INSTITUTO POLITÉCNICO DE BRAGANÇA
Escola Superior de Tecnologia e Gestão

Distributed AI training platform

Tiago Andrés Cerqueiro - a41783

Dissertation presented to the School of Technology and Management in the scope of the Master in Informatics.

Supervisors:

Prof. Rui Pedro Lopes

Prof. José Rufino

Bragança

October, 2025

Distributed AI training platform

Tiago Andrés Cerqueiro - a41783

Dissertation presented to the School of Technology and Management in the scope of the Master in Informatics.

Supervisors:

Prof. Rui Pedro Lopes

Prof. José Rufino

Bragança

October, 2025

Acknowledgment

Primeiramente quero agradecer aos meus orientadores, professor Rui Pedro Sanches de Castro Lopes e professor José Carlos Rufino Amaro pela ajuda, orientação, prontidão e paciência durante todo o processo. Agradeço também a todos os amigos que, ao longo destes anos, tornaram o percurso académico mais leve e enriquecedor, partilhando comigo momentos de companheirismo e motivação.

Por fim, deixo o meu profundo agradecimento à minha família, que sempre acreditou em mim e me apoiou incondicionalmente em todas as minhas decisões.

Abstract

Training large-scale artificial intelligence models has become a critical challenge in modern research, requiring distributed infrastructures capable of efficiently coordinating multiple devices. This dissertation presents a comparative analysis of three distributed deep learning training platforms: PyTorch Distributed Data Parallel (DDP), Apache Spark, and Determined AI, evaluating their performance, resource management capabilities, and usability in organizational environments. The methodology involved implementing and testing each framework on a three-node cluster equipped with NVIDIA GPUs, using the BERT-tiny model for sentiment classification on the IMDB dataset. Quantitative metrics of training time, model accuracy, and scaling efficiency were collected, complemented by qualitative evaluation of configuration complexity, orchestration features, and developer experience. Results demonstrate that PyTorch DDP offers the best absolute performance, completing 20 epochs of training in 499 seconds with 2 GPUs, while Determined AI introduces a 21% overhead but provides superior cluster management capabilities, including automatic scheduling, experiment tracking, and fault tolerance. Apache Spark presents significant overhead (187%) but integrates naturally into existing data processing pipelines. Framework selection depends on context: DDP is ideal for individual researchers prioritizing speed, Determined AI suits shared environments requiring reproducibility and centralized management, and Spark serves scenarios where training is integrated into broader big data workflows.

Keywords: Distributed Training, Deep Learning, Machine Learning, Parallel Computing

Resumo

O treino de modelos de inteligência artificial de grande escala tornou-se um desafio crítico para a investigação moderna, exigindo infraestruturas distribuídas capazes de coordenar múltiplos dispositivos de forma eficiente. Esta dissertação apresenta uma análise comparativa de três plataformas de treino distribuído de deep learning: PyTorch Distributed Data Parallel (DDP), Apache Spark e Determined AI, avaliando o seu desempenho, capacidades de gestão de recursos e usabilidade em ambientes organizacionais. A metodologia envolveu a implementação e teste de cada framework num cluster de três nós equipados com GPUs NVIDIA, utilizando o modelo BERT-tiny para classificação de sentimentos no dataset IMDB. Foram recolhidas métricas quantitativas de tempo de treino, precisão do modelo e eficiência de escalabilidade, complementadas por avaliação qualitativa de complexidade de configuração, funcionalidades de orquestração e experiência de desenvolvimento. Os resultados demonstram que PyTorch DDP oferece o melhor desempenho absoluto, completando 20 épocas de treino em 499 segundos com 2 GPUs, enquanto Determined AI introduz um overhead de 21% mas fornece capacidades superiores de gestão de cluster, incluindo escalonamento automático, tracking de experiências e tolerância a falhas. Apache Spark apresenta overhead significativo (187%) mas integra-se naturalmente em pipelines de processamento de dados existentes. A escolha do framework depende do contexto: DDP é ideal para investigadores individuais que priorizam velocidade, Determined AI adequa-se a ambientes partilhados que requerem reprodutibilidade e gestão centralizada, e Spark serve cenários onde o treino está integrado em workflows de big data mais amplos.

Palavras-chave: Treino Distribuído, Deep Learning, Machine Learning, Computação Paralela

Acknowledgement of use of AI tools

I acknowledge the use of ChatGPT4 (OpenAI, https://chatgpt.com/) and Claude (Antropic, https://claude.ai) for translation and text improvement, both semantically and syntactically.

Contents

List of Tables

List of Figures

Acronyms

AI Artificial Intelligence.

API Application Programming Interface.

CLI Command Line Interface.

CPU Central Processing Unit.

cuDNN CUDA Deep Neural Network library.

DDP Distributed Data Parallel.

DML Distributed Machine Learning.

FIFO First In, First Out.

FSDP Fully Sharded Data Parallel.

GPU Graphics Processing Unit.

HPC high-performance computing.

IMDB Internet Movie Database.

ML Machine Learning.

NCCL NVIDIA Collective Communications Library.

RAM Random Access Memory.

RDD Resilient Distributed Dataset.

SSH Secure Shell.

UI User Interface.

ZeRO Zero Redundancy Optimizer.

Chapter 1

Introduction

Training Artificial Intelligence (AI) models has become a central driver of progress in modern technological innovation. As models grow in size and complexity, their development no longer depends solely on advances in architecture design or algorithmic improvements, but increasingly on the efficiency and scalability of the training process itself. Large-scale deep learning models require vast computational resources and extended training times, presenting significant challenges when confined to a single machine or a single GPU.

Distributed training has emerged as a critical solution to these challenges, enabling the parallelization of computations across clusters of multiple nodes and devices. By coordinating workloads in a distributed manner, training times can be reduced, larger models can be supported, and overall efficiency can be improved. However, the effectiveness of distributed training depends not only on the underlying hardware but also on the software platforms and organizational frameworks that orchestrate computation, communication, and resource management across clusters. Efficient scheduling, resource allocation, and fault tolerance mechanisms are essential to ensure that multi-node systems operate coherently and without bottlenecks.

This work addresses these challenges by examining three distinct approaches to distributed deep learning training: i) PyTorch DDP [1], a framework-native solution offering fine-grained control and efficiency; ii) Determined AI [2], a specialized orchestration platform that abstracts complexity while enhancing reproducibility, experiment management,

and cluster scheduling; and iii) Apache Spark [3], a big data processing framework increasingly exploited for machine learning workloads due to its scalability and dynamic resource management capabilities. By conducting a systematic comparison of these platforms, this thesis seeks to highlight their trade-offs in terms of performance, scalability, and usability, and to provide practical insights into the selection of distributed training strategies and organizational frameworks for AI model development.

1.1 Context

The rapid evolution of deep learning has been characterized not only by the growing sophistication of neural network architectures but also by the unprecedented scale of data required for their training. As a result, the computational burden associated with training state-of-the-art models has increased dramatically, placing new demands on both hardware and software infrastructures. While high-performance GPUs have become essential for accelerating training, a single device is rarely sufficient for modern workloads. Distributing computations across multiple machines and GPUs has therefore become a necessity rather than an option.

Within this scenery, the management and organization of computational resources have become as important as the training algorithms themselves. Distributed machine learning relies on systems that can coordinate clusters efficiently, schedule tasks dynamically, and ensure resource utilization remains optimal. Cluster management, monitoring, and orchestration tools play a central role in enabling reproducible and scalable AI research environments.

A variety of distributed training solutions have emerged, each embodying a different philosophy. PyTorch DDP represents the baseline approach: a low-level, framework-integrated mechanism that provides direct control over distributed training while requiring careful manual configuration of the cluster environment. At a higher level of abstraction, Determined AI introduces a managed platform tailored for deep learning, offering experiment tracking, fault tolerance, and automated orchestration of distributed workloads.

In contrast, Apache Spark, originally designed for large-scale data analytics, extends its mature distributed computing engine to machine learning scenarios, offering robust scalability and resource scheduling capabilities at the cost of additional integration effort.

These three approaches reflect the diversity of strategies available, from direct control over communication primitives to specialized deep learning orchestration platforms and general-purpose big data frameworks. Understanding not only their computational performance but also their cluster-level organization, management overhead, and integration complexity is essential for guiding informed choices in real-world applications.

1.2 Motivation and Problem Statement

As deep learning systems continue to grow in scale and complexity, the selection of an appropriate distributed training framework has become a key decision in both academic research and industrial practice. A framework that achieves optimal raw performance may require complex manual setup, while one that simplifies orchestration and scheduling may trade off some efficiency in exchange for ease of use and manageability.

Understanding these trade-offs, between performance, scalability, and usability, is crucial for selecting the most suitable approach for a given computational environment. Moreover, from an organizational perspective, efficient orchestration systems can significantly reduce infrastructure overhead and improve experiment reproducibility. This balance between raw computational efficiency and system-level management is at the heart of modern distributed AI workflows.

1.3 Objectives

The main objective of this work is to perform a comparative analysis of distributed deep learning training across three representative frameworks: i) PyTorch DDP, a baseline, framework-native method offering fine-grained control and high efficiency; ii) Determined

AI, a specialized orchestration platform that automates experiment management, scheduling, and fault tolerance; iii) Apache Spark, a general-purpose distributed computing engine increasingly adapted for machine learning workloads due to its scalability and mature cluster management capabilities.

By conducting experiments under equivalent conditions, this study aims to evaluate how each of these systems performs in terms of a) Training performance, b) Cluster organization and resource management, and c) Usability and user experience.

Thus, the following main research questions guide the analysis:

- Which framework achieves the fastest training time under comparable setups?
- Which provides the most effective cluster management and scheduling capabilities?
- Which offers the most user-friendly configuration and workflow integration?

Since the objective of this work is to compare distributed training frameworks rather than to optimize model architecture, data parallelism was selected as the main training strategy. This ensures a fair and consistent basis for comparison, as it is supported natively by DDP, Spark, and Determined AI.

1.4 Structure of the document

The remaining of this document is organized into five chapters, as follows: Chapter 2 reviews the state of the art in distributed machine learning, covering parallelism strategies, distributed frameworks and libraries, and resource management tools; Chapter 3 describes the experimental methodology, including infrastructure setup, framework installation procedures, and evaluation metrics; Chapter 4 details the implementation specifics for each framework, including model configuration, training hyperparameters, and distributed training strategies; Chapter 5 presents the experimental results and comparative analysis across performance, organizational, and usability dimensions; finally, Chapter 6 concludes with research contributions, assessment of findings, and future work directions.

Chapter 2

Distributed ML Training

Machine Learning (ML) has established itself as one of the most influential areas of AI, powering applications ranging from recommendation systems and computer vision to natural language processing and autonomous driving [4]. As models become increasingly sophisticated and powerful, their development requires not only larger volumes of data but also significantly more computational resources.

In recent years, particularly with the rise of deep learning, model complexity has grown exponentially. Models with millions or even billions of parameters have become commonplace, reflecting the demand for greater representational capacity and higher accuracy in challenging tasks. However, this growth introduces critical difficulties:

- Limited memory: a single device is often insufficient to store both the model and the training data [5].
- Processing time: even with high-performance GPUs, training complex models can take days or weeks [6].
- Reduced scalability: simply adding more computing power to a single machine quickly becomes inefficient or financially unfeasible [7].

The traditional approach of performing training in a centralized manner (on a single machine or node within a cluster) therefore proves inadequate for modern requirements.

In this context, Distributed Machine Learning (DML) emerges as a fundamental solution.

DML enables the distribution of training across multiple computational resources, using parallelism to accelerate execution, improve scalability, and allow for the training of larger and more complex models. Beyond reducing the time needed to reach convergence, distributed training offers flexibility in resource management, adapting to both high-performance computing (HPC) environments and cloud-based infrastructures.

There are multiple approaches to DML, which differ in how computations and data are partitioned across participating nodes. Some strategies focus on dividing the dataset, others on splitting the model itself, while more advanced methods combine these techniques or introduce further optimizations.

