UNIVERSITY OF CALIFORNIA SAN DIEGO

Cerebro - An Efficient, End-to-End Platform for Scalable Deep Learning

A Thesis submitted in partial satisfaction of the
requirements for the degree of Master of Science

in

Computer Science

by

Pradyumna Sridhara

Committee in charge:

Professor Arun Kumar, Chair
Professor Alin Deutsch
Professor Rose Yu

2023

The Thesis of Pradyumna Sridhara is approved and is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2023

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge Prof.Arun Kumar for his support as the chair of my committee. His expertise in the domain and insightful discussions about the research were instrumental in completing this thesis. I am deeply grateful to him for providing an enriching environment at AdaLab in the past one and a half years which has helped me gain invaluable research experience. I hold great admiration for his mentorship, work ethics and unwavering transparency.

Additionally, I would like to express my gratitude to Prof. K.V. Subramaniam from PES University, who kindled my interest in the fields of Machine Learning Systems and Cloud Computing, and set me off on the journey that has led me here.

My sincere thanks to Vignesh Nanda Kumar for all his contributions and for being a constant companion in this project. My gratitude also to Yuhao Zhang and Kabir Nagrecha, whose inputs have greatly helped me in this work.

Special thanks to Shreyas S. Rao who always ensures that help is there when most needed, and to Bhanu Garg for making my time with this program fun.

My deep appreciation to Sarah Iwatsuki for supporting this research in her own special way, right from the very beginning.

And lastly, I would like to express my gratitude to my parents for their constant support throughout my life and for being exemplary role models.

To Ambi & Appiu for all their love and support.

ABSTRACT OF THE THESIS

Cerebro - An Efficient, End-to-End Platform for Scalable Deep Learning

by

Pradyumna Sridhara

Master of Science in Computer Science

University of California San Diego, 2023

Professor Arun Kumar, Chair

Deep Learning (DL) has emerged as a powerful tool for solving complex problems in various domains, including natural language processing and computer vision. DL, being an empirical process, requires tuning of hyperparameters and exploring neural network architectures which involve significant compute resources, storage, memory, time, and human effort. While tools exist to address challenges associated with large datasets or with large DL models, there is a notable scarcity of comprehensive solutions that efficiently handle both large-scale models as well as large-scale datasets. The advent of Transformers and Large Language Models(LLMs) have underlined these problems and made overcoming them ever more significant. Unlike big tech, these issues are particularly acute for small-scale companies and individuals. There is a

need to democratize large-scale DL.

As a response, we propose a novel end-to-end platform that provides efficient scaling of DL in a cluster, regardless of the size of the datasets or models. Our platform can preprocess data, train, validate, and test models, as well as visualize results - all under one roof. Cerebro achieves this through its novel scheduler which is a hybrid of task, data and model parallelism. Our design supports fault tolerance and cluster resource heterogeneity. Implementing Cerebro's user-friendly templates makes scaling DL effortless, allowing users to work seamlessly with the same familiarity as on their local machines. To evaluate our platform, we conducted experiments on various DL tasks, including image captioning and object detection. The experiments demonstrated that our platform provides efficient scaling of DL workloads, significantly reducing the time, effort, and resource costs required for large-scale model selection.

This thesis describes the methods and approaches taken in the design and development of the Cerebro platform. We also discuss in detail our experimental observations of Cerebro in action and outline the directions this work can take in the future.

Chapter 1

Introduction

Deep Learning (DL) has emerged as a transformative force revolutionizing various fields, ranging from computer vision and natural language processing to healthcare and autonomous driving systems. This paradigm shift has been driven by the ability of DL models to learn complex representations from large amounts of data, especially multi-modal non-tabular modalities such as images, video and text. DL, being an empirical process, necessitates the careful selection and tuning of hyperparameters to optimize model accuracy. These hyperparameters govern the behavior of the learning algorithm and consequently, tuning these hyperparameters is of utmost importance to achieve state-of-the-art results and extract the full potential of DL technology. However, this process of selecting the best model becomes challenging since the number of possible combinations grows exponentially with the number of hyperparameters [1]. DL's iterative nature involving repeated training and evaluation cycles, and the reliance on heuristic approaches and empirical observations make this process extremely compute resource intensive and demands significant time investments from ML Engineers. This problem is exacerbated as the size of datasets and models continues to grow [2]. Increasingly large-scale applications demand processing enormous datasets and training intricate models to extract valuable insights. This results in models, especially those based on transformers [3] such as GPT4 [4], taking several days to train.

Therefore, it is crucial to have systems that can scale the process of DL model building

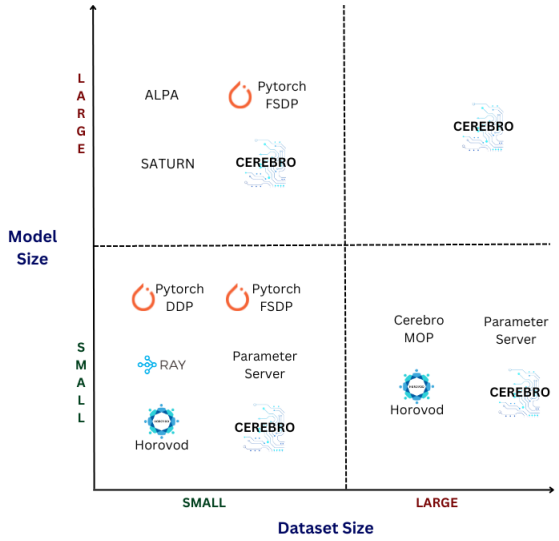


Figure 1.1. Comparison of the existing DL scaling landscape across dataset size and model size axes

onto multi-node clusters, allowing for parallel training of models. These systems must be able to scale on both the dataset size and the model size axes. Furthermore, to keep the process efficient and seamless, these systems must support the end-to-end DL workflow in a unified tool and not see the different parts of the pipeline as disjoint.

1.1 System Desiderata

We have the following key desiderata for a DL model selection system.

1. **Data Scalability.** Today's DL needs large datasets which often can't be trained on a single node, due to memory and compute constraints. Therefore, we need support for data scalability across multiple nodes.

2. **Model Scalability.** Since newer DL models are large and don't fit on a single GPU's memory, we need out-of-the-box model sharding across GPUs.

3. **Resource Efficiency and Convergence Guarantee.** The system must ensure judicial

and optimal usage of resources. These are of the following categories: *traditional and accelerated compute efficiency*: CPU and GPU utilization; *memory/storage efficiency*: the amount of memory and storage used; *communication efficiency*: amount of network bandwidth used by the system. The system also must ensure that other benefits are not at the cost of model accuracy. It must retain the expected convergence guarantees.

4. **Generality of Model Selection**: The system's design should allow for multiple model selection techniques such as grid search, random search and human-in-the-loop.

5. **Generality of Parallelism Selection**: Since no single model-sharding algorithm will be suitable for all models, the system must ensure the best available parallelism is chosen to shard each model.

6. **Fault Tolerance**: In a cloud setting, where resources may fail at any time, the system must be resilient to recover from resource loss and restart without intervention and with minimal overhead in efficiency.

7. **End-to-end Capability**: The system must support the entire DL pipeline from data pre-processing to training, validation, and testing, all under a single platform. Auxiliary features such as model visualizations may be supported.

1.2 Limitations of Existing Landscape

We examine the existing landscape of research and explore notable contributions in the field of scaling DL. We compare and contrast these works with the present study, highlighting the novel approaches and advancements that distinguish this work from the existing body of knowledge. A summary of this is depicted in 1.1

- **False dichotomy of model scalability and data scalability**: Most of the prior art in this domain restricts its focus to one of either data scalability [5], [6], [7] or model scalability

[8], [9], [10]. This leaves out a significant gap in applications where scaling on both these domains is needed. Figure 1.1 compares the existing landscape in this dichotomy.

- **DL workflow viewed in disjoint silos instead of a cohesive pipeline**: Prior art [5], [6], [7], [8], [9], [10] view data pre-processing, model building, and metrics visualization as independent entities. This results in multiple independent tools for each of these tasks. The user is expected to resolve the challenges that arise from this unnecessary bifurcation. These include but are not limited to, ensuring compatibility between these various entities, their setup, and installation, and most importantly, creating a pipeline that feeds the results of one entity as input to the next.

- **Lack of Generality**: Different scenarios call for different strategies of model selection and model sharding techniques. Therefore DL scaling tools must ensure that they support a variety of these strategies and techniques. Some prior art [11], [12] fails in this regard, partly due to the false dichotomy stated earlier.

1.3 Our Proposed Approach

We present Cerebro, a system for efficiently scaling DL model selection. As figure 1.1 shows, Cerebro can work with very large datasets as well as very large models. This is done using a novel scheduler that is a hybrid of task, data and, model parallelisms. Cerebro is designed to run on cloud clusters that provide accelerated computing resources such as GPUs. Our interface breaks down the DL model selection process into smaller templates, which the user can implement. Cerebro internally parallelizes these implementations, abstracting away the underlying complexity, and providing users with an experience that mimics working on a single machine.

We embrace the principle that data pre-processing is a crucial component of the DL pipeline and it aligns with our overarching objective of parallelizing operations across the

Table 1.1. Qualitative comparison of the existing scalable DL landscape on key desiderata

		Data Scalability	**Model Scalability**	**End-to -End Capability**	**Model Selection Aware**	**Model Parallelism Generality**
Data Parallel	Pytorch DDP	✓	✗	✗	✗	✗
	Horovod	✓	✗	✗	✗	✗
	Parameter Server	✓	✗	✗	✗	✗
Task Parallel	Ray	✗	✗	✓	✓	✗
Model Parallel	Pytorch FSDP	✗	✓	✗	✗	✗
Hybrid Parallel	Cerebro - MOP	✓	✗	✓	✓	✗
	Alpa	✗	✓	✗	✗	✓
	Saturn	✗	✓	✗	✓	✓
Data-Task -Model Parallel	***Cerebro (ours)***	✓	✓	✓	✓	✓

available cluster resources. We employ the Extract-Transform-Load (ETL) paradigm as it is well-suited for data preprocessing in our case [13]. Cerebro uses a two-step parallelization scheme for efficient data preprocessing - at the node level and at the process level. For model building, Cerebro uses Model-Hopper-Parallelism [12] a data system that can efficiently scale model selection over large datasets. Here, the dataset is sharded across the nodes of the cluster and each model train on a shard before hopping to the next node. To support large models that don't fit on a single GPU, we borrow from Saturn [14].

Chapter 2

Background and Related Works

Here, we explain the process of deep learning, its methodologies and challenges. We explore different patterns in distributed learning and deep-dive into some of these algorithms - including the prior art of Model Hopper Parallelism and Saturn, on which this work has been built.

Further, we contextualize DL within a broader framework, considering the accompanying infrastructure essential for its sustained deployment in real-world scenarios. We delve into the contemporary technology stack employed for DL development. We provide a concise overview of the orchestration platform used for the development of this work - Kubernetes, its necessity, advantages and its role in enabling DL via the cloud.

2.1 Deep Learning and Model Selection

Deep Learning [15] stems from Artificial Neural Networks, within the broader field of Machine Learning. Similar to the human brain, these artificial neural networks are composed of interconnected layers of artificial neurons and excel at capturing intricate patterns and extracting complex representations from data. DL focuses on the training and utilization of neural networks with multiple layers of artificial neurons.

Stochastic Gradient Descent (SGD) [16] serves as the fundamental optimization algorithm for neural networks and DL. It updates the model's parameters iteratively, to minimize a given

loss function, by randomly sampling mini-batches of data. SGD is a stochastic approximation of gradient descent since it uses the gradients calculated on a randomly selected subset of the data. SGD assumes that the training data is independent and identically distributed (IID), which means that all data points are sampled from the same underlying distribution. This assumption allows it to optimize the model parameters by iteratively updating them based on the gradients estimated from the mini-batches. Each iteration is called an epoch. Repeated sampling and updating allow SGD to gradually converge towards a good solution. One of the key advantages of SGD over other optimization algorithms is its compatibility with distributed learning techniques [17]. In some cases, parallel SGD can outperform its centralized counterpart algorithm [18]. The reader can refer to [19] and [20] for more technical details on SGD.

SGD is commonly employed as a heuristic optimization algorithm for specifically tackling non-convex optimization problems, a category to which DL belongs. In contrast to convex optimization problems that possess a single global minimum, non-convex optimization problems exhibit the presence of multiple local minima, which significantly complicates the search for the optimal solution. DL models, being trained on such non-convex objective functions, require careful fine-tuning of their hyperparameters - such as learning rate, batch size, number of neural network layers, etc. - to ensure better model performance and convergence [21]. With the multitude of hyperparameters available and the crucial impact of having the best configuration among them, their fine-tuning often involves exploring a wide range of permutations, assessing their impact on the model's behavior, and iteratively refining the choices based on evaluation metrics and validation results. The fine-tuning of hyperparameters, along with other design considerations such as the choice of neural network architecture, the use of certain techniques like regularization or dropout, and more, constitute the process of model selection.

