UNIVERSITY OF CALIFORNIA SAN DIEGO

Orchestration Systems to Support Deep Learning at Scale

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science

by

Kabir Nagrecha

Committee in charge:

Professor Arun Kumar, Chair
Professor Hao Zhang, Co-Chair
Professor Zhiting Hu
Professor Julian McAuley
Professor Jingbo Shang

2024

The Dissertation of Kabir Nagrecha is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2024

DEDICATION

This dissertation is dedicated to my parents, Amit and Anjali.
For their love, guidance, and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

It's hard to overestimate my gratitude to my Ph.D. advisor, Professor Arun Kumar, for his dedicated support over the years. I first began working with him as an undergraduate, and now — at the culmination of my Ph.D. — I can only look back and appreciate how his guidance and mentorship have shaped my journey as a researcher. I have been incredibly fortunate to have him as an advisor.

I'm also deeply grateful to my recent co-advisor, Professor Hao Zhang. Across paper iterations and review cycles, his feedback and advice have been invaluable. Besides my advisors, my sincere thanks also go to the other members of my thesis committee: Professor Jingbo Shang, Professor Julian McAuley, and Professor Zhiting Hu. I also thank Professor Victor Vianu, Professor Alin Deutsch, and Professor Babak Salimi for their much-appreciated feedback in the Database Lab seminars I've done over the years.

Deep learning systems research tends to blur the boundaries between industry and academia, and my Ph.D. has straddled both sides of the fence. I thank my fellowship mentors from Meta, Rakesh Kumar and Will Feng, for their practical feedback and support in getting my work adopted in the industry. At Netflix, where I spent most of my non-academic working hours, I must thank the many mentors who helped me throughout my internships and industry collaborations. Dr. Lingyi Liu, Pablo Delgado, Prasanna Padmanabhan, Evan Cox, and Faisal Siddiqi have all championed my work and helped ensure its real-world impact at Netflix and beyond.

I am also thankful to all my co-authors: Arun Kumar, Hao Zhang, Lingyi Liu, Pablo Delgado, Prasanna Padmanabhan, Supun Nakandala, Yuhao Zhang, Side Li, Advitya Gemawat, Pei Wang, and Nuno Vasconcelos. I learned a great deal from my collaborations with them, and this dissertation would not have been possible without them. I'd also like to thank my colleagues in the ADALab — Kyle Luoma, Xiuwen Zheng, Pradyumna Sridhara, and Vignesh Nanda Kumar — for the many discussions and talks we've had over the years.

Finally, none of this would have been possible without my parents. I am forever grateful

for their encouragement and unwavering support.

The material in this dissertation is based on the following publications:

- Chapter 3 contains material from "Hydra: A System for Large Multi-Model Deep Learning" by Kabir Nagrecha and Arun Kumar, and "Model-Parallel Model Selection for Deep Learning Systems" by Kabir Nagrecha, which appears in Proceedings of the 2021 International Conference on Management of Data. The dissertation author was the primary investigator and author of this paper.

- Chapter 4 contains material from "Saturn: An Optimized Data System for Large Model Deep Learning Workloads" by Kabir Nagrecha and Arun Kumar, which appears in Proceedings of VLDB Endowment Volume 17, Issue 4, 2023. The dissertation author was the primary investigator and author of this paper.

- Chapter 5 contains material from "InTune: Reinforcement Learning-based Data Pipeline Optimization for Deep Recommendation Models" by Kabir Nagrecha, Lingyi Liu, Pablo Delgado, and Prasanna Padmanabhan, which appears in Proceedings of the 17th ACM Conference on Recommender Systems, September 2023. The dissertation author was the primary investigator and author of this paper.

- Chapter 6 contains material from "Reinforcement Learning for Intra- & Inter-Node Recommender Data Pipeline Optimization" by Kabir Nagrecha, Lingyi Liu, and Pablo Delgado, which appears in ACM Transactions on Recommender Systems of the 17th ACM Conference on Recommender Systems, Volume 5. The dissertation author was the primary investigator and author of this paper.

- Chapter 7 contains material from "Routing Over LLMs Using Proxy Metrics for Relative Quality Estimation" by Kabir Nagrecha, Arun Kumar, and Hao Zhang, which is currently under submission. The dissertation author was the primary investigator and author of this paper.

My co-authors have kindly approved the inclusion of the aforementioned publications in my dissertation.

VITA

2021 B.S., University of California San Diego

2023 M.S., University of California San Diego

2024 Ph.D., University of California San Diego

PUBLICATIONS

Kabir Nagrecha, Arun Kumar, and Hao Zhang, "Routing Over LLMs Using Proxy Metrics for Relative Quality Estimation". *(Under Submission, 2024)*

Kabir Nagrecha, Lingyi Liu, and Pablo Delgado, "Reinforcement Learning for Intra- & Inter-Node Recommender Data Pipeline Optimization". *ACM Transactions on Recommender Systems (TORS) 5.*

Kabir Nagrecha and Arun Kumar, "Saturn: An Optimized Data System for Large Model Deep Learning Workloads". *Proceedings of the 2024 Very Large Data Bases Conference (VLDB 2024).*

Kabir Nagrecha, Lingyi Liu, Pablo Delgado, and Prasanna Padmanabhan, "InTune: Reinforcement Learning-based Data Pipeline Optimization for Deep Recommendation Models". *Proceedings of the 17th ACM Conference on Recommender Systems (RecSys 2023).*

Kabir Nagrecha and Arun Kumar, "Hydra: A System for Large Multi-Model Deep Learning".

Kabir Nagrecha, "Model-Parallel Model Selection for Deep Learning Systems". *Proceedings of the 2021 International Conference on Management of Data (SIGMOD 2021).*

ABSTRACT OF THE DISSERTATION

Orchestration Systems to Support Deep Learning at Scale

by

Kabir Nagrecha

Doctor of Philosophy in Computer Science

University of California San Diego, 2024

Professor Arun Kumar, Chair
Professor Hao Zhang, Co-Chair

Deep learning (DL)'s dramatic rise in popularity across the domain sciences and industry has been accompanied by a correspondingly aggressive increase in the scale and computational complexity of DL workloads. In order to adopt state-of-the-art techniques, practitioners must wrestle with systems challenges of performance, cost, and scalability. In this dissertation, we identify the need for *orchestration systems*, which ease scaling burdens across the DL lifecycle through holistic, workload-aware optimizations. Drawing on both established techniques from data management research and new bespoke algorithms, we build practical orchestration engines to optimize three common DL workloads in the large-scale setting: model selection, data

processing, and high-throughput serving. Our systems — which exploit workload- and context-specific opportunities — address a new layer of the large-scale DL optimization stack, more granular than current cluster managers and data systems, but still abstracted away low-level kernel & compiler optimizations. Empirical evaluations show that our orchestration techniques and systems can accelerate large-scale DL workloads by a large margin, even in complex, real-world settings. Our approach introduces a new technical lens, unifying systems, databases, and DL research, ultimately focused on democratizing and amplifying state-of-the-art DL innovations. Some of the systems proposed in this dissertation have already been adopted in production-scale industry pipelines, demonstrating the value of such orchestration optimizers for real-world DL.

Chapter 1

Introduction

1.1 Motivation

Over the past several years, deep learning (DL) has rapidly evolved into a critical component of the data analytics landscape. In industry, DL models now underpin key applications such as web search, recommendations, robotics, and business intelligence. In the domain sciences, DL has enabled substantial advances in fields ranging from drug discovery to climate modeling.

Yet this rapid evolution of state-of-the-art DL practice has come at a cost. DL workloads now routinely demand vast computational resources, at scales which are out of reach for most practitioners. While large technology companies can afford the cluster capacities necessary to develop and deploy such large-scale DL solutions, domain scientists and small-to-medium enterprises (SMEs) now run the risk of being cut off from state-of-the-art innovations in DL.

Recent advances in DL practice, e.g., the use of large language models (LLMs), have only exacerbated this need. State-of-the-art models in both natural language processing (NLP) and computer vision (CV) now routinely demand dozens or even hundreds of GPUs, and are trained on multi-petabyte datasets. Figure 1.1 depicts how the search for higher accuracy has motivated this upscaling. We observe a new and emerging need in DL practice: the need for cost- & performance- optimizations that can alleviate the computational burdens presented by large-scale DL workloads.