2.1 Types of ML Parallelism

DML relies on different parallelism strategies to split workloads across computational resources. The main approaches include data parallelism, model parallelism (with pipeline and tensor parallelism variants), hybrid parallelism, and more recent optimizations such as the Zero Redundancy Optimizer (ZeRO). Each of these methods addresses different challenges related to memory usage, communication overhead, and model scalability.

2.1.1 Data Parallelism

Data parallelism is the most commonly used approach in distributed training due to its relative simplicity and efficiency. In this strategy, a full replica of the model is stored on each device (e.g., GPU), while the training dataset is divided into smaller subsets, as shown in Figure 2.1. Each worker processes a mini-batch of data independently during the forward and backward passes, computing gradients locally. At the end of each iteration, gradients are aggregated across workers, commonly through an all-reduce operation, and the model parameters are updated synchronously to maintain consistency.

For example, in a configuration with three GPUs, training a ResNet-50 [8] model with a batch size of 120 could be parallelized by assigning 40 samples to each GPU. Each

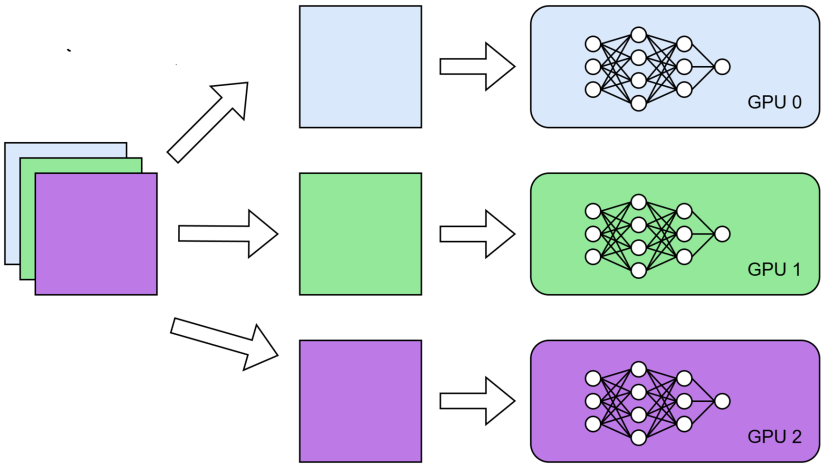

Figure 2.1: Data Parallelism.

device computes its gradients independently, which are then averaged across all GPUs before updating the weights [9] [10]. This ensures that the training trajectory matches that of centralized training, while execution is significantly accelerated.

Data parallelism is particularly effective for medium to large models that fit into the memory of a single device, such as ResNet or BERT [11]. Popular implementations include PyTorch DDP [1] and TensorFlow MirroredStrategy [12], which are widely used in both academic and industrial settings.

2.1.2 Model Parallelism

While data parallelism works well when the entire model can fit on each device, this assumption fails for very large-scale models that exceed the memory capacity of a single GPU. In such cases, model parallelism becomes necessary. Instead of replicating the model across devices, this approach splits the model itself, assigning different portions of the architecture to different workers. Model parallelism is achived in two ways (pipeline parallelism and tensor parallelism), as next described.

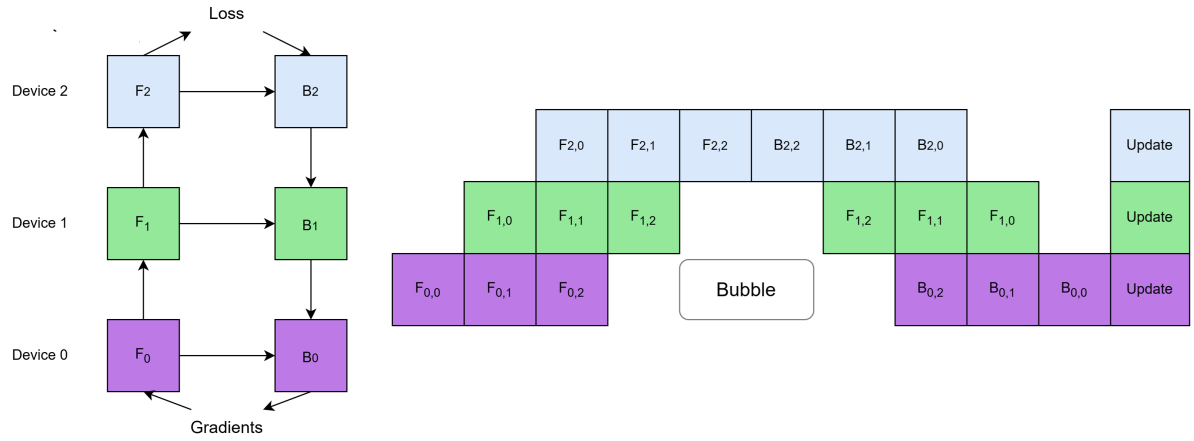

Figure 2.2: Pipeline Parallelism.

Pipeline Parallelism

Pipeline parallelism divides the model into sequential stages, much like an assembly line. Each stage of the network (e.g., a group of layers) is placed on a different device. Input mini-batches are further split into micro-batches that are passed through the pipeline, enabling multiple stages to process different micro-batches simultaneously .

As an example, consider a 12-layer Transformer model divided into 3 stages, each stage with 4 layers and allocated to a separate GPU. While the 1st stage processes micro-batch 1, the second stage can process micro-batch 0, and so on. This improves hardware utilization compared to naively splitting the model, though it introduces “pipeline bubbles” (periods where some devices are idle at the start and end of training iterations), as shown in Figure 2.2. Frameworks such as GPipe [13] and DeepSpeed implement this technic.

Tensor Parallelism

Tensor parallelism, in contrast, splits operations within a layer across devices, as demonstrated in Figure 2.3 . For instance, in a fully connected layer with a large weight matrix of size 4096 x 4096, the computation can be divided among four GPUs, each responsible for a 1024 x 4096 partition. This allows models with extremely large layers to be trained

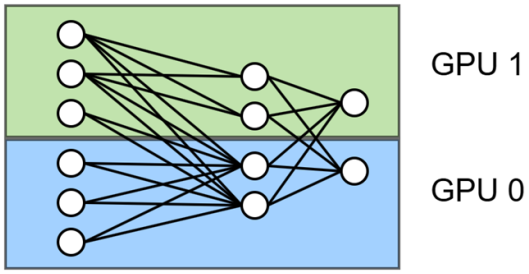

Figure 2.3: Tensor Parallelism.

efficiently by distributing the computational load .

Tensor parallelism is especially important in very large language models (LLMs), where the size of individual layers can exceed the capacity of a single device. Frameworks such as NVIDIA Megatron-LM rely heavily on tensor parallelism to train models with hundreds of billions of parameters, such as the Megatron-Turing NLG [14].

2.1.3 Hybrid Parallelism

For extremely large-scale models, neither data parallelism nor model parallelism alone is sufficient. Hybrid parallelism combines multiple strategies (often data parallelism with either pipeline or tensor parallelism) to balance scalability and memory efficiency.

A common configuration is to divide a trillion-parameter model into multiple pipeline stages, with each stage further parallelized across several GPUs using tensor parallelism, while data parallelism is applied across multiple nodes. For instance, a model could be split into 8 pipeline stages, each stage distributed across 4 GPUs with tensor parallelism, and then replicated across 64 nodes with data parallelism. This hierarchical structure enables the training of cutting-edge models such as GPT-4 [15] or Anthropic's Claude.

DeepSpeed [16] and PyTorch Fully Sharded Data Parallel (FSDP) [17] also support hybrid approaches, combining sharding and parallelism strategies in an unified interface.

2.1.4 Zero Redundancy Optimizer (ZeRO)

ZeRO [18] represents a major advancement in the scalability of data-parallel training. Traditional data parallelism replicates not only the model but also optimizer states and gradients on each device, leading to significant memory overhead. ZeRO, introduced as part of Microsoft's DeepSpeed library [16], partitions these elements across devices, thereby eliminating redundancy.

ZeRO is implemented in 3 stages of increasing partitioning granularity: at Stage 1, optimizer states are partitioned; at Stage 2, gradients are also partitioned; and at Stage 3, parameters themselves are sharded across devices. For example, in an 8-GPU system with ZeRO Stage 3, parameters, gradients, and optimizer states are evenly distributed among GPUs, enabling the training of models that would otherwise exceed device memory limits.

This approach has enabled breakthroughs in training trillion-parameter models, as it significantly reduces memory consumption while maintaining training efficiency. ZeRO is often combined with hybrid parallelism strategies to maximize both speed and scalability.

To summarize these strategies, Table 2.1 highlights the main types of parallelism, their goals, and representative use cases. This overview shows how each technique contributes to machine learning scaling: from data parallelism applied in models like ResNet-50, to hybrid parallelism and ZeRO enabling training of trillion-parameter models as GPT-4.

Table 2.1: Characterization of different types of ML parallelism

Type	**Splitting Strategy**	**Goal**	**Use Cases**
Data Parallelism	Data batches	Accelerate training	ResNet-50 on ImageNet
Pipeline Parallelism	Model layers	Fit large models	GPT-3
Tensor Parallelism	Layer operations	Reduce memory per device	Megatron-Turing NLG
Hybrid Parallelism	Data + model	Scale model and data	GPT-4, Claude
ZeRO	Optimizer states	Memory efficiency	Training 20B+ models

2.2 Distributed ML Frameworks and Libraries

The rapid growth of deep learning models, especially large-scale neural networks with billions or even trillions of parameters, has driven the development of distributed training frameworks. These libraries provide abstractions that enable to scale models beyond the limitations of a single device, while handling communication, synchronization, and efficiency optimizations. They can be broadly categorized into core deep learning frameworks with distributed Application Programming Interfaces (APIs), scaling and optimization libraries.

2.2.1 Core Deep Learning Frameworks

PyTorch and TensorFlow are the dominant deep learning frameworks, with mature and highly optimized distributed training support. They provide the essential building blocks for scalable model development, combining flexibility, extensibility, and high performance.

PyTorch [19] is characterized by its dynamic computation graph, user-friendly API, and deep integration with Python's scientific ecosystem. Its distributed training is primarily enabled through DDP, which serves as the standard approach for synchronous data parallelism. In DDP, each process maintains a full replica of the model and computes gradients locally. These gradients are then aggregated across all participating processes using efficient collective communication primitives, such as NVIDIA Collective Communications Library (NCCL) [1] (for GPU communication), Gloo (for Central Processing Unit (CPU) or cross-platform setups), or Message Passing Interface (MPI). This design allows for near-linear scaling efficiency when training on multiple GPUs or nodes.

The simplicity and robustness of DDP made it a standard in both research and production. It has been extensively used at Meta AI for large-scale projects such as transformer model training and foundation model experimentation. Beyond core training, PyTorch also integrates with distributed data loading, mixed precision (via torch.amp), and gradient checkpointing, allowing users to optimize performance and memory usage at scale.

TensorFlow [20], developed by Google, adopts a more static computation graph approach, emphasizing reproducibility and deployment readiness. It provides multiple distribution strategies under the tf.distribute module, the most common being MirroredStrategy and MultiWorkerMirroredStrategy. MirroredStrategy supports multi-GPU training within a single machine by replicating model variables across devices and synchronizing updates using an all-reduce algorithm. MultiWorkerMirroredStrategy extends this functionality to multi-node environments, coordinating gradient synchronization across multiple workers via the CollectiveOps API.

TensorFlow's ecosystem integrates seamlessly with TensorFlow Extended (TFX) for model deployment pipelines and TensorBoard for visualization and monitoring, enabling end-to-end workflows from training to serving. It has been critical in large-scale industrial applications such as Google's training of BERT and TPU (Tensor Processing Unit) based transformer architectures, showcasing its scalability and production-grade reliability.

Overall, PyTorch and TensorFlow provide the computational and communication foundations for most distributed machine learning pipelines. Their modular APIs allow researchers and engineers to tailor distributed execution to specific hardware configurations, balancing flexibility and performance across diverse environments.

2.2.2 Scaling and Optimization Libraries

As models grew larger, new libraries extended the capabilities of core frameworks:

- Horovod [21], created at Uber, simplified multi-GPU and multi-node training with a unified API. By introducing the ring-allreduce algorithm for gradient synchronization, Horovod reduced communication overhead and helped accelerate adoption of distributed training in production systems.