2.2 Distributed Learning Techniques and Algorithms

As datasets grow larger and models become more complex, the computational demands of DL model selection and training also escalate. This creates the need for distributed learning, which allows for the parallelization of DL computation across multiple computing resources enabling faster and more efficient training by distributing the workload. In the context of distributed DL, these computing resources can be nodes and/or GPUs within a node - we will refer to them simply as 'workers'. Distributed Learning is also employed to overcome the resource limitations posed by single, centralized systems. There are several techniques for parallelizing DL training. Here, we discuss three primary categories of techniques relevant to this work - Task Parallelism, Data Parallelsim, and Model Parallelism. For a more in-depth study of these techniques, the reader can refer to [22] and [23].

Task Parallelism is perhaps the simplest technique, where tasks can be executed concurrently on multiple workers by replicating the dataset onto each worker and executing the set of tasks independently from each other. In the context of DL, tasks are models or model configurations in the model-selection space. There are no communication costs and no loss of convergence efficiency. Its advantage is in its simplicity and its ability to train multiple models in parallel. The main drawback of this technique is the dataset replication, which does not scale to large datasets. It is highly resource inefficient - of both memory and storage. One 'solution' can be to read the data from shared remote storage, but this is merely a trade-off between storage costs and network costs. It would overload the network with huge amounts of redundant data. Ray [11], Celery, and Dask [24] fall into this category.

Data Parallelism refers to partitioning the dataset into smaller shards and distributing it to multiple workers, where each worker then operates the same task on its shard simultaneously, allowing for parallel execution across multiple processing units. Such techniques are used to train DL models on large-scale datasets which might not fit on a single worker. During training, the model parameters or gradients are synchronized among the workers to ensure consistency.

This involves network communication between the different workers to perform collective operations such as gradient averaging or model aggregation. Data Parallelism suffers from high communication costs and sometimes, poor convergence. Examples of this technique include Parameter Server [6] and Horovod [5]. This technique is particularly useful in situations were the data is by its nature distributed, as in the case of Federated Learning [25].

Model Parallelism refers to the partitioning or sharding of the neural architecture graph into smaller sub-graphs and each worker then operates on its sub-graph. Once the computations are complete or have reached an intermediate stage, they are synchronized among all or some of the workers. Different ways of doing this step have led to different model parallelism algorithms [26]. However, this introduces a synchronization barrier between different workers and is often the bottleneck in model parallelism algorithms. This also mandates, in some way, inter-worker communication which adds overheads. This can be mitigated to an extent, by using technologies such as "NVLink" in newer Nvidia GPUs to speed up communication. Model Parallelism is useful when models don't fit on a single GPU's memory. Some of the popular model parallelism techniques include PyTorch FSDP [8] (based on ZeRO [27]), GPipe [28] and Alpa [9].

2.2.1 Model Hopper Parallelism

Model Hopper Parallelism (MOP for short) is a novel hybrid of task parallelism and data parallelism that offers efficient model selection on a cluster. It combines the advantages of both task and data parallelism while mitigating the disadvantages of either. MOP takes advantage of SGD's robustness to the random ordering of data that it operates over [19]. This is due to the property of the data being independent and identically distributed (IID). The basic idea of MOP is as follows: Assume there is a set S of training configurations, and there is a set M consisting of W number of workers that can participate in the training. We need to train each configuration in S for e epochs.

First, the dataset is partitioned into W partitions, such that each worker gets one partition or shard. Next, MOP picks W configurations from S, assigns one configuration to each worker,

and trains them on that worker's shard. This constitutes a "sub-epoch" since each training configuration has only seen a $\frac{1}{W}^{th}$ fraction of the entire dataset. At the end of a sub-epoch, the worker is marked as idle. Note that each worker might complete its sub-epoch at different times. Next, for every idle worker m, the MOP scheduler picks a configuration s from set S, such that: 1. configuration s has not visited worker m before, and; 2) configuration s is not currently being trained on any worker in M. This updated configuration s is then scheduled on worker m and trained over its shard. This essentially means that configuration s "hopped" from a previous worker to the new worker m. Hence the name Model Hopper. Similarly, all configurations hop across to all workers, visiting each worker exactly once. When all configurations have visited all workers, we complete one full epoch. This process is then repeated for e epochs, completing the model selection process. Though models are being trained on multiple workers in parallel, MOP is theoretically identical to running model training on a single machine over the entire dataset.

2.2.2 Saturn

Saturn is a system designed to optimize multi-large model DL workloads. It focuses on 3 main challenges in cases where models don't necessarily fit on a single GPU's memory - tackling parallelism selection, resource apportioning, and scheduling. Saturn operates in the context of multi-GPU clusters. Here, parallelism selection refers to the process of determining the method and granularity of model-sharding. There are several popular model-sharding techniques or "parallelisms" that are each better suited for specific models. Furthermore, there might be novel parallelisms and reinforced versions of existing parallelisms in the future. Saturn aims to incorporate these parallelisms by providing a 'parallelism interface' to enable it to work with other components. Resource allocation refers to the process of assigning computing resources (such as GPUs) to the model training tasks. Multiple models are being trained on the same workers, each with different resource requirements and these requirements may change over time as the models are trained. The goal of resource allocation is to use the available resources efficiently and minimize overall runtime. Finally, the challenge of scheduling is about

the sequence of resource allocation among the several models. Different sequences can have different runtimes on the task and these differences can be significant. Therefore, the challenge is to find the best sequence of model-worker pairs that achieves task completion in the least amount of time.

Saturn argues that the three challenges of parallelism selection, resource apportioning, and scheduling are interdependent and have to be addressed simultaneously. For example, insufficient allocation of resources to a task may result in prolonged execution times, negatively affecting overall performance. Conversely, inadequate scheduling of tasks can lead to idle periods on GPUs and overload on others. Therefore, addressing the interdependence of parallelism selection, resource apportioning, and scheduling is crucial. This is together called the SPASE problem.

To address this, Saturn proposes the joint optimization of the SPASE problem, which involves using mixed-integer linear programs (MILP) that optimizes all three components of the SPASE problem - parallelism selection, resource allocation, and schedule construction - simultaneously. The MILP solver takes into account factors such as model size, dataset size, and training complexity to determine how best to allocate resources among the different models. To assist the MILP solver, Saturn employs a profiler. The profiler's role is to estimate the runtime of each model on different parallelisms and configurations. It does this by running a small number of iterations (mini-batches) of each model on different parallelisms and configurations and measuring their performance. The profiler then extrapolates these performance values to estimate the end-to-end runtime of each model on different parallelisms and configurations.

In this way, Saturn can achieve practically optimal solutions to all three dimensions of the SPASE problem, thereby enabling faster and more efficient model selection in large multi-model DL tasks.

2.3 Deep Learning Pipeline

Employing and maintaining a DL system in the real-world in a complex process with several steps. It begins with data acquisition, where relevant data is collected from various sources, such as databases, APIs, or sensor devices. Once the data is obtained, it undergoes a cleaning process to remove outliers, errors, missing values, or even offensive or inappropriate data [29], ensuring high-quality and reliable input for the DL model. Following data cleaning, the pipeline moves to data preprocessing. Here, the raw data is transformed into a format suitable for deep learning models. This typically involves tasks such as feature extraction, normalization, scaling, and handling categorical variables. Extract, Transform and Load (ETL) techniques are commonly employed to efficiently process and prepare the data for training.

The preprocessed data is used to train the deep learning model. Training involves feeding the data into the model, computing the gradients, and updating the model's parameters using optimization algorithms like SGD. The objective is to optimize the model's performance by minimizing a defined loss function. To assess the model's performance and prevent overfitting, a separate validation set is utilized. The trained model is evaluated on the validation set, and the hyperparameters are fine-tuned based on these results. Iterative adjustments and validations are performed until satisfactory performance is achieved. Following validation, the pipeline proceeds to the testing phase. The model's performance is evaluated on an independent test set to obtain unbiased estimates of its accuracy and generalization. This step provides insights into the model's performance on real-world data. The final model is then deployed for inferencing. Figure 2.1 shows the entire process pictorially. DL code constitutes only a fraction of real-world ML Systems [30]. Several tools and technologies are needed to support this infrastructure.

Large-scale GPU clusters are becoming more popular [31] leaving Cloud platforms as a natural choice to provision such resources when needed and tear-down when done. However, managing cloud infrastructure is a complex undertaking due to the highly flexible and adaptable nature of cloud resources, the extent of their scalability, and the diverse set of resources and

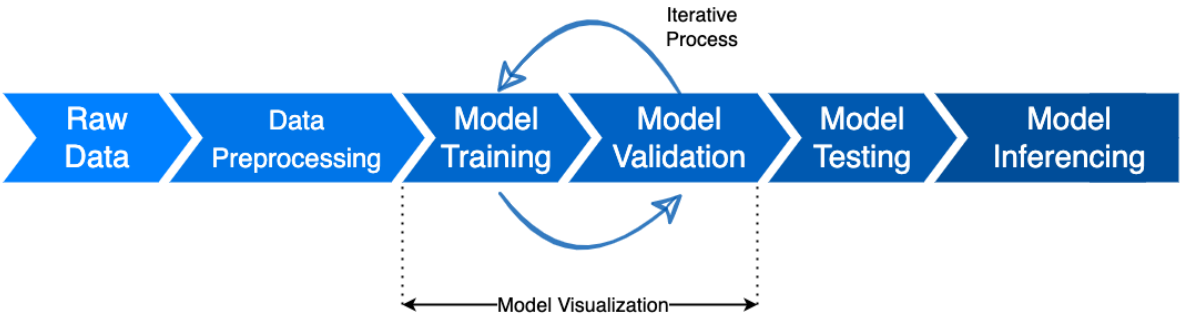


Figure 2.1. The end-to-end Deep Learning pipeline

services they offer. The need for a robust orchestration platform to manage these resources is vital [32]. This is where Kubernetes, an open-source container orchestration platform, proves to be a valuable solution. Kubernetes simplifies the management of cloud infrastructure by providing automated deployment, scaling, and management of containerized applications [33]. It offers features such as automatic load balancing, self-healing capabilities, and declarative configuration, which greatly enhance the resilience and scalability of applications running on cloud resources. The use of such orchestration technologies enables developers to focus on the applications they build rather than on the infrastructure that it runs on. Kubernetes has emerged as a leading choice for managing complex cloud infrastructure [34].

Chapter 3

System Architecture

Cerebro is designed to have a modular, microservices-inspired architecture, that enables various components to work with each other while remaining decoupled. This allows Cerebro to easily scale individual components, increasing fault tolerance [35]. Modularity allows the use of different technologies and frameworks for each service, providing the flexibility to choose the best tool for the job. With future advancements, better-suited frameworks or tools can be swapped out on the technology stack while retaining others.

To ensure the seamless delivery of all system features and capabilities, the adoption of a cluster management system was necessary. Kubernetes was chosen due to its capacity to easily manage multiple cluster resources via container orchestration. Furthermore, for the system to be production-ready, it was imperative to implement a layer of reliable automation for such tasks, for which Kubernetes proved to be an ideal candidate [36]. Although our architecture is agnostic of the cluster-management system, we have included certain aspects of Kubernetes nomenclature for easier explainability.

In this chapter, we present the architecture of Cerebro, explaining the roles and responsibilities of each of the various components involved. Later, we delve into the data flow, with a step-by-step description of the process orchestration. And finally, we explore some of the design considerations and tradeoffs that influenced the development of Cerebro and present a comprehensive justification for them.

3.1 Components

Here, we explain each component in Cerebro's architecture as a standalone service. Figure 3.1 gives an overview of these components. The next section 3.2 deals with how these components interact with each other as a sequence of events.

All Cerebro components mentioned exist in the context of a multi-node cloud cluster. We support public clouds, on-premise private clouds as well as hybrid clouds. We provision a dedicated Cerebro control-plane node, namely the Controller node. We recommend that Cerebro's control-plane node be provisioned on-demand. All other nodes in the cluster are designated as worker nodes. Worker nodes tend to have higher amounts of resources, including accelerated computing units such as GPUs. These need not be on-demand nodes.

3.1.1 Cerebro Controller

The Cerebro Controller is the central component in our framework. It orchestrates all data flow within the system. It is scheduled to run on the Controller node as a Kubernetes Deployment[1] for automatic restarts on node or application failure, although not designed to be fully fault-tolerant. It runs the JupyterNotebook user interface as an independent Python process. The user will initiate all workflows through JupyterNotebook. Additionally, it runs the Model Monitoring visualization tool, also as a Python process.