This nascent space of large-scale DL systems has, thus far, been dominated by *low-level*

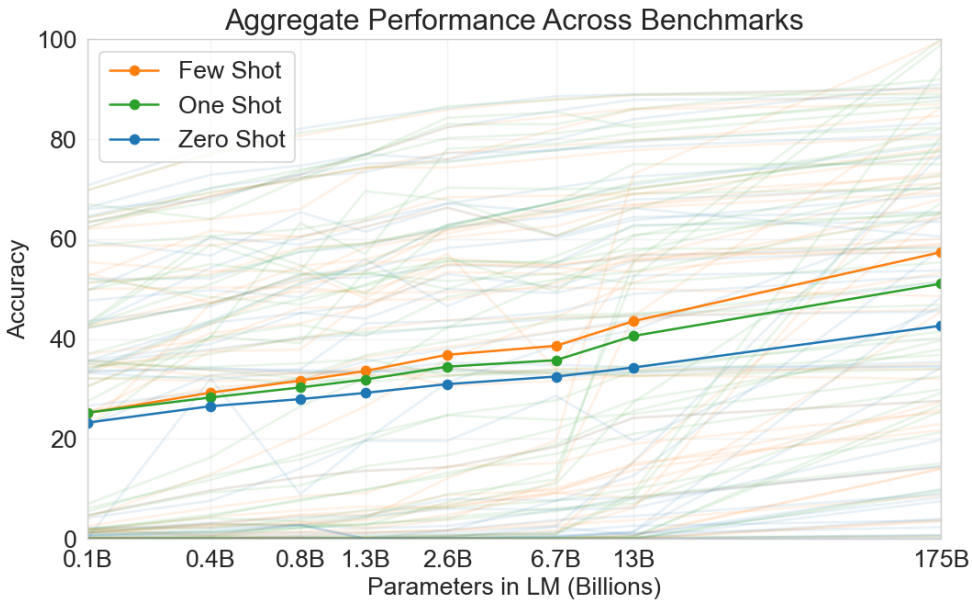

Figure 1.1. Illustration of the relationship between model quality and model size in the context of NLP. Similar trends are now emerging in CV applications as well. Figure taken from a 2020 technical report on GPT-3 [29].

optimization efforts, e.g., new kernels [43] and operations [55], or DL-specific compilation techniques [241, 35]. These approaches — operating at the lowest levels of abstraction — have created effective building blocks for large-scale DL. But we observe that there remains significant scope for large-scale DL optimization at a *higher level of abstraction*, i.e., the orchestration layer. Recent works [232, 99, 152] have observed that large-scale DL workloads typically exist in the context of *broader systems*, *operational lifecycles*, and *infrastructure environments*. For instance, model selection workloads — where the user explores multiple training jobs & models simultaneously to identify the best configuration for their use case — introduce new challenges with batched multi-model training, hyperparameter optimization, and efficient resource allocation, none of which can be resolved by granular per-model optimizations alone.

This orchestration layer is uniquely situated to exploit opportunities for *workload-aware* optimization. Lower-level techniques lack context on the broader system and user needs, while higher-level optimizations (e.g., cluster schedulers [226]) operate at a level of abstraction where the distinct characteristics of large-scale DL are lost. *Orchestration systems for large-scale*

DL can account for specific development patterns (e.g., model selection workloads), exploit application-specific quirks (e.g., quality degradation tolerances), and leverage meta-learning (i.e., deriving new optimization strategies from workload observation). Furthermore, such optimizations are largely orthogonal to lower-level innovations and advances. A new system to accelerate model selection might easily benefit from new kernels as well — thus combining the impact of the two.

By optimizing in this way, not only do we amplify the reach of today's state-of-the-art architectures and techniques, by making them more accessible and cost-effective, but we also open the door to *further* upscaling and innovations. Rather than relying on additional hardware or innovations at the device level, we enable researchers and practitioners to do *more with less*, such that the systems serve as a catalyst for new DL innovations by sidestepping the limitations of current hardware platforms. Just as techniques like *spilling* [174] enable large-scale databases to operate beyond the constraints of memory, so too can these novel orchestration systems enable DL models to operate beyond the constraints of their existing hardware accelerators.

In this dissertation, we present new *workload-aware orchestration systems* to optimize and accelerate large-scale DL. We decompose DL workloads into a three-stage pipeline — data processing, training/fine-tuning, and inference — to understand how scaling challenges have uniquely impacted each phase of the process. In each case, we identify opportunities for orchestration optimizations that exploit the unique workload characteristics presented by large-scale DL, ultimately enabling significant cost & performance improvements. Thus, the thesis of this dissertation is as follows:

> *Workload-aware orchestration systems can enable significant optimizations in the lifecycle of large-scale DL workloads, accelerating performance and reducing the costs of state-of-the-art DL practice, democratizing and amplifying the impact of DL research.*

Our proposed solutions draw on existing techniques from data management research, adapted for large-scale DL, as well as a combination of new algorithmic and architectural innovations. We find that our techniques — each targeting a different scaling burden in a different stage of the DL pipeline — can accelerate performance and reduce costs considerably, by as much as 6-7X in some cases. These works have already seen strong impact and adoption in practice, and have been used to enable new products and research at both large technology companies and academic labs.

1.2 Technical Contributions

In this dissertation, we select three critical phases of the DL pipeline: data pre-processing, model training/fine-tuning, and inference. We chose these phases for their unique challenges, importance, and exposure to different scaling dimensions of DL. Data processing highlights the growing size of training datasets and the complexity of data transformation operations. Training and fine-tuning require consideration of model sizes and resourcing constraints. Inference combines the challenges of model scale and complexity with the demands of serving a high-throughput, latency-constrained application. Figure 1.2 summarizes and positions our contributions.

HYDRA: Model-Parallel Model Selection for Memory Intensive Architectures. The first of our contributions tackles the challenges of *model selection*, i.e., the empirical process of evaluating multiple training configurations and hyperparameters, for large-model architectures. While efficient model selection typically involves running multiple models in parallel, to better optimize end-to-end makespan, large-model architectures often require multiple GPUs to amortize their high memory demands over the combined capacities of several devices. These two objectives run counter to one another — the more GPUs needed per-model, the lower the degree of task-parallelism that can be achieved on a limited-resource compute cluster. HYDRA introduces a spilled execution pattern, distributing memory demands across a memory hierar-

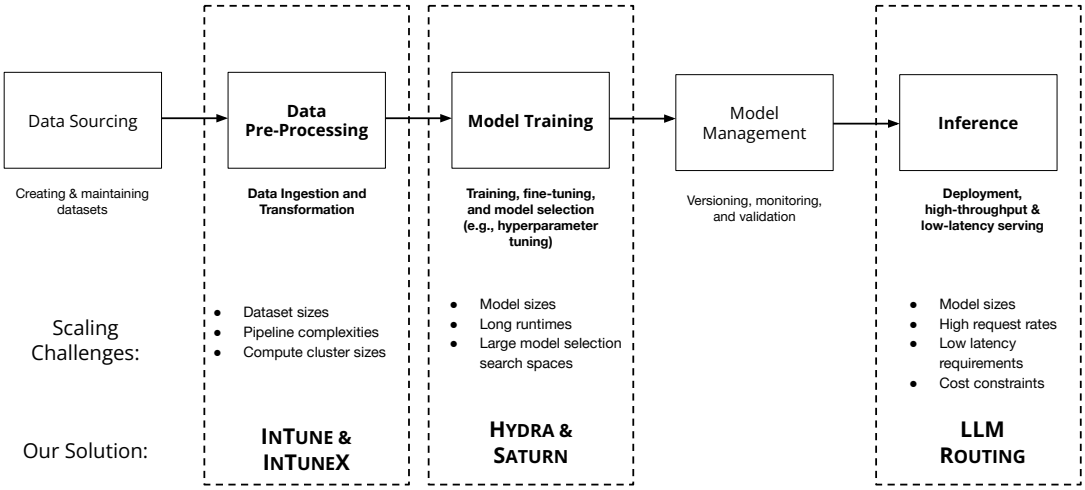

Figure 1.2. The DL lifecycle can broadly be divided into several stages. We target three stages — data pre-processing, training, and inference. The systems we discuss in this dissertation tackle the scaling challenges seen in each of these three stages.

chy (i.e., system DRAM and GPU memory) rather than across multiple GPUs, and exploits the reduced memory demands to introduce higher degrees of task parallelism in multi-model workloads. Combined with a new and efficient scheduling algorithm, Sharded-LRTF, that minimizes makespan across multiple models, HYDRA achieves up to 1.5X higher training throughput than even state-of-the-art model parallel techniques that do not exploit multi-model task parallelization.

