



École Nationale Supérieure
d'Informatique et d'Analyse des Systèmes
- Rabat

Distributed Systems
Rapport : projet

---

# Scalable Machine Learning : Distributed Training Methods for Large Models and Datasets

---

***Élèves :***
Mohamed STIFI

***Enseignant :***
Pr. Noureddine KERZAZI

10 janvier 2025

# Acknowledgments

First and foremost, we express our heartfelt gratitude to The Almighty Allah for granting me the strength and patience needed to successfully complete this work.

I would also like to extend my sincere thanks to my supervisor, Pr. Noureddine KERZAZI, for her invaluable guidance, trust, and generosity. His unwavering support and the resources he provided were essential to the successful completion of this project. Throughout the duration of my work, he offered insightful guidance and demonstrated remarkable patience, helping me navigate the challenges I encountered.

Finally, I would like to thank all those who have contributed, both directly and indirectly, to the completion of this project. Your support has been invaluable, and your contributions have played a crucial role in helping me reach this gratifying outcome.

# Abstract

This report explores the transition from CPU-based to GPU-based machine learning training, with a focus on distributed training methods such as data parallelism and model parallelism. The study evaluates the performance of these methods across six models—VGG11, NBoW, Seq2Seq, ConvAutoencoder, CNN, and MLP—using single-GPU and multi-GPU setups. The experiments demonstrate that distributed training significantly reduces training time while maintaining or improving model performance in most cases. Hybrid parallelism, which combines data and model parallelism, shows particular promise in accelerating training for large models. However, challenges such as synchronization overhead and model complexity are observed, especially in models like CNN. The findings underscore the importance of distributed training methods in modern machine learning workflows, particularly for large-scale models and datasets.

# Résumé

Ce rapport explore la transition de l'entraînement en apprentissage automatique basé sur CPU à celui basé sur GPU, en se concentrant sur les méthodes d'entraînement distribuées telles que le parallélisme de données et le parallélisme de modèles. L'étude évalue les performances de ces méthodes sur six modèles — VGG11, NBoW, Seq2Seq, ConvAutoencoder, CNN et MLP — en utilisant des configurations avec un seul GPU et plusieurs GPU. Les expériences montrent que l'entraînement distribué réduit significativement le temps d'entraînement tout en maintenant ou en améliorant les performances du modèle dans la plupart des cas. Le parallélisme hybride, qui combine le parallélisme de données et de modèles, montre un potentiel particulier pour accélérer l'entraînement des grands modèles. Cependant, des défis tels que la surcharge de synchronisation et la complexité des modèles sont observés, en particulier pour des modèles comme le CNN. Les résultats soulignent l'importance des méthodes d'entraînement distribuées dans les flux de travail modernes en apprentissage automatique, notamment pour les modèles et ensembles de données à grande échelle.

# Table des matières

**Introduction** **6**

**1 Acceleration Methods in Machine Learning Training : From CPU to Multi-GPUs** **7**
1.1 From CPU to Single GPU in ML Training . . . . . 7
1.1.1 Benefits of Using a Single GPU . . . . . 7
1.1.2 Limitations of Single GPU Training . . . . . 7
1.2 From Single GPU to Multi-GPUs : Distributed Training . . . . . 8
1.2.1 Types of Distributed Training . . . . . 8
1.2.2 Advantages of Multi-GPU Training . . . . . 9
1.2.3 Challenges of Distributed Training . . . . . 9
1.3 Conclusion . . . . . 10

**2 Distributed Training : Data and Model Parallelism** **11**
2.1 Data Parallelism . . . . . 11
2.1.1 Problem Addressed by Data Parallelism . . . . . 11
2.1.2 Data Parallelism Training Algorithm . . . . . 11
2.1.3 Advantages of Data Parallelism . . . . . 12
2.1.4 Types of Data Parallelism . . . . . 12
2.2 Model Parallelism . . . . . 13
2.2.1 Problem Addressed by Model Parallelism . . . . . 14
2.2.2 Model Parallelism Training Algorithm . . . . . 14
2.2.3 Advantages of Model Parallelism . . . . . 15
2.2.4 Types of Model Parallelism . . . . . 15
2.2.5 Challenges and Considerations . . . . . 16
2.3 Distributed Training Algorithm and Explanation . . . . . 17
2.3.1 Algorithm . . . . . 18
2.3.2 Explanation . . . . . 18
2.4 Conclusion . . . . . 20

**3 Experimental Evaluation of Distributed Training Methods** **21**
3.1 Experimental Environment . . . . . 21
3.1.1 Experimental Setup . . . . . 21
3.1.2 Models . . . . . 22
3.1.3 Training and Evaluation . . . . . 22
3.1.4 Data Parallelism and Model Parallelism . . . . . 22
3.2 Experimental Results and Analysis . . . . . 23
3.2.1 VGG11 . . . . . 23

3.2.2 NBoW . . . . . 25
3.2.3 Seq2Seq . . . . . 27
3.2.4 ConvAutoencoder . . . . . 29
3.2.5 CNN . . . . . 31
3.2.6 MLP . . . . . 34
3.3 Conclusion . . . . . 37

**Conclusion** **38**

# Introduction

Machine learning models have grown increasingly complex, requiring significant computational resources to train effectively. Traditional CPU-based training methods are often insufficient for handling large datasets and complex architectures, leading to the adoption of GPU-based training and, more recently, distributed multi-GPU training. This report investigates the transition from CPU to GPU-based training and explores the benefits and challenges of distributed training methods, including data parallelism and model parallelism.

The primary objective of this study is to evaluate the effectiveness of distributed training in accelerating machine learning workflows. We conduct experiments on six models—VGG11, NBoW, Seq2Seq, ConvAutoencoder, CNN, and MLP—using both single-GPU and multi-GPU setups. The performance metrics, including training time, validation time, loss, and accuracy, are analyzed to compare the efficiency of distributed training methods.

The report is structured as follows : Chapter 1 provides an overview of the transition from CPU to GPU-based training and introduces distributed training methods. Chapter 2 delves into the concepts of data parallelism and model parallelism, outlining their algorithms, advantages, and challenges. Chapter 3 presents the experimental setup and results, comparing the performance of single-GPU and multi-GPU training across the selected models. Finally, the report concludes with a summary of the findings and their implications for future machine learning workflows.

# Chapitre 1

# Acceleration Methods in Machine Learning Training : From CPU to Multi-GPUs

Machine learning (ML) models often require significant computational resources, especially when dealing with large datasets and complex architectures. To address this challenge, the field has seen a gradual shift from traditional CPU-based training to GPU-based training, and ultimately to distributed multi-GPU training. This chapter explores these transitions, their motivations, limitations, and advantages, with a focus on achieving efficient and fast ML training.

## 1.1 From CPU to Single GPU in ML Training

The initial step in accelerating ML training involves transitioning from CPU-based computations to GPU-based computations. GPUs are highly optimized for parallel processing, making them particularly well-suited for operations commonly used in ML, such as matrix multiplications and convolutions.

### 1.1.1 Benefits of Using a Single GPU

— **Parallelism :** GPUs can handle thousands of simultaneous threads, enabling efficient execution of parallelizable tasks.
— **High Throughput :** GPUs have high memory bandwidth and computational throughput, leading to faster training times compared to CPUs.
— **Optimized Libraries :** Frameworks such as TensorFlow and PyTorch provide GPU-optimized libraries (e.g., cuDNN) for ML training.

### 1.1.2 Limitations of Single GPU Training

While using a single GPU significantly accelerates ML training, certain limitations remain :

— **Memory Constraints :** GPUs often have limited memory, which can restrict the size of the models and datasets that can be trained.

— **Scalability :** As model sizes and datasets grow, a single GPU may become a bottleneck.
— **Overhead :** The process of transferring data between the CPU and GPU introduces latency, which can impact performance.

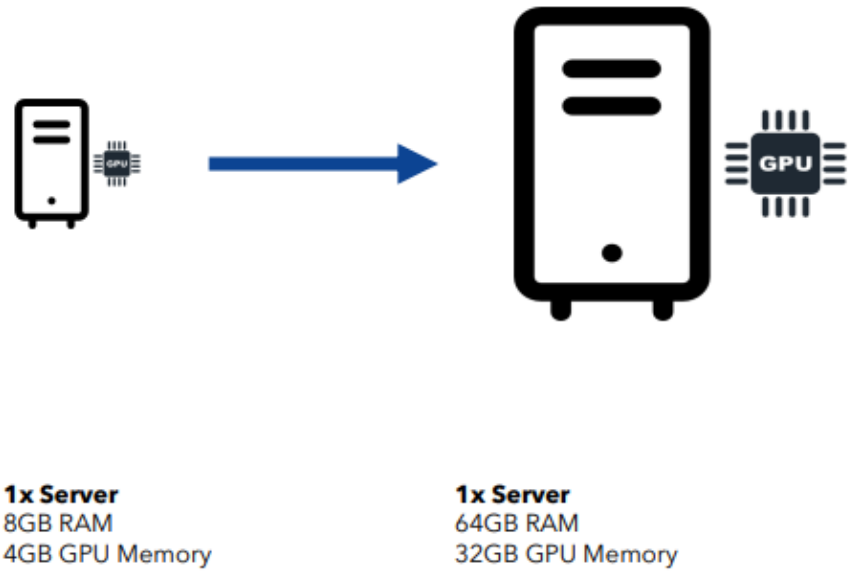


FIGURE 1.1 – Transition from CPU to Single GPU in ML Training

## 1.2 From Single GPU to Multi-GPUs : Distributed Training

To overcome the limitations of single GPU training, researchers and practitioners have adopted distributed training techniques using multiple GPUs. Distributed training leverages the computational power of multiple GPUs to handle larger models and datasets more efficiently.

### 1.2.1 Types of Distributed Training

Distributed training can be broadly categorized into :
— **Data Parallelism :** The dataset is divided into smaller chunks, and each GPU processes a subset of the data using a shared model. After processing, gradients are aggregated to update the model (Figure 1.2).
— **Model Parallelism :** The model itself is divided across GPUs, with each GPU handling a different part of the model. This is particularly useful for very large models that cannot fit into the memory of a single GPU.