- DeepSpeed, developed by Microsoft, introduced several innovations, including the ZeRO , which partitions optimizer states, gradients, and parameters across devices to dramatically reduce memory requirements. DeepSpeed has been used to train GPT-3 and other very large language models [22].

- Megatron-LLM [14], from NVIDIA, specializes in tensor parallelism, enabling efficient training of extremely large transformer-based language models. It was a core component in training Megatron-Turing NLG, a 530-billion-parameter model.

- FairScale and PyTorch FSDP extend PyTorch with parameter and optimizer sharding techniques, lowering memory footprints while maintaining efficient parallelism.

These libraries push the limits of distributed training, making it feasible to train models at unprecedented scales. Table 2.2 provides a comparative overview of the major distributed machine learning frameworks and libraries, including both native framework capabilities (PyTorch DDP, TensorFlow) and specialized optimization libraries (Horovod, DeepSpeed, Megatron-LM, FSDP), highlighting their primary focus areas, key technical features, and representative applications in production environments.

Table 2.2: Distributed Machine Learning Frameworks and Libraries

Framework / Library	**Main Focus**	**Key Features**	**Use Cases**
PyTorch DDP	Native DL framework	Synchronous data parallelism, easy integration	Training ResNet or BERT across GPUs
TensorFlow	Native DL framework	Multi-GPU and multi-node training APIs	Large-scale BERT training at Google
Horovod	Distributed optimization	Unified API, ring-allreduce communication	Scalable multi-node training at Uber
DeepSpeed	Large-scale optimization	ZeRO optimizer, mixed parallelism, memory efficiency	Training GPT-3-scale models
Megatron-LLM	Model parallelism	Tensor and pipeline parallelism for transformers	Megatron-Turing NLG (530B params)
PyTorch FSDP / FairScale	Memory-efficient training	Fully sharded model and optimizer states	Efficient large model fine-tuning

2.3 Resource Management and Orchestration

Efficient distributed machine learning does not depend solely on algorithms and infrastructure, but also on how organizations manage and allocate their computational resources. In environments such as research institutions, cloud providers, or enterprise AI departments, the coordination between users, infrastructure, and resource management systems plays a key role in ensuring scalability, cost-effectiveness, and high utilization of hardware resources. This section explores how these organizational workflows operate and how orchestration tools support them in practice.

2.3.1 Organizational Workflow in Distributed Environments

In a typical organizational environment, users such as data scientists or AI engineers submit requests for computational resources to train machine learning models. These requests often specify requirements such as the number of GPUs, CPU cores, memory allocation, and expected duration of the training job.

The resource management team, or, increasingly, automated schedulers, evaluate these requests and assign resources based on availability, priority, and efficiency goals. The aim is to maximize resource utilization while minimizing idle time and preventing bottlenecks.

In many institutions, this process follows a structured workflow:

1. **Job Submission**: The user submits a training job through an internal platform or API, describing the model, dataset, and resource needs.

2. **Scheduling and Allocation**: The management system evaluates current workloads and allocates available resources, queueing jobs if the system is at capacity.

3. **Monitoring and Optimization**: During execution, the system monitors GPU/CPU usage, memory consumption, and job progress. It can dynamically reallocate resources or reschedule jobs to maintain optimal cluster efficiency.

4. **Completion and Reporting**: Once training finishes, results are logged, metrics are stored, and reports are made available for reproducibility and auditing.

This workflow ensures that distributed resources, often expensive clusters with tens or hundreds of GPUs, are shared fairly among users while supporting continuous experimentation. Platforms such as Determined AI and Kubernetes-based systems have integrated these concepts, providing automated job submission, priority scheduling, and real-time performance tracking.

2.3.2 Orchestration Platforms and Tools

Distributed training requires the efficient coordination of multiple nodes, GPUs, and CPUs to ensure models are trained efficiently and scalably. Beyond the technical execution of ML jobs, resource management also plays a key role in cluster organization, determining how resources are shared across users and preventing bottlenecks or idle hardware. Orchestration tools therefore act both as workload schedulers and as cluster managers, ensuring fairness, reproducibility, and high utilization of expensive computational resources. Some of the most prominent of these tools are briefly described bellow:

- **Kubernetes**: A widely used container orchestration platform that facilitates the distribution and management of machine learning workloads across clusters [23]. It provides elasticity, fault tolerance, and scalability, enabling ML frameworks such as TensorFlow and PyTorch to run seamlessly on large infrastructures. Extensions such as Kubeflow add ML-specific features, including distributed training operators and experiment tracking.

- **Apache Spark**: A distributed data processing platform originally designed for big data analytics. Its main job is to run data transformations (like Resilient Distributed Datasets (RDDs) and DataFrames) across many nodes, but can also be applied in ML. Its MLlib library implements scalable algorithms for regression, classification, and clustering. Spark operates through a cluster of *executors*, processes distributed across *worker nodes*, that perform computations in parallel under the coordination of a central *driver*, as shown in the Figure 2.4. This same mechanism can be leveraged

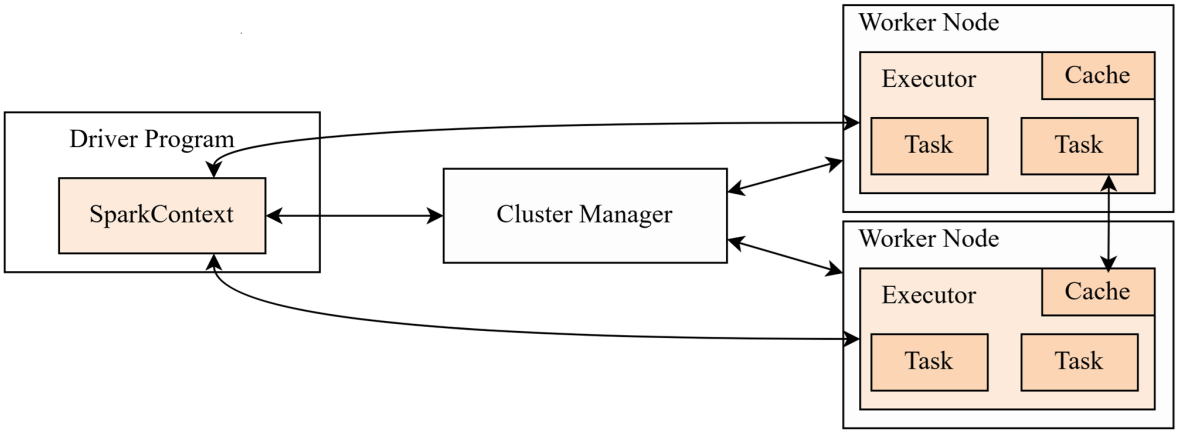

Figure 2.4: Spark System Architecture

for distributed deep learning, where each executor runs part of the training process on a dedicated GPU or node.

- **DeterminedAI**: A complete master + agent system (like Kubernetes, but simpler and focused on ML), container-native platform created specifically for distributed deep learning. Unlike general-purpose orchestrators, it provides built-in support for GPU scheduling, experiment tracking, fault tolerance, and hyperparameter optimization. Both a training framework and a cluster manager, it abstracts away much of the complexity of distributed training. The platform also simplifies collaboration with a centralized multi-user system. By assigning users to workspaces with specific roles, it enables secure, efficient sharing of GPU resources across the entire team, preventing contention and maximizing productivity on a shared infrastructure.

The orchestration landscape for distributed machine learning is diverse, with each platform offering distinct trade-offs between flexibility, ease of use, and ML-specific optimizations. Table 2.3 summarizes the primary characteristics and typical applications of the main orchestration tools discussed above, highlighting how they address different aspects of resource management and distributed training coordination.

Table 2.3: Resource Management and Orchestration Tools

Tool / Platform	**Primary Function**	**Key Features**	**Use Cases**
Kubernetes	Container orchestration	Elastic scaling, fault tolerance, workload scheduling	Managing ML jobs across GPU clusters
Apache Spark	Distributed data and ML engine	In-memory computation, MLlib library, scalability	Parallel model training via TensorFlowOnSpark
Determined AI	Deep learning orchestration platform	GPU scheduling, hyperparameter search, fault tolerance	Distributed training with built-in experiment management

2.4 Summary

This chapter reviewed the state of the art in distributed machine learning, focusing on three core dimensions: parallelism strategies, distributed frameworks and libraries, and resource management tools. Together, these elements define how large-scale models are efficiently trained across multiple devices and clusters.

The chapter began by outlining the main parallelism techniques: data, model (pipeline and tensor), hybrid, and ZeRO, which collectively enable faster training, better scalability, and improved memory efficiency.

Next, it explored the key frameworks and libraries that implement these strategies. Foundational tools like PyTorch and TensorFlow provide distributed APIs, while advanced systems such as Horovod, DeepSpeed, Megatron-LM, and FSDP optimize performance for large-scale models. Broader platforms like Apache Spark, Kubeflow, and DeterminedAI extend these capabilities with integrated data processing and experiment management.

Finally, resource management and orchestration tools were examined, including Kubernetes, Apache Spark, and DeterminedAI, which ensure scalable, fault-tolerant execution in distributed environments. These systems are essential for coordinating resources, scheduling workloads, and maintaining reproducibility in modern AI infrastructures.

Chapter 3

Methodology

In this chapter, the fundamental details of the methodology used to evaluate the tested platforms are presented: the computational infrastructure, the used tools and their installation procedures, and the metrics used to evaluate the results collected.

3.1 Infrastructure

The experimental environment is designed to reproduce a realistic distributed training setup similar to those found in research labs and organizational clusters. The experimental cluster comprises three virtual nodes (virtual machines) in a KVM-based virtualization cluster, each equipped with a passthrough GPU and a multi-core vCPU, capable of supporting deep learning workloads. Each virtual node has the following specifications:

- CPU: AMD EPYC Zen1 7351 (16-core). This processor architecture provides solid multi-threaded performance for data preprocessing and communication tasks, both critical in distributed training.
- Main Memory: 32 GB, providing sufficient capacity for batch-level data loading and model state management during training.
- Secondary Storage: 64 GB of an SSD-based vDisk, ensuring fast access to datasets and checkpoints.

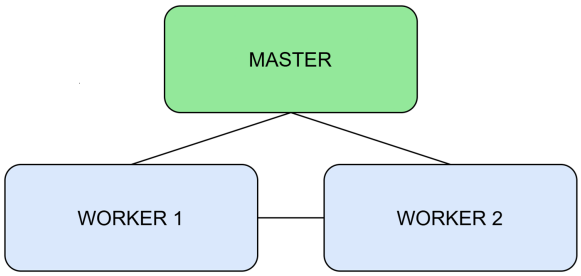

Figure 3.1: Experimental TestBed Architecture

- GPU: NVIDIA GeForce GTX 1060 3 GB (driver 535.247.01, CUDA 12.2 and CUDA Deep Neural Network library (cuDNN) enabled). It provides CUDA [24] support for running deep learning workloads with frameworks such as PyTorch and TensorFlow, and allows evaluation of GPU utilization, communication overhead, and scaling efficiency in small-scale distributed tests.

- Operating System: Ubuntu 24.04.3 LTS, a stable and popular Linux distribution for AI and HPC workloads. Its compatibility with recent NVIDIA drivers and CUDA versions ensures optimal GPU performance and supports modern frameworks.

This setup represents the experimental configuration used throughout this study. Although modest, the specific GPU model used was selected for two main reasons: it provides enough computational power suitable for experimentation without excessive energy consumption, and there were plenty of units available in the virtualization cluster used.

As illustrated in Figure 3.1, the three (virtual) nodes used play different roles: there is one master node, responsible for managing job distribution and coordination, and two worker nodes, which handle the actual model training and computation tasks.

3.2 Tools and Installation

This section describes the main frameworks and tools employed in this study. Each framework provides a distinct approach to distributed deep learning training, ranging from

low-level implementations with manual process management to high-level orchestration platforms that automate scheduling, monitoring, and metadata handling.

PyTorch DDP PyTorch DDP is selected as the baseline framework due to its direct integration with PyTorch and its ability to provide fine-grained control over distributed execution. DDP enables parallel training by replicating the model across multiple GPUs and synchronizing gradients during backpropagation using high-performance collective communication backends such as NCCL.