This work is the subject of Chapter 3, and is a joint work with Arun Kumar. A short paper on this work appeared at the SIGMOD conference in 2021 [140]. The techniques and system architecture have since been adopted in the industry for Transformer model selection workloads. A longer-form technical report was later posted to the online arXiv repository. The code for HYDRA is open-source and available on GitHub: https://github.com/knagrecha/hydra.

SATURN: Joint Optimization for Multi-Large-Model Deep Learning Workloads. SATURN addresses three interconnected challenges faced in multi-model workloads with large-model architectures: parallelization, resource apportionment, and scheduling. We observe that such workloads, whether driven by model selection or simply by multi-user cluster configurations,

introduce several optimization challenges that are typically addressed in isolation (or else not addressed at all). Parallelization, for instance, determines how an individual model should be partitioned over its available GPUs. Many strategies exist, e.g., pipelining, data parallelism, hybrids. We discuss these strategies in depth in a separate technical report [141]. Each technique presents different advantages in terms of its memory usage and performance, and the best technique to choose in a given situation depends on the model architecture at hand, the number of GPUs available, and even the specifications of the given hardware (e.g., communication bandwidths, accelerator memory capacities). While bespoke strategies may suffice in the single-model context where resources are provisioned up front, in the multi-model context, even the number of GPUs assigned to each job (i.e., the resource apportionment) may be optimized for end-to-end efficiency, complicating the parallelization decisions. In addition, *runtime-aware scheduling* decisions can improve makespan considerably, but are subject to constraints based on the resource demands and runtimes of each individual job. Thus, we formulated a joint optimization problem — SPASE — combining the challenges of **S**electing a **P**arallelism, **A**llocating resources, and constructing a runtime **S**chedul**e**. SATURN defines this problem as a mixed integer linear programming (MILP) problem, and solves it using a combination of an out-of-the-box commercial solver (Gurobi [65]) and scheduling mechanisms designed to enable introspective solution adjustments over time.

We find that SATURN achieves up to 1.96X higher training throughput than existing training techniques, even on multiple nodes and heterogeneously-sized machine instances. This work is the subject of Chapter 4, and is a joint work with Arun Kumar. A paper on this work will appear at the VLDB conference in 2024 [143]. It has been adopted at several companies, including Netflix, where it is used for LLM fine-tuning. The code for SATURN is open-source and available on GitHub: https://github.com/knagrecha/saturn.

INTUNE: Reinforcement Learning for CPU Resource Optimization in Data Ingestion Pipelines. INTUNE addresses a key — but thus far, largely underserved — bottleneck

in the DL lifecycle: data ingestion. We primarily focus on the recommendation setting, since it provides as an effective demonstration the importance of this DL optimization stage. Recommender systems account for an outsized proportion of industry DL workloads, and account for as much as *50%* of compute usage at large technology companies [224]. Interestingly, though, our empirical studies of a real-world industry recommender cluster demonstrate that the primary bottleneck in recommender model training is often *CPU-driven* data processing, not GPU-driven model execution. Data ingestion pipelines, where data is read from some disk or online data store, then processed through a series of transformation operations (e.g., user-defined functions/UDFs, batching stages, shuffles), are often highly complex and compute-intensive — and in the recommender system case, often more compute-intensive than the model itself! To address this challenge, INTUNE introduces a *reinforcement learning system* for data pipeline optimization; a system that can automatically design and parallelize individual stages of the data pipeline to maximize end-to-end efficiency.

Our evaluations of INTUNE on both academic and industry workloads show that this RL-based approach can offer as much as 2.29X higher ingestion throughput than existing state-of-the-art data pipeline optimizers. We also find that INTUNE is more robust against resource over-subscription and unexpected cluster changes than existing tooling. This work is the subject of Chapter 5, and is a joint work with collaborators from Netflix: Lingyi Liu, Pablo Delgado, and Prasanna Padmanabhan. Netflix provided access to their compute cluster and execution traces for analyses and experimentation. A paper on this work appeared at the ACM Conference on Recommender Systems in 2023 [146]. The system has been adopted at Netflix, and is currently being tested at Meta, Snap, Google, and Uber.

INTUNEX: Reinforcement Learning for Cluster Management & Node Optimization for Large-Scale Data Ingestion Pipelines. While INTUNE tackles the problem of maximizing data pipeline throughput for a training job isolated to a single node, larger-scale industry users frequently employ a *disaggregation* strategy, where data processing stages are shifted from the

trainer node to secondary machines. The advantage of this approach is simple: by moving beyond the resource limits of a single machine, the pipeline can now be replicated across multiple machines for higher processing throughput. However, *resource-efficient* replication is not straightforward. A typical compute cluster might contain several different node types, e.g., one with 128 CPUs and 500GB of RAM, and another with 64 CPUs and 2TB of RAM. Each might have different associated costs-of-usage, based on internal demand or external cloud provider pricing. Determining which nodes to use (and how many of each are needed) is a complex orchestration problem in and of itself, complicated further by the individual optimization problems that must be solved on each active node. With INTUNEX, we build on the original RL optimization problem presented in INTUNE to create a *multi-agent* RL solver that simultaneously optimizes individual nodes while also finding the most cost-effective multi-node parallelization scheme for a given job.

Our evaluations show that INTUNEX achieves comparable single-node performance to the original INTUNE system while also reducing overall cluster costs by as much as 25% — a considerable saving, when many data-centers cost many tens of millions of dollars to maintain. This work is the subject of Chapter 6, and is a joint work with collaborators from Netflix: Lingyi Liu, and Pablo Delgado. Netflix provided access to their compute cluster and execution traces for analyses and experimentation. A paper on this work will appear in the ACM Transactions on Recommender Systems Journal. The system has been adopted at Netflix.

Routing Over LLMs Using Proxy Metrics for Relative Quality Estimation. Over the last year, LLM inference has quickly become a critical area of DL systems optimization. While LLMs, and more broadly, Generative AI technologies, have shown impressive capabilities on complex, real-world tasks that require reasoning & human-like interactions, practical adoption is often challenging due to the considerable performance overheads of serving LLMs at scale. While using smaller LLMs may alleviate the pain of serving a large-scale model (e.g., serving LLaMA-7B [204] in-place of LLaMA-70B), smaller models typically produce lower-quality outputs.

In this paper, we look to understand this degradation behavior, and identify a straightforward method to estimate the degree of degradation based only on the input query. Unlike prior works — which try and predict LLM output quality directly — we re-imagine the problem as one of *relative* quality estimation, i.e., how will Model A perform relative to Model B, under the condition of Query X? We use the resultant prediction to select a candidate LLM that will provide the best inference performance while respecting a user-configured quality degradation threshold.

We evaluate our procedure on three diverse datasets representing challenging LLM applications — MT-Bench for chatbots, HumanEval for code generators, and a conversational recommendation dataset from a streaming service. Our experiments show up to 6.92X improvements in inference performance while maintaining output quality close to the largest candidate LLM. This work is the subject of Chapter 7, and is a joint work with Arun Kumar and Hao Zhang.

1.3 Summary and Impact

DL practice has quickly grown both in scale and penetration [7]. Many companies — both small and large — now employ MLOps practitioners to help with DL training, serving, and optimization. As a result, there has been a broad upswell of interest in *systems for DL* research. While most systems for DL works have focused on low-level kernel optimizations or novel compilation techniques, relatively little attention has been paid to the *orchestration* layer. In this dissertation, we propose various orchestration systems for three different stages of the DL lifecycle: (1) data ingestion, (2) training, and (3) serving. We show that this orchestration layer can unlock significant performance benefits and practical efficiency improvements for DL engineers and researchers, both in the domain sciences and industry.