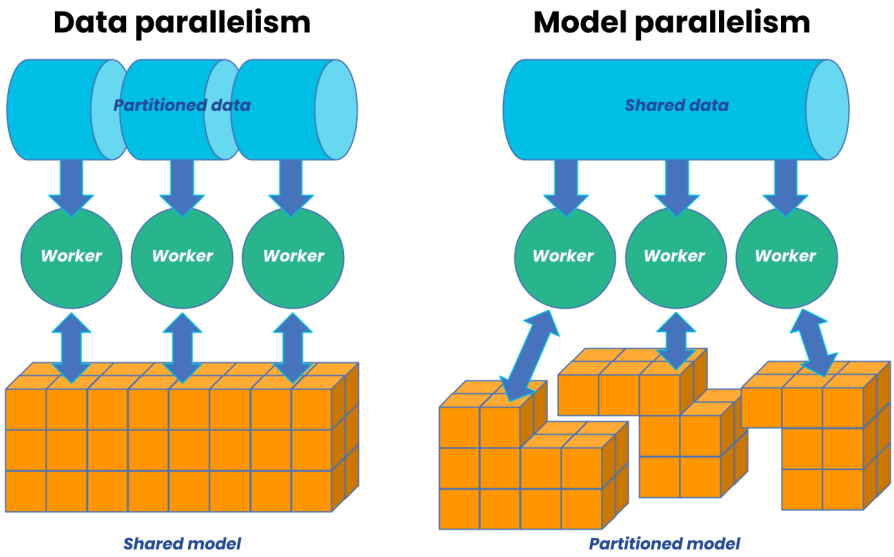


FIGURE 1.2 – Data Parallelism and Model Parallelism in Distributed Training

### 1.2.2 Advantages of Multi-GPU Training

Distributed training offers several key advantages :

— **Scalability :** By distributing the workload, multi-GPU setups can handle larger models and datasets that would otherwise be infeasible.

— **Reduced Training Time :** Parallel processing across multiple GPUs significantly reduces the time required for training.

— **Flexibility :** Distributed training frameworks such as Horovod and NCCL make it easier to scale ML workloads across multiple GPUs.

### 1.2.3 Challenges of Distributed Training

Despite its advantages, distributed training also presents challenges :

— **Communication Overhead :** Synchronizing gradients and parameters across GPUs introduces communication overhead.

— **Complexity :** Implementing distributed training requires additional effort and expertise.

— **Hardware Costs :** Multi-GPU setups can be expensive to acquire and maintain.

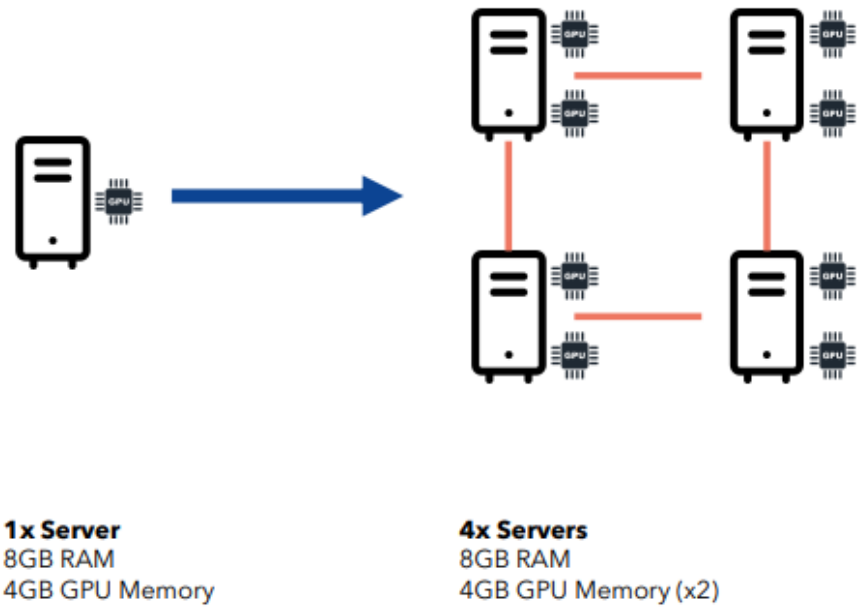


FIGURE 1.3 – Scaling from Single GPU to Multi-GPU Training

## 1.3 Conclusion

The transition from CPU-based training to GPU-based training and subsequently to distributed multi-GPU training represents a significant milestone in the field of machine learning. While each step offers substantial performance improvements, it also introduces new challenges that must be addressed. By leveraging the power of GPUs and distributed training, researchers and practitioners can train larger and more complex models efficiently, paving the way for further advancements in machine learning.

# Chapitre 2

# Distributed Training : Data and Model Parallelism

Distributed training is the process of training ML models across multiple machines or devices, with the goal of speeding up the training process and enabling the training of larger models on larger datasets.

## 2.1 Data Parallelism

### Introduction

Data parallelism is a fundamental technique in distributed machine learning that addresses the challenges of training large models on extensive datasets. As models and datasets grow in size, the computational resources required for training also increase significantly. Data parallelism offers a solution by distributing the workload across multiple processing units, typically GPUs, to accelerate the training process.

### 2.1.1 Problem Addressed by Data Parallelism

Training deep learning models, especially on large datasets, presents several challenges :

- **Memory Constraints** : Large models may not fit within the memory of a single GPU, limiting the batch size and slowing down training.
- **Training Time** : Extensive datasets can result in prolonged training times, sometimes extending to weeks or months.
- **Resource Utilization** : Single-GPU setups underutilize available computational resources, leading to inefficiencies.

Data parallelism mitigates these issues by distributing the dataset across multiple GPUs, allowing each GPU to process a subset of the data concurrently. This approach not only reduces memory constraints but also significantly decreases training time by leveraging the collective power of multiple GPUs.

### 2.1.2 Data Parallelism Training Algorithm

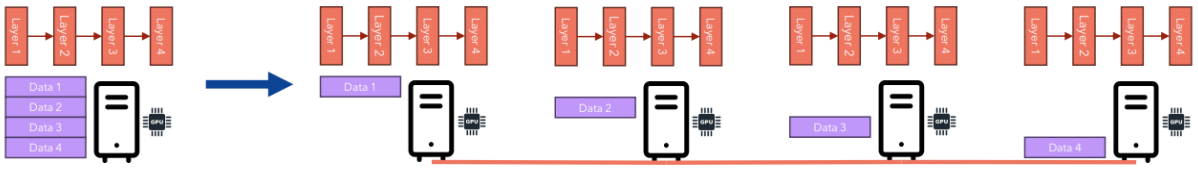


FIGURE 2.1 – Data Parallelism

The data parallelism training algorithm involves the following steps :

1. **Model Replication** : The model is replicated across all available GPUs. Each GPU holds an identical copy of the model.
2. **Data Partitioning** : The training dataset is divided into smaller mini-batches, with each GPU processing a distinct mini-batch.
3. **Local Gradient Calculation** : Each GPU computes the gradients of the model parameters based on its assigned mini-batch.
4. **Gradient Synchronization** : The gradients computed by each GPU are averaged to produce a single set of gradient values. This step ensures that all model replicas remain consistent.
5. **Model Update** : The model parameters are updated using the averaged gradients, and the updated parameters are broadcast to all GPUs.
6. **Iteration** : The process repeats for the next set of mini-batches until the training is complete.

This algorithm ensures that the training process is both efficient and scalable, making it suitable for large-scale machine learning tasks.

### 2.1.3 Advantages of Data Parallelism

Data parallelism offers several advantages :

— **Scalability** : It allows for the efficient use of multiple GPUs, enabling the training of larger models and datasets.
— **Reduced Training Time** : By distributing the workload, data parallelism significantly reduces the time required for training.
— **Memory Efficiency** : It alleviates memory constraints by allowing smaller batch sizes to be processed in parallel.
— **Ease of Implementation** : Compared to model parallelism, data parallelism is relatively straightforward to implement, especially with frameworks like PyTorch and TensorFlow.

### 2.1.4 Types of Data Parallelism

Data parallelism can be categorized based on the level of distribution and synchronization :

— **Single-Node Data Parallelism** : Involves multiple GPUs within a single machine. The GPUs share the same memory and communication overhead is minimal.

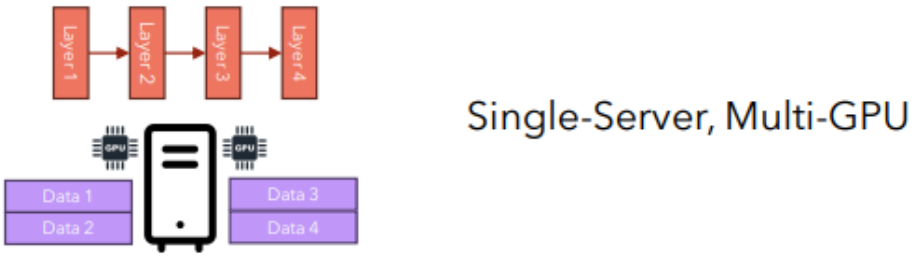


FIGURE 2.2 – Single-Server, Multi-GPU

— **Multi-Node Data Parallelism** : Extends data parallelism across multiple machines, each with its own set of GPUs. This approach is suitable for extremely large datasets and models but introduces additional communication overhead.

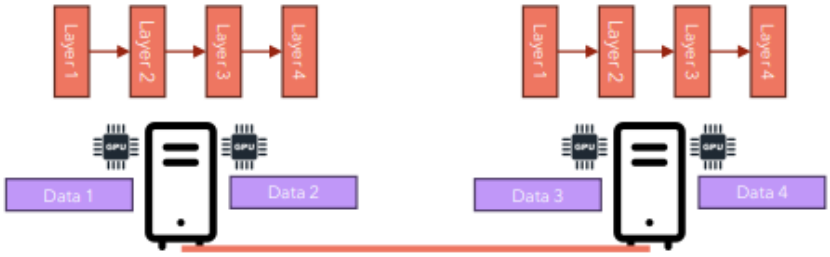


FIGURE 2.3 – Multi-Server, Multi-GPU

— **Hybrid Parallelism** : Combines data parallelism with model parallelism to address both large datasets and large models. This approach is more complex but offers greater flexibility and efficiency.

### Conclusion

Data parallelism is a powerful technique for accelerating machine learning training, particularly for large-scale datasets. By distributing the workload across multiple GPUs, it addresses memory constraints, reduces training time, and improves resource utilization. While it is most effective when the model fits within a single GPU, its scalability and ease of implementation make it a popular choice for distributed training. In the next section, we will explore the Model parallelism as a solution when the model can't fits within a single GPU.

## 2.2 Model Parallelism

### Introduction

Model parallelism is a technique used in distributed machine learning to address the challenges posed by extremely large models that cannot fit within the memory of a single GPU. Unlike data parallelism, which distributes the data across multiple GPUs, model

parallelism distributes the model itself. This approach is particularly useful for training state-of-the-art models, such as large transformers, which have millions or even billions of parameters.

### 2.2.1 Problem Addressed by Model Parallelism

Training large models presents several significant challenges :

— **Memory Constraints** : The model may be too large to fit within the memory of a single GPU, making it impossible to train using traditional methods.

— **Computational Load** : Even if the model fits, the computational load may be too high for a single GPU, leading to excessively long training times.

— **Resource Utilization** : Single-GPU setups may underutilize available computational resources, leading to inefficiencies.