This framework represents a low-level, "framework-native" approach, where the user explicitly configures the training processes, environment variables (e.g., MASTER_ADDR, MASTER_PORT, WORLD_SIZE), and data samplers to ensure synchronized and efficient distributed computation.

The installation includes the latest stable release of PyTorch with CUDA and cuDNN support, as well as complementary Python libraries such as Transformers, Datasets, and Evaluate for model handling and performance assessment. This setup ensures an optimized training environment while allowing direct comparison with higher-level orchestration systems such as Determined AI and Apache Spark.

Determined AI Determined AI is installed in standalone mode, with one master node and two agent nodes, reflecting the cluster topology used for DDP experiments. Unlike DDP, which requires manual coordination of processes and resources, Determined AI provides an integrated orchestration layer that automates key aspects of distributed training, including experiment scheduling, hyperparameter optimization, and GPU allocation.

The platform's architecture follows a Master-Agent model: a) the Master node acts as the central controller, managing experiments, allocating resources, and monitoring task execution; b) the Agents (worker nodes) register with the master and execute training tasks within isolated environments, typically Docker containers.

Experiments can be launched via the Web or Command Line interfaces, with results automatically tracked and visualized in real time. Determined's declarative configuration

model, based on YAML files, allows precise definition of resources, hyperparameters, and storage locations for checkpoints, ensuring reproducibility and scalability.

This platform abstracts away much of the operational complexity of distributed systems, offering a higher level of automation and monitoring while still leveraging PyTorch DDP under the hood for gradient synchronization.

PostgreSQL PostgreSQL is deployed as a core component of the Determined AI infrastructure. Serving as the system's metadata store, it maintains persistent and structured records of experiments, trial configurations, hyperparameters, user information, and checkpoint references. By externalizing metadata from the Determined master process, PostgreSQL ensures data durability, transactional consistency, and fault tolerance, allowing experiments and user sessions to be resumed or analyzed even after service restarts.

The database runs as a background service on the master node, configured for local access and optimized for concurrent I/O operations. Standard SQL interfaces allow for direct querying of experiment and user metadata when required. This integration is essential for Determined AI's centralized experiment and account management, providing a reliable storage layer for long-term research tracking and user administration.

Apache Spark Apache Spark is installed in standalone cluster mode, with one master and two worker nodes. Originally designed for large-scale data processing, it provides a robust general-purpose distributed computing engine that can also be adapted for deep learning workloads. In this setup, PyTorch training is integrated through custom launcher scripts that leverage Spark's barrier execution mode, ensuring all executors start training simultaneously, a crucial requirement for synchronization in distributed deep learning.

Spark's scheduler manages the allocation of CPU, GPU, and memory resources across workers, while its Web interface provides detailed visibility into task distribution, execution stages, and resource usage.

Although Spark is not a deep learning-specific platform like Determined AI, its flexibility and scalability make it valuable for comparing how a general-purpose distributed

framework performs in parallel model training scenarios.

Containerized Execution (Docker) While all frameworks are installed natively on the host machines, both Determined AI and PyTorch DDP rely on containerized environments to execute training jobs. Docker [25] containers provides an isolated and reproducible runtime for each experiment, ensuring that dependencies, library versions, and system configurations remain consistent across all nodes.

In this configuration, containers are automatically managed by each framework. For instance, Determined AI launches every experiment inside a container derived from a specified image, while still leveraging the host's native CUDA drivers and communication libraries (e.g., NCCL).

This hybrid setup combines the advantages of containerization with native performance, reflecting real-world deployment environments where clusters balance isolation and flexibility. It ensures that results remain reproducible while maintaining compatibility with GPU hardware and low-level communication layers. Each framework is tested using the same model, dataset, and hyperparameters to guarantee experimental fairness and comparability across all distributed setups.

General Cluster Configuration Beyond framework-specific installations, several common infrastructure configurations were required across all experimental setups to enable distributed communication and fault tolerance:

- Setting up Secure Shell (SSH) key-based communication between nodes.
- Configuring environment variables such as MASTER_ADDR, MASTER_PORT, and WORLD_SIZE;
- Framework-specific fault tolerance mechanisms (manual restart for DDP, automatic recovery for Determined AI, executor restart for Spark)

This approach mirrors how distributed systems are typically deployed in organizational

environments, where multiple users may request computational resources that must be efficiently allocated and tracked.

3.3 Metrics and Evaluation

To assess the effectiveness of the selected distributed training technologies, multiple evaluation criteria are considered, encompassing both performance and usability dimensions. The core quantitative metrics are ***training time*** and ***model accuracy***, which together provide a comprehensive measure of computational efficiency and training quality.

Initially, all experiments are conducted on a single node to establish a baseline for comparison. This baseline serves as a reference point for evaluating the impact of distributed or parallelized training on performance. Subsequently, the same training processes are executed using distributed configurations (across multiple nodes and GPUs), allowing for direct comparison in terms of execution time reduction and accuracy preservation.

To ensure consistency and fairness across tests, all systems train the same model under identical hyperparameters and dataset conditions. The selected model is BERT [11] from the Hugging Face library, a transformer-based architecture widely used for natural language processing tasks. BERT was chosen due to its accessibility, reproducibility, and compatibility with distributed training frameworks such as the ones exploited in this work (PyTorch, Apache Spark, and Determined AI).

In addition to raw performance, qualitative aspects related to cluster management and user experience are also evaluated. These include:

- ***Cluster management and scheduling efficiency***: assessing how effectively each framework allocates and monitors computational resources, handles fault recovery, and manages parallel jobs.
- ***Ease of configuration and workflow integration***: evaluating the clarity of setup procedures, the complexity of configuration files, and the level of automation in experiment orchestration and monitoring.

The main evaluation therefore follows these steps:

1. **Single-node baseline**: measure training time and final model accuracy.

2. **Distributed execution**: repeat the same process using each distributed training system (DDP, Spark, Determined AI).

3. **Management and usability assessment**: compare the frameworks in terms of resource scheduling, monitoring tools, and ease of integration.

4. **Comparison and analysis**: evaluate how much training time decreases, whether accuracy is maintained or degraded, and which framework provides the best overall balance between performance, management, and usability.

3.4 Summary

This chapter presented the methodological framework for comparative evaluation of distributed deep learning platforms, encompassing infrastructure specification, tool installation procedures, and evaluation metrics.

The experimental infrastructure comprises a three-node virtualized cluster, with each node equipped with an NVIDIA GTX 1060 GPU, 16-core AMD EPYC processor, and 32GB Random Access Memory (RAM). This configuration provides a controlled testbed suitable for assessing distributed training characteristics in research-oriented environments.

Three distinct frameworks were deployed: PyTorch DDP as a baseline framework-native approach, Determined AI as a specialized orchestration platform with automated experiment management, and Apache Spark as a general-purpose distributed computing engine adapted for deep learning. Supporting components including PostgreSQL, Docker containerization, and common cluster infrastructure (SSH authentication, environment variables) were configured to enable distributed coordination across all systems.

The evaluation methodology combines quantitative performance metrics (training time and model accuracy) with qualitative assessment of cluster management and usability. All frameworks train identical BERT models on the Internet Movie Database (IMDB) dataset using consistent hyperparameters, with experiments conducted first on single nodes to establish baselines, then scaled to distributed configurations for comparative analysis.

Chapter 4

Implementation

In this chapter, the experimental setup is thoroughly described, detailing the configuration of the training environment, the hyperparameters used in the baseline model, and the procedures followed to install, configure, and adapt the code for execution across multiple distributed training frameworks.

4.1 ML Model and Dataset

As already assumed, the goal of this work is not to develop a new model but to measure and compare the performance of different distributed training technologies. For that reason, it was decided to use a well-established model, BERT from Hugging Face, performing sequence classification on two classes (positive/negative sentiment). The dataset used was the IMDB Movie Reviews, containing 50,000 labeled reviews. The data was split evenly, using 25,000 reviews for training and 25,000 for validation.

During initial tests, however, a hardware limitation emerged: the 3 GB of global memory of the GPUs used, made it impossible to train the full-size BERT model. To overcome this limitation, a smaller model variant was adopted, prajjwal1/bert-tiny, which retains the same structure but requires fewer resources, allowing the experiments to proceed.

4.2 Training Hyperparameters

All experiments used identical hyperparameters to ensure fair comparison across frameworks. The hyperparameters at stake were the follwing:

- Learning Rate: 5×10^{-5} – value selected based on the recommendations from Devlin et al. [11], who identified the range of 2×10^{-5} to 5×10^{-5} as optimal for fine-tuning BERT-based models on classification tasks. Given that BERT-tiny has significantly fewer parameters (4.4M) than BERT-base (110M), it was selected the upper end of this range to enable faster convergence while maintaining training stability.

- Learning Rate Schedule: Linear warmup followed by linear decay, with warmup steps set to 500 steps (about 6% of total training steps). This schedule prevents early-stage instability, common in transformer fine-tuning, while gradually reducing the learning rate to avoid overshooting the optimum.

- Batch Size: 32 per GPU, resulting in an effective batch size of 64 across two GPUs. This aligns with typical batch sizes used for BERT fine-tuning on single-sentence classification tasks [11].

- Weight Decay: 0.01, following the AdamW optimizer configuration recommended by Loshchilov & Hutter [26].

- Training Duration: 20 epochs (repeated 5 times per configuration) and 100 epochs (single run per configuration for extended validation). The 20-epoch experiments provided the primary performance metrics, while the 100-epoch runs validated that per-epoch training time remains constant across different training durations.

These hyperparameters represent established best practices for transformer fine-tuning and ensure that observed performance differences reflect framework characteristics rather than suboptimal training configurations.

For comparability, the Python time module was used to measure how long it took to complete each full training run.

4.3 Distributed Training Strategy

Regardless of the distributed framework used (PyTorch DDP, Determined AI, or Apache Spark), all implementations in this work rely on data parallelism with synchronous gradient aggregation. In this paradigm, each worker processes a unique subset of the training data, computes gradients locally, and then synchronizes those gradients with other workers to maintain consistent model weights across all devices, as shown in Figure 4.1.

The synchronization is performed using collective communication primitives through the NCCL backend. Specifically, gradient synchronization employs the all-reduce operation, which averages gradients across all participating processes. This operation is automatically handled by PyTorch's DDP wrapper, which all three frameworks utilize internally. The all-reduce ensures that, after every training step, all replicas of the model share identical parameters, guaranteeing convergence equivalence to single-GPU training.

Additionally, validation metrics (accuracy, loss) are aggregated across workers using explicit all-reduce operations with the SUM reduction operator, allowing each process to compute globally consistent evaluation results.

The way in which gradient synchronization is achieved is represented in Figure 4.1.

4.4 Setup, Configuration, and Implementation

After defining the baseline setup, the distributed training technologies were installed, and the training code was adapted for execution under each platform.

