring-based communication pattern. In this method, each rank sends and receives data in a circular fashion, which helps in reducing the communication overhead in large distributed systems.

The steps for Ring All-Reduce are as follows:

- Each rank computes gradients for its local model parameters.
- Every rank sends its local gradients to the next rank in a ring-like fashion, and it receives gradients from the previous rank.
- Upon receiving gradients, each rank adds them to its own gradients, ensuring that all ranks accumulate gradients from all other ranks.
- After each rank has received gradients from all others, the gradient is averaged by dividing it by the total number of ranks.
- Finally, the averaged gradients are sent to all other ranks so that each rank has the same gradient values for the parameter update.

I manually implemented the gradient synchronization for **All-Reduce** and **Ring All-Reduce** as follows:

- **All-Reduce**: After each epoch, gradients were computed on each rank, then manually summed and broadcasted to all ranks using custom logic to share the gradients across all processes.
- **Ring All-Reduce**: Implemented a cyclic communication pattern where each rank exchanged its gradients with its neighboring rank using `dist.isend()` and `dist.irecv()`.
- We have Import time function to measure training time.

**Imports used in the code:**

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, Subset
import torch.multiprocessing as mp
import time
import datetime
```

**Command to run the file:**

```
python -m torch.distributed.launch --nproc_per_node=4 your_script.py
```

**Distributed Training Setup: TCPStore and mp.spawn()**

**1. TCPStore**

The `TCPStore` is used to initialize the process group for distributed training. It manages communication between ranks by providing a central store to synchronize process states. The store is initialized with the local IP, port, and rank-specific configurations. It facilitates rank communication using `dist.init_process_group`.

**2. Using mp.spawn()**
The `mp.spawn()` function is used to spawn multiple processes, each running the `train_worker` function. It allows parallel execution of the training loop across multiple ranks. The number of processes is determined by `world_size`, and `join=True` ensures the main process waits for all workers to finish.

In this Implementation, we compared the training times for All-Reduce and Ring All-Reduce for the LeNet-5 model on the MNIST dataset, using a **learning rate** of **0.002** which is typical for gradient-based optimization in convolutional neural networks like LeNet-5. This learning rate was chosen to ensure stable and consistent convergence during training.

**Training Accuracy Comparison: All-Reduce vs Ring All-Reduce**

| **Metric** | **Rank 0** | **Rank 1** | **Rank 2** | **Rank 3** |
|---|---|---|---|---|
| All-Reduce Training Accuracy (%) | 50.80 | 52.92 | 65.92 | 58.44 |
| All-Reduce Validation Accuracy (%) | 25.15 | 27.15 | 31.83 | 32.92 |
| Ring All-Reduce Training Accuracy (%) | 40.60 | 52.44 | 54.52 | 55.72 |
| Ring All-Reduce Validation Accuracy (%) | 20.81 | 26.45 | 28.07 | 28.07 |

Table 1: Comparison of Training and Validation Accuracy for All-Reduce vs Ring All-Reduce

**Comments on the Results**

- **Training Accuracy Comparison:**