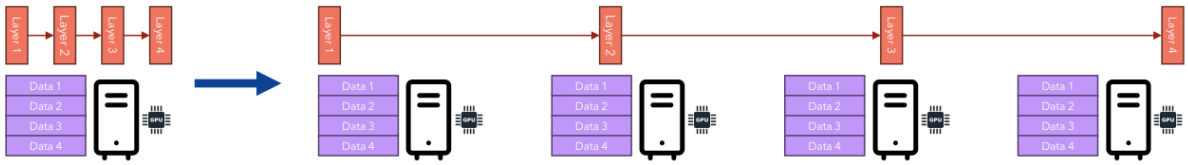


FIGURE 2.4 – Model Parallelism

Model parallelism addresses these issues by splitting the model across multiple GPUs, allowing each GPU to handle a portion of the model. This approach not only reduces memory constraints but also distributes the computational load, making it feasible to train extremely large models.

### 2.2.2 Model Parallelism Training Algorithm

The model parallelism training algorithm involves the following steps :


FIGURE 2.5 – Model Parallelism

1. **Model Partitioning** : The model is divided into smaller sub-models, each assigned to a different GPU. This partitioning can be done at the layer level, with each GPU handling a specific set of layers.
2. **Forward Pass** : Each GPU performs the forward pass on its assigned portion of the model. The output from one GPU is passed as input to the next GPU in the sequence.

3. **Backward Pass** : During the backward pass, gradients are computed for each portion of the model. The gradients are passed sequentially from one GPU to the previous one.
4. **Parameter Update** : Each GPU updates its portion of the model parameters based on the computed gradients.
5. **Iteration** : The process repeats for the next batch of data until the training is complete.

This algorithm ensures that the model is trained efficiently, even when it is too large to fit within the memory of a single GPU.

### 2.2.3 Advantages of Model Parallelism

Model parallelism offers several advantages :

— **Memory Efficiency** : By distributing the model across multiple GPUs, model parallelism reduces the memory footprint on each GPU, making it possible to train larger models.

— **Computational Load Distribution** : The computational load is distributed across multiple GPUs, reducing the burden on any single GPU and potentially speeding up training.

— **Scalability** : Model parallelism allows for the training of extremely large models that would be impossible to train using single-GPU setups.

### 2.2.4 Types of Model Parallelism

Model parallelism can be categorized based on the level of distribution and the techniques used :

— **Layer-wise Parallelism** : The model is divided at the layer level, with each GPU handling a specific set of layers. This approach is straightforward but may lead to inefficiencies due to sequential dependencies.

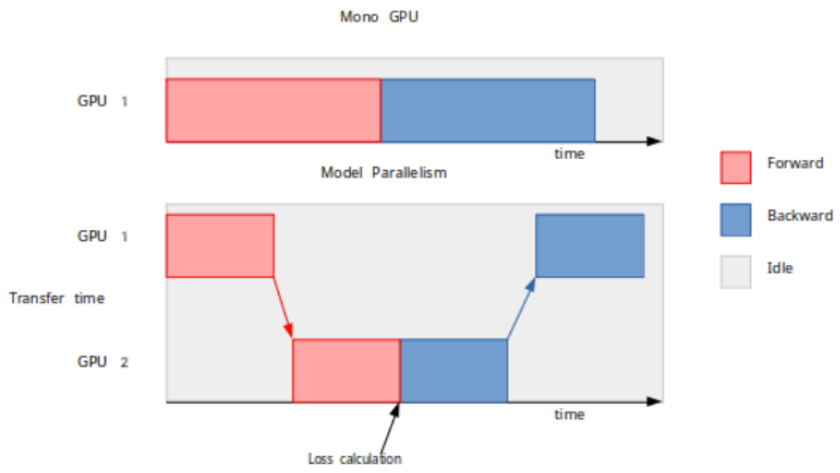


FIGURE 2.6 – Layer-wise Parallelism : Model Parallelism

Note that Layer-wise Parallelism does not accelerate the learning because a GPU

must wait for preceding layers of the model to be treated by another GPU before being able to execute its own layers. The learning is even a little longer because of the time of data transfer between GPUs

— **Pipeline Parallelism** : The model is divided into stages, and each stage is assigned to a different GPU. Data is passed through the pipeline in micro-batches, allowing for quasi-simultaneous processing and reducing idle time.

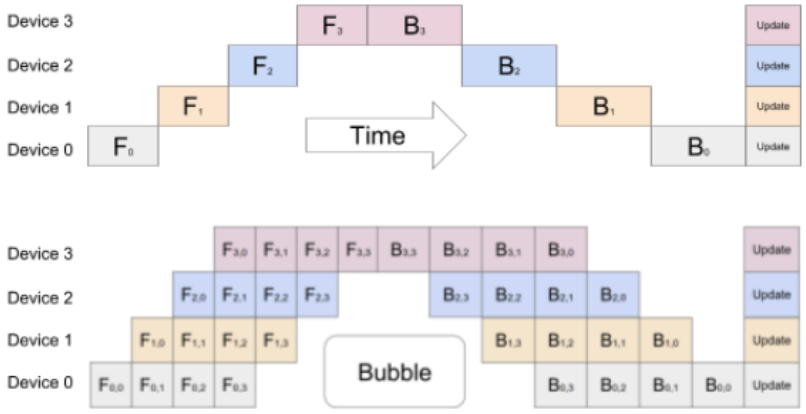


FIGURE 2.7 – Pipeline Parallelism : Model Parallelism

The illustration below shows the quasi-simultaneity and the gain obtained by using the splitting technique where the data batch is divided into 4 micro-batches, for a training distributed on 4 GPUs.

— **Hybrid Parallelism** : Combines model parallelism with data parallelism to address both large models and large datasets. This approach is more complex but offers greater flexibility and efficiency.

### 2.2.5 Challenges and Considerations

While model parallelism offers significant advantages, it also presents several challenges :

— **Implementation Complexity** : Implementing model parallelism requires careful partitioning of the model and efficient communication between GPUs, which can be complex and error-prone.

— **Communication Overhead** : The need to transfer data between GPUs can introduce communication overhead, potentially offsetting some of the performance gains.

— **Load Balancing** : Ensuring that the computational load is evenly distributed across GPUs can be challenging, especially for models with heterogeneous layers.

## Conclusion

Model parallelism is a powerful technique for training extremely large models that cannot fit within the memory of a single GPU. By distributing the model across multiple GPUs, it addresses memory constraints and distributes the computational load, making it feasible to train state-of-the-art models. While it presents implementation challenges,

the benefits of model parallelism make it an essential tool in the arsenal of modern machine learning practitioners. In the next section, we will explore the an overview of the implementation data and model parallelism.

## 2.3 Distributed Training Algorithm and Explanation

### Introduction

Implementing distributed training using data and model parallelism involves a series of well-defined steps. This section provides an overview of the algorithm used to set up and execute distributed training, focusing on the key components and their roles in the process. The algorithm is divided into six main steps, each of which is crucial for the successful implementation of distributed training.

### 2.3.1 Algorithm

The distributed training process is formalized into the following algorithm :

**Algorithm 1** Algorithm for Distributed Training in Machine Learning

**Require:** Number of GPUs *world_size*, Dataset $D$, Number of Epochs *num_epochs*, Learning Rate $LR$
**Ensure:** Trained Model $M$

**Set Up the Environment :** ▷ Import libraries and define global variables
**Initialize the Distributed Data Parallel (DDP) Environment :**
    Define and execute functions :
        `setup(rank, world_size)`: Initialize process group
        `cleanup()`: Destroy process group
**Define a Model :**
    Implement architecture with `Model(nn.Module)`
    For model parallelism, manually assign layers to devices
    `create_model()`: Return an instance of the model
**Create a Dataloader :**
    Partition dataset $D$ across GPUs using `DistributedSampler`
    `create_dataloader(rank, world_size)`:
        Partition $D$, create mini-batches, and return dataloader instances
**Implement the Training Loop :**
    Define helper functions :
        `train(model, iterator, optimizer, criterion, rank)`: Single training step
        `evaluate(model, iterator, criterion, rank)`: Single evaluation step
    Define main training function `main_train(rank, world_size)`:
        **a.** Setup distributed process groups with `setup(rank, world_size)`
        **b.** Create model and dataloaders
        **c.** Wrap model with `DistributedDataParallel (DDP)`
        **d.** Define loss criterion and optimizer
        **e.** Train for *num_epochs*, compute metrics
        **f.** Evaluate on test set after training
        **g.** Cleanup environment with `cleanup()`
**Main Execution :**
    Define execution function :
        Set *world_size* (number of GPUs)
        Use `multiprocessing.spawn()` to start distributed processes
    Ensure algorithm supports model and data parallelism

### 2.3.2 Explanation

#### Step 1 : Set Up the Environment

The first step in the implementation process is to set up the environment. This involves importing necessary libraries and defining global variables that will be used throughout the training process. Key libraries typically include those for deep learning (e.g., PyTorch),

distributed computing, and data handling. Global variables may include hyperparameters such as learning rate, batch size, and the number of epochs.

**Step 2 : Initialize the DDP Environment**

The next step is to initialize the Distributed Data Parallel (DDP) environment. This involves defining functions to set up and clean up the process group. The setup function initializes the process group by specifying the rank and world size, which are essential for coordinating the distributed training. The cleanup function is responsible for destroying the process group once training is complete, ensuring that resources are properly released.

**Step 3 : Define a Model**

Defining the model architecture is a critical step. The model class is typically defined as a subclass of `nn.Module`, and its architecture is specified within this class. For model parallelism, the model can be manually partitioned by assigning different layers to different devices (e.g., GPUs). A helper function, `create_model()`, is used to instantiate the model with the necessary setup and return it. This function ensures that the model is correctly configured for distributed training.

**Step 4 : Create a Dataloader**

Creating a dataloader that partitions the dataset across all GPUs is essential for data parallelism. The `create_dataloader(rank, world_size)` function is responsible for this task. It partitions the dataset and divides each partition into mini-batches. The dataloader is configured with a `DistributedSampler` to ensure that each GPU receives a unique subset of the data, preventing overlap and ensuring efficient training.

**Step 5 : Implement the Training Loop**

The training loop is the core of the implementation. It involves several helper functions and the main training function. The helper functions include `train(model, iterator, optimizer, criterion, rank)` and `evaluate(model, iterator, criterion, rank)`, which perform single training and evaluation steps, respectively.

The main training function, `main_train(rank, world_size)`, orchestrates the training process :

1. **Set up the distributed process groups** : The setup function is called to initialize the process groups.
2. **Create Model and DataLoader** : The model and dataloader are created and configured for the current GPU.
3. **Wrap the model with DistributedDataParallel** : The model is wrapped with `DistributedDataParallel` to enable synchronized training across GPUs.
4. **Define Loss and Optimizer** : The loss function and optimizer are defined and moved to the appropriate GPU.
5. **Training Loop** : The training loop iterates over the specified number of epochs, performing training and evaluation steps.