4.4.1 Pytorch Distributed Data Parallel

The first technology deployed was PyTorch DDP, since it serves as the foundation and is the simplest to implement. Once the local version of the training script was validated, the next step was to enable parallelization across multiple machines. To achieve this, it was first necessary to establish reliable communication between the nodes. DDP uses NCCL as the backend for inter-process communication, which operated over TCP (port

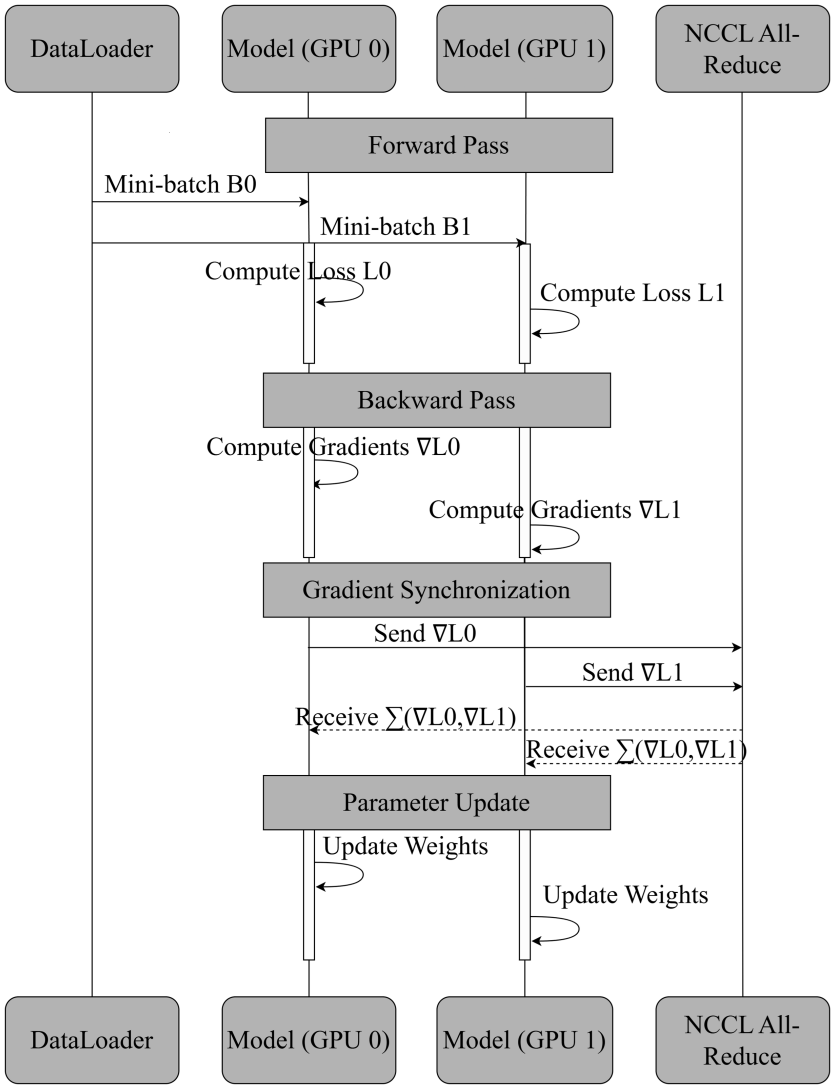

Figure 4.1: How gradient synchronization works.

12355). For convenience and smoother node management, SSH keys were configured to allow passwordless access, and the machines' hostnames were defined in `/etc/hosts` to simplify identification within the network.

After setting up connectivity, a Dockerfile was created to build a consistent execution environment across all nodes. This ensured that every machine had the same versions of Python, CUDA, PyTorch, and all necessary dependencies (defined in requirements.txt). The Docker image also simplified replication of the setup and reduced compatibility issues during distributed execution.

```
docker run --gpus all --rm -it \
  --network host \
  -v /home/a41783/Desktop/DDP:/app \
  ddp-image \
  torchrun \
    --nnodes=2 \
    --nproc_per_node=1 \
    --node_rank=0 \
    --master_addr=192.168.217.244 \
    --master_port=12355 \
    train.py
```

Listing 4.1: Docker command for 2-node distributed training with PyTorch DDP

To start the distributed training, it was used the command of Listing 4.1. This command launches the training inside a Docker container while giving it direct access to the host GPU (`-gpus all`) and network stack (`-network host`) to allow proper communication between nodes. The volume mount (`-v`) makes the training script and dataset available inside the container. The `torchrun` utility is responsible for orchestrating the distributed execution. The parameters specify that the training involves two nodes (`-nnodes=2`), with one process per node (`-nproc_per_node=1`). The `-node_rank` identifies the current machine's role (0 for the master, 1 for the worker), while `-master_addr` and `-master_port` define how the nodes find each other.

This setup ensures that the DDP processes are correctly initialized and synchronized across machines, allowing the training to be distributed efficiently while keeping the environment reproducible and isolated within Docker.

4.4.2 Apache Spark

After successfully configuring and validating the DDP setup, the next step was to install and test Apache Spark. For this, the standalone version was chosen, as it represents the most direct and "pure" form of using Spark without additional abstractions or managed

environments. The installation began by downloading the compressed archive from the latest official version available at the time (`spark-3.5.1-bin-hadoop3.tgz`).

Once the archive was extracted, several configuration files had to be adapted according to the role of each machine within the cluster. On the master node, within the `spark-env.sh` file, the `SPARK_UI_PORT` was set to `8081` to avoid conflicts with other services, and the `SPARK_MASTER_HOST` variable was assigned the master node's IP address. Additionally, the hostnames of all worker nodes were added to the workers file so that Spark could correctly identify and connect to them when starting the cluster.

On the worker nodes, the configuration was slightly more complicated. Similarly, the master IP and worker identifiers were defined, but when checking the Spark Web interface, it became clear that Spark was not detecting the GPUs available on the machines, as if they were invisible to the system. To address this issue, a custom GPU discovery script and a corresponding resource description file were created – see Listing 4.2 and 4.3. These ensured that Spark could recognize and register each GPU as a usable resource.

```
#!/bin/bash
echo '{"name": "gpu", "addresses": ["0"]}'
```

Listing 4.2: Shell script to discover GPU

```
{
    "id": {
        "componentName": "spark.worker",
        "resourceName": "gpu"
    },
"addresses": ["0"]
}
```

Listing 4.3: Json file defining gpu as worker resource

Here, "0" simply indicates that each worker machine contains a single GPU.

After confirming that the GPUs were correctly detected and available in the Spark Web interface, the next step was to adapt the existing DDP training code to run under Spark's

execution model. The main modification involved replacing the manual distributed setup and process management with Spark's TorchDistributor, which provides an integrated way to launch PyTorch distributed jobs on Spark executors.

The training logic itself remained identical to the previous DDP implementation. However, process initialization, resource allocation, and job launching were now managed directly by Spark. This required reorganizing the project into two main components:

- A **launcher script**, responsible for creating the Spark session and invoking the TorchDistributor to distribute the training workload across executors.
- A **training script**, containing the actual model training logic.

The launcher script (Listing 4.4), creates a Spark session and uses TorchDistributor to coordinate the distributed training across two processes on two separate executors (worker nodes). The distributor handles all the complexity of setting up the distributed environment, making the code significantly simpler compared to the manual DDP setup.

```
from pyspark.sql import SparkSession
from pyspark.ml.torch.distributor import TorchDistributor
import train_spark
def wrapped_run_fn():
    train_spark.run_fn()
if __name__ == "__main__":
    spark = SparkSession.builder.appName("DDP-Spark").getOrCreate()
    distributor = TorchDistributor(num_processes=2, local_mode=False)
    distributor.run(wrapped_run_fn)
    spark.stop()
```

Listing 4.4: Spark launcher script for distributed training

The training script (Listing 4.5), keeps the same core logic as the standalone DDP implementation, with the key difference being how the distributed process group is initialized. In the Spark version, this initialization is triggered by the TorchDistributor, which automatically sets the required environment variables before calling the training function.

```
def run_fn():
    # TorchDistributor sets up MASTER_ADDR, MASTER_PORT, RANK, etc.
    dist.init_process_group(backend="nccl", init_method="env://")
    rank = dist.get_rank()
    world_size = dist.get_world_size()
    print(f"Initialized process group: rank {rank}, world_size {
        world_size}")
    main(rank, world_size)
```

Listing 4.5: Training script entry point for Spark-based distributed training

In this new structure, explicitly calling torch.distributed.init_process_group() was no longer necessary, as Spark automatically handled the distributed process group setup.

Once the environment and scripts were configured, the Spark cluster operated as expected: the master node coordinated the job execution, while the two worker machines performed the distributed computations. The Spark Web interface provided real-time visibility into the cluster, including running applications, resource utilization, GPU assignment, and execution times, allowing for effective monitoring and performance assessment of the distributed training process.

4.4.3 DeterminedAI

When installing Determined AI using Linux packages (instead of via a pre-built Docker image), the first step was to install PostgreSQL, since Determined relies on a PostgreSQL database to store experiment and trial metadata. On the master machine, after installing PostgreSQL and enabling the database service to start automatically at boot, a dedicated database was created for Determined's use. It was also configured a system account with the necessary privileges to allow Determined to connect to the database. This setup followed the commands provided in Determined's documentation (see Listing 4.6).

Once the database was configured, the next step was to install Determined AI itself, using the latest available Debian packages (`determined-master_0.38.1_linux_amd64.deb`

```
sudo -u postgres psql
postgres=# CREATE DATABASE determined;
postgres=# CREATE USER determined WITH ENCRYPTED PASSWORD 'password';
postgres=# GRANT ALL PRIVILEGES ON DATABASE determined TO determined;
postgres=# \c determined
determined=> GRANT ALL ON SCHEMA public TO determined;
```

Listing 4.6: Commands for the connection between Determined and PostgreSQL

for the master node, and `determined-agent_0.38.1_linux_amd64.deb` for the workers).

After installation, the configuration files were customized to reflect the cluster architecture and resource distribution.

On the master node, the `master.yaml` file was updated to define the communication port (`port:8080`) and to specify the scheduler type. The latter was set to `fair_share scheduler` instead of First In, First Out (FIFO), since fair share provides better resource balancing and scalability across users or experiments. It was also configured the connection parameters for the PostgreSQL database and defined the directory for checkpoint storage, where intermediate model states are saved.

On the agent nodes, the `agent.yaml` configuration file was similarly updated. Each agent was assigned the IP address of the master machine through the `master_host` property, along with the port number (`8080`). It was also added a unique identifier for each worker using the `agent_id property` (`AGENT_ID_1` and `AGENT_ID_2`), and defined the available GPU resources per node, using `slot_type:cuda` and `visible_gpus:0` (since each machine contained a single GPU).

Compared to the manual DDP setup, the Determined AI configuration significantly abstracts away the complexity of distributed management. Instead of launching processes manually using torchrun and handling environment variables for rank and world size, Determined automatically manages experiment scheduling, process orchestration, and inter-node communication. All experiment parameters, such as number of GPUs, batch size, hyperparameters, and checkpointing details, are declared inside a YAML configuration file, making the workflow both more structured and reproducible.

Within the training script itself, the Determined API (det.core) replaces manual logging and checkpointing. It provides access to the experiment context, handles metric reporting, and ensures proper saving of checkpoints. This integration enables fault tolerance, hyperparameter tuning, and automatic tracking of results, while still leveraging PyTorch's native DDP backend under the hood. In practice, this means that while the core training loop remains the same as in the original DDP implementation, the orchestration, monitoring, and experiment management are fully handled by Determined AI.

During early tests, an issue occurred with some Python dependencies not installing correctly from the `requirements.txt` file inside the containerized environment used by Determined. To fix this, a small initialization script (`startup-hook.sh`) was created to install all required libraries before each training session started. This solved the dependency problem and ensured consistent environment setup across runs.

4.5 Summary

This chapter described the implementation of distributed training experiments across PyTorch DDP, Apache Spark, and Determined AI, establishing the technical foundation for comparative evaluation.

The experimental configuration employed BERT-tiny for sentiment classification on the IMDB dataset, with hyperparameters following established best practices (5×10^{-5} learning rate, batch size 32 per GPU, AdamW optimization). All implementations used data parallelism with synchronous gradient aggregation via NCCL, ensuring training equivalence across frameworks.

Framework implementations revealed distinct orchestration approaches: PyTorch DDP required manual cluster coordination via torchrun and SSH; Apache Spark demanded custom GPU discovery scripts and code restructuring for TorchDistributor integration; Determined AI needed extensive PostgreSQL and master-agent setup but provided automated experiment management through YAML configuration.

Chapter 5

Experimental Evaluation

This chapter presents the experimental evaluation of the distributed deep learning frameworks under scrutiny in this work: PyTorch DDP, Apache Spark, and Determined AI.

5.1 Experimental Conditions

The evaluation was conducted according to the methodology described in Chapter 3, with the objective of answering the three research questions formulated in Section 1.3. All experiments were conducted on the three-node cluster described in Section 3.1.

The training task consisted of fine-tuning BERT-tiny (prajjwal1/bert-tiny) for binary sentiment classification on the IMDB Movie Reviews dataset, using identical hyperparameters across all frameworks: learning rate of 5×10^{-5}, batch size of 32 per GPU, and AdamW optimizer with linear warmup and decay.