Our work has been broadly adopted in both industry and academia, demonstrating the real-world potential of such orchestration systems. At the time of writing this document, we are aware of the following adoption cases:

- HYDRA has been used at Meta for various model selection workloads, for both deep recommendation models and Transformers. Meta engineers are now exploring how to integrate HYDRA's techniques with custom hardware platforms that support higher bandwidth interconnects.

- HYDRA was also used by researchers at UC San Diego and the Lawrence Berkeley National Laboratory to train 3D-CNN models for DL-driven physical simulations.

- SATURN has been used by Netflix for LLM fine-tuning jobs — some of which are used as part of production, customer-facing applications.

- INTUNE and INTUNEX have been used by Netflix to optimize data ingestion on recommender model training clusters, providing considerable cost-efficiency wins along with improved experimentation velocity. Several other companies are currently evaluating the techniques for their own recommendation clusters as well.

- Our work on routing over LLMs has been tested and adopted as part of some upcoming conversational chatbot products. Preliminary estimates show that the router will manage more than 100 queries per second.

Chapter 2

Background

In this chapter, we cover preliminary concepts necessary for the remainder of this dissertation. This includes standard machine learning (ML) and DL terminology, systems concepts and theory, and relevant data management techniques. We refer interested readers to more in-depth surveys and literature [141, 57, 191, 174, 199] for further background.

2.1 Deep Learning Terminology

Basics. Deep learning, a subset of machine learning, utilizes a class of algorithms known as *artificial neural networks* (ANNs). ANNs consist of a *computational graph* of nodes (individually referred to as "neurons"). Each of these neurons performs a simple transformation operation on the data it receives, passing the data forward through the graph. These transformation operations commonly involve some non-linear function applied to the sum of the neuron's inputs. Typically, groups of neurons are organized into standard building-blocks, known as *layers*, providing an intermediate layer of abstraction for graph definitions. Popular DL tools such as PyTorch and TensorFlow offer methods to specify and compose multiple layers together to construct a *model*, i.e., a complete computational graph.

The hierarchical, multi-level nature of such DL networks allows them to express complex relationships in data, even with raw feature inputs. Intermediate layers produce *latent* representations of the original inputs within the model, bypassing the need for manual feature engineering

up-front. As a result, DL models have become broadly popular for analytics on a variety of modalities, including text, image, audio, and tabular data.

Training. Individual neurons within a model typically record some sort of *state*, also known as a parameter. The process of *training* involves updating a model's parameters to improve its output quality for some given objective. The objective is usually defined through some *loss function*, which quantifies the discrepancies between a model's output and the "ideal" result. The ultimate goal of training is to configure the model parameters in such a way that the loss function is minimized. This process is typically performed using an optimization algorithm, with gradient descent being the most common. In gradient descent, the parameters are updated in the direction that most steeply decreases the loss function. This direction is determined by the gradient of the loss function with respect to the parameters.

Stochastic Gradient Descent (SGD) and its variants are often used to efficiently optimize deep learning models. In contrast to standard (or "batch") gradient descent, which computes the gradient using the entire training dataset, SGD computes the gradient using a single example at a time. Mini-batch SGD strikes a balance between the two: it computes the gradient using a small subset of the data, or "mini-batch". This approach benefits from computational efficiencies of parallel processing while still maintaining a level of noise in the gradient estimates that can help escape local minima. The size of the mini-batch, a key hyperparameter, impacts the model's learning speed and final performance.

Model Selection. Model selection in deep learning is the process of identifying the most suitable model architecture and hyperparameters for a given task. The architecture refers to the design of the neural network, such as the number of layers, the number of neurons per layer, and the type of layers (dense, convolutional, recurrent, etc.). Hyperparameters include the learning rate, batch size, number of training epochs, and regularization parameters, among others. Selecting the optimal architecture and hyperparameters often involves a combination of empirical testing, domain knowledge, and theoretical considerations. Due to model selection needs,

building a single model often requires training many dozens (or hundreds!) of configurations in advance. Model selection is often run as part of an automated pipeline [140, 100, 116] or else as part of a human-in-the-loop exploration procedure [110].

Inference. Inference in the context of deep learning refers to the process of using a trained model to make predictions on new, unseen data. After a model has been trained, its learned parameters are used to process input data, propagate it through the network's layers, and output a prediction. Inference can be performed on a single example or on a batch of examples, much like during training. However, unlike training, inference does not involve updating the model's parameters; the model is simply applied to the new data as it is. The efficiency of this process is critical in many applications, especially those requiring real-time predictions. As such, various techniques are used to speed up inference, such as model pruning, quantization, and the use of dedicated inference hardware.

LLMs. Large language models, or LLMs, refer to a new class of Transformer-based [211] model architectures [30] known for their zero-shot capabilities on a wide variety of tasks, including question-answering [195], problem solving [203], and content creation [53]. LLM inputs, or prompts, and their outputs, are represented as sequences of "tokens", each of which could be translated to individual words, letters, or common phrases.

2.2 Parallelization

A variety of techniques exist for parallelizing DL training. Each offers different advantages or drawbacks, which tend to vary depending on the workload. Understanding the data access and communication patterns of each strategy is critical to evaluating them in the context of large-model training.

Model parallelism. A common approach to parallelizing a model's execution is to partition, or shard, a neural architecture graph into subgraphs, and assign each subgraph, or model shard, to a different device. In a feedforward network, these shards might refer to groups

of stacked layers. The speedup potential of model parallelism depends largely on the architecture and the sharding strategy. Sequential model sharding on a feedforward network, for example, will offer no scope for parallel execution, since different accelerators must execute in sequence. In other cases, the neural computational graph offers natural opportunities for inter-operator parallelism.

Pipeline parallelism. Pipelining is a more optimized modification of the sequentially-sharded model parallel setting. It partitions an SGD minibatch into smaller "microbatches," then shuttles the microbatches through the model partitions [229, 82, 94, 118]. This enables different model shards to concurrently run different microbatches. The speedup of pipelining is heavily tied to the partitioning scheme and the number of microbatches. Prior work has underscored the importance of tuning these knobs via either expert knowledge or automated heuristics [118].

Tensor parallelism. Another approach is to actually partition the individual operators in the network. Some operators, such as embedding tables (which map categorical inputs to continuous vectors through a key-value lookup), can be sharded width-wise with minimal overheads. Others, such as matrix multiplies, can still be partitioned (e.g. using parallel GEMM strategies [80]) but involve more communication steps. These width-wise sharding strategies, more generally known as tensor parallelism as they require input tensor partitioning, can enable more performant intra-operator parallelism versus inter-operator model parallelism, but require more effort and thought to implement. In addition, the performance benefits are tempered by the fact that most tensor parallel operators require at least one (potentially slow) all-gather communication step to re-aggregate partitioned outputs.

Data parallelism. Historically, the most common parallel deep learning execution strategy has been to simply replicate a model across multiple accelerators and send different mini-batches to each copy. Such techniques, broadly classified as *data parallelism*, can be classified into two subcategories — asynchronous data parallelism and synchronous data parallelism. Asynchronous data parallelism techniques, like Parameter Server [113], let replicas run out-of-sync, with occasional update steps bringing them back in line. Synchronous data parallelism

techniques, like those provided under the PyTorch DDP [114] package, enforce strict per-mini-batch synchronization requirements. This ensures that the training updates are mathematically equivalent to single-processor training (so long as the effective batch size remains consistent).

Hybrids. Recently, some hybridizations of the above techniques have been proposed. Fully-sharded data parallelism [173, 4] combines data parallel replication with tensor-parallel partitioning, by sharding the overall model, but using all-to-all communication steps to fully replicate on individual operators during execution. Other techniques, like *3D parallelism*, have combined data parallelism with both tensor parallelism and pipelining, for higher end-to-end efficiency. Typically, such complex schemes are implemented to maximize communication efficiency by accounting for specific hardware interconnect rates and cluster configurations [158]. Some other works [241, 89, 207] implement custom hybrid parallelizations through a compilation process, solving for an optimal configuration given a specific model and cluster architecture.

2.3 Data Processing

A typical DL dataset is composed of historical user interactions with the target application. A streaming service, for example, might record user interactions (i.e. plays, ratings) with movies and shows. *Millions* of such interactions could be recorded every day. DL models deployed in production environments must be retrained regularly to account for the dataset updates.