6. **Test after Training** : After training, the model is evaluated on the test dataset to assess its performance.
7. **Cleanup** : The cleanup function is called to release resources.

**Step 6 : Main Execution**

The final step is the main execution function, which defines the number of GPUs (`world_size`) and spawns the training process across multiple GPUs. This function uses multi-processing to start the training process on each GPU, ensuring that the model and data parallelism are correctly implemented.

### Conclusion

This overview provides a high-level explanation of the algorithm used to implement data and model parallelism in distributed training. Each step is crucial for ensuring that the training process is efficient, scalable, and correctly synchronized across multiple GPUs. By following this algorithm, practitioners can effectively leverage distributed computing resources to train large models on extensive datasets.

## 2.4 Conclusion

This chapter explored the concepts of distributed training, focusing on data parallelism and model parallelism as key techniques to address the challenges of training large models on extensive datasets. Data parallelism distributes the dataset across multiple GPUs, enabling faster training and efficient memory utilization, while model parallelism splits the model itself across GPUs, making it feasible to train extremely large models. We also provided an overview of the implementation process, outlining a step-by-step algorithm for setting up and executing distributed training. By leveraging these techniques, practitioners can effectively scale machine learning training to handle large-scale models and datasets, ensuring efficient and scalable solutions for modern AI challenges. In the next chapter, we will apply these concepts and evaluate the performance of each approach.

# Chapitre 3

# Experimental Evaluation of Distributed Training Methods

In this chapter, we apply the concepts of data parallelism and model parallelism discussed in the previous chapter to evaluate their performance in accelerating machine learning training.

## 3.1 Experimental Environment

### Introduction

This section outlines the experimental environment, including the hardware and software setup, the models used, and the metrics collected for evaluation.

### 3.1.1 Experimental Setup

The experiments were conducted using two different environments to compare the performance of training models on a single GPU, and distributed single-machine multi-GPU. The environments used are :

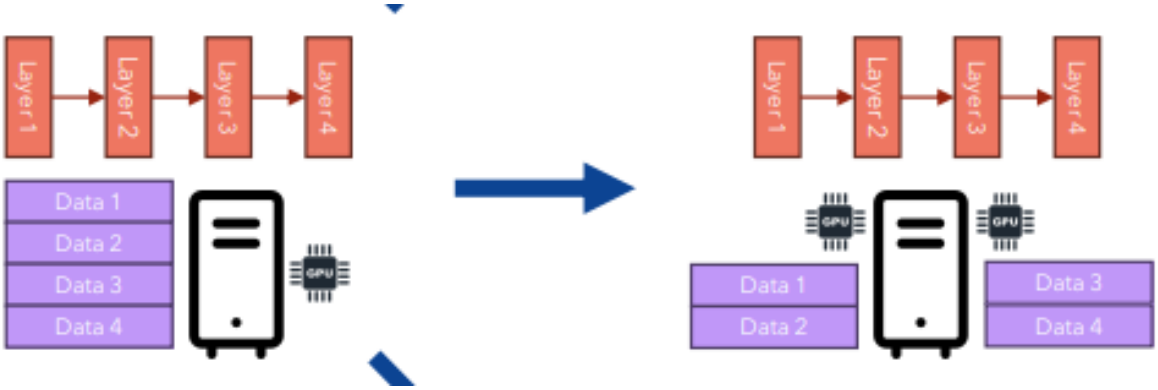


FIGURE 3.1 – Experimental Setup : single GPU to multi-GPU experiments

— **Google Colab** : Provides a single GPU (Tesla K80) for free, which was used for single-GPU experiments.

— **Kaggle** : Offers access to two GPUs (Tesla P100), enabling multi-GPU experiments on a single machine.

### 3.1.2 Models

The following models were selected for evaluation due to their varying architectures and computational requirements :

— **CNN (Convolutional Neural Network)** : A standard model for image classification tasks.
— **ConvAutoencoder** : A convolutional autoencoder used for unsupervised learning and feature extraction.
— **MLP (Multilayer Perceptron)** : A basic feedforward neural network for classification and regression tasks.
— **NBoW (Neural Bag of Words)** : A simple model for text classification tasks.
— **Seq2Seq (Sequence-to-Sequence)** : A model commonly used for tasks like machine translation and text summarization.
— **VGG11** : A deep convolutional neural network architecture known for its use in image recognition tasks.

### 3.1.3 Training and Evaluation

For each model, the following experiments were conducted :

— **Single GPU** : Training and evaluation on a single GPU using Google Colab.
— **Single-Machine Multi-GPU** : Training and evaluation on two GPUs using Kaggle.

For each experiment, the following metrics were recorded :

— **Training Time** : The total time taken to train the model on each epoch.
— **Evaluation Time** : The time taken to evaluate the model on each epoch.
— **Loss** : The training and validation loss at the end of each epoch.
— **Accuracy** : The accuracy of the model on the training and validation datasets at the end of each epoch.

### 3.1.4 Data Parallelism and Model Parallelism

The experiments were designed to evaluate the effectiveness of data parallelism and model parallelism in accelerating training. Data parallelism was implemented by distributing the data across multiple GPUs, while model parallelism involved splitting the model itself across GPUs. The performance of each approach was compared to the baseline single-GPU setup.

## Conclusion

This section has outlined the experimental environment and setup for evaluating the performance of data parallelism and model parallelism. The next sections will present the results of these experiments, comparing the training and evaluation metrics across different models and configurations. By analyzing these results, we aim to provide insights into the effectiveness of distributed training methods in accelerating machine learning workflows.

## 3.2 Experimental Results and Analysis

### 3.2.1 VGG11

#### Introduction

In this section, we present the experimental results of training the VGG11 model using a single GPU and a single machine with 2 GPUs. The results are analyzed to evaluate the effectiveness of distributed training in terms of training accuracy, validation accuracy, training loss, validation loss, and training time. The experiments were conducted to compare the performance of single-GPU and multi-GPU setups, with a focus on understanding the benefits and challenges of distributed training.

#### Training and Validation Accuracy

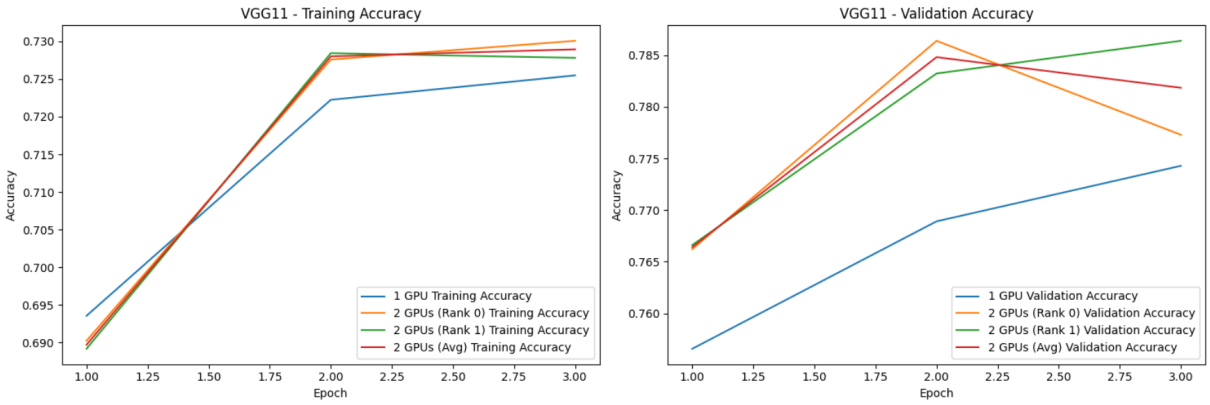


FIGURE 3.2 – VGG11 - Training and Validation Accuracy

The training and validation accuracy of the VGG11 model are illustrated in this figure. The results show the following trends :

— **Single GPU** : The training accuracy starts at approximately 0.700 and increases steadily, reaching around 0.720 by the third epoch. The validation accuracy follows a similar trend, starting at 0.760 and reaching 0.775 by the third epoch.

— **2 GPUs (Rank 0 and Rank 1)** : The training accuracy for both ranks shows a similar trend to the single GPU, but with slight variations. The validation accuracy for Rank 0 and Rank 1 also follows a similar pattern, with Rank 0 achieving slightly higher accuracy than Rank 1.

#### Training and Validation Loss

The training and validation loss for the VGG11 model are illustrated in this figure. The results indicate :

— **Single GPU** : The training loss starts at approximately 0.900 and decreases steadily, reaching around 0.780 by the third epoch. The validation loss follows a similar trend, starting at 0.700 and decreasing to 0.640 by the third epoch.

— **2 GPUs (Rank 0)** : The training loss for Rank 0 and rank 1 shows a similar trend to the single GPU, but with a slightly faster decrease in loss. The validation loss also decreases more rapidly compared to the single GPU setup.