To ensure statistical reliability, each configuration was evaluated over 5 runs with fixed random seeds, allowing for calculation of mean performance metrics and standard deviations. The primary experiments were conducted over 20 epochs, a duration sufficient for model convergence on this task, as validated by extended 100-epoch experiments that confirmed linear time scaling and no significant accuracy improvements beyond 20 epochs.

5.2 Experimental Configurations

Three distinct experimental configurations were evaluated:

- **Baseline (Single-GPU):** Each framework was first evaluated in single-GPU mode to establish baseline performance metrics. This configuration serves as the reference point for assessing the benefits and overhead of distributed training.

- **Distributed (2 GPUs):** The primary distributed configuration employed two GPUs (one per worker node) to evaluate scaling behavior, communication overhead, and the effectiveness of gradient synchronization mechanisms.

- **Extended Validation (100 Epochs):** A subset of experiments was conducted over 100 epochs to verify that framework performance characteristics remain consistent over longer training durations and that the 20-epoch measurements are representative of steady-state behavior.

5.3 Experimental Results

This section presents the quantitative results obtained from training BERT-tiny on the IMDB sentiment classification task across the three evaluated frameworks. All measurements represent mean values computed over five independent runs, with standard deviations reported where variability is observed.

5.3.1 Training Performance

Tables 5.1 to 5.3 present the complete training time and accuracy results for PyTorch DDP, Apache Spark, and Determined AI, respectively. The results are organized by GPU configuration (1 GPU vs. 2 GPUs) and training duration (20 epochs vs. 100 epochs).

Table 5.1: PyTorch DDP Training Results

Configuration	Epochs	Mean Time (s)	Mean Accuracy
1 GPU	20	**677.10**	0.8284
2 GPUs	20	**498.99**	0.8259
1 GPU	100	**3399.48**	0.8273
2 GPUs	100	**2510.63**	0.8314

Table 5.2: Determined AI Training Results

Configuration	Epochs	Mean Time (s)	Mean Accuracy
1 GPU	20	833.14	0.8279
2 GPUs	20	605.56	0.8289
1 GPU	100	3924.99	0.8358
2 GPUs	100	2822.77	0.8228

Table 5.3: Apache Spark Training Results

Configuration	Epochs	Mean Time (s)	Mean Accuracy
1 GPU	20	2520.89	0.8249
2 GPUs	20	1431.39	0.8236
1 GPU	100	12487.48	0.8290
2 GPUs	100	7203.63	0.8194

Single-GPU Performance

In single-GPU configuration with 20 epochs, PyTorch DDP achieved the fastest training time (677.10 s), followed by Determined AI (833.14 s), and Apache Spark (2520.89 s). The performance ranking remained consistent in the 100-epoch experiments, with DDP completing training in 3399.48 s, Determined AI in 3924.99 s, and Spark in 12487.48 s.

These results establish DDP as the baseline for performance, with Determined AI introducing a 23% overhead and Spark introducing a 272% overhead in the 20-epoch tests. The substantial overhead in Spark reflects its architecture as a general-purpose distributed computing framework rather than a specialized deep learning platform.

Distributed Training Performance (2 GPUs)

When scaling to two GPUs, all frameworks demonstrated performance improvements, though with varying degrees of effectiveness. For 20-epoch training, DDP completed

in 498.99 s, Determined AI in 605.56 s, and Spark in 1431.39 s. This represents time reductions of 26.3%, 27.3%, and 43.2% relative to their respective single-GPU baselines.

The 100-epoch experiments showed similar patterns, with DDP achieving 2510.63 s, Determined AI 2822.77 s, and Spark 7203.63 s. The consistency between 20-epoch and 100-epoch relative performance confirms that the observed characteristics are representative of steady-state behavior rather than dominated by initialization or warm-up effects.

Training Time Linearity Validation

To verify that framework overhead remains constant over extended training periods, the ratio of per-epoch training time between 20-epoch and 100-epoch experiments was calculated. Table 5.4 presents these results.

Table 5.4: Training Time Linearity: Per-Epoch Comparison

Framework	**20 Epochs (s/epoch)**	**100 Epochs (s/epoch)**	**Ratio**
DDP (1 GPU)	33.86	34.00	1.00
DDP (2 GPUs)	24.95	25.11	1.01
Determined AI (1 GPU)	41.66	39.25	0.94
Determined AI (2 GPUs)	30.28	28.23	0.93
Spark (1 GPU)	126.04	124.87	0.99
Spark (2 GPUs)	71.57	72.04	1.01

The ratios are around 1.0 (range: 0.93-1.01), showing that training time scales linearly with the number of epochs across all frameworks. This validates the use of 20-epoch experiments as representative of framework performance characteristics and confirms that framework overhead does not compound over extended training runs.

5.3.2 Model Accuracy

Table 5.5 consolidates validation accuracy results across all frameworks and configurations for the primary 20-epoch experiments.

Validation accuracy remained consistent across all frameworks and configurations,

Table 5.5: Validation Accuracy Comparison (20 Epochs)

Framework	**1 GPU**	**2 GPUs**
PyTorch DDP	0.8284	0.8259
Determined AI	0.8279	0.8289
Apache Spark	0.8249	0.8236

with values ranging from 0.8236 to 0.8289. This consistency demonstrates that all frameworks correctly implement synchronous gradient aggregation, ensuring training equivalence regardless of the number of GPUs employed. The negligible variation confirms that the choice of framework does not impact final model quality, only training efficiency.

The 100-epoch results (Tables 5.1-5.3) show no systematic accuracy improvement beyond 20 epochs, as shown in Table 5.6. In some cases, accuracy remained stable (e.g., DDP 2 GPU: 0.8259 at 20 epochs vs. 0.8314 at 100 epochs, +0.55%), while in others it slightly decreased (e.g., Spark 2 GPU: 0.8236 at 20 epochs vs. 0.8194 at 100 epochs, -0.42%). This confirms that the model converges within 20 epochs and that extended training does not give improvements for this task and model configuration.

Table 5.6: Validation Accuracy Comparison (100 Epochs)

Framework	**1 GPU**	**2 GPUs**
PyTorch DDP	0.8273	0.8314
Determined AI	0.8358	0.8228
Apache Spark	0.8290	0.8194

5.4 Framework Comparison

This section provides direct comparative analysis of the 3 frameworks, focusing on performance metrics that enable quantitative assessment of relative strengths and weaknesses.

5.4.1 Absolute Performance Comparison

Figure 5.1 and Table 5.7 present a consolidated view of training time across frameworks and configurations for the primary 20-epoch experiments.

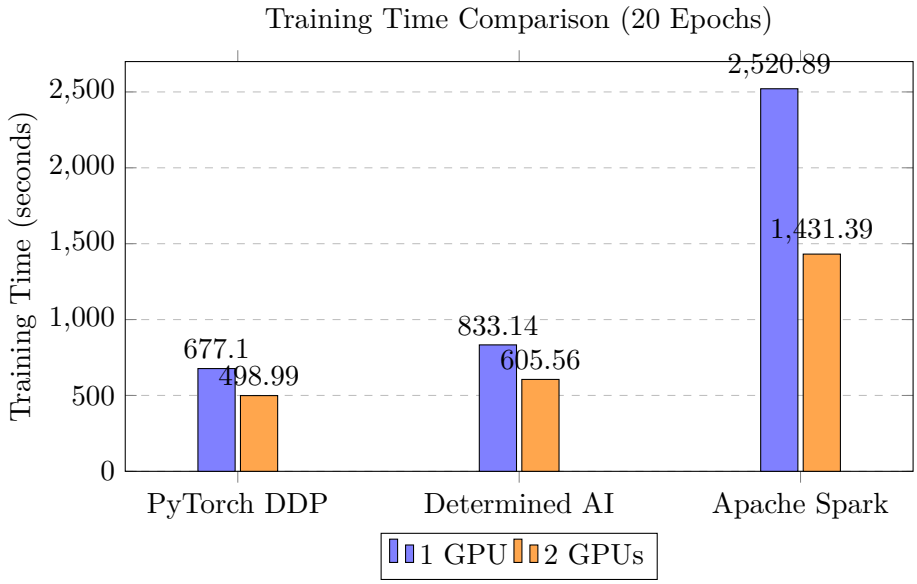

Figure 5.1: Comparison of training times across frameworks using 1 and 2 GPUs.

Table 5.7: Training Time Comparison (20 Epochs)

Framework	**1 GPU (s)**	**2 GPUs (s)**	**Overhead vs DDP (2 GPUs)**
PyTorch DDP	677.10	498.99	1.00× (baseline)
Determined AI	833.14	605.56	1.21×
Apache Spark	2520.89	1431.39	2.87×

PyTorch DDP had the best performance, with the fastest training times in all configurations. Determined AI had a 21% overhead relative to DDP in the 2-GPU configuration, a modest cost for its significantly enhanced orchestration capabilities. Apache Spark presented a substantially higher overhead, requiring 2.87× the training time of DDP in distributed configuration, reflecting the architectural costs of its general-purpose design.

In absolute terms, for a 20-epoch training run on 2 GPUs, DDP completed in approximately 8.3 minutes, Determined AI in 10.1 minutes, and Spark in 23.9 minutes. For researchers prioritizing rapid iteration and time-to-result, DDP's performance advantage is significant. However, the additional 1.8 minutes required by Determined AI may be worth in exchange for automated experiment tracking, fault tolerance, and multi-user resource management, capabilities that DDP lacks without additional tools.

5.4.2 Speedup and Scaling Efficiency

Speedup is the ratio of single-GPU training time (T1) to distributed training time (T2):

$$\text{Speedup} = \frac{T_{1\text{ GPU}}}{T_{2\text{ GPUs}}}$$

Scaling efficiency quantifies how effectively additional resources are utilized:

$$\text{Efficiency} = \frac{\text{Speedup}}{N_{\text{GPUs}}} \times 100\%$$

Table 5.8 presents these metrics for all frameworks.

Table 5.8: Speedup and Scaling Efficiency (20 Epochs)

Framework	**Speedup (2 GPUs)**	**Scaling Efficiency**
PyTorch DDP	1.36×	68%
Determined AI	1.38×	69%
Apache Spark	1.76×	88%

All frameworks exhibited sub-linear scaling, with speedups ranging from 1.36× to 1.76× when doubling GPU count from one to two. This is expected behavior in distributed deep learning, as gradient synchronization, data loading, and other non-parallelizable operations impose overhead that grows with the number of workers.

DDP and Determined AI got almost the same scaling efficiency (68% and 69%, respectively – both frameworks utilize PyTorch's DDP mechanism internally for gradient synchronization), confirming that Determined AI's orchestration layer does not introduce additional communication overhead. The slightly lower efficiency compared to ideal (100%) indicates that approximately 32% of execution time is consumed by communication, synchronization barriers, and sequential operations that cannot be parallelized.

Surprisingly, Spark achieved superior scaling efficiency (88%), despite its substantially longer absolute training times. However, this should not be interpreted as Spark being more efficient in absolute terms, once its per-epoch time remains 2.87× that of DDP.

5.4.3 Computational Overhead Analysis

To quantify framework-specific overhead, per-epoch training times were calculated and compared against the DDP baseline. Table 5.9 presents this analysis.

Table 5.9: Per-Epoch Training Time and Overhead (20 Epochs, 2 GPUs)

Framework	**Time per Epoch (s)**	**Overhead vs DDP**
PyTorch DDP	24.95	– (baseline)
Determined AI	30.28	+5.33 s (+21%)
Apache Spark	71.57	+46.62 s (+187%)

Determined AI's 5.33 s per-epoch overhead (21%) can be attributed to its orchestration layer, which handles experiment tracking, metric reporting, and checkpoint management. While measurable, this overhead is relatively modest and may be justified by the operational benefits it provides.

Spark's 46.62 s per-epoch overhead (187%) is substantially larger. Spark's performance may be acceptable in scenarios where its broader ecosystem capabilities (data preprocessing, ETL (Extract, Transform, Load) pipelines, integration with data lakes) provide added value beyond pure training performance.

5.5 Organizational Evaluation

Beyond raw performance, the practical utility of a distributed training framework depends significantly on operational factors such as ease of deployment, management capabilities, and developer experience. Next, these factors are compared for the 3 platforms studied.