In the data pre-processing phase, the Cerebro Controller is responsible for initializing worker nodes to install required Python packages, downloading and shading the user-provided metadata, initiating the data pre-processing task on all workers, and finally, the cleaning up of worker resources once the job is done. During the pre-processing, the controller will update the progress (aggregated across all workers) to the user, via JupyterNotebook. For the next phase, the Cerebro Controller places trials with different parallelisms for each of the model configurations present. The collection and storing of the resulting data is also orchestrated by the Cerebro

[1] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

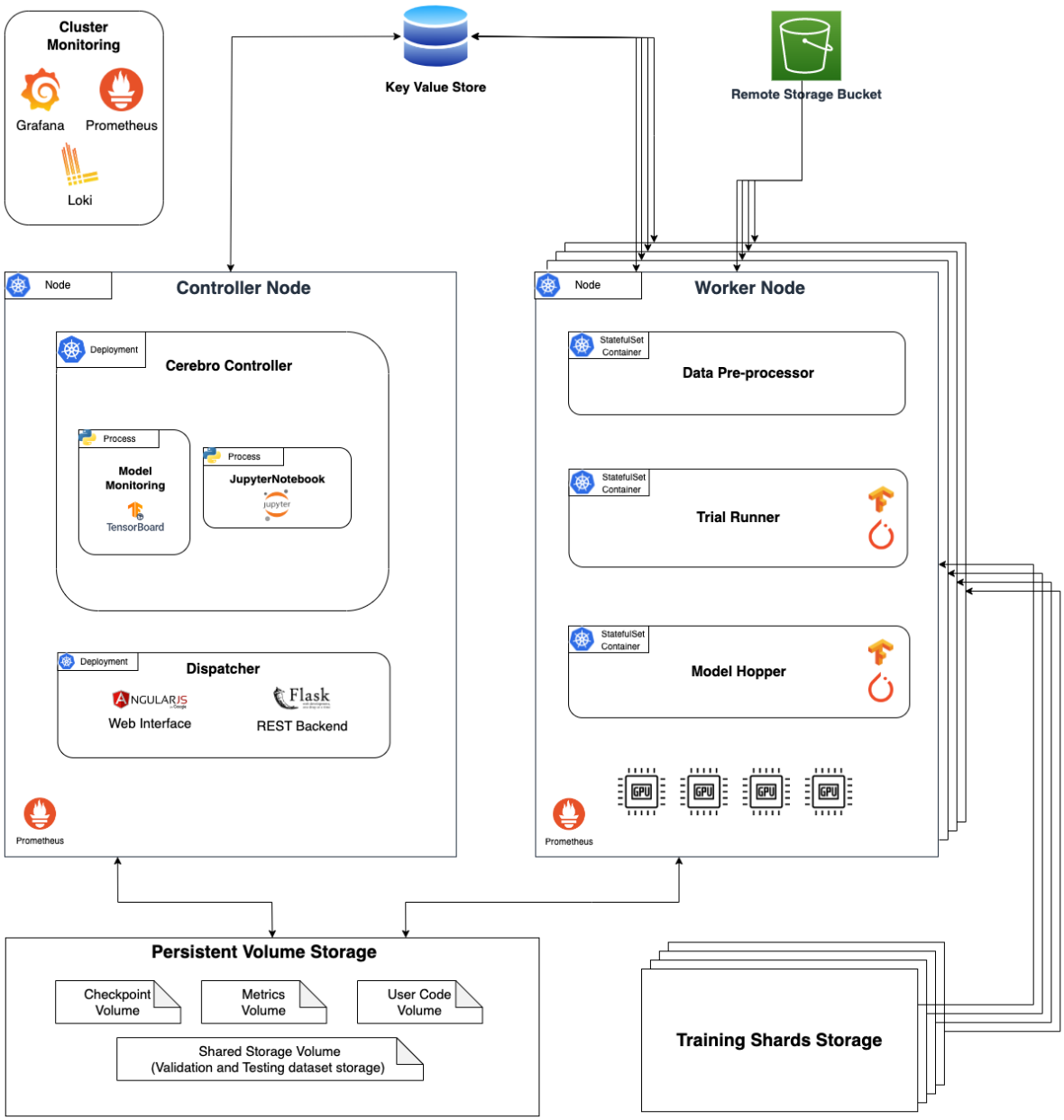


Figure 3.1. Cerebro's architecture and technology stack of our implementation

Controller. In the model-building phase, the Cerebro Controller, via the unified scheduler, is responsible for generating the task schedule. This dictates which model configuration should run on which node and using which parallelism. It also maintains the state of the tasks and all other data that is required to orchestrate Model Hopper Parallelism. Also, it is responsible for maintaining the state of the DL workload pipeline, to recover in the event of a node/application failure.

Apart from the JupyterNotebook and the Model Monitoring tool, other Cerebro Controller tasks are not a continually running process, rather it is invoked through the user's interaction with Cerebro.

3.1.2 Dispatcher

The Dispatcher houses the landing web user interface of Cerebro. It runs on the Controller node as a Kubernetes Deployment. It also has a backend web service to assist the fulfillment of the web page's services. These services include collecting and saving the Dataset Locators from the user. And also, the distribution of the users' code files to the Cerebro Controller and all Workers. Once this data is collected from the user, the UI presents the user with links to the JupyterNotebook, Model Monitoring tool, and Cluster Monitoring tool. The Dispatcher is a continually running web server, available throughout the lifecycle of the DL workload pipeline.

3.1.3 Data Pre-processor

The Data Pre-Processor or ETL phase of the pipeline downloads the datasets from any remote storage and processes it so that it is ready to be consumed by the models for training or inference. Cerebro sees the Data Pre-processing as a purely data-parallel workload. On each worker node, the Pre-processor runs as a StatefulSet[2] Container in Kubernetes and each instance get its partition of data. This partition is then further divided into smaller shards and each CPU core processes this shard using the user-provided code. The division of each node's partition

[2]https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

into smaller shards is carried out by the Pre-processor, whereas the division of the dataset into partitions is done upstream. The processed data is saved on the cluster attached storage.

3.1.4 Model Hopper

Similar to the TrialRunner, an instance of this component runs on each node of the cluster, as a StatefulSet Container. It fetches a model configuration and a parallelism scheme from upstream. The specific model in question is read from its checkpoint on the attached checkpoint volume. The model training code is run on the entire data shard of its node. If the epoch for that model ends on the worker in question, the model validation task is executed. Testing and Validation phases are done in a semi-task-parallel manner. Training and validation metrics specified by the user are aggregated and pushed downstream. This component too, makes use of any accelerated computing units on the node. PyTorch or Tensorflow is used for running the DL workloads.

3.1.5 Key-Value Store

The Key-Value Store, or KVS for short, is a managed database to store Key-Value pairs that are used to maintain the state of the DL workflow pipeline. It is used to store the information provided by the user, such as the data pre-processing code, the model building. and model evaluation codes. It is also used to store Cerebro's orchestration information such as results of the current stage of the model training phase, among others. This component can be deployed in high availability mode via Kubernetes, to protect against failures which can in turn bring down the cluster. It can also be housed on the on-demand provisioned Controller node. In Cerebro, the KVS is exposed via an API service that can be called by any of the other components for read-write operations. Since there are multiple actors at play in parallel, the managed database must exercise concurrency control for these read-write operations.

A note on fault tolerance: Fault tolerant components of Cerebro, upon failure, are designed to be automatically restarted. They recover by querying their last successful state

with KVS, and redoing operations if deemed necessary to catch up with the current state of the pipeline. All necessary data for the component to recover is available in the KVS as they are written ahead of time, before each task. The rest of the components proceed till they reach a pre-designated checkpoint and wait for all faulty component instances to catch up. If these components fail to catch up after a specified amount of time, the design allows for a separate control sequence can be triggered to halt the entire operation and notify the user of the error. We employ write-ahead logging (WAL) since it is best suited for distributed microservices architectures [37] and more specifically for the use-case of Cerebro.

3.1.6 Storage

Cerebro needs shared as well as standalone persistent storage services to function. This storage is independent of the node and should not cease to exist on node failures. From the shared storage block, we allocate multiple volumes, each for a dedicated purpose. These volumes are mounted onto multiple Pods[3], each of which can read or write on the volume. We have the following shared volumes -

- **Checkpoint Volume** - Here, we read and write the checkpoints of each model configuration after every epoch
- **Metrics Volume** - Model Training and Validation metrics are written to this volume, during the model-building phase. The Model Monitoring tool will read from here and present them as visualizations to the user
- **User Code Volume** - User-uploaded code files, that are referenced while running the DL workload pipeline are hosted here. This is mounted onto all Worker node pods.
- **Shared Storage Volume** - This is a large shared storage space to house processed validation and testing data. It is also used for sharing the partitioned metadata files between the Cerebro Controller and the respective Workers

[3]https://kubernetes.io/docs/concepts/workloads/pods/

In addition to these shared volumes, we have one standalone volume per Worker node. This is a large volume that is for storing training data, both before and after pre-processing.

3.1.7 Monitoring

Cerebro supports both Model monitoring as well as Cluster Monitoring out of the box. These can be third-party libraries that are installed to run as a cluster-wide application. Model monitoring involves continuous polling and fetching of training and validation metrics from the Metrics storage volume. This is then visualized as time-series graphs and presented to the user as a web interface. These metrics are user-specific and are defined in the model training or validation code specification. Tools that can be configured for this purpose include Tensorboard and Weights and Biases. An internal log monitoring system can also be included, to gather, in one place, logs from the various components running on multiple nodes. This can be exposed only to the cluster administrator and not the end user. Tools like Loki and Prometheus can be used here.

3.1.8 Future Work: Trial Runner

[4] An instance of the Trial Runner component runs on each node of the cluster, as a StatefulSet Container. It fetches a model configuration and a parallelism scheme from upstream, specific to each instance of the TrialRunner. The model-building code is run on a small sample of the processed data on its node, for that particular model configuration + parallelism scheme combination. The metrics from this trial run are then sent downstream. This component makes use of any accelerated computing units on the node, if present. The number of such units used depends on the parallelism scheme, but usually, all available units are put to use. PyTorch or Tensorflow is used for running the DL workloads.

[4] At the time of writing, this component is still a work-in-progress

Initialize Data Preprocessing

```
class MyETLSpec(ETLSpec):
    def __init__(self):
        self.is_feature_download = []

    def initialize_worker(self):
        pass

    def row_preprocessor(self, row, mode, object_dir):
        return None, None
```

Initialize Model Building

```
class MySubEpochSpec(SubEpochSpec):
    def __init__(self):
        pass

    def initialize_worker(self):
        pass

    def create_models(self):
        return list()

    def train(self, models, checkpoint, dataloader, hyperparams):
        return dict()

    def test(self, models, checkpoint, dataloader, hyperparams):
        return dict()
```

Figure 3.2. Data Preprocessing and Model Building abstractions that the user implements

3.2 Data Pre-processing and Model Building

In this section, we take a step-by-step approach to explain the flow of data and control through the components in our architecture. We delve into each phase of the DL workload pipeline.

3.2.1 User begins DL Workload

Figure 3.3 showcases the steps executed immediately after the user triggers the beginning of the DL workload pipeline. The user auxiliary code files and Dataset Locators to Cerebro via the Dispatcher. The Dispatcher saves the Dataset Locators to the KVS, from where multiple actors will read, in later steps. The auxiliary code files are mounted to each Worker node via the User Code Volume. The user provides DL workload functions such as the Data Pre-processing routines, the model train and validation routines, via the JupyterNotebook. Here, the Cerebro Controller is invoked wherein these functions are encoded as strings and saved in the KVS for

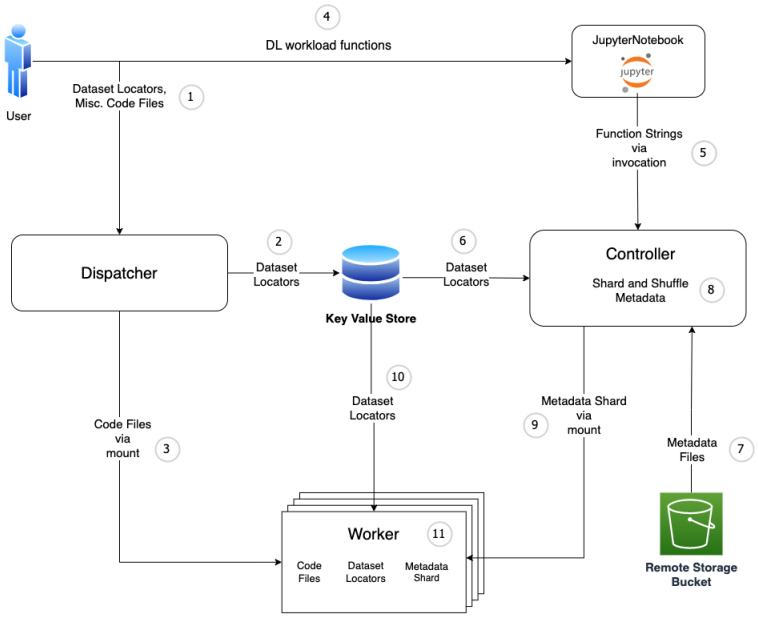


Figure 3.3. User triggers the start of the DL workload pipeline

later access. It then pulls the Dataset Locator URIs from the KVS and using that, the metadata files from the remote storage bucket. Finally, these metadata files are shuffled and sharded into as many shards as there are Workers. Metadata shards are then transferred to their respective workers via the Shared Storage Volume mount. The Worker nodes now have access to their Metadata shard, Dataset Locator URIs from the KVS, as well as the auxiliary code files.