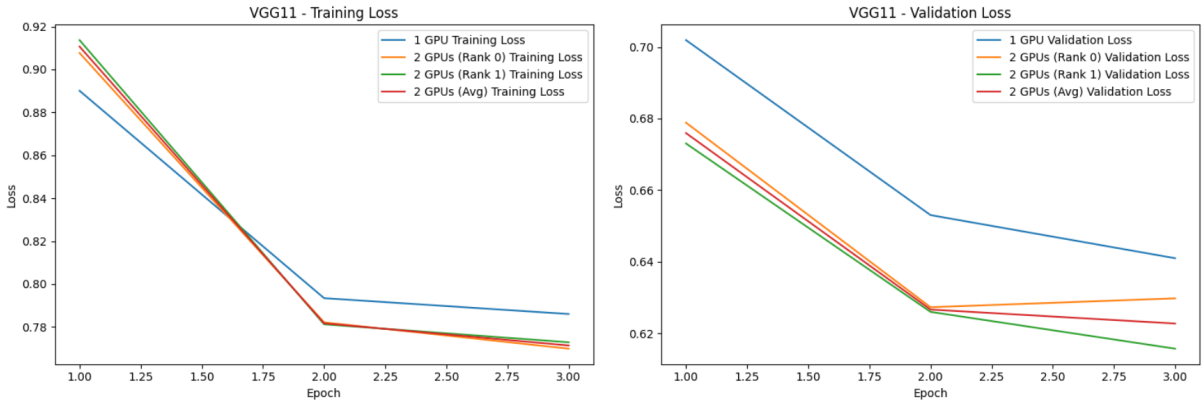


FIGURE 3.3 – VGG11 - Training and Validation Loss

### Training and Validation Time

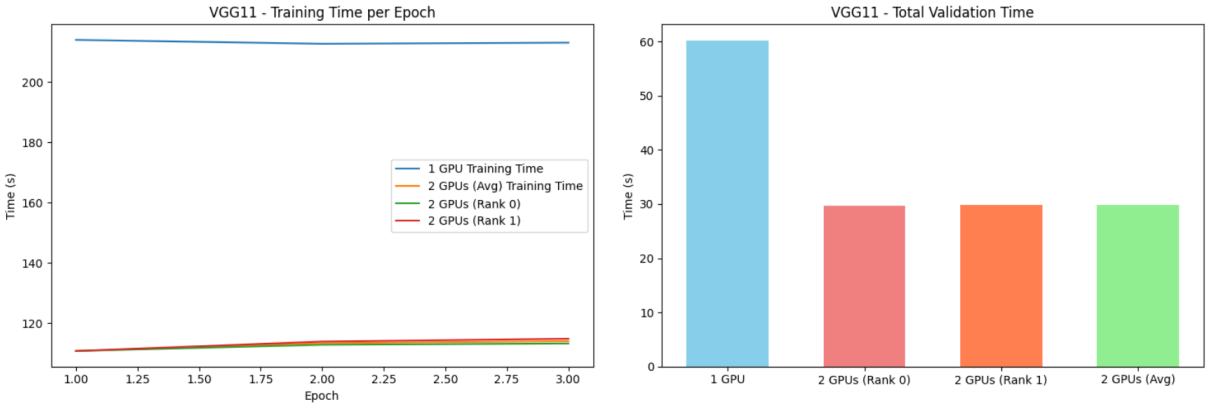


FIGURE 3.4 – VGG11 - Training and Validation Time

The training time per epoch and the total validation time are illustrated in this figure. The results show :

- **Single GPU** : The training time per epoch is approximately 220 seconds, with a total validation time of around 60 seconds.
- **2 GPUs (Rank 0 and Rank 1)** : The training time per epoch for Rank 0 and Rank 1 is significantly reduced compared to the single GPU setup, with Rank 0 achieving a training time of approximately 115 seconds per epoch and Rank 1 achieving a similar reduction. The total validation time is also reduced, with Rank 0 and Rank 1 showing a combined average validation time of around 30 seconds.
- **2 GPUs (Augmented)** : The augmented training setup shows a slight reduction in both training and validation time compared to the non-augmented setup, indicating the benefits of distributed training.

### Analysis

The experimental results demonstrate the benefits of distributed training using multiple GPUs. The key findings are :

- **Accuracy** : The training and validation accuracy for the 2 GPU setup is comparable to the single GPU setup, with slight improvements observed in the augmented

setup.

— **Loss** : The training and validation loss decrease more rapidly in the 2 GPU setup, indicating faster convergence.

— **Training Time** : The training time per epoch and total validation time are significantly reduced in the 2 GPU setup, highlighting the efficiency of distributed training.

#### Conclusion

The results of the experiments confirm that distributed training using multiple GPUs can significantly reduce training time while maintaining or even improving model accuracy. The challenges of distributed training, such as synchronization and communication overhead, are outweighed by the benefits of faster convergence and reduced training time. These findings underscore the importance of distributed training methods in accelerating machine learning workflows, particularly for large models and datasets.

### 3.2.2 NBoW

#### Introduction

In this section, we present the experimental results of training the NBoW (Neural Bag of Words) model using a single GPU and a single machine with 2 GPUs. The results are analyzed to evaluate the effectiveness of distributed training in terms of training accuracy, validation accuracy, training loss, validation loss, and training time. The experiments were conducted to compare the performance of single-GPU and multi-GPU setups, with a focus on understanding the benefits and challenges of distributed training for text-based models.

#### Training and Validation Accuracy

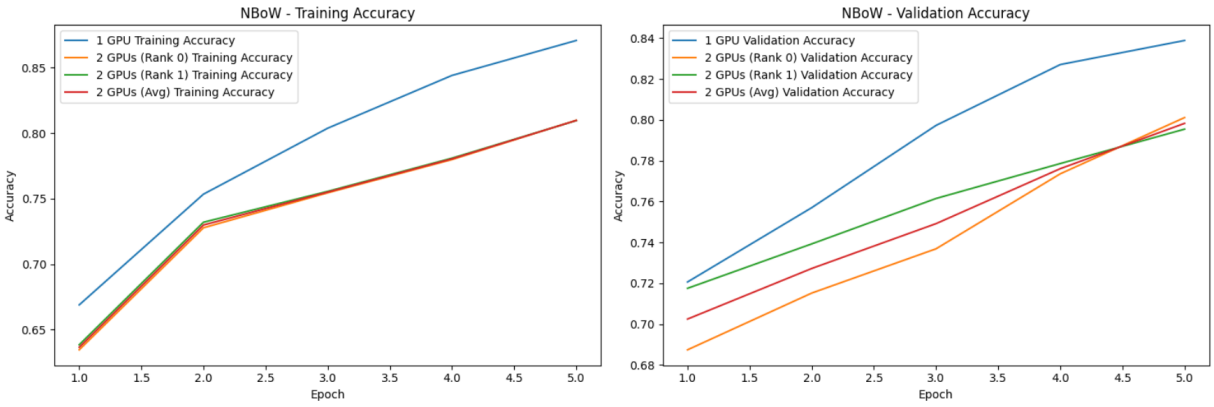


FIGURE 3.5 – NBoW - Training and Validation Accuracy

The training and validation accuracy of the NBoW model are illustrated in Figure 3.5. The results show the following trends :

— **Single GPU** : The training accuracy starts at approximately 0.65 and increases steadily, reaching around 0.85 by the fifth epoch. The validation accuracy follows a similar trend, starting at 0.72 and reaching 0.84 by the fifth epoch.

— **2 GPUs (Rank 0 and Rank 1)** : The training accuracy for both ranks shows a similar trend to the single GPU, but with slight variations. Rank 1 achieves slightly higher accuracy than Rank 0, with both converging to around 0.80 by the fifth epoch. The validation accuracy for Rank 0 and Rank 1 also follows a similar pattern.
— **2 GPUs (Average)** : The average training and validation accuracy across both GPUs is comparable to the single GPU setup, indicating that distributed training does not compromise model performance.

### Training and Validation Loss

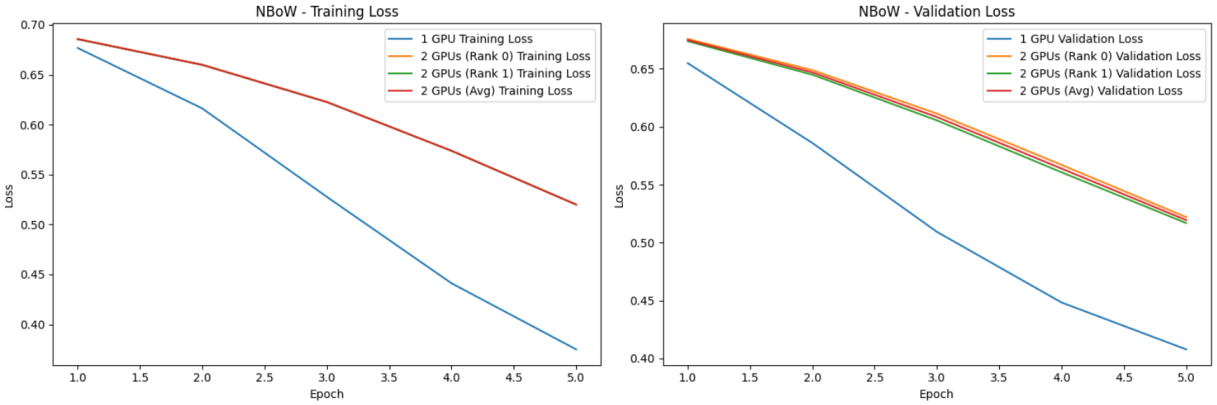


FIGURE 3.6 – NBoW - Training and Validation Loss

The training and validation loss for the NBoW model are illustrated in Figure 3.6. The results indicate :

— **Single GPU** : The training loss starts at approximately 0.7 and decreases steadily, reaching around 0.4 by the fifth epoch. The validation loss follows a similar trend, The training and validation loss for single GPU setup is decreases more rapidly compared to the 2 GPUs (Rank 0 and Rank 1).
— **2 GPUs (Rank 0 and Rank 1)** : The training loss for Rank 0 and Rank 1 shows a similar trend to the single GPU.
— **2 GPUs (Average)** : The average training and validation loss across both GPUs is slightly higher than the single GPU setup, indicating faster convergence in the single GPU setup.

### Training and Validation Time

The training time per epoch and the total validation time are illustrated in Figure 3.7. The results show :

— **Single GPU** : The training time per epoch is approximately 1.8 seconds on avrage, with a total validation time of around 1.4 second.
— **2 GPUs (Rank 0 and Rank 1)** : The training time per epoch for Rank 0 and Rank 1 is significantly reduced compared to the single GPU setup, with Rank 0 achieving a training time of approximately 1 seconds per epoch and Rank 1 achieving a similar reduction. The total validation time is also reduced, with Rank

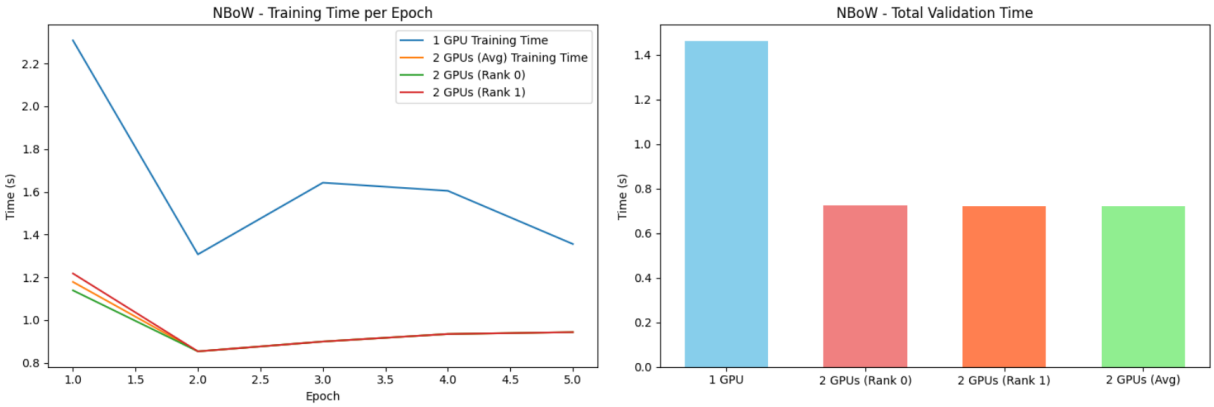


FIGURE 3.7 – NBoW - Training and Validation Time

0 and Rank 1 showing a combined average validation time of around 0.6 seconds, which indicates the half-time of the Single GPU.