While one dataset might be shared over many models, each model might target a different aspect of the application — for example, in a streaming service, one model might be used for ordering rows in the UI, while another might be used for video ordering. Individual models need different features (i.e. columns) of the base dataset and might require different preprocessing pipelines. As a result, many processing pipelines are run *online*, during the model training procedure, rather than being done offline — which would bloat storage costs by creating multiple variations of the same dataset. We will now describe a typical online transformation pipeline — using the recommendation setting as an example.

Samples are loaded from the base dataset in a disk read operation. Each sample is represented as a dictionary of key-value pairs, mapping feature names to values. These samples are used to fill up a batch for SGD training. This is repeated until some significant number of batches are in memory. They are then shuffled to encourage some randomness in the SGD procedure to improve model robustness [57].

For each batch, a custom dictionary lookup operation is used to extract relevant feature columns. In this case, we will say that product ID, user ID, user country, and total product view time are the relevant columns. Note that this dictionary lookup could be fairly expensive on a feature-rich dataset.

The first three columns are categorical, while the fourth is continuous. Some random noise is applied to the continuous variable to augment the data and improve model robustness. To improve training times, several batches will be "prefetched" at once into GPU memory to overlap the next pipeline loading phase with model execution, trading memory for performance. At this point, the pipeline has finished producing a training batch for model consumption.

Millions (or even billions) of recorded interactions will have to run through these stages to feed and train the model. The throughput rate needed from this pipeline is dependent on the GPU-driven model execution speed. To improve pipeline performance, two levels of parallelism are possible.

First, pipelining. This simply exploits the stage-by-stage processing structure. Stages can be overlapped in a similar way to CPU instruction pipelining [40] to improve throughput. Maximizing pipeline performance requires a delicate balancing act. Each transformation stage within the pipeline must take the same amount of time to avoid idling [118] .

Second, per-stage replication. Replicating pipeline stages across multiple processors can improve per-stage throughput significantly. The effect of this replication interplays with the balance of stage performances, thus impacting pipeline-parallel throughput. Solving this complex, joint optimization problem effectively can yield significant performance benefits. At a coarse-grained level, the entire pipeline itself could be copied across multiple machines [9], but

this would discard the opportunity presented by the joint optimization problem.

These challenges are complicated further by the possibility of *machine resizing*. Many clusters now use techniques such as auto-scaling, interruption & reassignment, or even machine multi-tenancy. In such cases, external decision-making may cause a job to actually receive *new* or different resources across the course of its lifecycle. This setting has become increasingly popular in recent years as new multi-model training tools [142, 136] have emerged. Effectively parallelizing such jobs even as the underlying resource pool is actively shifting requires a level of adaptability and flexibility not present in existing tooling.

2.4 Deep Reinforcement Learning

The general aim of reinforcement learning (RL) is to train an "agent", or actor, using data collected from exploring an environment. The agent can choose from a set of actions in the environment based on the current state. The state is updated as a result of the action and a reward is computed to reflect the benefit produced as a result of the agent's action. The new state and reward are used to modify the agent in a way that encourages reward-positive actions and discourages reward-negative actions.

A variety of techniques can be used to construct this feedback loop. The Deep Q-Network (DQN) approach uses a DL model as its agent with and SGD for the feedback loop. For the purposes of this dissertation, we will primarily focus on this mixed DL-RL setting, also known as *deep reinforcement learning*.

DQN Technique. The agent model is trained to approximate an unknown function Q, where $Q(s,a)$ yields the reward for execution action a in the environment state $state$. Then, this DL model can compute an expected *total* reward for all possible a's at a given state s, then select the action that maximizes the expected reward. The action space should be relatively small to make this search feasible, as excessively large action spaces are known to reduce model accuracy [202]. It has become common practice to employ *action space shaping* [93],

reducing and combining actions to simplify the space. In multi-discrete action spaces (e.g. a keyboard), wherein multiple simultaneous actions can be taken at once, the potential action space is exponential with a degree of the maximum number of simultaneous actions.

Selecting an action from a space requires understanding both the *immediate* and *long-term* reward. To predict "overall" reward of an action, the Optimal Action Value Function is used [28] to shape the agent's behaviors and teach it expected rewards over time. Thus, the agent learns a model of its environment and how its actions will change its state and impact its rewards.

This design works well in settings where responsiveness and adaptability are important. The agent can actively make decisions in response to environmental changes, a positive contrast against static one-shot optimizers.

Online vs Offline RL. An RL agent can be built in either the "offline" or "online" setting. Offline RL agents are trained in a simulation environment to understand how the various factors of their environment impact performance. They rely on the assumption that the final, live environment will be reasonably similar to the offline simulation settings.

Online RL, by contrast, tunes the agent as it actively interacts with the target application. This is more flexible and adaptive, but historically, long convergence times have been a significant concern. Some recent works have proposed a hybrid of the two, initially pre-training the RL model on offline simulation data then re-tuning it online [164, 146]. The effectiveness of this hybrid is largely dependent on the specifics of the target application.

Chapter 3

HYDRA: Optimized Hybrid Task-and-Spilled Parallelism for Multi-Large-Model Workloads

In this chapter, we dive deeper into our techniques to combine task parallelism and model spilling to enable high-efficiency model selection on large-model workloads. While prior large-model systems generally focus on training *one model at a time* with maximum efficiency, HYDRA recognizes that DL practitioners often train models in bulk due to model selection needs, e.g., hyperparameter tuning, architecture selection, etc. This gap often leads to significant system inefficiency. We approach this problem from first principles and propose a new information system architecture — HYDRA — for scalable multi-model training that adapts and blends ideas from classical RDBMS design with task parallelism from the ML world. We propose a suite of techniques to optimize system efficiency holistically, including a highly general parameter-spilling design that enables large models to be trained even with a single GPU, a novel multi-query optimization scheme that blends model execution schedules efficiently and maximizes GPU utilization, and a double buffering idea to hide latency. Experiments with real benchmark large-scale multi-model DL workloads show that HYDRA is over 7x faster than regular model parallelism and 1.8-4.5X faster than state-of-the-art industrial tools for large-scale model training.