3.2.2 Data Pre-processing

As Figure 3.4 shows, the data pre-processing pipeline begins with the user sending the pre-processing code to the Controller via the JupyterNotebook. A copy of this code is saved in KVS as an encoded string. The data pre-processor container on the Worker nodes pulls this from the KVS. They also have access to their Metadata partition via the shared storage volume mount. This partition is further split into smaller shards and each CPU core on that node is assigned the task of processing its shard. This might involve pulling object files from the Remote Storage bucket. This occurs when there are multi-modal objects such as images or video files in the dataset. Throughout this process, the progress of pre-processing the shards from all cores are

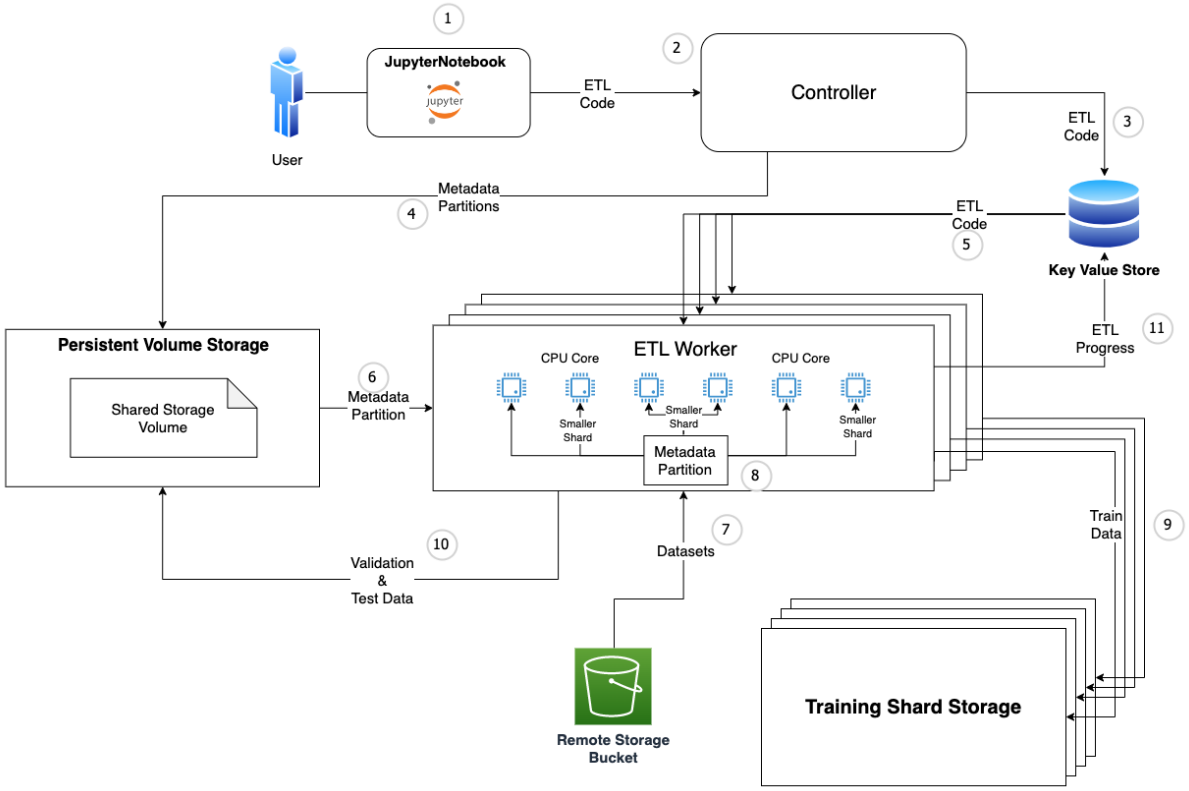


Figure 3.4. Dataflow in the Pre-processing phase of the DL workload pipeline

aggregated and saved to KVS. This information is later conveyed to the user via the Conroller. Once pre-processing is complete, this data is stored on the Training Shard Storage volume, if the data in question was the training data. Validation and Test data is similarly stored on the Shared Storage volume. This completes the Data Pre-processing phase of the pipeline.

3.2.3 Model Building

The data flow of the model-building phase is shown in Figure 3.5. Similar to the Data Pre-processing phase, the user invokes via the JupyterNotebook. The model train and validation function strings are saved in KVS. The Cerebro Controller also formulates a schedule for each Worker, containing three pieces of information - (worker_id, model configuration, parallelism scheme) and pushes this to KVS. The Worker then reads the model configuration and parallelism scheme associated with its worker_id from the KVS. This information is routed through the KVS so that,

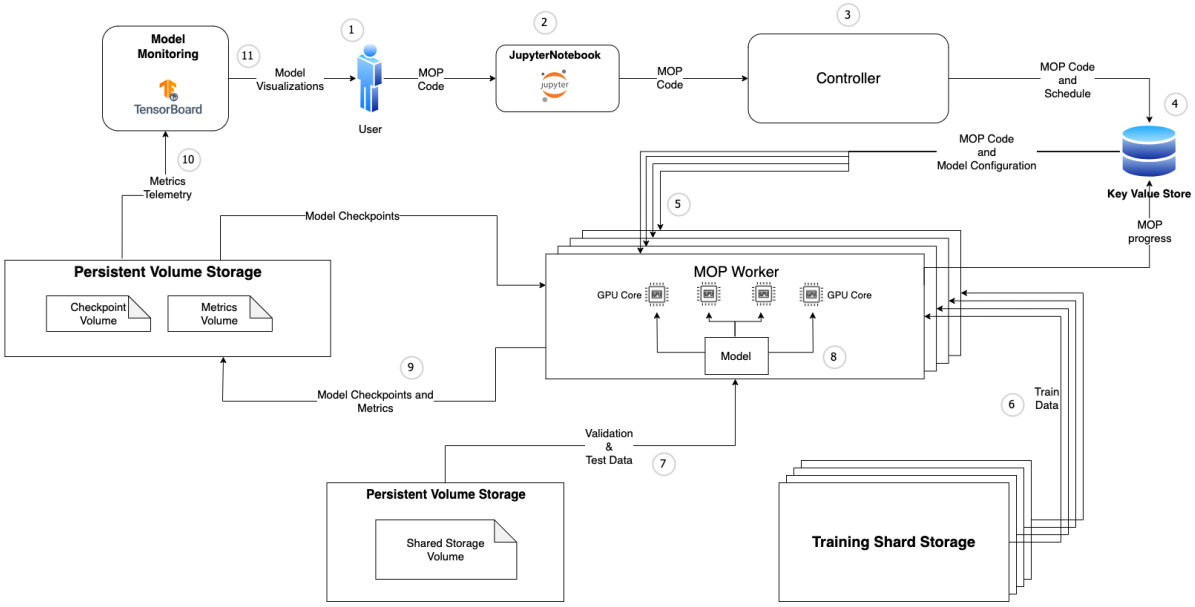


Figure 3.5. Dataflow in the Model Building phase of the DL workload pipeline

in the event of a failure, the workers can resume their task by consulting the KVS, after restart. The Worker then loads the model's checkpoint into memory from the Checkpoint volume, the train data from the Training Shard Storage volume, and as needed, the validation and test data from the Shared Storage volume. Model checkpoints and metrics are written to their respective volumes periodically. The Cerebro Controller monitors the completion of every epoch on every worker and schedules the next model configuration to run, till all epochs for all model configurations are complete.

3.2.4 Future Work: Model Trials

[5] The trial runner dataflow is similar to that of Model Building. Here, the Cerebro Controller's schedule ensures that all model configurations are sampled on all possible parallelism schemes to determine which pair works best. This information is then saved back to KVS. As Figure 3.6 shows, we need only the Training data shard to be mounted to the trial-runner container. Few batches of training data are sampled from the entire dataset.

[5] At the time of writing, this component is still a work-in-progress

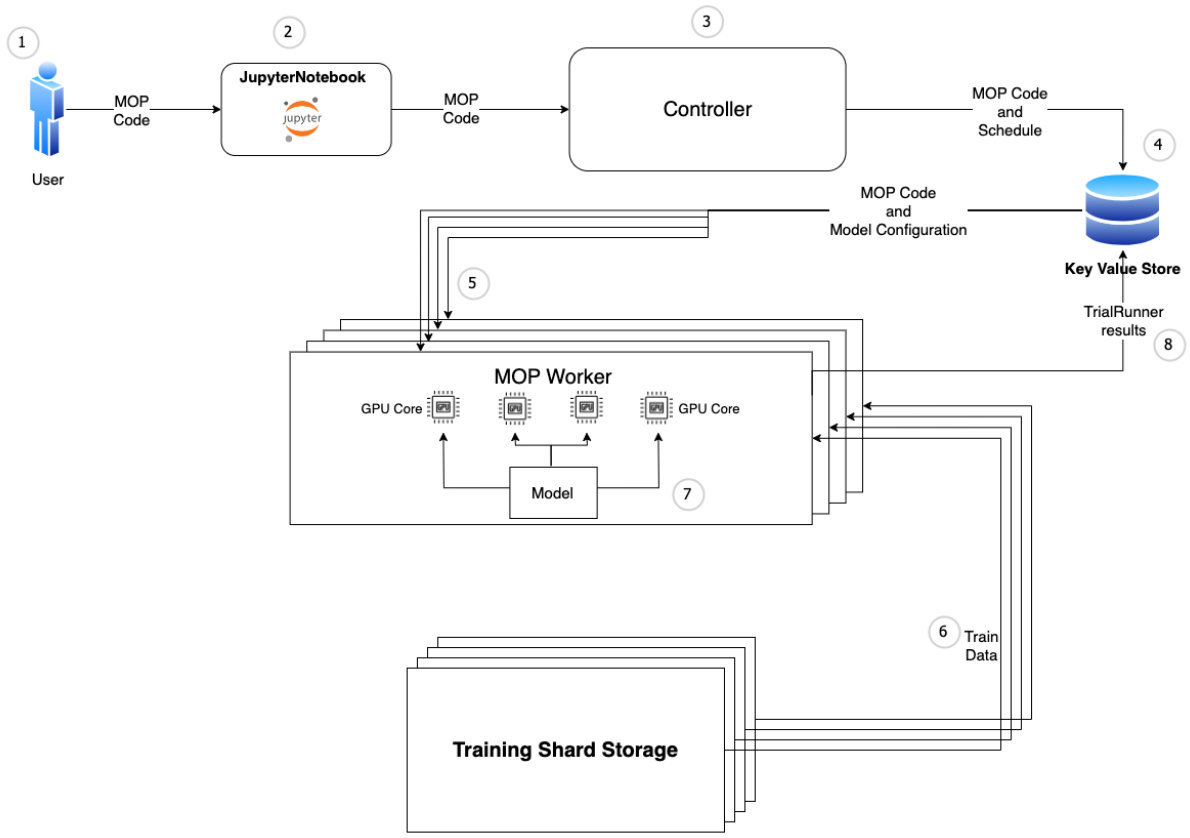


Figure 3.6. Dataflow in the Trial-Runner phase of the DL workload pipeline (this component is still a work-in-progress)

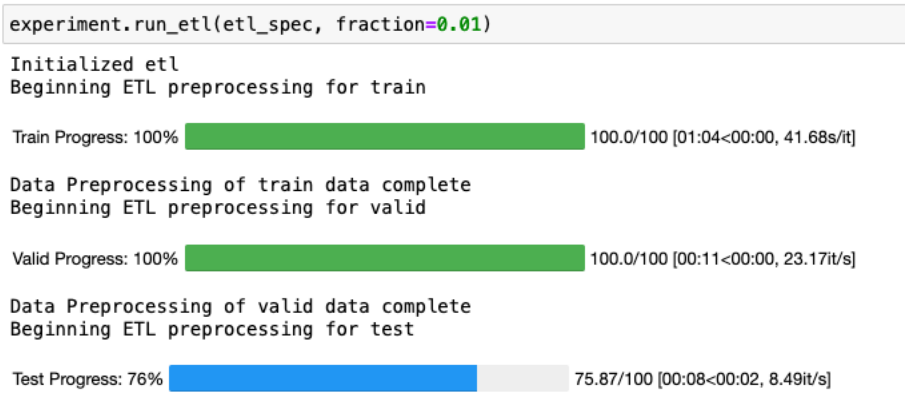


Figure 3.7. A screenshot of the JupyterNotebook showing the progress of Data Preprocessing

3.3 Cerebro's User Interface

The user interacts with the Cerebro system via JupyterNotebook. Unlike other systems such as Ray, which use Python decorators for this purpose, we believe that JupyterNotebook is a more natural way to interact with Machine Learning systems as studies have shown [38]. The user implements data pre-processing and model-building templates as shown in Figure 3.2. This abstraction design has evolved over several iterations to ensure maximum flexibility in covering most of the user's DL applications. Cerebro interacts with information snippets and progress via the same interface as shown in Figure 3.7.