5.5.1 Setup and Configuration Complexity

Table 5.10 summarizes the installation and configuration requirements for each framework. A brief discussion on setup and configuration complexity for each framework follows.

Table 5.10: Setup Complexity Comparison

Aspect	**PyTorch DDP**	**Determined AI**	**Apache Spark**
Installation Steps	PyTorch + NCCL	Master: PostgreSQL + Determined ; Agent: Determined	Spark download + extraction
Configuration Files	None (environment variables only)	master.yaml + agent.yaml	spark-env.sh + workers + GPU discovery scripts
Manual Coordination	SSH to each node, launch torchrun manually	Web UI or CLI submission (centralized)	spark-submit from master (centralized)

PyTorch DDP

DDP represents the simplest installation, requiring only PyTorch with CUDA support and NCCL. Configuration is minimal, and environment variables (`MASTER_ADDR`, `MASTER_PORT`, `WORLD_SIZE`, `RANK`) are enough for cluster coordination. However, this simplicity comes with operational costs: users must manually connect via SSH to each node and launch training processes with correct rank assignments, making it error-prone and unsuitable for multi-user environments without additional orchestration tooling. It is also important to mention that for every change made in the program, the image has to be rebuilt, and that process is also done manually.

The primary challenge with DDP is not technical complexity, but operational overhead. For a single researcher working alone, the manual launch process is manageable. For teams sharing a cluster, the lack of resource scheduling, experiment tracking, and fault tolerance becomes a significant limitation.

Determined AI

Determined AI requires an extensive initial setup, including PostgreSQL database installation and configuration on the master node, plus Determined master and agent installations. Configuration files (`master.yaml` and `agent.yaml`) must specify checkpoint storage paths, and GPU resources. This additional complexity is offset by operational benefits: once installed, users interact with the system through a polished Web interface

or via Command Line, submitting experiments with simple YAML configuration files.

The master-agent architecture centralizes cluster management, enabling features such as fair-share scheduling, automatic fault recovery, and experiment reproducibility.

Apache Spark

Spark's installation is straightforward (download and extract), but configuration proved challenging. The primary difficulty encountered was GPU detection: Spark didn't automatically discover GPUs, requiring custom discovery scripts (`spark-gpu-discovery.sh`) and resource descriptors (`worker-resources.json`). Troubleshooting this issue consumed significant time during initial deployment.

Additionally, integrating PyTorch with Spark via TorchDistributor required restructuring code into launcher and training scripts, adding complexity compared to standalone PyTorch. The spark-submit interface provides centralized job submission, but lacks the user-friendly experiment management features of Determined AI.

5.5.2 Cluster Management Capabilities

Table 5.11 compares cluster management and orchestration capabilities. An expanded discussion on the way in which each of the platforms caters for these features, follows.

Resource Scheduling and Multi-User Support

DDP provides no native resource scheduling. If two users attempt to run experiments simultaneously, manual coordination is required to avoid GPU conflicts. This limits scalability, unless external cluster management tools (e.g., Slurm, Kubernetes) are used.

Determined AI excels in this dimension, offering sophisticated resource management with fair-share scheduling, priority queues, and user workspaces with role-based access control. Multiple users can submit experiments concurrently, and the scheduler automatically allocates GPUs based on availability and configured policies. This makes Determined AI particularly well-suited for shared research clusters and organizational deployments.

Table 5.11: Cluster Management Features

Feature	**PyTorch DDP**	**Determined AI**	**Apache Spark**
Resource Scheduling	Manual	Automatic (fair-share, priority)	Automatic (FIFO, fair)
Multi-User Support	No	Yes (workspaces, roles)	Yes (basic queuing)
Experiment Tracking	Manual (logs, TensorBoard)	Built-in (database, Web UI)	Manual (logs, Spark UI)
Checkpoint Management	Manual (custom code)	Automatic (configurable)	Manual (custom code)
Fault Tolerance	Restart from checkpoint (manual)	Automatic retry and recovery	Executor restart (limited)
Hyperparameter Search	Manual implementation	Built-in (grid, random, adaptive)	Manual implementation
Monitoring	TensorBoard, nvidia-smi	Web UI (real-time metrics)	Spark Web UI (task-level)

Spark provides basic resource scheduling through its cluster manager, supporting FIFO and fair scheduling modes. While adequate for preventing resource conflicts, it lacks Determined AI's advanced features such as priority scheduling, user workspaces, and experiment-aware resource allocation.

Experiment Tracking and Reproducibility

Experiment tracking is a critical weakness in both DDP and Spark. Users must manually implement logging, typically using TensorBoard for visualization and custom scripts for hyperparameter and metric storage. This increases development overhead and makes it difficult to compare experiments done at different times or by different team members.

Determined AI provides comprehensive experiment tracking from his kit. All hyperparameters, metrics, and system information are automatically stored in PostgreSQL and visualized in the Web UI. Experiments can be easily compared, filtered, and sorted. Checkpoints are automatically managed according to configurable policies (e.g., save best, save every N epochs), eliminating code and ensuring reproducibility.

Fault Tolerance

None of the frameworks provide seamless fault tolerance in the tested configurations. DDP requires manual intervention to restart failed training runs, though PyTorch's checkpoint mechanism allows resumption from saved states if implemented by the user. Spark can automatically restart failed executors but does not provide transparent checkpoint-resume for deep learning workloads without custom implementation.

Determined AI offers the most robust fault tolerance, automatically retrying failed trials and resuming from the last saved checkpoint. While this feature was not tested in this study (due to stable hardware), it represents a significant operational advantage in production environments where node failures are inevitable.

Hyperparameter Search

Hyperparameter optimization capabilities differ significantly across frameworks. DDP provides no built-in search functionality, requiring users to manually implement custom scripts for launching multiple training runs and managing results.

Determined AI offers comprehensive built-in support, including grid search, random search, and adaptive methods such as ASHA (Asynchronous Successive Halving Algorithm). Users specify search spaces in YAML files, while the platform automatically manages parallel trial execution, early stopping, and result tracking through its Web UI.

Spark requires manual implementation. While its parallel execution capabilities can run multiple configurations simultaneously, users must write custom orchestration code without access to specialized deep learning search algorithms or automated early stopping.

Monitoring

Real-time monitoring vary across platforms. DDP relies on external tools: TensorBoard for training metrics and nvidia-smi for system monitoring. This fragmented approach complicates obtaining unified views of experiment progress and resource usage.

Determined AI provides comprehensive monitoring through its Web UI, displaying

real-time training metrics, resource utilization, and experiment progress in a unified dashboard. Users can monitor concurrent experiments, compare trends, and inspect logs without SSH access. All metrics are stored in PostgreSQL for historical analysis, significantly reducing operational overhead.

Spark's Web UI offers task-level monitoring (executor status, resource allocation) optimized for general distributed computing rather than deep learning. Training-specific metrics require manual logging via TensorBoard or custom solutions, and the interface lacks visualization features.

5.5.3 Usability and Developer Experience

While performance and management capabilities determine what a framework can achieve, usability determines how efficiently developers can leverage those capabilities. This section examines the developer experience across three key dimensions: the operational overhead of launching and monitoring experiments and the ease of integrating each framework into existing research workflows.

Ease of Experiment Launch

DDP requires SSH access to each node and manual execution of torchrun with correct parameters. This is error-prone and tedious, particularly for larger clusters. The process must be repeated for every experiment, and debugging often involves examining logs across multiple machines.

Determined AI provides the most streamlined experience. Users submit experiments with a single command (e.g., `det experiment create experiment.yaml`) from any machine with CLI access. Progress can be monitored in real-time through the Web user interface, and logs are centralized and searchable.

Spark offers an intermediate approach: spark-submit provides centralized job submission, but users must still construct correct command-line arguments and manage Python script paths. The Spark Web user interface provides visibility into task execution, but is

less user-friendly than Determined AI's interface.

Learning Curve and Documentation

DDP has the shallowest learning curve for users already familiar with PyTorch. The distributed training modifications are minimal (wrap model with DDP, use DistributedSampler), and extensive community resources and tutorials are available. However, mastering cluster management and solving communication issues requires additional expertise.

Determined AI has a steeper initial learning curve due to its declarative configuration model and orchestration concepts (trials, experiments, searchers). In terms of documentation, Determined has close to none, being basically the github everything it has. Sadly, as of the date of finishing this thesis, Determined AI project was abandoned, meaning no more documentation or community forums for beginners to consult.

Spark has the steepest learning curve, particularly for users without prior Spark experience. Understanding concepts such as executors, barrier mode, and the master-worker architecture is necessary to use Spark effectively for distributed deep learning. Additionally, the integration with PyTorch via TorchDistributor is less mature than native PyTorch DDP, with fewer examples and community resources available. Documentation exists but is often oriented toward Spark's core data processing capabilities rather than machine learning use cases.

Integration with Existing Workflows

DDP integrates seamlessly with existing PyTorch codebases, requiring minimal modifications to convert single-GPU scripts to distributed training. Standard PyTorch ecosystem tools (TensorBoard, PyTorch Lightning, Hugging Face Transformers) work without additional configuration. This makes DDP the natural choice for researchers already familiar with the PyTorch ecosystem who need distributed training with minimal workflow impact.

Determined AI requires more significant code restructuring, as training scripts must be adapted to use Determined's API (det.core). However, this investment yields benefits in terms of experiment management and reproducibility. Determined provides integrations

with popular frameworks (PyTorch, TensorFlow) and can coexist with existing tools like TensorBoard, though its native Web UI provides superior experiment tracking.

Spark requires the most substantial code restructuring, as training logic must be packaged for Spark's executor-based execution model. The TorchDistributor abstraction simplifies this compared to raw Spark RDD operations, but the separation into launcher and training scripts adds complexity. Additionally, Spark's Python environment must be carefully managed to ensure consistent library versions across executors.

5.6 Discussion

This section synthesizes the experimental findings and qualitative assessments to address the three research questions posed in Chapter 1, discusses the trade-offs between frameworks, and provides practical recommendations for framework selection.

5.6.1 Performance Efficiency

Question: Which framework achieves the fastest training time under comparable setups?

Answer: PyTorch DDP. It provides the best performance efficiency, achieving training times of 677 s (1 GPU) and 499 s (2 GPUs) for 20 epochs, 23% faster than Determined AI and 272% faster than Spark in single-GPU configuration. The performance advantage is maintained across all tested configurations, with DDP completing 100-epoch training in 3399 s compared to Determined AI's 3925 s and Spark's 12,487 s on a single GPU.

All three frameworks maintained equivalent model accuracy (0.824-0.829 validation accuracy), confirming that the performance differences reflect framework overhead rather than differences in training effectiveness. This validates that all implementations correctly synchronize gradients and maintain training equivalence to single-GPU execution.

However, the 21% overhead introduced by Determined AI (an additional 1.8 minutes for a 20-epoch run on 2 GPUs) may be acceptable in scenarios where its orchestration capabilities provide value. The performance cost is modest in absolute terms and may be justified by gains in productivity, reproducibility, and multi-user resource management.

5.6.2 Organizational and Management Complexity

Question: Which framework provides the most effective cluster management and scheduling capabilities?

Answer: Determined AI. Provides the most comprehensive cluster management and organizational capabilities, being the best option for shared research environments, organizational deployments, and scenarios requiring reproducibility and experiment tracking.

While DDP offers the simplest installation (PyTorch + NCCL), its lack of native resource scheduling, experiment tracking, and multi-user support makes it unsuitable for organizational use without significant additional tooling. Manual process launching via SSH is error-prone and doesn't scale to multi-user scenarios. Organizations adopting DDP must invest in complementary infrastructure (Slurm for scheduling, MLflow for experiment tracking, custom checkpoint management) to achieve enterprise-grade capabilities.

For organizational deployments, the recommendation is clear: Determined AI offers the best balance of performance and operational capabilities for dedicated deep learning clusters. Its 21% performance overhead is a reasonable cost for the substantial productivity and management benefits it provides.

5.6.3 Usability and Developer Experience

Question: Which framework offers the most user-friendly configuration and workflow integration? **Answer:** It depends on user expertise and workflow requirements.