3.1 Introduction

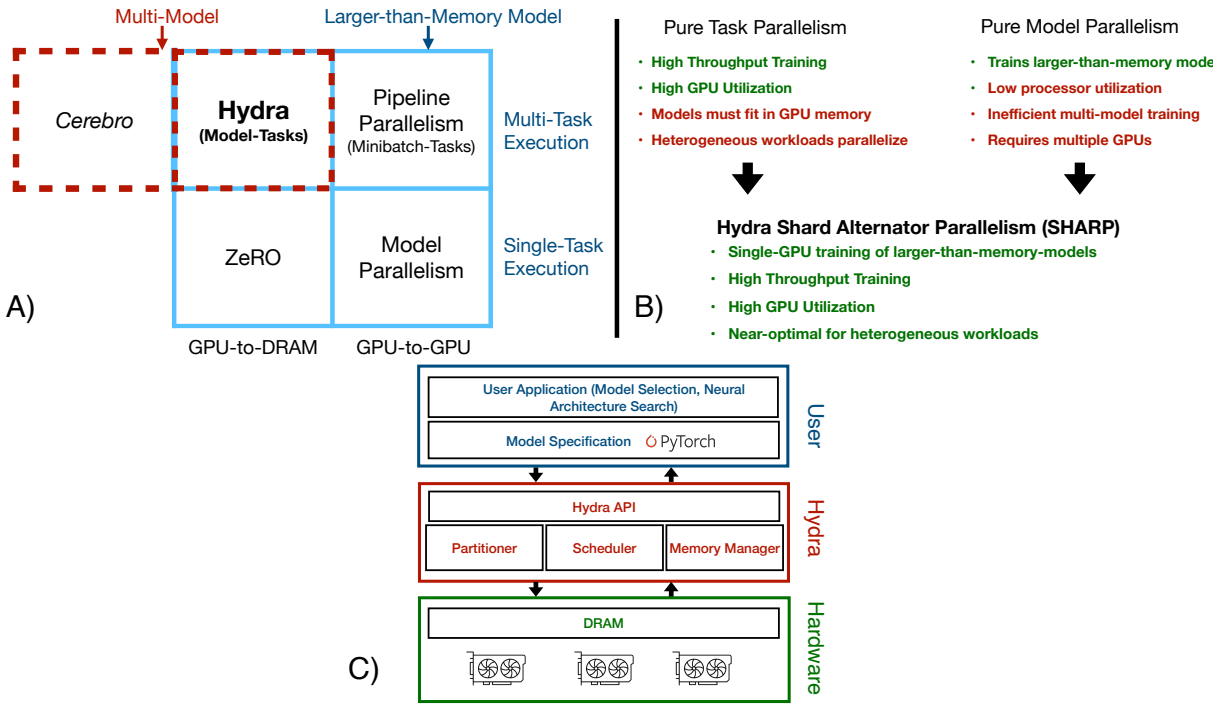

Figure 3.1. A) HYDRA introduces the first known hybrid of model- and task- parallelism. HYDRA trains multiple models (like Cerebro [151]), exploits task parallelism more effectively than pipelining, and expands the GPU-to-DRAM offloading optimizations of ZeRO and DeepSpeed [178, 173]. B) HYDRA combines the benefits of model- and task-parallelism, hybridizing them to eliminate the weaknesses of both and offer their added strengths. C) HYDRA in the context of user-interactions and hardware.

Example.

Consider the example of a political scientist building a text classifier for sentiment analysis on Twitter posts (tweets) to understand polarization between gun rights and gun control supporters. Their dataset is a few GB in size, much like the majority of DL training jobs according to recent polls [3]. So, a single-node multi-GPU setting suffices for their task. They download a few different state-of-the-art BERT models from HuggingFace [223] and prepare to fine-tune them. They aim tune relevant hyperparameters of all models to help raise accuracy. However, their GPUs do not have enough memory to even hold one model, and execution causes

PyTorch or TensorFlow to crash and impede their application.

There are two key issues here. First, the GPU memory bottleneck prevents users from fully exploring the possibilities of state-of-the-art DL models. Even for cloud users, who are not bound by their own local resources, this limitation forces them to spend more money on renting more expensive machines with larger GPUs. Second, the multi-task nature of the model building process amplifies the hassle of grappling with large models many times over, leading to more costs and DL user frustration.

System Desiderata.

We have the following key desiderata to support a large-scale multi-model training system.

- **Out-of-the-box Model Scalability.** We desire an approach that can easily scale to very large models even if the user has only one GPU. It should not force users to get multiple GPUs but nor should it require them to make special manual efforts to adjust their model or manually handle resources when they do have multiple GPUs.

- **High Throughput.** Given the ubiquity of needing to train multiple models, we desire an approach that parallelizes execution to achieve higher overall throughput across a model training job's lifecycle.

- **Resource Efficiency.** We desire a system that is aware of its hardware environment and maximizes the utility of available GPUs. Not only do we wish to avoid *wasting* resources, but we also aim to exploit the full *potential* of what resources we do have.

- **No Effect on Accuracy.** We desire a system that does not directly affect the data being processed or modify the execution patterns of models. Actually altering the model's training procedure tends to cause accuracy degradation, something that must be avoided if we are to offer a seamless training experience for DL practitioners.

We do *not* focus on data scalability, nor are we focused on scaling across multi-node clusters. We are not in the setting of scaling training to 1000 GPUs or PB-sized datasets on massive clusters like the Googles, NVIDIAs, and OpenAIs of the world. We aim to democratize large DL models to regular users in the domain sciences, enterprises, small Web firms, etc. From our own conversations with such data scientists at UCSD and at some companies, we note that a majority operate on single-node multi-GPU settings. It suffices for their data scale and avoids the hassle of operating multi-node clusters. Thus, in this work, we focus on studying the single-node multi-GPU setting in depth for HYDRA. We leave it to future work to extend our techniques to multi-node clusters, say, by integrating it with Cerebro[151, 100] or DeepSpeed [178, 173, 167].

Limitations of Existing Landscape.

There has been a flurry of recent work in the ML systems world aimed at addressing some of these issues. However, each fails on one or another of our desiderata.

1) False Dichotomy of Task- and Model- Parallelism

Existing systems do not address the problems of multi-model training and large-scale model training at once — they choose to focus on either one or the other. Current multi-model systems cannot train models that do not fit into GPU memory [99, 151, 100, 109, 111], and existing larger-than-GPU-memory model techniques [82, 70, 22, 89, 173, 178, 132, 99, 79] do not consider the possibility of multi-model execution. This is problematic, as these systems' inability to consider all aspects of their workloads prevents them from optimizing the system holistically to improve model training throughput and maximize resource efficiency. Figure 3.1A) illustrates a few critical examples of this, as well as the gap our work fills.

2) Low Scalability & Resource Efficiency

Most techniques aiming to support the training of models larger than the memory capacity of a single GPU use "model parallelism" [140] as a starting point. Unfortunately, model parallelism offers poor scalability, and tends to suffer from low resource utilization due to sequential dependencies in model architectures that prevent parallel device execution. Systems

that build off of model parallelism generally inherit these weaknesses to a greater or lesser degree. We elaborate on these tradeoffs further in Section 8.1.

3) Inability to Train Larger-than-GPU-Memory Models

Considering instead systems that are built specifically for multi-model execution and model selection, none actually support larger-than-GPU-memory model training [99, 151, 100, 109, 111]. All focus on the challenge of processing a large number of model configurations in parallel, an approach known as *task parallelism*. As such, attempting to train larger-than-GPU-memory models with such systems will generally lead to crashes and other undesirable behaviors. We elaborate on these systems further in Section 8.1.

Overall, we observe two issues with prior art. First, *they conflate scalability with parallelism*. Parallelism is not a fix for scalability but complementary. One must address the scalability bottleneck from first principles. Second, *they all fail to recognize or exploit a key source of higher degree of parallelism in DL workloads: training multiple models in one go*, e.g., during model selection such as hyperparameter tuning or neural architecture engineering [99]. By focusing on this new aim of training multiple model tasks, we introduce a new design motivator that has not been present on prior works.

Our Approach.

We present HYDRA, a new system for large-scale multi-model DL training that optimizes workloads holistically. HYDRA is easy-to-use, exposing only higher level APIs to shield DL users from having to manually optimize execution. Figure 3.1(C) illustrates our high-level architecture and its placement in between the user-facing APIs and the hardware (more details in Section 3). The user provides a set of model specifications in a popular DL tool (we focus on PyTorch), as shown in Listing 3.1, but beyond this initial step, all training procedures are managed and executed by HYDRA. Our target setting is single nodes with multiple GPUs, a common setting in natural language processing *NLP) where pretrained models tend to be very large, but fine-tuning datasets tend to be relatively small [77]. We focus on supporting workloads

with several large-scale sequentially defined models (e.g. Transformers) that are too large to fit into the memory of a single GPU, but sufficiently small to fit into DRAM.

Techniques in HYDRA.

HYDRA achieves our previously described desiderata with a suite of data systems techniques, some novel and some old (inspired by RDBMSs), albeit repurposed to DL systems. But our main novelty is also in how we assemble this "right" set of techniques and adapt them to build a fully automated system for large multi-model DL workloads. At the core of HYDRA is a novel hybrid form of parallel DL execution that *blends task parallelism and model parallelism* to improve overall resource efficiency in multi-GPU multi-model cases. Figure 3.1B) positions our hybrid parallelism approach, which we call Shard Alternator Parallelism (SHARP). Basically, model parallelism's main con is that it keeps only one GPU busy at a time, while task parallelism's main con is that it requires a model to fully fit in a GPU's memory. SHARP obviates both these issues. While Section 4 will present the details of how SHARP works, our intuition is to "break multiple models down and put them back together in a blended, better way."

In order to support SHARP, we design several other components to generalize model parallelism into a more flexible and more efficient form.

First, we devise a *model spilling* technique, an analogue to "data spilling" in RDBMSs where parts of the data are put at a lower level of the memory hierarchy. We blend that with model parallelism's notion of model shards by breaking up a large model, only loading some model shards onto GPUs, while the rest sit in DRAM. This is akin to sharding a large table in an RDBMS and loading only the active shard from disk to buffer manager. Model spilling provides a great deal of flexibility in execution management and scheduling for SHARP.