Chapter 4

Unified Scheduler

Optimal scheduling plays a crucial role in effective resource utilization especially when dealing with parallel systems, which in our case, are multi-GPU multi-node clusters. Since DL workloads are compute-heavy, a sub-optimal scheduler can significantly impact the time taken for training. In this chapter, we highlight the significance of an optimized parallel job scheduler and delve into the inner workings of Cerebro's scheduler. Also mentioned are the details of the iterative modifications that lead to the scheduler's current version. Our approach integrates elements from the Model Hopper Parallelism scheduler [12] and Saturn's scheduler [14] to create a unified system capable of harnessing the combined power of data, task, and model parallelism within a single job.

4.1 The Scheduling Problem and its Complexity

Here, the jobs are that of training multiple model configurations generated in the process of model selection. The resources on which these jobs have to be scheduled include the cluster's nodes (and by extension, the data partitions on that node) and the several GPUs available on these nodes. To enable large model training, we need to also select the best parallelism strategy for each model configuration. Therefore, an optimal schedule involves choosing the sequence of which model configuration to train on which node's data partition, using how many GPUs and what parallelism strategy such that the overall makespan is minimal. We provide a formal

definition of this scheduling problem in Section 4.3.

Based on prior art, we can logically break down the problem into two parts - task-data parallelism scheduling from Model Hopper Parallelism and task-model parallelism scheduling from Saturn. Model Hopper Parallelism uses a hybrid data-task parallelism where several models have to be trained on multiple nodes and the data has been partitioned across all nodes. Each model must train on each data partition exactly once and the ordering of this sequence is irrelevant. This is an instance of the open-shop scheduling problem, which is known to be NP-Hard [39]. On the other hand, Saturn shards the models and trains them on a given number of GPUs using the best parallelism strategy for each model. Since the model's layers have to be trained in sequence, the ordering becomes relevant. This is an instance of the job-shop scheduling problem which is also NP-Hard [40]. Saturn tries to simplify this problem by using inputs from the user on GPU allocation.

The nature of this scheduling problem allows us to treat it as analogous to multi-query optimization from RDBMS [41] to be solved via joint optimization. In Section 4.3, we represent the problem as mixed-integer linear programming (MILP). As input to this MILP formulation, we need training time estimates for each model configuration. With this data, we can solve the MILP using an optimizer such as Gurobi [42]. However, given the complexity of the problem, we found that the optimizer was unable to reach convergence in a reasonable timeframe of 300 seconds. We explored alternative heuristical approaches to approximate an optimal solution, as described in Section 4.4. From our simulation results, we can see that the random scheduler is a very good approximation of the MILP-solved scheduler, but without the added dependencies.

4.2 Cerebro's Unified Scheduler

Before scheduling the model selection process, we first do a trial run to determine the best model sharding parallelism for each model, borrowing from Saturn. This is done by sampling a few mini-batches of train data and running model training for each {model configuration, model

parallelism strategy} combination. This data is then extrapolated to give an estimate of the actual runtimes. The parallelism that yields the least runtime is then fixed as the best strategy for that model configuration. Since the design supports heterogeneity of node resources, this exercise has to be specific to each node on the cluster. With this data, we can proceed with the model selection scheduling. Cerebro's heuristic unified scheduler works on two levels - at the cluster level and at the node level. At the cluster level, a worker is represented by a node. We partition the data proportional to the number of GPUs on the worker, such that workers with more GPUs get a higher portion of the data. This is to ensure that, in a heterogenous cluster, nodes with a lesser number of GPUs are not stragglers, which can cause significant delays [43]. We proceed by allocating one model per worker and have the models hop around workers as they complete a sub-epoch, as per Model Hopper Parallelism. This is done by randomly choosing any eligible {model, worker} pair. The training of these models is done at the node level, where a worker is represented by a GPU. The model is sharded as per its parallelism strategy. We allocate all the GPUs on that node to train the given model. This level is treated as a black box by the upper-level scheduler.

4.2.1 Multi-GPU MOP Scheduler

A simpler variation of the above scheduler was designed as a baseline benchmark. Here, we assume that models fit on GPU memory and do not need sharding. Unlike the unified scheduler where one model is allocated per node, here we allocate one model per GPU. All data shards are of equal size and each GPU has its own data shard. Model Hopper Parallelism then ensures that models hop over to each GPU.

4.2.2 Alternate Schedulers for Cerebro

Since the unified scheduler is random, it is not always guaranteed to approximate the optimal solution well. This is due to the inherent variance present in the schedule search space. To reduce this variation in the outcomes and obtain a more stable estimate of the optimized

schedule, we can employ Monte Carlo Averaging [44]. From the current point in the training, subsequent schedules can be simulated multiple times via dry-runs and the best schedule can be picked. This is at a minimal cost of both compute and time.

To move closer to the optimal schedule, a more complex but effective method would be to warm-start the scheduler and interleave the MILP solver with model training. We begin with a randomized schedule, and while the first sub-epochs are being trained, we warm-start the MILP solver with the random schedule and run it to convergence in parallel. This optimal result can then be used for all subsequent scheduling, without the downside of having to wait for MILP convergence.

Additionally, to counter variances generated from fault recovery, we suggest that these optimizations be run multiple times throughout the lifespan of the scheduling process to ensure better results.

4.3 Formal Problem Statement as MILP

In this section, we describe Cerebro's scheduler as a formal mixed-integer linear program (MILP). We define the MILP input variables as shown in 4.1. The variables whose values are selected by solving the program are also shown.

Equation 4.1 specifies that Makespan must be higher than start time + time taken by the last task.

$$\begin{gathered} C \geq I_{t,n,g} + R_{t,n,s} - U \times (1 - B_{t,n,s}) \\ \forall s \in S_{t,n}, \forall t \in T, \forall n \in N, \forall g \in GPU_n \end{gathered} \tag{4.1}$$

Equation 4.2 specifies that exactly one strategy should be selected for a task on any node.

$$\begin{gathered} \sum_{x \in B_{t,n}} x = 1 \\ \forall t \in T, \forall n \in N \end{gathered} \tag{4.2}$$

Table 4.1. List of variables and their description used for formulating the Cerebro scheduler problem as MILP

MILP Inputs	
Symbol	Description
N	Number of nodes available for execution
T	Number of model tasks to train
GPU_n	Number of GPUs available in each node
$S_{t,n}$	GPU configurations available for a task on a node
$G_{t,n,s}$	Number of GPUs requested for a task on a node with a particular strategy
$R_{t,n,s}$	Runtime of a task on a node with a particular strategy
MILP Selected Variables	
Symbol	Description
$B_{t,n,s} \in 0,1$	Binary variable for whether task t uses strategy s on node
$P_{t,n,g} \in 0,1$	Binary variable for whether task t trained on GPU g of node n
$O_{t,n,s} \in 0,1$	Binary variable for whether task t ran on node n
$I_{t,n,g} \in \mathbb{R}^+$	Start time of training of task t on GPU g of node n
$A_{n,t_1,t_2,g} \in 0,1$	Binary variable of whether task t_1 ran before task t_2 on GPU g of node n
$X_{t,n_1,n_2} \in 0,1$	Binary variable of whether task t ran on node n_1 before node n_2

Equation 4.3 and 4.4 specify that number of GPUs alloted to a task on a node should be equal to the number of GPUs requested for that task on the node.

$$\sum_{x \in P_{t,n}} x \geq G_{t,n,s} - U \times (2 - B_{t,n,s} - O_{t,n,s}) \quad \forall s \in S_{t,n}, \forall t \in T, \forall n \in N \tag{4.3}$$

$$\sum_{x \in P_{t,n}} x \leq G_{t,n,s} + U \times (2 - B_{t,n,s} - O_{t,n,s}) \quad \forall s \in S_{t,n}, \forall t \in T, \forall n \in N \tag{4.4}$$

Equation 4.5 and 4.6 specify all GPUs running a task on a node have to start at the same time.

$$\frac{\sum_{x \in I_{t,n}} x}{G_{t,n,s}} \leq I_{t,n,g} + U \times (3 - P_{t,n,g} - B_{t,n,s} - O_{t,n,s}) \quad \forall s \in S_{t,n}, \forall t \in T, \forall n \in N, \forall g \in GPU_n \tag{4.5}$$

$$\frac{\sum_{x \in I_{t,n}} x}{G_{t,n,s}} \geq I_{t,n,g} - U \times (3 - P_{t,n,g} - B_{t,n,s} - O_{t,n,s}) \quad \forall s \in S_{t,n}, \forall t \in T, \forall n \in N, \forall g \in GPU_n \tag{4.6}$$

Equation 4.7 and 4.8 specify that two tasks shouldn't run on the same GPU at the same time.

$$I_{t_1,n,g} \leq I_{t_2,n,g} - R_{t_1,n,s} + U \times (3 - P_{t_1,n,g} - P_{t_2,n,g} - B_{t_1,n,s} + A_{n,t_2,t_1}) \quad \forall s \in S_{t_1,n}, \forall t_1 \in T, \forall t_2 \in (T - \{t_1\}), \forall n \in N, \forall g \in GPU_n \tag{4.7}$$

$$I_{t_1,n,g} \geq I_{t_2,n,g} + R_{t_2,n,s} - U \times (4 - P_{t_1,n,g} - P_{t_2,n,g} - B_{t_2,n,s} - A_{n,t_2,t_1}) \quad \forall s \in S_{t_2,n}, \forall t_1 \in T, \forall t_2 \in (T - \{t_1\}), \forall n \in N, \forall g \in GPU_n \tag{4.8}$$

Alternately, equations 4.7 and 4.8 can be combined into a single bi-linear equation 4.9 as below

$$\begin{aligned}(2\cdot A_{n,t_1,t_2,g}-1)\cdot I_{t_1,n,g}+A_{n,t_1,t_2,g}\cdot R_{t_1,n,s_1}\leq(1-2\cdot A_{n,t_2,t_1,g})\cdot I_{t_2,n,g}\\ -A_{n,t_2,t_1,g}\cdot R_{t_2,n,s_2}+U\times(3-P_{t_1,n,g}-P_{t_2,n,g}-(A_{n,t_1,t_2,g}\cdot B_{t_1,n,s_1})\\ -(A_{n,t_2,t_1,g}\cdot B_{t_2,n,s_2}))\\ \forall s_1\in S_{t_1,n},\forall s_2\in S_{t_2,n},\forall t_1,t_2\in T \text{ such that } t_1<t_2,\forall n\in N,\forall g\in GPU_n\end{aligned}\tag{4.9}$$

Equation 4.10 and 4.11 specify that two nodes shouldn't run the same task at the same time.

$$\begin{aligned}I_{t,n_1,g1}\leq I_{t,n_2,g2}-R_{t,n_1,s}+U\times(3-P_{t,n_1,g1}-P_{t,n_2,g2}-B_{t,n_1,s}+X_{t,n_2,n_1})\\ \forall s\in S_{t,n_1},\forall t\in T,\forall n_1\in N,\forall n_2\in(N-\{n_1\}),\forall g1\in GPU_{n_1},\forall g2\in GPU_{n_2}\end{aligned}\tag{4.10}$$

$$\begin{aligned}I_{t,n_1,g1}\geq I_{t,n_2,g2}+R_{t,n_2,s}-U\times(4-P_{t,n_1,g1}-P_{t,n_2,g2}-B_{t,n_2,s}-X_{t,n_2,n_1})\\ \forall s\in S_{t,n_2},\forall t\in T,\forall n_1\in N,\forall n_2\in(N-\{n_1\}),\forall g1\in GPU_{n_1},\forall g2\in GPU_{n_2}\end{aligned}\tag{4.11}$$

Alternately, equations 4.10 and 4.11 can be combined into a single bi-linear equation 4.12 as below

$$\begin{aligned}(2\cdot X_{t,n_1,n_2}-1)\cdot I_{t,n_1,g}+X_{t,n_1,n_2}\cdot R_{t,n_1,s_1}\leq(1-2\cdot X_{t,n_2,n_1})\cdot I_{t,n_2,g}\\ -X_{t,n_2,n_1}\cdot R_{t,n_2,s_2}+U\times(3-P_{t,n_1,g}-P_{t,n_2,g}-(X_{n,t_1,t_2}\cdot B_{t,n_1,s_1})\\ -(X_{t,n_2,n_1}\cdot B_{t,n_2,s_2}))\\ \forall s_1\in S_{t,n_1},\forall s_2\in S_{t,n_2},\forall t\in T,\forall n_1,n_2\in N, \text{ such that } n_1<n_2,\forall g\in GPU_n\end{aligned}\tag{4.12}$$