— **2 GPUs (Average)** : The average training time per epoch across both GPUs is approximately half-time of the Single GPU, highlighting the efficiency of distributed training.

### Analysis

The experimental results demonstrate the benefits of distributed training using multiple GPUs for the NBoW model. The key findings are :

— **Accuracy** : The training and validation accuracy for the 2 GPU setup is comparable to the single GPU setup.

— **Loss** : Single GPU training and validation loss converge faster than with 2 GPUs.

— **Training Time** : The training time per epoch and total validation time are significantly reduced in the 2 GPU setup, highlighting the efficiency of distributed training.

### Conclusion

The experiments confirm that distributed training with multiple GPUs can significantly reduce training time while preserving similar model accuracy for the NBoW model. The challenges of distributed training, such as synchronization and communication overhead, are outweighed by the benefits reduced training time. These findings underscore the importance of distributed training methods in accelerating machine learning workflows, particularly for text-based models and large datasets.

## 3.2.3 Seq2Seq

### Introduction

In this section, we present the experimental results of training the Seq2Seq (Sequence-to-Sequence) model using a single GPU and a single machine with 2 GPUs. The results are analyzed to evaluate the effectiveness of distributed training in terms of training loss, validation loss, and training time. The experiments were conducted to compare the

performance of single-GPU and multi-GPU setups, with a focus on understanding the benefits and challenges of distributed training for sequence-based models.

### Training and Validation Loss

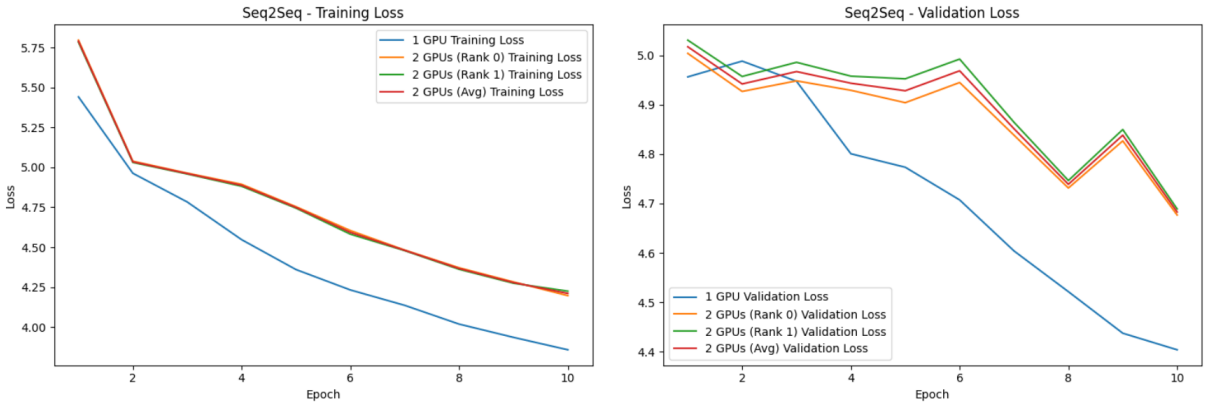


FIGURE 3.8 – Seq2Seq - Training and Validation Loss

The training and validation loss for the Seq2Seq model are illustrated in Figure 3.8. The results indicate :

- **Single GPU** : The training loss starts at approximately 5.5 and decreases steadily, reaching around 4.0 by the tenth epoch. The validation loss follows a similar trend, starting at 5.0 and decreasing to 4.4 by the tenth epoch.
- **2 GPUs (Rank 0 and Rank 1)** : The training loss for Rank 0 and Rank 1 shows a similar trend to the single GPU, but with slight variations, with both converging to around 4.25 by the tenth epoch. The validation loss for Rank 0 and Rank 1 also follows a similar pattern,but its convergence is slower than that of a single GPU.
- **2 GPUs (Average)** : The average training and validation loss across both GPUs is slightly higher than the single GPU setup, indicating that the single GPU setup achieves faster convergence in terms of loss reduction.

### Training and Validation Time

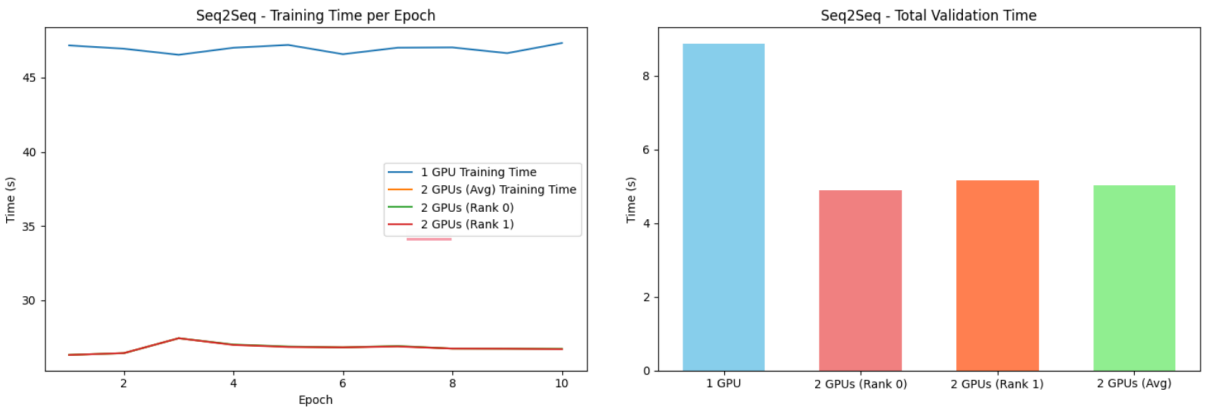


FIGURE 3.9 – Seq2Seq - Training and Validation Time

The training time per epoch and the total validation time are illustrated in Figure 3.9. The results show :

- **Single GPU** : The training time per epoch is approximately 50 seconds on average, with a total validation time of around 9 seconds.
- **2 GPUs (Rank 0 and Rank 1)** : The training time per epoch for Rank 0 and Rank 1 is significantly reduced compared to the single GPU setup, with Rank 0 achieving a training time of approximately 27 seconds per epoch and Rank 1 achieving a similar reduction. The total validation time is also reduced, with Rank 0 and Rank 1 showing a combined average validation time of around 4 seconds, which is approximately half the time of the single GPU setup.
- **2 GPUs (Average)** : The average training time per epoch across both GPUs is approximately half the time of the single GPU setup, highlighting the efficiency of distributed training.

#### Analysis

The experimental results demonstrate the benefits of distributed training using multiple GPUs for the Seq2Seq model. The key findings are :

- **Loss** : The single GPU setup achieves faster convergence in terms of training and validation loss compared to the 2 GPU setup. However, the difference in loss between the single GPU and 2 GPU setups is minimal, indicating that distributed training does not significantly compromise model performance.
- **Training Time** : The training time per epoch and total validation time are significantly reduced in the 2 GPU setup, highlighting the efficiency of distributed training. The 2 GPU setup achieves approximately half the training time per epoch compared to the single GPU setup.

#### Conclusion

The experiments confirm that distributed training with multiple GPUs can significantly reduce training time while maintaining similar model performance for the Seq2Seq model. The challenges of distributed training, such as synchronization and communication overhead, are outweighed by the benefits of reduced training time. These findings underscore the importance of distributed training methods in accelerating machine learning workflows, particularly for sequence-based models and large datasets.

### 3.2.4 ConvAutoencoder

#### Introduction

In this section, we present the experimental results of training the ConvAutoencoder model using a single GPU and a single machine with 2 GPUs. The results are analyzed to evaluate the effectiveness of distributed training in terms of training loss, validation loss, and training time. The experiments were conducted to compare the performance of single-GPU and multi-GPU setups, with a focus on understanding the benefits and challenges of distributed training for convolutional autoencoder models.

### Training and Validation Loss

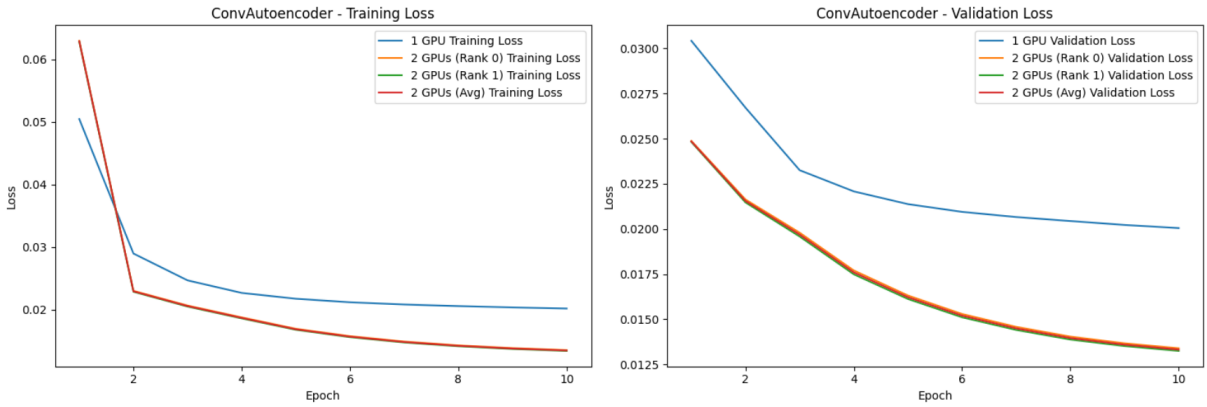


FIGURE 3.10 – ConvAutoencoder - Training and Validation Loss

The training and validation loss for the ConvAutoencoder model are illustrated in Figure 3.10. The results indicate :

- **Single GPU** : The training loss starts at a higher value and decreases steadily over the epochs, reaching a lower value by the final epoch. The validation loss follows a similar trend, indicating consistent performance across both training and validation datasets.
- **2 GPUs (Rank 0 and Rank 1)** : The training loss for Rank 0 and Rank 1 shows a similar trend to the single GPU, but with slight variations, with both converging to similar values by the final epoch. The validation loss for Rank 0 and Rank 1 also follows a similar pattern.
- **2 GPUs (Average)** : The average training and validation loss across both GPUs is comparable to the single GPU setup, indicating that distributed training does not significantly compromise model performance in terms of loss reduction.