  **All-Reduce** consistently resulted in higher **training accuracy** compared to **Ring All-Reduce** across all ranks. For example, **Rank 2** achieved **65.92%** accuracy with **All-Reduce**, but only **54.52%** with **Ring All-Reduce**.
- **Reason for the Difference:**
  **All-Reduce** converges faster since it synchronizes gradients across all ranks simultaneously. This allows each rank to update its model with a **global view** of the gradients, leading to faster adjustments in the model weights and more rapid convergence during training.
  - **Faster Convergence:** The gradient synchronization happens in one step for all ranks. This leads to quicker model updates, making **All-Reduce** more effective for achieving higher accuracy faster.
  - **Gradient Sharing:** With **All-Reduce**, all the gradients are aggregated immediately, allowing each rank to benefit from the complete information of all other ranks, leading to more precise weight updates and improved performance.

  On the other hand, **Ring All-Reduce** uses a **cyclic gradient communication** pattern, where each rank only shares its gradients with one neighboring rank at a time. This results in a **gradual synchronization** of the gradients, meaning each rank only gets partial gradient information before moving forward. This process takes longer compared to **All-Reduce**.
  - **Cyclic Communication:** In **Ring All-Reduce**, the communication follows a "ring" pattern, where each rank communicates with its immediate neighbors. While this method reduces the communication overhead, it also delays the time it takes for a rank to access the full gradient information, leading to slower convergence.

- **Slower Gradient Sharing:** Unlike **All-Reduce**, which gathers all the gradients at once, **Ring All-Reduce** must wait until it has received gradients from all other ranks, thus causing **slower updates** and less efficient training.

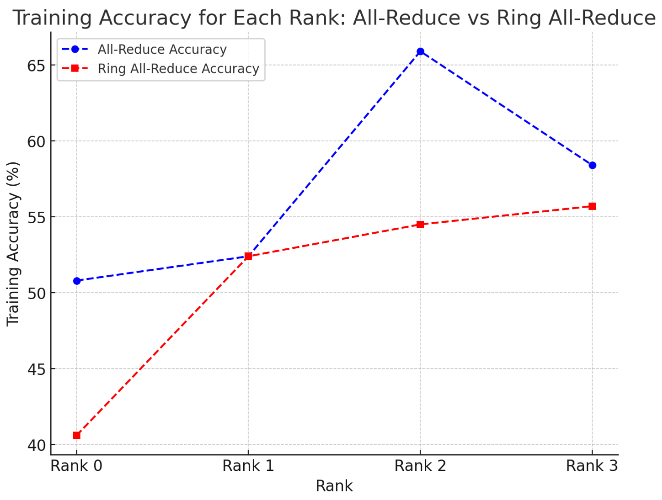


Figure 3: Comparison of Training Accuracy for All-Reduce vs Ring All-Reduce

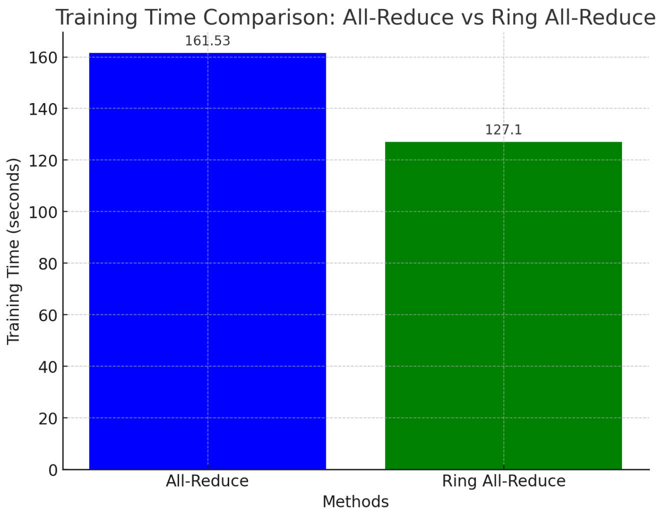


Figure 4: Comparison of Training Time for All-Reduce vs Ring All-Reduce

### Training Time: All-Reduce vs Ring All-Reduce Comments on the Results

- **Training Time Comparison:** As shown in the bar graph, the training time for **All-Reduce** is higher than for **Ring All-Reduce**. Specifically, **All-Reduce** took **161.53 seconds**, while **Ring All-Reduce** completed the training in **127.10 seconds**.
- **Why the Difference?**
  The main reason behind this difference is the synchronization method.

- **All-Reduce** synchronizes gradients across all ranks simultaneously, which requires more communication overhead. While this method leads to higher accuracy, it takes more time to complete the process.
- **Ring All-Reduce**, on the other hand, uses a cyclic gradient sharing method where each rank only shares gradients with its neighbor ranks in a ring. This reduces the communication overhead but takes longer for convergence.

**Conclusion:** While **Ring All-Reduce** reduces the communication overhead, it results in faster convergence compared to **All-Reduce**. Thus, **Ring All-Reduce** is more efficient in terms of training time, but **All-Reduce** is typically better when it comes to model accuracy.

**Problem 3:** Eyeriss proposes a row-stationary dataflow. Consider the given input feature map and kernel. Compute the convolution output at each clock cycle by following the row-stationary dataflow approach. Assume that both multiplication and addition operations require one clock cycle.

Input feature map