Equation 4.13 specifies that start times for all tasks cannot be negative.

$$\begin{aligned}I_{t,n,g}\geq 0\\ \forall t\in T,\forall n\in N,\forall g\in GPU_n\end{aligned}\tag{4.13}$$

Equation 4.14 specifies that two nodes can only have one ordering of execution for a particular

task.

$$X_{t,n_1,n_2} + X_{t,n_2,n_1} = 1$$
$$\forall t \in T, \forall n_1 \in N, \forall n_2 \in (N - \{n_1\}) \tag{4.14}$$

4.4 Scheduler Simulations

To evaluate heuristic approximations of the MILP solved schedule, we conduct simulations of several scheduling strategies. We assume a linear speed-up in training time with each additional GPU as observed by experiments from [10]. All GPUs are assumed to be identical and all models fit on the GPU's memory. The training time extrapolation is computed using the formula -

$$\begin{aligned}&\text{Training time for 1 model on 1 worker}\\ &= \frac{\text{training time for 1 model on 1 shard on 1 GPU} \times \text{\# of data shards on the worker}}{\text{\# of GPUs used on that worker}}\end{aligned} \tag{4.15}$$

We have defined the following strategies used in the simulations:

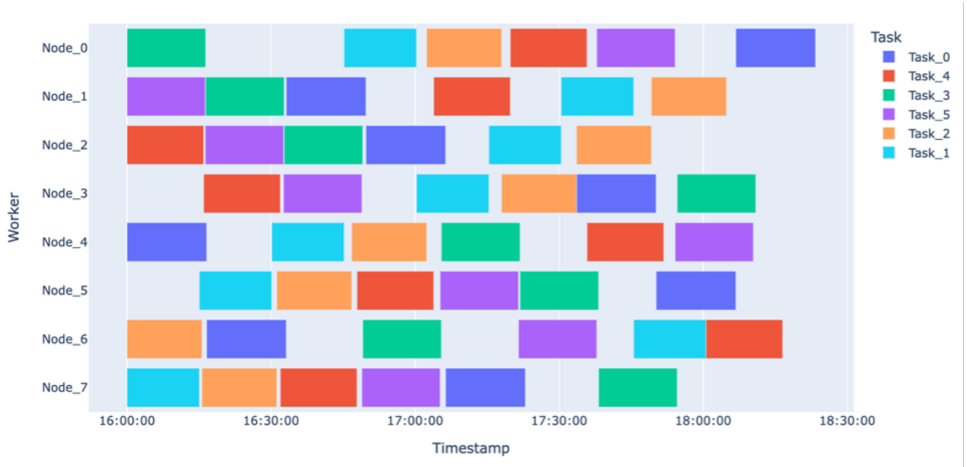
- **Multi-GPU MOP Scheduler**: We allocate one model per GPU where each GPU has its shard. The concept of a cluster node is not relevant. The training time under this schedule is considered the benchmark, relative to which we measure other strategies. Eligible {model, worker} pairs are randomly picked and scheduled.
- **Random Scheduler**: Here, we allocate one model per node, and the model trains on all GPUs of the node. Data is sharded proportional to the number of GPUs on the node. Eligible {model, worker} pairs are randomly picked and scheduled.
- **Cerebro-MILP**: Here, a model can run on one or more GPUs. Data is sharded proportional to the number of GPUs on the node. Each model has to visit a node exactly once. Scheduling is given by the Gurobi optimizer, by solving the MILP equation from Section

4.3. The cutoff time for the Gurobi solver was 300s and could not reach convergence in that time.

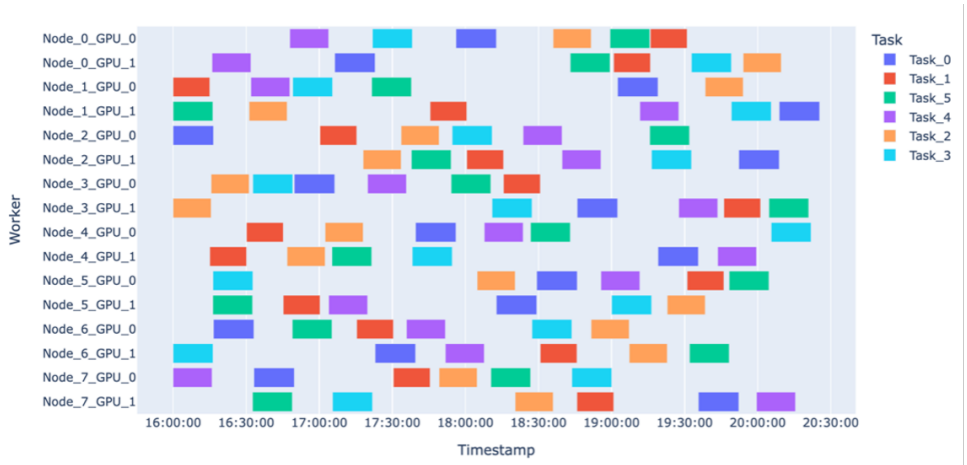
To make the simulations as realistic as possible, we consider different scenarios of model selection search space sizes and different heterogeneous cluster configurations. We have the following scenarios, grouped on the number of models to work with:

- **number of models < number of nodes**: Figure 4.1 shows that Cerebro-MILP outperforms the other two schedulers. However, the random scheduler's makespan is quite close to that of Cerebro-MILP.
- **number of nodes < number of models < number of GPUs**: Surprisingly, Figure 4.2 shows that the Random Scheduler's makespan is better than that of Cerebro-MILP, although by a thin margin. This is because the MILP solver hasn't converged on the best solution.
- **number of GPUs < number of models**: In this case too, as per Figure 4.3, the Random Scheduler's solution is better than that of Cerebro-MILP.

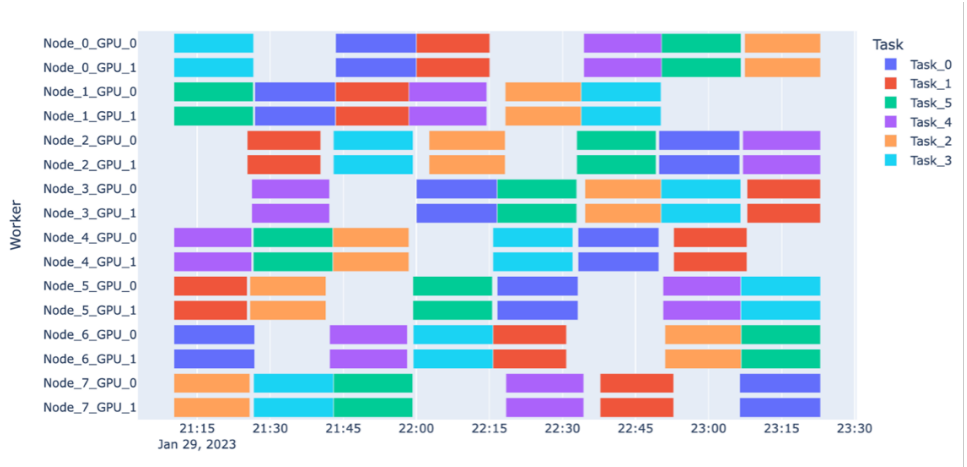
From these results, we can conclude that the randomized scheduler is a more practical option. We have used this scheduler as part of Cerebro's design. Section 4.2.2 suggests optimizations on these schedulers. We leave it to future works to compare these optimizations.



(a) Random Scheduler Makespan: 8603s

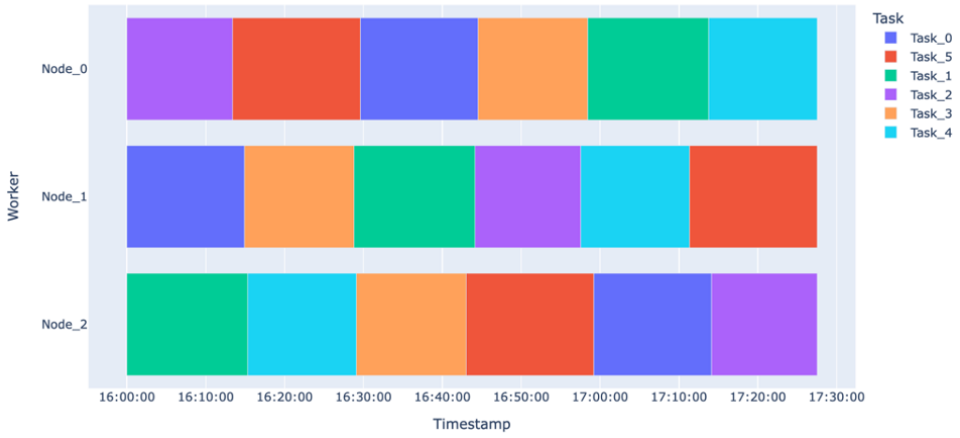


(b) Multi-GPU MOP Makespan: 15926s

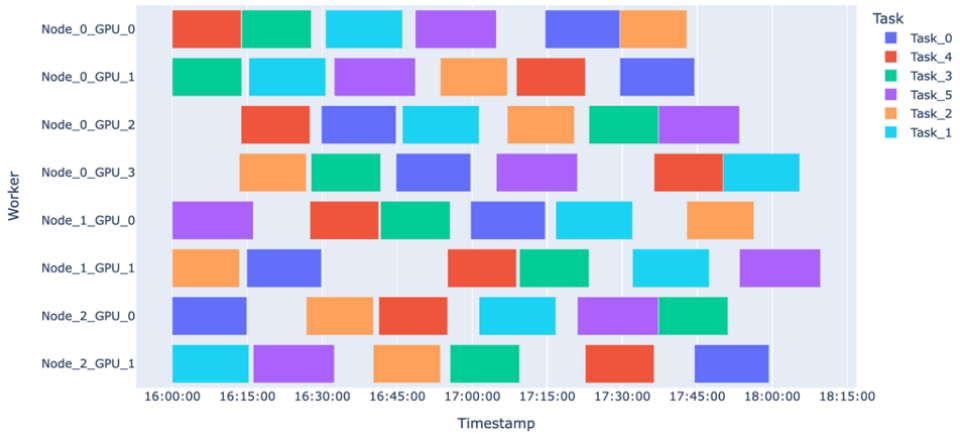


(c) Cerebro-MILP Makespan: 7967s

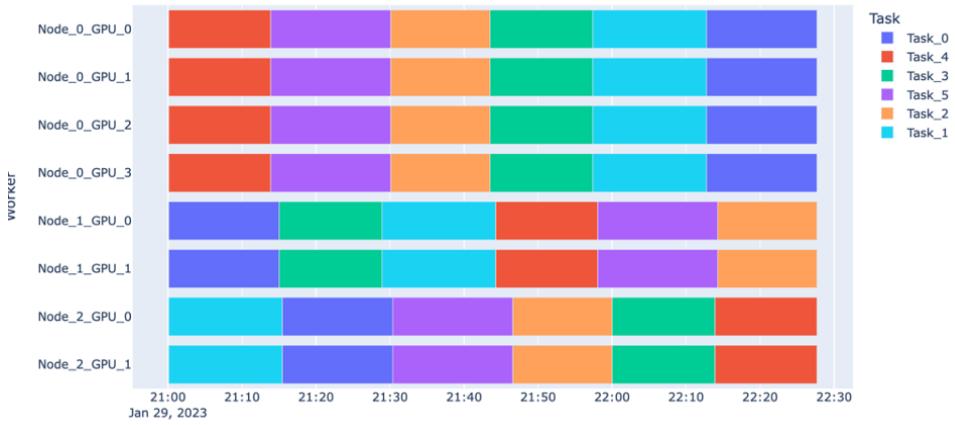
Figure 4.1. Gantt chart of simulation depicting number of models $<$ number of nodes. (Task Count: 6; Node Count: 8; GPU Counts: 2 each)