Next, we fully *automate the partitioning* of all models to respect GPU memory constraints with a lightweight and highly general approach.

Third, we adopt the classic RDBMS trick of *double buffering* to reduce the latency in between execution of model shards on a device. It overlaps GPU computation with loading from

DRAM and works in lockstep with SHARP.

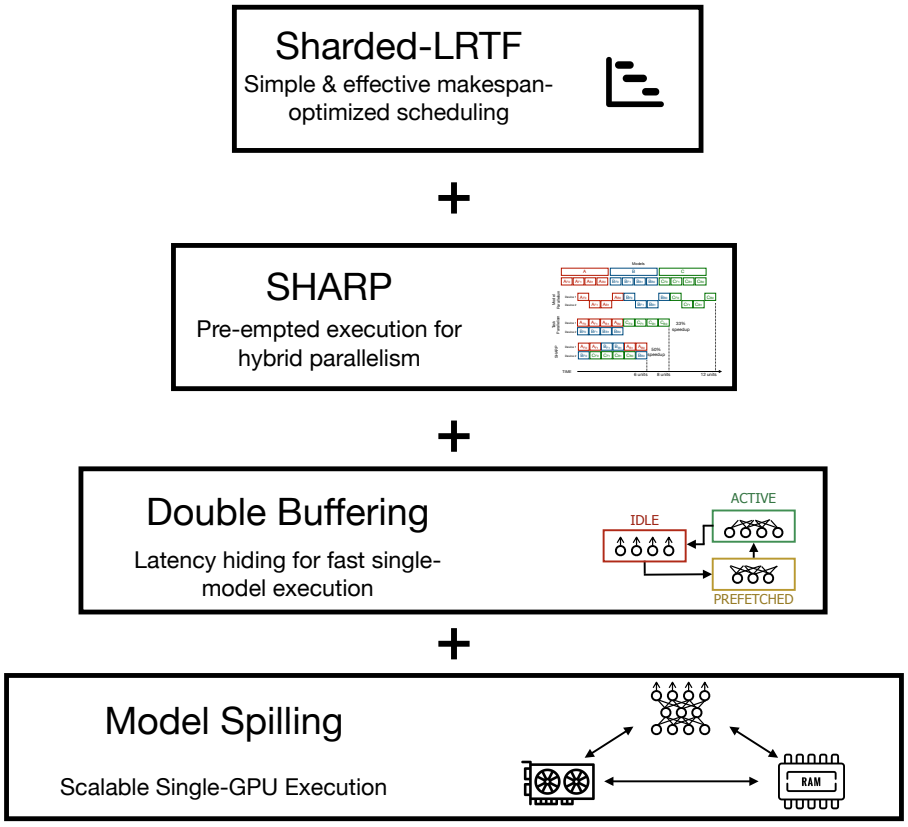

Figure 3.2. HYDRA's layered optimization stack combining scalability (spilling), low-latency per-model execution (double-buffering), hybrid parallelism (SHARP), and efficient scheduling (Sharded-LRTF).

Finally, we put all these techniques together to formulate a formal scheduling problem for our setting. We propose a simple greedy algorithm we call *Sharded-Longest-Remaining-Time-First (Sharded-LRTF)* to tackle it. We show using simulations that Sharded-LRTF offers near-optimal results for both homogeneous and heterogeneous sets of models. Figure 3.2 shows how these optimizations build on top of one another.

We prototype all of our ideas on top of PyTorch to create HYDRA. We evaluate it empirically on two key large-model benchmark workloads and datasets: hyperparameter tuning for BERT-Large on the WikiText-2 [133] dataset and neural architecture evaluation for Vision Transformer [49] on the CIFAR-10 dataset. HYDRA substantially outperforms all prior approaches,

yielding near-linear speedups on an 8-GPU machine for both workloads. In particular, it offers almost 7.5x speedups over regular model parallelism, with a substantial speedup over pipeline parallelism. HYDRA also reports the highest GPU utilization. We then dive deeper into the behavior of HYDRA by varying model scales, number of models trained together, and number of GPUs. We also report an ablation study showing the impact of our two key optimization techniques: SHARP and double buffering.

In summary, this work makes the following contributions:

- To the best of our knowledge, this is the first work to study the union of scalability and parallelism from first principles for multi-model training of very large sequential DL models. Our current focus is single-node multi-GPU settings.
- Inspired by RDBMSs, we present a suite of scaling and efficiency techniques combining automated model partitioning, model shard spilling, and double buffering.
- We devise a novel hybrid form of DL execution called SHARP combining task parallelism and model parallelism that mitigates the major cons of both.
- We cast our multi-model shared training as a form of multi-query optimization and build a simple scheduler featuring an efficient greedy algorithm called Sharded-LRTF.
- We implement all our ideas in a system we call HYDRA. A thorough empirical evaluation with real multi-model large DL workloads shows that HYDRA substantially outperforms prior state-of-the-art open source and industrial systems.

3.2 Architecture of HYDRA

HYDRA is designed to be a lightweight wrapper around the popular DL tool PyTorch.[1] We do not need any internal code of the DL tool to be altered, which can help ease practical

[1]It is a relatively simply engineering effort to add support for TensorFlow too but we skip it in our current version for tractability.

adoption. Figure 3.1 illustrates the overall architecture of HYDRA and how it handles the models. There are 4 main components: *API*, *Automated Partitioner*, *Memory Manager*, and *Scheduler*. We briefly explain the role of each component.

- **API.** The user specifies a set of models to be trained using standard PyTorch APIs. Listing 3.1 provides an example. Note that HYDRA can also scale the training of single model on a single device. The neural architectures are *fully automatically* inferred by HYDRA; no custom annotations are needed. The user just needs to provide the model(s), a data loading function, and model selection specification, e.g., hyperparameter search grid or metaheuristics.

```
from hydra import ModelTask, ModelOrchestrator

# define some models, model_0 and model_1...
# define some dataloaders, dataloader_0 and dataloader_1
# define some task specs, loss_functions, learning rates, etc...

task_0 = ModelTask(model_0, loss_fn, dataloader_0, lr_0, epochs_0)
task_1 = ModelTask(model_1, loss_fn, dataloader_1, lr_1, epochs_1)
orchestra = ModelOrchestrator([task_0, task_1])
orchestra.train_models()
```

Listing 3.1. Example usage of the API of HYDRA.

- **Automated Partitioner.** HYDRA automatically ascertains the memory size(s) of the GPU(s). Then it automatically partitions the model(s) given into model shards that respect the GPU memory constraints. At a given point in time, a GPU runs computations for only only one model shard. Section 4.3 explains our sharding process in more detail.

- **Memory Manager.** After the model shards are constructed, HYDRA puts them all in the machine's DRAM. It then moves shards up the memory hierarchy into the GPU based on the schedule produced by the *Scheduler*. When a shard's computation is completed, it is moved back to DRAM. Intermediate outputs within a model across shards are also written to DRAM by this component. All of this happens transparently to the DL user. This is the crux of how HYDRA achieves seamless scalability to very large models. Section 4.2

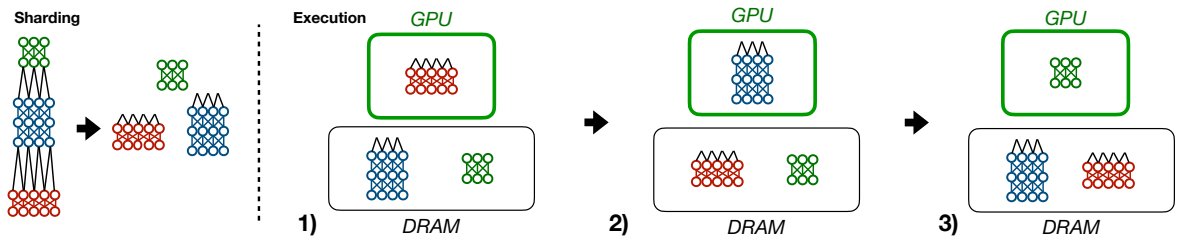

Figure 3.3. Illustration of model spilling as a temporal schematic. HYDRA places inactive shards at a lower level of the memory hierarchy (DRAM here), from which they are re-activated later.

explains this "model spilling" technique in more detail.