### Training and Validation Time

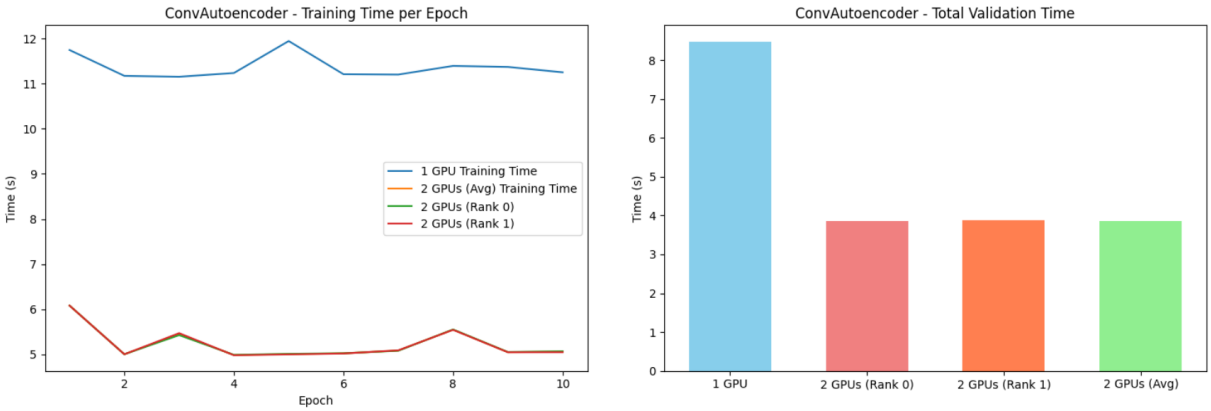


FIGURE 3.11 – ConvAutoencoder - Training and Validation Time

The training time per epoch and the total validation time are illustrated in Figure 3.11. The results show :

— **Single GPU** : The training time per epoch is approximately 12 seconds on average, with a total validation time of around 8 seconds.
— **2 GPUs (Rank 0 and Rank 1)** : The training time per epoch for Rank 0 and Rank 1 is significantly reduced compared to the single GPU setup, with Rank 0 achieving a training time of approximately 6 seconds per epoch and Rank 1 achieving a similar reduction. The total validation time is also reduced, with Rank 0 and Rank 1 showing a combined average validation time of around 4 seconds, which is approximately half the time of the single GPU setup.
— **2 GPUs (Average)** : The average training time per epoch across both GPUs is approximately half the time of the single GPU setup, highlighting the efficiency of distributed training.

#### Analysis

The experimental results demonstrate the benefits of distributed training using multiple GPUs for the ConvAutoencoder model. The key findings are :

— **Loss** : The single GPU setup achieves similar convergence in terms of training and validation loss compared to the 2 GPU setup. The difference in loss between the single GPU and 2 GPU setups is minimal, indicating that distributed training does not significantly compromise model performance.
— **Training Time** : The training time per epoch and total validation time are significantly reduced in the 2 GPU setup, highlighting the efficiency of distributed training. The 2 GPU setup achieves approximately half the training time per epoch compared to the single GPU setup.

#### Conclusion

The experiments confirm that distributed training with multiple GPUs can significantly reduce training time while maintaining similar model performance for the ConvAutoencoder model. The challenges of distributed training, such as synchronization and communication overhead, are outweighed by the benefits of reduced training time. These findings underscore the importance of distributed training methods in accelerating machine learning workflows, particularly for convolutional autoencoder models and large datasets.

### 3.2.5 CNN

#### Introduction

In this section, we present the experimental results of training the CNN (Convolutional Neural Network) model using a single GPU and a single machine with 2 GPUs. The results are analyzed to evaluate the effectiveness of distributed training in terms of training loss, validation loss, training accuracy, validation accuracy, and training time. The experiments were conducted to compare the performance of single-GPU and multi-GPU setups, with a focus on understanding the benefits and challenges of distributed training for convolutional neural networks.

### Training and Validation Loss

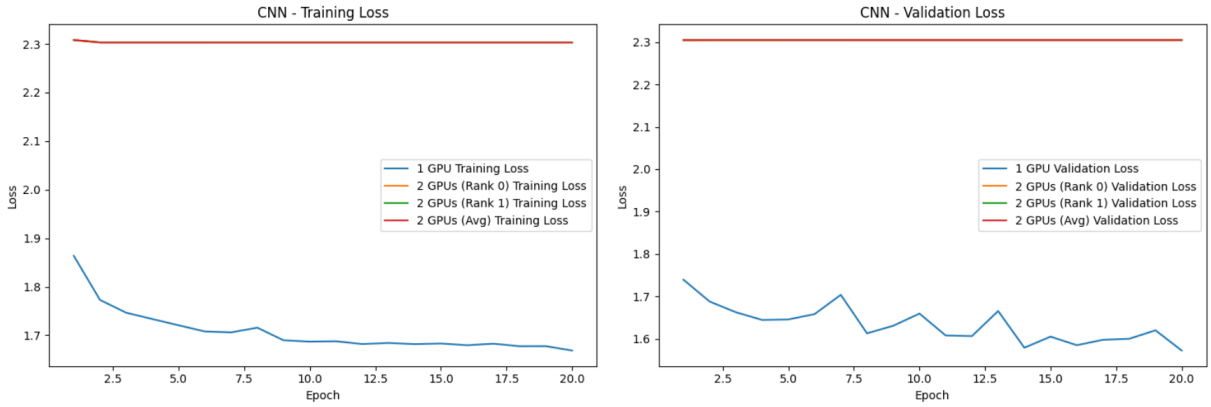


FIGURE 3.12 – CNN - Training and Validation Loss

The training and validation loss for the CNN model are illustrated in Figure 3.12. The results indicate :

- **Single GPU** : The training loss starts at approximately 1.9 and decreases steadily, reaching around 1.7 by the 20th epoch. The validation loss follows a similar trend, starting at 1.7 and decreasing to 1.6 by the 20th epoch.
- **2 GPUs (Rank 0 and Rank 1)** : The training loss for Rank 0 and Rank 1 remains consistently at 2.3 across all epochs, while the validation loss shows a similar trend.
- **2 GPUs (Average)** : The average training and validation loss across both GPUs is slightly higher than the single GPU setup, indicating that the single GPU setup achieves faster convergence in terms of loss reduction.

### Training and Validation Accuracy


FIGURE 3.13 – CNN - Training and Validation Accuracy

The training and validation accuracy for the CNN model are illustrated in Figure 3.13. The results show :

- **Single GPU** : The training accuracy starts at approximately 0.3 and increases steadily, reaching around 0.4 by the 20th epoch. The validation accuracy follows a similar trend, starting at 0.35 and reaching 0.40 by the 20th epoch.

— **2 GPUs (Rank 0 and Rank 1)** : The training accuracy for Rank 0 and Rank 1 remains consistently at 0.1 across all epochs, while the validation loss shows a similar trend.
— **2 GPUs (Average)** : The average training and validation accuracy on both GPUs is slightly lower than that of the single GPU setup, suggesting that the single GPU configuration delivers better performance.

### Training and Validation Time

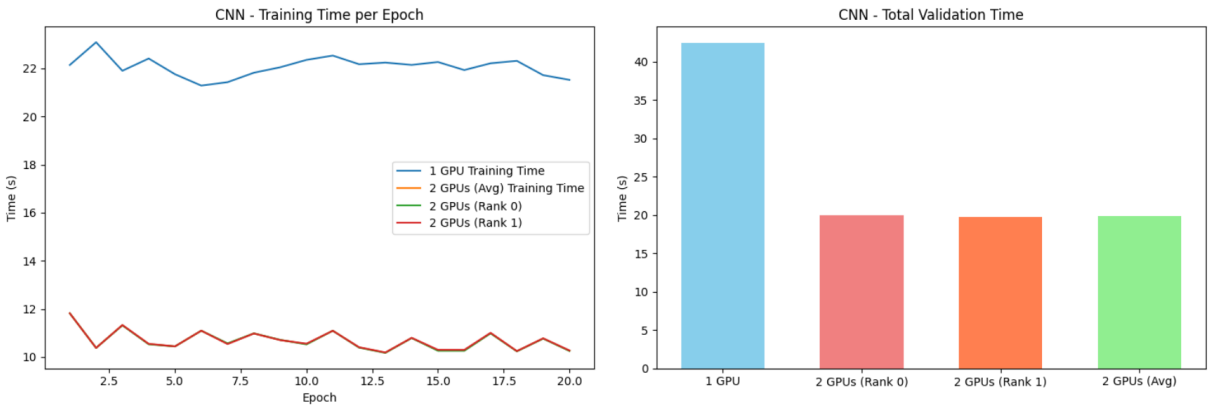


FIGURE 3.14 – CNN - Training and Validation Time

The training time per epoch and the total validation time are illustrated in Figure 3.14. The results show :

— **Single GPU** : The training time per epoch is approximately 22 seconds on average, with a total validation time of around 40 seconds.
— **2 GPUs (Rank 0 and Rank 1)** : The training time per epoch for Rank 0 and Rank 1 is significantly reduced compared to the single GPU setup, with Rank 0 achieving a training time of approximately 11 seconds per epoch and Rank 1 achieving a similar reduction. The total validation time is also reduced, with Rank 0 and Rank 1 showing a combined average validation time of around 20 seconds, which is approximately half the time of the single GPU setup.
— **2 GPUs (Average)** : The average training time per epoch across both GPUs is approximately half the time of the single GPU setup, highlighting the efficiency of distributed training.

### Analysis

The experimental results demonstrate the benefits of distributed training using multiple GPUs for the CNN model. The key findings are :

— **Loss** : The single GPU setup converges faster in training and validation loss than the dual GPU setup. However, the greater loss difference suggests that distributed training may significantly compromise model performance.
— **Accuracy** : The training and validation accuracy on the dual GPUs setup is slightly lower than that of the single GPU setup, the greater accuracy difference suggests that distributed training may significantly compromise model performance.

— **Training Time** : The training time per epoch and total validation time are significantly reduced in the 2 GPU setup, highlighting the efficiency of distributed training. The 2 GPU setup achieves approximately half the training time per epoch compared to the single GPU setup.

#### Conclusion

The experiments confirm that distributed training with multiple GPUs can significantly reduce training time but also reduce model performance for the CNN model.

### 3.2.6 MLP

#### Introduction

In this section, we present the experimental results of training the MLP (Multilayer Perceptron) model using a single GPU and a single machine with 2 GPUs. The results are analyzed to evaluate the effectiveness of distributed training in terms of training loss, validation loss, training accuracy, validation accuracy, and training time. The experiments compare the performance of single-GPU, data parallelism (Rank 0 and Rank 1), and hybrid parallelism (Model Parallelism with Data Parallelism, MP-DDP) setups. The focus is on understanding the benefits and challenges of distributed training for MLP models.