For PyTorch experts seeking minimal friction: DDP provides the most natural development experience. Converting existing single-GPU PyTorch code to DDP requires minimal changes (typically 5-10 lines: initialize process group, wrap model, use DistributedSampler). The PyTorch ecosystem's extensive documentation, tutorials, and community support make troubleshooting straightforward. However, the manual cluster management and lack of experiment tracking impose ongoing operational burdens.

For teams prioritizing reproducibility and collaboration: Determined AI offers

superior long-term usability despite a steeper learning curve. The declarative configuration model (YAML) separates hyperparameters from code, enhancing reproducibility. The Web UI provides intuitive experiment comparison and metric visualization, reducing the need for custom dashboards. Centralized logging and debugging tools streamline troubleshooting. Once users internalize Determined's concepts (trials, experiments, searchers), productivity increases significantly compared to manually managing DDP experiments.

For users embedded in Spark ecosystems: Spark's usability is acceptable but not exceptional. The TorchDistributor API simplifies PyTorch integration compared to raw Spark operations, but the executor-based model requires code restructuring that feels foreign to PyTorch users. Documentation is adequate but less comprehensive than PyTorch's. Debugging distributed failures can be challenging due to the driver-executor communication model.

Overall, DDP provides the best developer experience for individual researchers who prioritize rapid prototyping and are comfortable with manual cluster management. Determined AI provides the best developer experience for teams and organizations where collaboration, reproducibility, and long-term maintainability outweigh the learning effort.

5.6.4 Framework Trade-offs and Positioning

The experimental results reveal that framework selection involves navigating three primary trade-off dimensions:

Performance vs. Ease of Use

DDP offers maximum performance but minimal automation. Users gain control and no orchestration overhead at the cost of manual resource management and experiment tracking. This is optimal for:

- Single-user research environments
- Scenarios where speed is the focus

Determined AI sacrifices 21% performance for comprehensive automation. This trade-off favors scenarios where:

- Multiple users share computational resources
- Reproducibility and experiment tracking are critical
- Fault tolerance is required (long-running experiments, unstable hardware)
- Development time and operational overhead outweigh raw training speed

Control vs. Automation

DDP offers maximum control, users explicitly manage every aspect of distributed execution. This flexibility enables advanced use cases (custom communication patterns, mixed precision strategies, gradient accumulation) but requires expertise to implement correctly.

Determined AI prioritizes automation over control. Common workflows (checkpointing, metric logging, hyperparameter tuning) are handled automatically, but customization requires working within Determined's abstractions. This is appropriate when standard workflows suffice and productivity is more valuable than flexibility.

Spark occupies a middle ground, offering programmable task orchestration while automating resource management. However, its generality makes it less optimized for deep learning compared to specialized frameworks.

Specialization vs. Generality

DDP and Determined AI are purpose-built for deep learning, with APIs and optimizations tailored to neural network training. This specialization offers superior performance and usability within their target domain but offers limited capabilities beyond model training.

Spark is a general-purpose distributed computing framework adaptable to deep learning. Its generality enables unified pipelines that combine data preprocessing, model training, and inference, valuable when training is one component of a larger data workflow.

However, this generality comes at the cost of deep learning-specific optimizations, resulting in lower performance for pure training workloads.

5.7 Summary

This chapter presented a comprehensive evaluation of PyTorch DDP, Determined AI, and Apache Spark for distributed deep learning training. The key findings are:

If the focus is a simple implementation with the fastest times, DDP is the best choice.

If it's possible to sacrifice some time to have a very complete interface, automation and a better organization management, then Determined is the best choice.

For Spark the only reason to choose it over the other two frameworks would be if the code is running in a Spark infrastructure and there is a lot of data processing.

Chapter 6

Conclusions

This thesis presented a comprehensive comparative analysis of three distributed deep learning training frameworks: PyTorch DDP, Apache Spark, and Determined AI. The study evaluated these frameworks across three critical dimensions: training performance, cluster management capabilities, and developer usability, with the objective of providing practical guidance for framework selection in real-world distributed AI deployments.

6.1 Research contributions and assessment

This work makes several contributions to distributed machine learning practice:

- Empirical benchmarking under controlled conditions, providing quantitative performance data that bridges the gap between theoretical capabilities and actual deployment results.
- Organizational perspective on framework evaluation, examining resource scheduling, multi-user coordination, and operational overhead, factors critical for real world environments yet frequently overlooked in academic comparisons.
- Actionable guidance based on measured trade-offs, enabling searchers to select frameworks aligned with their specific constraints rather than relying on vendor documentation alone.

- The study successfully answered the three research questions posed initially:
  - Performance: PyTorch DDP achieved the fastest training times across all configurations, making it optimal when raw speed is the priority.
  - Management: Determined AI provided superior organizational capabilities, recommended for shared research clusters and production environments requiring reproducibility and automated orchestration.
  - Usability: Context-dependent, DDP suits individual experts prioritizing rapid iteration, while Determined AI better serves teams needing collaboration infrastructure and long-term maintainability.

6.2 Future work

Several research directions would extend this work:

- Scalability at larger scales: Testing on 8-64 GPU clusters would reveal how communication overhead and efficiency evolve with increased parallelism, identifying practical scaling limits for each framework.
- Model parallelism evaluation: Extending beyond data parallelism to include pipeline and tensor parallelism would assess frameworks designed for extreme-scale models (DeepSpeed, Megatron-LM) that exceed single-device memory.
- Heterogeneous hardware: Real-world clusters often mix GPU generations and architectures. Understanding how frameworks handle resource heterogeneity would increase practical applicability.
- Broader framework coverage: Including Horovod, Ray, and PyTorch FSDP would provide a more complete landscape of distributed training options, each with distinct architectural trade-offs.

6.3 Final Remarks

Distributed machine learning frameworks embody a fundamental tension between performance, automation, and flexibility. No single solution dominates, each occupies a distinct position in the trade-off space between raw speed, operational convenience, and ecosystem integration. Based on this work, Determined AI would be the strongest choice for organizational deployments, offering comprehensive management features and an intuitive interface at a modest 21% performance cost. However, the project appears to have been discontinued, raising questions about its long-term viability. As distributed training continues evolving through advances in communication optimization and memory efficiency, the comparative methodology established here enables systematic evaluation of emerging frameworks against proven baselines. The ultimate goal remains democratizing access to large-scale AI training, reducing barriers that prevent researchers and organizations from developing increasingly sophisticated models that advance the state of the art.

Bibliography

[1] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "Pytorch distributed: Experiences on accelerating data parallel training", *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3005–3018, Aug. 2020, ISSN: 2150-8097. DOI: 10.14778/3415478.3415530. [Online]. Available: https://doi.org/10.14778/3415478.3415530.

[2] Determined-ai, *Determined: An open-source machine learning platform for distributed training, hyperparameter tuning and experiment tracking*, https://github.com/determined-ai/determined.

[3] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing", *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016, ISSN: 0001-0782. DOI: 10.1145/2934664. [Online]. Available: https://doi.org/10.1145/2934664.

[4] X. Tu, Z. He, Y. Huang, Z.-H. Zhang, M. Yang, and J. Zhao, "An overview of large ai models and their applications", *Visual Intelligence*, vol. 2, Dec. 2024. DOI: 10.1007/s44267-024-00065-8.

[5] S. Han, H. Mao, and W. J. Dally, *Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding*, 2016. arXiv: 1510.00149 [cs.CV]. [Online]. Available: https://arxiv.org/abs/1510.00149.

[6] S. Wang, H. Zheng, X. Wen, and S. Fu, "Distributed high-performance computing methods for accelerating deep learning training", *Journal of Knowledge Learning*

and Science Technology ISSN: 2959-6386 (online), vol. 3, no. 3, pp. 108–126, Sep. 2024. DOI: 10.60087/jklst.v3.n3.p108-126. [Online]. Available: https://jklst.org/index.php/home/article/view/230.

[7] R. Mayer and H.-A. Jacobsen, "Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools", *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020, ISSN: 0360-0300. DOI: 10.1145/3363554. [Online]. Available: https://doi.org/10.1145/3363554.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[9] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour", *arXiv preprint arXiv:1706.02677*, 2017.

[10] X. Li, C. Guo, K. Qian, M. Zhang, M. Yang, and M. Xu, "Near-lossless gradient compression for data-parallel distributed dnn training", in *Proceedings of the 2024 ACM Symposium on Cloud Computing*, ser. SoCC '24, Redmond, WA, USA: Association for Computing Machinery, 2024, pp. 977–994, ISBN: 9798400712869. DOI: 10.1145/3698038.3698541. [Online]. Available: https://doi.org/10.1145/3698038.3698541.

[11] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, Version Number: 2, 2018. DOI: 10.48550/ARXIV.1810.04805. [Online]. Available: https://arxiv.org/abs/1810.04805.

[12] TensorFlow, *Distributed training with tensorflow*, https://www.tensorflow.org/guide/distributed_training, Accessed: 2024-10-27, 2024.

[13] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism", *ArXiv*, vol. abs/1811.06965, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:260441206.

[14] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia, "Efficient large-scale language model training on GPU clusters using megatron-LM", en, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, St. Louis Missouri: ACM, Nov. 2021, pp. 1–15, ISBN: 978-1-4503-8442-1. DOI: 10.1145/3458817.3476209. [Online]. Available: https://dl.acm.org/doi/10.1145/3458817.3476209.

[15] OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, R. Avila, I. Babuschkin, S. Balaji, V. Balcom, P. Baltescu, H. Bao, M. Bavarian, J. Belgum, and B. Zoph, *Gpt-4 technical report*, Mar. 2023. DOI: 10.48550/arXiv.2303.08774.

[16] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters", in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '20, Virtual Event, CA, USA: Association for Computing Machinery, 2020, pp. 3505–3506, ISBN: 9781450379984. DOI: 10.1145/3394486.3406703. [Online]. Available: https://doi.org/10.1145/3394486.3406703.

[17] Q. Zhou, Q. Anthony, L. Xu, A. Shafi, M. Abduljabbar, H. Subramoni, and D. K. D. Panda, "Accelerating distributed deep learning training with compression assisted allgather and reduce-scatter communication", in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 134–144. DOI: 10.1109/IPDPS54959.2023.00023.

[18] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "Zero-offload: Democratizing billion-scale model training", Cited by: 228,

2021, pp. 551–564. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85107474478&partnerID=40&md5=e4f78d63be8a83a1cb056ecdea5a547a.

[19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library", Cited by: 33482, vol. 32, 2019. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85090176877&partnerID=40&md5=b8ecd177286b903fcb2493893027cce0.

[20] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, *Tensorflow: A system for large-scale machine learning*, 2016. arXiv: 1605.08695 [cs.DC]. [Online]. Available: https://arxiv.org/abs/1605.08695.

[21] A. Sergeev and M. D. Balso, *Horovod: Fast and easy distributed deep learning in tensorflow*, 2018. arXiv: 1802.05799 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1802.05799.

[22] J. Duan, S. Zhang, Z. Wang, L. Jiang, W. Qu, Q. Hu, G. Wang, Q. Weng, H. Yan, X. Zhang, X. Qiu, D. Lin, Y. Wen, X. Jin, T. Zhang, and P. Sun, *Efficient training of large language models on distributed infrastructures: A survey*, 2024. arXiv: 2407.20018 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2407.20018.

[23] D. Kaul, "Ai-driven self-healing container orchestration framework for energy-efficient kubernetes clusters", *Emerging Science Journal*, vol. 2024, pp. 1–13, Dec. 2024.

[24] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda", in *ACM SIGGRAPH 2008 Classes*, ser. SIGGRAPH '08, Los Angeles, California: Association for Computing Machinery, 2008, ISBN: 9781450378451. DOI: 10.1145/1401132.1401152. [Online]. Available: https://doi.org/10.1145/1401132.1401152.

[25] M. De Benedictis and A. Lioy, "Integrity verification of docker containers for a lightweight cloud environment", *Future Generation Computer Systems*, vol. 97, pp. 236–246, 2019, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2019.02.026. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X18327201.

[26] I. Loshchilov and F. Hutter, *Decoupled Weight Decay Regularization*, Version Number: 3, 2017. DOI: 10.48550/ARXIV.1711.05101. [Online]. Available: https://arxiv.org/abs/1711.05101.