(a) Random Scheduler Makespan: 5252s

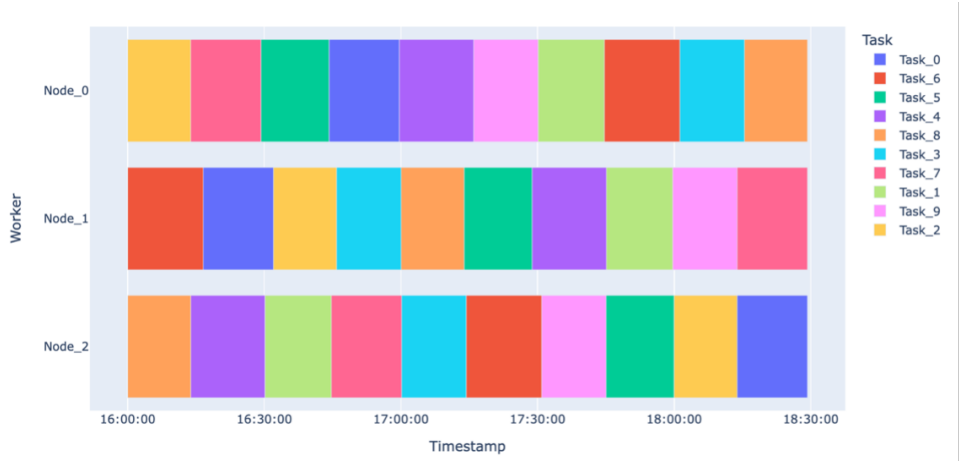


(b) Multi-GPU MOP Makespan: 7780s

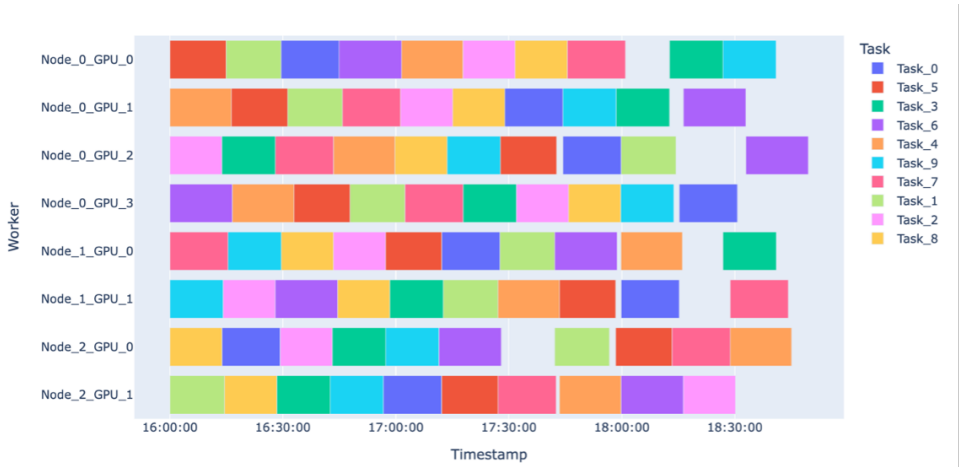


(c) Cerebro-MILP Makespan: 5255.36s

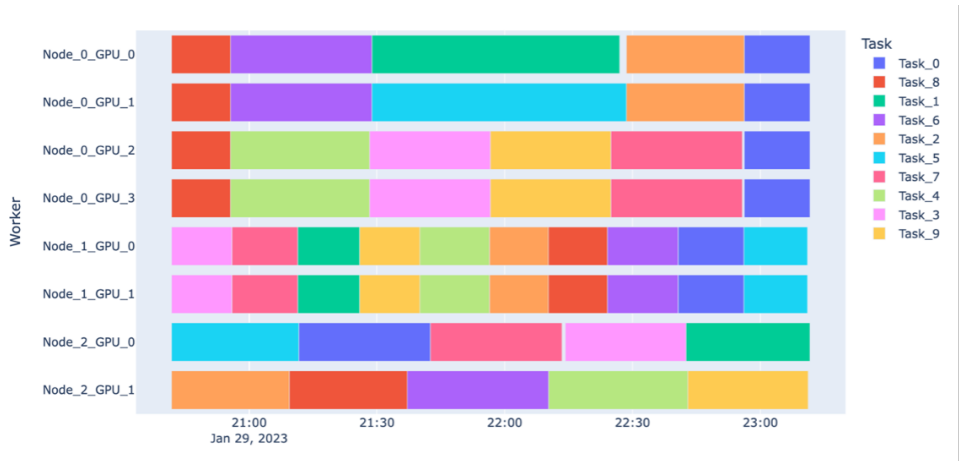
Figure 4.2. Gantt chart of simulation depicting number of nodes $<$ number of models $<$ number of GPUs. (Task Count: 6; Node Count: 3; GPU Counts: [4, 2, 2])



(a) Random Scheduler Makespan: 8956s



(b) Multi-GPU MOP Makespan: 10172s



(c) Cerebro-MILP Makespan: 8994.59s

Figure 4.3. Gantt chart of simulation depicting number of models $<$ number of nodes. (Task Count: 10; Node Count: 3; GPU Counts: [4, 2, 2])

Chapter 5

Implementation and Experiments

Here we describe our implementation of Cerebro and its evaluation on Image Captioning and Image Classification tasks.

5.1 Implementation Details

The current version of Cerebro implements the Multi-GPU MOP scheduler, described in 4.2.1. The Unified Scheduler version that supports model sharding capabilities, along with the trial runner, is a work in progress. Cerebro was developed on Amazon AWS, using EKS to provison a Kubernetes cluster. It supports S3 as the remote storage bucket and uses EFS for storage. Since it is fully cloud-native, Cerebro can be extended to other clouds as well. For the Key-Value Store, we use an on-prem Redis instance. The dispatcher hosts the backend web server using the micro web framework Flask and provides a web interface using AngularJS. The Cerebro application logic is implemented entirely in Python3, with current support for PyTorch as the choice for the Machine Learning framework. Future versions will extend this support to include Tensorflow and other frameworks. We have written scripts to automate the entire installation process. This includes the provisioning of a multi-node Kubernetes cluster, installation of Cerebro components, and the deprovisioning of the cluster when experiments are completed. All this can be executed via a single command. Cluster specifications are defined in a separate YAML file.

Table 5.1. Model Selection Hyperparameter search space for the Microsoft COCO experiment

Hyperparameters		
Learning rate	1e-2	1e-3
Embed size	256	512
Hidden size	256	512
Batch size	128	

5.2 Microsoft COCO

The Microsoft Common Objects in Context (COCO) dataset [45] is a widely used benchmark dataset in the field of computer vision. It consists of a large collection of images with diverse object categories and complex scenes. We chose this dataset since it provides a good validation of Cerebro's multi-modal capabilities. The COCO 2017 dataset, which was used here, contains over 330,000 images, each annotated with object labels, segmentations, and keypoint annotations. The dataset covers 80 different object categories, including common everyday objects such as people, animals, vehicles, and household items. For our experiments, we used the image captioning task. The model is comprised of a CNN encoder and an RNN decoder, borrowing from [46]. The implementation code is based on [47]. For validating the model, we referred to [48], which uses a bleu-4 score.

The experiment was conducted on an 8 node cluster, with 1 Nvidia T4 GPU (16 GB GPU memory) per node with a total cluster main memory of 1TB. The processed train data amounted to 332GB and the validation data was 14GB in size. The parallelized pre-processing via ETL took approximately 12m. We explored a model selection search space of 8 configurations, as shown in Table 5.1. Each configuration was run for 8 epochs. The experiment took 8h 50m to run. Figure 5.1 shows the Bleu-4 scores obtained on the validation data and Figure 5.2 shows the decreasing mini-batch loss on the train data.

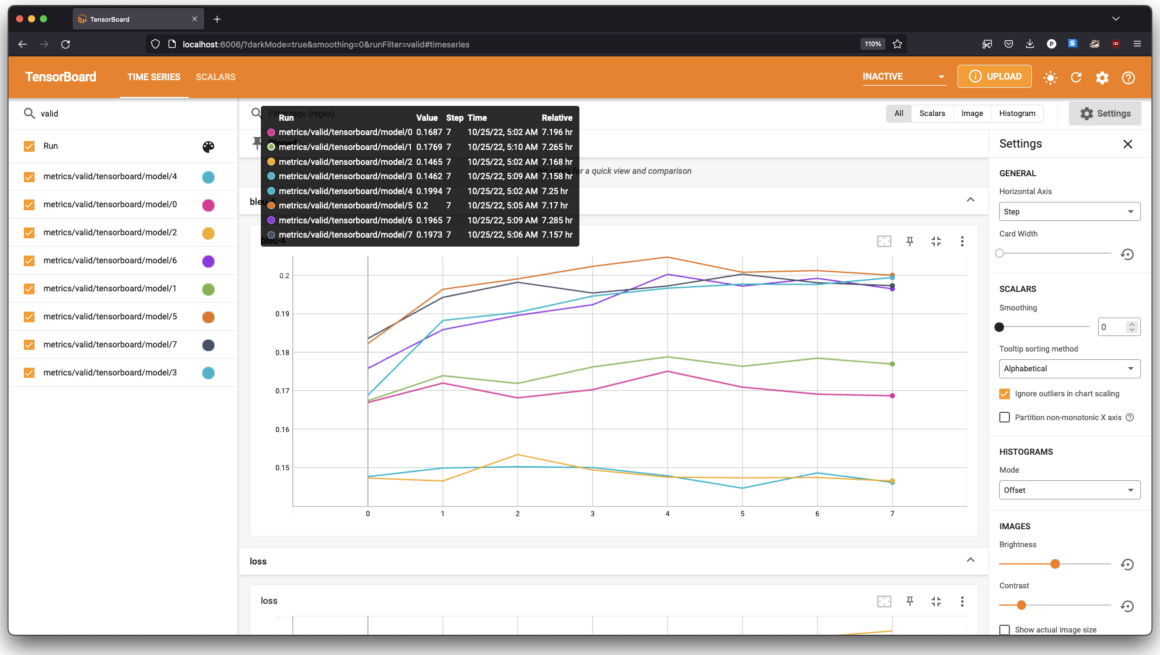


Figure 5.1. BLEU-4 score of models on the Microsoft COCO validation dataset (higher value is better), as seen on Cerebro's Model Visuzalization dashboard

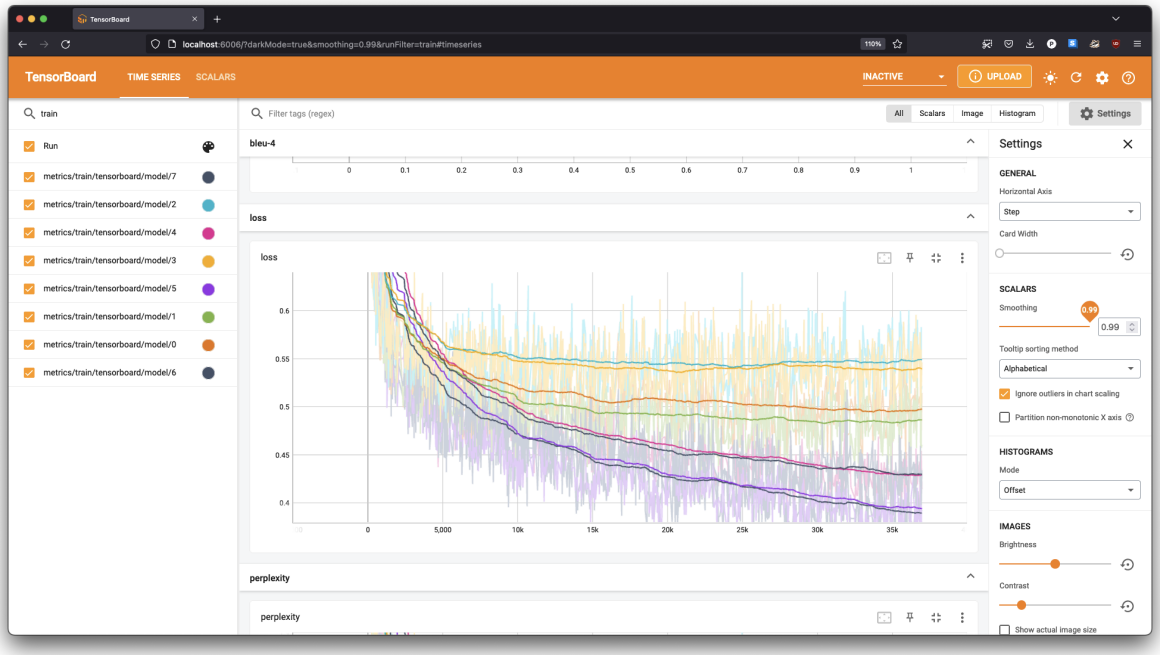


Figure 5.2. Sub-epoch loss of the models on the Microsoft COCO training dataset (lower value is better), as seen on Cerebro's Model Visuzalization dashboard

Table 5.2. Model Selection Hyperparameter search space for the Imagenet experiment

Hyperparameters		
Learning rate	1e-2	1e-3
Lambda value	1e-3	1e-4
Batch size	128	256
Model	VGG16	Resnet-50

5.3 Imagenet

The ImageNet dataset is a widely recognized and extensively used benchmark in the field of machine learning. The dataset was initially introduced as part of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2010. More details about the dataset can be found at [49]. It consists of over one million images categorized into 1,000 different classes and the task is object detection. To test Cerebro's capability to work with multiple Neural Network Architectures, we used two different models as part of our search space. One based on VGG16 [50] and the other based on ResNet50 [51]. Model definition code has been borrowed from [52].