| | | |
|---|---|---|
| 1 | 3 | 9 |
| 4 | 5 | 6 |
| 7 | 4 | 9 |

Kernel

| | |
|---|---|
| -1 | 2 |
| 2 | -1 |

**solution:** We assume that computations occur using a row-stationary dataflow. The convolution operation involves sliding the kernel over the input feature map and performing element-wise multiplication followed by summation at each position. Using row-stationary dataflow, intermediate results (partial sums) are accumulated row-wise in local registers before being moved to higher memory levels.Let us denote the processing elements as follows:

The output dimensions are determined as:

$$\textbf{Output Dimensions} = \textbf{(Input Dimensions - Kernel Dimensions + 1)}$$

For a $3 \times 3$ input and $2 \times 2$ kernel:

$$\text{Output Dimensions} = (3 - 2 + 1) \times (3 - 2 + 1) = 2 \times 2$$

**Computation for Each Output Position** - We slide the kernel over the input feature map and compute the convolution as follows:

1. **Top-left corner Region:**

$$\begin{bmatrix} 1 & 3 \\ 4 & 5 \end{bmatrix} \quad \begin{bmatrix} -1 & 2 \\ 2 & -1 \end{bmatrix}$$

Element-wise multiplication:

$$(-1 \cdot 1) + (2 \cdot 3) + (2 \cdot 4) + (-1 \cdot 5) = -1 + 6 + 8 - 5 = \mathbf{8}$$

**Multiply** $-1 \times 1 = -1$
**Multiply** $2 \times 3 = 6$
**Multiply** $2 \times 4 = 8$
**Multiply** $-1 \times 5 = -5$
**Sum:** $-1 + 6 + 8 - 5 = 8$

**2. Top-right corner Region:**

$$\begin{bmatrix} 3 & 9 \\ 5 & 6 \end{bmatrix} \quad \begin{bmatrix} -1 & 2 \\ 2 & -1 \end{bmatrix}$$

Element-wise multiplication:

$$(-1 \cdot 3) + (2 \cdot 9) + (2 \cdot 5) + (-1 \cdot 6) = -3 + 18 + 10 - 6 = \mathbf{19}$$

**Multiply** $-1 \times 3 = -3$
**Multiply** $2 \times 9 = 18$
**Multiply** $2 \times 5 = 10$
**Multiply** $-1 \times 6 = -6$
**Sum:** $-3 + 18 + 10 - 6 = 19$

**3. Bottom-left corner Region:**

$$\begin{bmatrix} 4 & 5 \\ 7 & 4 \end{bmatrix} \quad \begin{bmatrix} -1 & 2 \\ 2 & -1 \end{bmatrix}$$

Element-wise multiplication:

$$(-1 \cdot 4) + (2 \cdot 5) + (2 \cdot 7) + (-1 \cdot 4) = -4 + 10 + 14 - 4 = \mathbf{16}$$

**Multiply** $-1 \times 4 = -4$
**Multiply** $2 \times 5 = 10$
**Multiply** $2 \times 7 = 14$
**Multiply** $-1 \times 4 = -4$
**Sum:** $-4 + 10 + 14 - 4 = 16$

**4. Bottom-right corner Region:**

$$\begin{bmatrix} 5 & 6 \\ 4 & 9 \end{bmatrix} \quad \begin{bmatrix} -1 & 2 \\ 2 & -1 \end{bmatrix}$$

Element-wise multiplication:

$$(-1 \cdot 5) + (2 \cdot 6) + (2 \cdot 4) + (-1 \cdot 9) = -5 + 12 + 8 - 9 = \mathbf{6}$$

**Multiply** $-1 \times 5 = -5$
**Multiply** $2 \times 6 = 12$
**Multiply** $2 \times 4 = 8$
**Multiply** $-1 \times 9 = -9$
**Sum:** $-5 + 12 + 8 - 9 = 6$

**Final Output Feature Map** -The resulting output feature map is:

$$\begin{bmatrix} 8 & 19 \\ 16 & 6 \end{bmatrix}$$

**Clock Cycle Analysis** -Using row-stationary dataflow:
- Each multiplication takes 1 clock cycle.
- Each addition takes 1 clock cycle.
- For a $2 \times 2$ kernel applied to a $3 \times 3$ input feature map:
- Multiplications per position:4
- Additions per position: 3
- Total cycles per position: 4(multiplications) + 3(additions) = 7 clock cycles.
- Total positions: $2 \times 2 = 4$.
- Total clock cycles for convolution: $7 \times 4 = 28$ clock cycles.

**Thus tolal number of cclock cycles require are 28.**