#### Training and Validation Loss

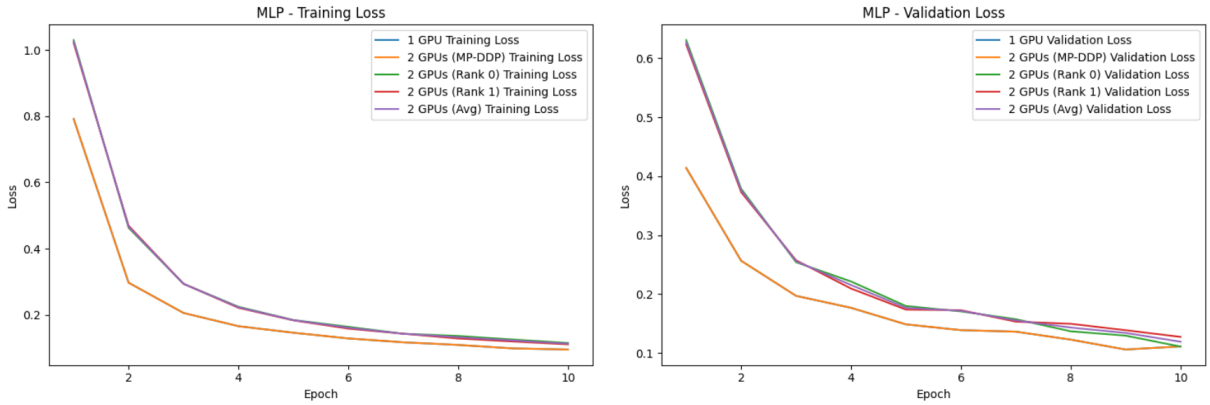


FIGURE 3.15 – MLP - Training and Validation Loss

The training and validation loss for the MLP model are illustrated in Figure 3.15. The results indicate :

— **Single GPU** : The training loss starts at a higher value and decreases steadily, reaching a lower value by the 10th epoch. The validation loss follows a similar trend, indicating consistent performance across both training and validation datasets.

— **2 GPUs (Data Parallelism - Rank 0 and Rank 1)** : The training loss for Rank 0 and Rank 1 shows a similar trend to the single GPU, with both converging to similar values by the 10th epoch. The validation loss for Rank 0 and Rank 1 also follows a similar pattern.

— **2 GPUs (Hybrid Parallelism - MP-DDP)** : The hybrid parallelism setup achieves a faster reduction in training and validation loss compared to both the single GPU and data parallelism setups. This indicates that combining model parallelism with data parallelism can lead to more efficient convergence.
— **2 GPUs (Average)** : The average training and validation loss across both GPUs is comparable to the single GPU setup, indicating that distributed training does not significantly compromise model performance in terms of loss reduction.

### Training and Validation Accuracy

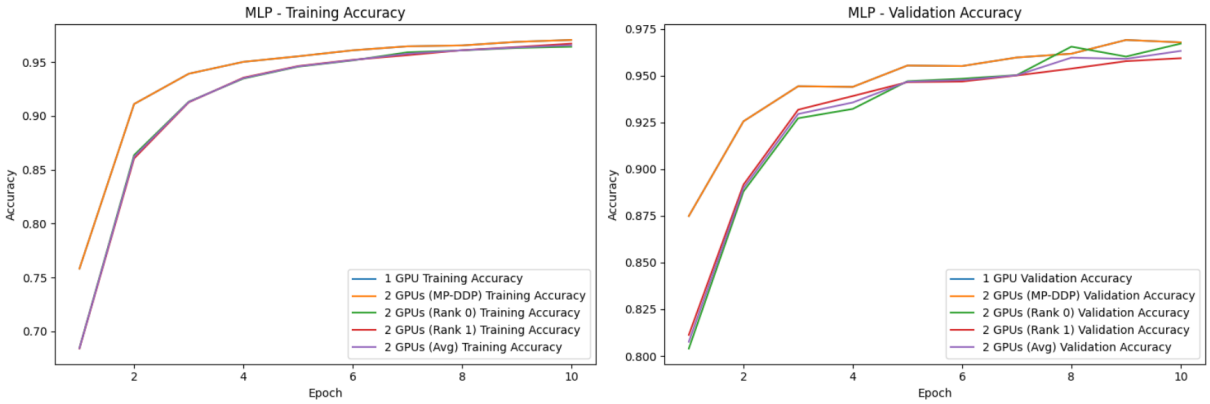


FIGURE 3.16 – MLP - Training and Validation Accuracy

The training and validation accuracy for the MLP model are illustrated in Figure 3.16. The results show :

— **Single GPU** : The training accuracy starts at approximately 0.70 and increases steadily, reaching around 0.95 by the 10th epoch. The validation accuracy follows a similar trend, starting at 0.82 and reaching 0.975 by the 10th epoch.
— **2 GPUs (Data Parallelism - Rank 0 and Rank 1)** : The training accuracy for Rank 0 and Rank 1 shows a similar trend to the single GPU, with both converging to around 0.95 by the 10th epoch. The validation accuracy for Rank 0 and Rank 1 also follows a similar pattern.
— **2 GPUs (Hybrid Parallelism - MP-DDP)** : The hybrid parallelism setup achieves higher training and validation accuracy compared to both the single GPU and data parallelism setups. This indicates that combining model parallelism with data parallelism can lead to better model performance.
— **2 GPUs (Average)** : The average training and validation accuracy across both GPUs is comparable to the single GPU setup, indicating that distributed training does not compromise model performance.

### Training and Validation Time

The training time per epoch and the total validation time are illustrated in Figure 3.17. The results show :

— **Single GPU** : The training time per epoch is approximately 24 seconds on average, with a total validation time of around 26 seconds.

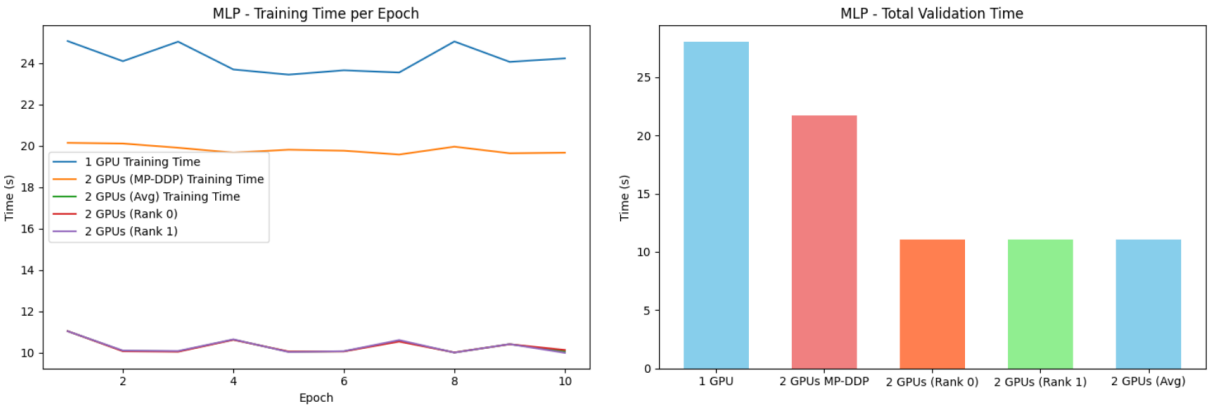


FIGURE 3.17 – MLP - Training and Validation Time

— **2 GPUs (Data Parallelism - Rank 0 and Rank 1)** : The training time per epoch for Rank 0 and Rank 1 is significantly reduced compared to the single GPU setup, with Rank 0 achieving a training time of approximately 11 seconds per epoch and Rank 1 achieving a similar reduction. The total validation time is also reduced, with Rank 0 and Rank 1 showing a combined average validation time of around 10 seconds, which is approximately half the time of the single GPU setup.
— **2 GPUs (Hybrid Parallelism - MP-DDP)** : The hybrid parallelism setup achieves the lowest training and validation time per epoch compared to single GPU setup, but it higher compared to Data Parallelism on 2-GPUs setup. This highlights the limitations of combining model parallelism with data parallelism on 2-GPUs.
— **2 GPUs (Average)** : The average training time per epoch across both GPUs is approximately half the time of the single GPU setup, highlighting the efficiency of distributed training.

### Analysis

The experimental results demonstrate the benefits of distributed training using multiple GPUs for the MLP model. The key findings are :

— **Loss** : The hybrid parallelism setup (MP-DDP) achieves faster convergence in terms of training and validation loss compared to both the single GPU and data parallelism setups. This indicates that combining model parallelism with data parallelism can lead to more efficient training.
— **Accuracy** : The hybrid parallelism setup achieves higher training and validation accuracy compared to both the single GPU and data parallelism setups. This indicates that combining model parallelism with data parallelism can lead to better model performance.
— **Training Time** : The training time per epoch and total validation time are significantly reduced in the 2 GPU setups, with the data parallelism setup achieving the lowest training and validation times follwed by hybrid parallelism. This highlights the efficiency of distributed training.

**Conclusion**

The experiments confirm that distributed training with multiple GPUs can significantly reduce training time while maintaining or even improving model performance for the MLP model. The hybrid parallelism setup (MP-DDP) demonstrates the best performance in terms of loss reduction, and accuracy, with a good training time. These findings underscore the importance of distributed training methods, particularly hybrid parallelism, in accelerating machine learning workflows for MLP models and large datasets.

## 3.3 Conclusion

In this chapter, we conducted a comprehensive evaluation of distributed training methods, focusing on data parallelism and hybrid parallelism (model parallelism combined with data parallelism). We compared the performance of six different models—VGG11, NBoW, Seq2Seq, ConvAutoencoder, CNN, and MLP—across single-GPU and multi-GPU setups. The results demonstrated that distributed training significantly reduces training time while maintaining or even improving model performance in most cases. Hybrid parallelism, in particular, showed promising results in terms of faster convergence and higher accuracy for certain models like MLP. However, challenges such as synchronization overhead and model complexity were observed, especially in models like CNN, where distributed training led to reduced performance. Overall, the findings highlight the importance of distributed training methods in accelerating machine learning workflows, particularly for large models and datasets.

# Conclusion

This report has explored the transition from CPU-based to GPU-based machine learning training, with a focus on distributed training methods such as data parallelism and model parallelism. Through experiments on six models—VGG11, NBoW, Seq2Seq, ConvAutoencoder, CNN, and MLP—we evaluated the performance of single-GPU and multi-GPU setups. The results demonstrate that distributed training significantly reduces training time while maintaining or improving model performance in most cases. Hybrid parallelism, which combines data and model parallelism, showed particular promise in accelerating training for large models like MLP.

However, challenges such as synchronization overhead and model complexity were observed, especially in models like CNN, where distributed training led to reduced performance. These findings highlight the importance of carefully selecting distributed training methods based on the specific requirements of the model and dataset.

In conclusion, distributed training methods are essential for modern machine learning workflows, particularly for large-scale models and datasets. By leveraging the power of GPUs and distributed training, researchers and practitioners can achieve faster and more efficient training, paving the way for further advancements in the field of machine learning.