The experiment was conducted on an 8 node cluster, with 4 Nvidia T4 GPUs on each node. The processed train data amounted to 180GB and the validation data was 28.5 GB in size. We explored a model selection space of 16 configurations over 8 epochs, as shown in Table 5.2. Each epoch took approximately 2.5h to complete. Figure 5.3 shows the Top-5 accuracy graph on the training data for various model configurations. Figure 5.4 shows the Top-5 accuracy (percentages shown as decimal values) on the validation data.

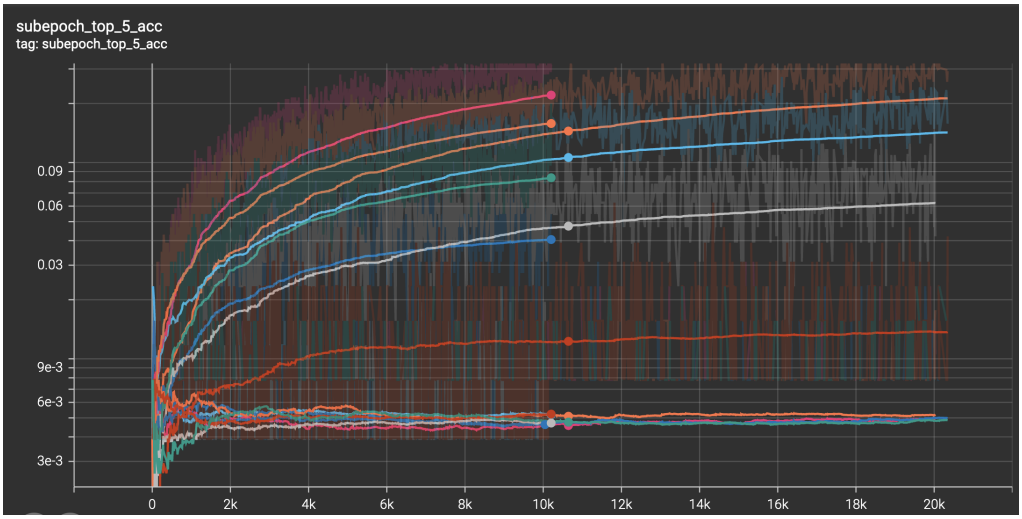


Figure 5.3. Imagenet models' sub-epoch Top-5 accuracy on training data (higher value is better)

Run	Value	Step	Time	Relative
metrics/valid/tensorboard/model/0	0.02226		3/24/23, 10:03 AM	2.414 hr
metrics/valid/tensorboard/model/1	0.04631		3/24/23, 9:15 AM	2.57 hr
metrics/valid/tensorboard/model/10	0.02053		3/24/23, 10:02 AM	2.498 hr
metrics/valid/tensorboard/model/11	0.2644		3/24/23, 9:13 AM	2.54 hr
metrics/valid/tensorboard/model/12	0.0205		3/24/23, 10:08 AM	2.749 hr
metrics/valid/tensorboard/model/13	0.5806		3/24/23, 9:12 AM	2.505 hr
metrics/valid/tensorboard/model/14	0.01781		3/24/23, 10:04 AM	2.572 hr
metrics/valid/tensorboard/model/15	0.7916		3/24/23, 9:16 AM	2.496 hr
metrics/valid/tensorboard/model/2	0.02137		3/24/23, 10:12 AM	2.65 hr
metrics/valid/tensorboard/model/3	0.2028		3/24/23, 9:18 AM	2.722 hr
metrics/valid/tensorboard/model/4	0.03705		3/24/23, 10:14 AM	2.695 hr
metrics/valid/tensorboard/model/5	0.4889		3/24/23, 9:11 AM	2.462 hr
metrics/valid/tensorboard/model/6	0.02242		3/24/23, 10:12 AM	2.673 hr
metrics/valid/tensorboard/model/7	0.9124		3/24/23, 9:13 AM	2.534 hr
metrics/valid/tensorboard/model/8	0.01701		3/24/23, 10:01 AM	2.725 hr
metrics/valid/tensorboard/model/9	0.09806		3/24/23, 9:11 AM	2.475 hr

Figure 5.4. Imagenet models' Top-5 accuracy on validation data (higher value is better)

Chapter 6

Future Work

In this thesis, we studied Cerebro and how it can efficiently scale deep learning. In the future, several avenues can be explored to further enhance the capabilities and scalability of the Cerebro platform. Expanding the platform's support to work with Tensorflow will provide users with a wider range of options and flexibility in choosing their preferred frameworks for model development. Incorporating more parallelism techniques such as GPipe will support more variety of models to be sharded efficiently. Different scheduler optimization suggested in 4.2.2 can be implemented and compared.

In the near future, we plan to collaborate with the San Diego Supercomputer Center (SDSC), where Cerebro will power their Voyager AI platform [53]. This platform is focused on domain science research that is increasingly dependent upon artificial intelligence at scale.

By pursuing these directions, the Cerebro platform can continue to evolve and serve as a robust and efficient solution for scaling DL on multi-node clusters, empowering researchers and users in their Deep Learning endeavors.

References

[1] M. Feurer and F. Hutter, "Hyperparameter Optimization," in *Automated Machine Learning: Methods, Systems, Challenges* (F. Hutter, L. Kotthoff, and J. Vanschoren, eds.), The Springer Series on Challenges in Machine Learning, pp. 3–33, Cham: Springer International Publishing, 2019.

[2] P. Villalobos, "Trends in Training Dataset Sizes," Sept. 2022.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Kaiser, and I. Polosukhin, "Attention is All you Need,"

[4] OpenAI, "GPT-4 Technical Report," Mar. 2023. arXiv:2303.08774 [cs].

[5] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," Feb. 2018. arXiv:1802.05799 [cs, stat].

[6] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling Distributed Machine Learning with the Parameter Server," pp. 583–598, 2014.

[7] 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," June 2020. arXiv:2006.15704 [cs].

[8] M. X. P. G. Q. D. V. M. O. Caggiano, Sam Shleifer, "Fully Sharded Data Parallel: faster AI training with fewer GPUs," July 2021.

[9] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, J. E. Gonzalez, and I. Stoica, "Alpa: Automating Inter- and {Intra-Operator} Parallelism for Distributed Deep Learning," pp. 559–578, 2022.

[10] K. Nagrecha and A. Kumar, "Hydra: A System for Large Multi-Model Deep Learning," Aug. 2022. arXiv:2110.08633 [cs].

[11] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A Distributed Framework for Emerging AI Applications,"

[12] S. Nakandala, Y. Zhang, and A. Kumar, "Cerebro: a data system for optimized deep learning model selection," *Proceedings of the VLDB Endowment*, vol. 13, pp. 2159–2173, Aug. 2020.

[13] P. Vassiliadis, "A Survey of Extract–Transform–Load Technology," *International Journal of Data Warehousing and Mining (IJDWM)*, vol. 5, pp. 1–27, July 2009. Publisher: IGI Global.

[14] K. Nagrecha Feb. 2023.

[15] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[16] H. Robbins and S. Monro, "A Stochastic Approximation Method," *The Annals of Mathematical Statistics*, vol. 22, pp. 400–407, Sept. 1951. Publisher: Institute of Mathematical Statistics.

[17] M. Zinkevich, M. Weimer, L. Li, and A. Smola, "Parallelized Stochastic Gradient Descent," in *Advances in Neural Information Processing Systems*, vol. 23, Curran Associates, Inc., 2010.

[18] X. Lian, C. Zhang, H. Zhang, C.-J. Hsieh, W. Zhang, and J. Liu, "Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent," in *Advances in Neural Information Processing Systems*, vol. 30, Curran Associates, Inc., 2017.

[19] L. Bottou, "Curiously Fast Convergence of some Stochastic Gradient Descent Algorithms," 2009.

[20] D. P. Bertsekas, "A New Class of Incremental Gradient Methods for Least Squares Problems," *SIAM Journal on Optimization*, vol. 7, pp. 913–926, Nov. 1997.

[21] B. A.-L. Probst Philipp, Bischl Bernd, "Tunability: Importance of Hyperparameters of Machine Learning Algorithms,"

[22] K. Nagrecha, "Systems for Parallel and Distributed Large-Model Deep Learning Training," Jan. 2023. arXiv:2301.02691 [cs].

[23] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A Survey on Distributed Machine Learning," *ACM Computing Surveys*, vol. 53, pp. 1–33, Mar. 2021.

[24] M. Rocklin, "Dask: Parallel Computation with Blocked algorithms and Task Scheduling," (Austin, Texas), pp. 126–132, 2015.

[25] H. B. McMahan, E. Moore, D. Ramage, and S. Hampson, "Communication-Efficient Learning of Deep Networks from Decentralized Data,"

[26] A. Castelló, M. F. Dolz, E. S. Quintana-Ortí, and J. Duato, "Analysis of model parallelism for distributed neural networks," in *Proceedings of the 26th European MPI Users' Group Meeting*, EuroMPI '19, (New York, NY, USA), pp. 1–10, Association for Computing Machinery, Sept. 2019.

[27] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models," May 2020.

[28] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "GPipe: efficient training of giant neural networks using pipeline parallelism," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, no. 10, pp. 103–112, Red Hook, NY, USA: Curran Associates Inc., Dec. 2019.

[29] S. Gehman, S. Gururangan, M. Sap, Y. Choi, and N. A. Smith, "RealToxicityPrompts: Evaluating Neural Toxic Degeneration in Language Models," Sept. 2020. arXiv:2009.11462 [cs].

[30] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden Technical Debt in Machine Learning Systems,"

[31] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of {Large-Scale} {Multi-Tenant} {GPU} Clusters for {DNN} Training Workloads," pp. 947–960, 2019.

[32] A. Tosatto, P. Ruiu, and A. Attanasio, "Container-Based Orchestration in Cloud: State of the Art and Challenges," in *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, pp. 70–75, July 2015.

[33] B. Burns, J. Beda, K. Hightower, and L. Evenson, *Kubernetes: Up and Running*. "O'Reilly Media, Inc.", Aug. 2022.

[34] "The State of Kubernetes 2022| VMware Tanzu." https://hello-tanzu.vmware.com/state-of-kubernetes-2022/.

[35] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The Journey So Far and Challenges Ahead," *IEEE Software*, vol. 35, pp. 24–35, May 2018. Conference Name: IEEE Software.

[36] J. Shah and D. Dubaria, "Building Modern Clouds: Using Docker, Kubernetes & Google Cloud Platform," in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 0184–0189, Jan. 2019.

[37] F. Cristian, "Understanding fault-tolerant distributed systems," *Communications of the ACM*, vol. 34, pp. 56–78, Feb. 1991.

[38] JetBrains, "Data science - the state of developer ecosystem in 2022 infographic." https://www.jetbrains.com/lp/devecosystem-2022/.

[39] T. Gonzalez and S. Sahni, "Open Shop Scheduling to Minimize Finish Time," *Journal of the ACM*, vol. 23, pp. 665–679, Oct. 1976.

[40] A. S. Manne, "On the Job-Shop Scheduling Problem," *Operations Research*, vol. 8, pp. 219–223, Apr. 1960. Publisher: INFORMS.

[41] T. K. Sellis, "Multiple-query optimization," *ACM Transactions on Database Systems*, vol. 13, pp. 23–52, Mar. 1988.

[42] "Gurobi optimization." https://www.gurobi.com/.

[43] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective Straggler Mitigation: Attack of the Clones,"

[44] C. Robert and G. Casella, *Monte Carlo Statistical Methods*. Springer Science & Business Media, Mar. 2013. Google-Books-ID: lrvfBwAAQBAJ.

[45] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, "Microsoft COCO: Common Objects in Context," Feb. 2015. arXiv:1405.0312 [cs].

[46] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and tell: A neural image caption generator," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (Boston, MA, USA), pp. 3156–3164, IEEE, June 2015.

[47] T. Nguyen, "Image captioning." https://github.com/ntrang086/image_captioning.

[48] T.-Y. Lin, "Microsoft COCO Caption Evaluation." https://github.com/tylin/coco-caption, June 2023. original-date: 2015-03-17T22:44:42Z.

[49] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, vol. 115, pp. 211–252, Dec. 2015.

[50] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," Apr. 2015. arXiv:1409.1556 [cs].

[51] 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)*, (Las Vegas, NV, USA), pp. 770–778, IEEE, June 2016.

[52] Y. Zhang, "Cerebro-DS: Cerebro on Data Systems." https://github.com/makemebitter/cerebro-ds, Aug. 2022. original-date: 2020-11-23T19:01:20Z.

[53] S. D. S. Center, "Voyager User Guide." https://sdsc.edu/support/user_guides/voyager.html.