- **Scheduler.** This component is the core orchestrator of what shard gets placed on what GPU and when. It uses SHARP, our novel hybrid of task- and model parallelism. It ensures that the shards of a given model are scheduled in a way that respects the *sequential dependency* inherent in the DL model's forward pass and backward pass. We formalized this scheduling problem as an MILP, compared a few alternative scheduling algorithms, and devised a simple new algorithm that best suits our system setting. We also devised a buffering technique to further raise resource utilization. Sections 3.3.7.1– 3.3.7.3 explain our ideas in more detail.

3.3 Techniques in HYDRA

We now dive into the techniques in HYDRA to achieve seamless scalability and resource-efficient parallelism for training multiple large DL models in one go. Our techniques are inspired by a suite of classical ideas in RDBMSs, viz., spilling, sharding, multi-query optimization, and double buffering [174, 184], but our work is among the first to study them in the context of DL training. While the individual techniques may not be highly novel in the context of data management systems, the way we identify the right set of techniques, adapt them for DL, and synthesize them in HYDRA is novel. This enables HYDRA to offer state-of-the-art results in this important DL systems setting.

3.3.1 Intuition, Motivating Challenges, and Technical Novelty

HYDRA's core optimization is based on a relatively simple insight. Model parallelism's dependencies are too rigid to utilize resources efficiently, and task parallelism is too coarse-grained with its uninterrupted execution requirement. Blending the two together solves both problems. Task parallelism allows us to sidestep dependencies in model parallelism by using blocked or idle devices to train alternate tasks, similar to the difference between sleeping and busy waiting in CPU processor design [51]. Meanwhile, model parallelism allows us to break down previously atomic training tasks into "chunks", based on shards, and then reassemble shards of different models in a more efficient way. This is the same idea that underpins the speedups seen in prior hybrid parallel works [151, 116].

To exploit this intuition, our system must demonstrate two key characteristics.

1) Shard Movement Flexibility. For our model shards to be taken apart and put back together again effectively, we must introduce a degree of flexibility that is not present in model parallelism. Model parallel shards *cannot* be moved around between devices, nor can they be offloaded or delayed. But if we are to blend schedules across models, we must support the possibility of shards being temporarily put off or shifted. To solve this problem, we introduce *model spilling*, wherein we demote and promote shards of the model between GPU memory and DRAM. We then layer on communication optimizations with *double-buffering*. Note that this concept is *not novel*, spilling is an age-old idea from RDBMSs. Offloading has also been explored before [132, 99, 79]. It is our generalization of offloading tensors into spilling full sub-models (shards) and our redesign of RDBMS spilling for DL that is novel.

2) Shard Orchestration Once we obtain sufficient flexibility to orchestrate models freely, we must move and schedule them efficiently. Model schedules must be blended together to achieve optimal makespans, but the choice of shards and models at each timestep must be decided by some scheduler. We introduce our blended parallel scheme, Shard Alternator Parallelism (SHARP), and a greedy scheduler, Sharded-Longest-Remaining-Time-First (Sharded-LRTF).

The basic concept of scheduling across multiple different tasks is not inherently novel — prior work in multi-query optimization has explored this for the RDBMS setting, and hybrid parallel works[151, 116] have explored it in the context of data parallelism and task parallelism. However, this is the first work to expand this concept to the model-task hybrid setting, and the first to devise a scheduler for this problem.

3.3.2 Model Spilling

In traditional model parallelism, a model is divided into shards; each shard is placed on a different (GPU) device. But all shards must be loaded across multiple GPUs for a forward/backward pass to work at all. We observe that this is an overkill: due to the sequential dependency across layers inherent in DL models, only one device is typically "active" with computation at any point in time. The other devices are merely repositories for inactive model shards.

Exploiting the above observation, we use a simple idea in HYDRA to avoid making all shards active: *spill to DRAM*. Only an active shard is promoted to a GPU's memory, while the rest "wait" in DRAM. This is akin to sharding a large table and staging reads between disk and DRAM in RDBMSs, except we focus on a higher level in the memory hierarchy and apply it to a large model instead. All this means HYDRA scales to arbitrarily large models *on a single device*. So, even a trillion-parameter DL model can now be trained on a single GPU out of the box, given sufficient DRAM. This can already makes a qualitative difference for DL users with limited resources.

Model spilling is a direct reimagining of model parallelism, serving as a substitute rather than a complement. It retains the notion of shards and chunks of layers and does not alter the execution pattern, unlike other memory offload systems [173, 178, 132]. A natural analogy might be busy waiting versus blocking in CPU execution. In model parallelism, unused shards still block the CPU, busy-waiting in place while their dependencies are resolved. SHARP sends these unused shards "to sleep", allowing processors to work on other models in the meantime. Spilling can be used as a direct replacement of existing model-parallel setups. The only downside

is that model spilling introduces GPU-DRAM interactions, which are slower than the GPU-GPU interactions of model parallelism. The latency costs can be mitigated or even eliminated, however, as shown in Sec 3.3.6.

The use of model spilling, or indeed any sharded execution strategy, poses two open questions: Who does the sharding and how? How to efficiently train multiple models together with sharding in a multi-GPU setup? Our next two techniques tackle precisely these questions.

3.3.3 Automated Model Partitioning

Both traditional model parallelism and our model spilling depend on some sort of "cut point" to split the neural computational graph because shards must consist of disjoint subsets of layer groups. Prior art [159] uses some basic heuristics, albeit restricted to a specific class of models. Unfortunately, their approach is not general enough for our purpose. While one could use sophisticated graph partitioning algorithms for "optimal" partitioning, we find that is not worthwhile for two reasons. First, this stage is anyway only a tiny part of the overall runtime, which is dominated by the actual training runs. Second, due to the marginal utility of over-optimizing here, it will just make system engineering needlessly complex.

We prefer simplicity that still offers generality and good efficiency. Thus, we use a *dynamic greedy approach* based on toy "pilot runs." Algorithm 1 presents it succinctly. The basic idea is to pack as much of a model as possible on to a GPU. If the set of GPUs is heterogeneous, we use the smallest-memory GPU to ensure cross-device compatibility of shards. We treat a DL model as an ordered list of layer indices, with the layers being "cut-points" in the graph to enable smooth partitioning. HYDRA then iterates through these indices to run "toy" passes with a *single mini-batch once*. If the run is successful, the Partitioner raises the shard size by appending the next set of layers. If the GPU throws an out-of-memory error, we remove the set of layers appended last. Thus, in this dynamic way, we find the near-maximum set of layers that fit in GPU memory; this set is then cut off from the model as its own shard. The Partitioner continues this process for the remaining the layers, until that model is fully partitioned We record runtime

Algorithm 1: Dynamic model partitioning algorithm.

Input: Model as a sequence of layers L with size m; data mini-batch B; GPU G
Output: Array of partition indices A

```
Append 0 to S
for i = 0 to m − 1 do
    Place L[i] and B on G
    B′ ← Forward pass through L[i] with B
    T ← New tensor with same shape as B′
    Backpropagate T through L[i] without freeing its memory
    if G out of memory then
        Append i to S
        for j = 0 to i − 1 do
            Release all memory consumed by L[j]
            Append i to A
        end for
    end if
end for
```

statistics for later use by our Scheduler.

3.3.4 SHARP

We now present one of our key novel techniques: Shard Alternator Parallelism (SHARP), a hybrid of classical model parallelism and task parallelism. We define our basic unit of computation, *shard unit*, as follows: the subset of computations of a forward or backward pass on a model's shard. Thus, a full forward or backward pass of a model is a *sequence* of shard units.[2] Overall, the scheduling goal is to execute all shard units of all models given by the user for all epochs.

Figure 3.4 illustrates the basic idea of SHARP contrasted with both task- and model parallelism. After a model's shards are created (Section 3.3.3), shard units are naturally set. The key difference in SHARP is that a given model's shard units do not necessarily run *immediately*

[2]In recent ML literature, this unit is also called a "microbatch" [82]. We prefer to use the more standard terminology of "unit" from the operations research and systems literatures instead because the term "microbatch" may cause confusion on whether the mini-batch *data* is split further, which is not the case. A shard unit splits the *computations* (not data) of a forward/backward pass of a whole mini-batch.

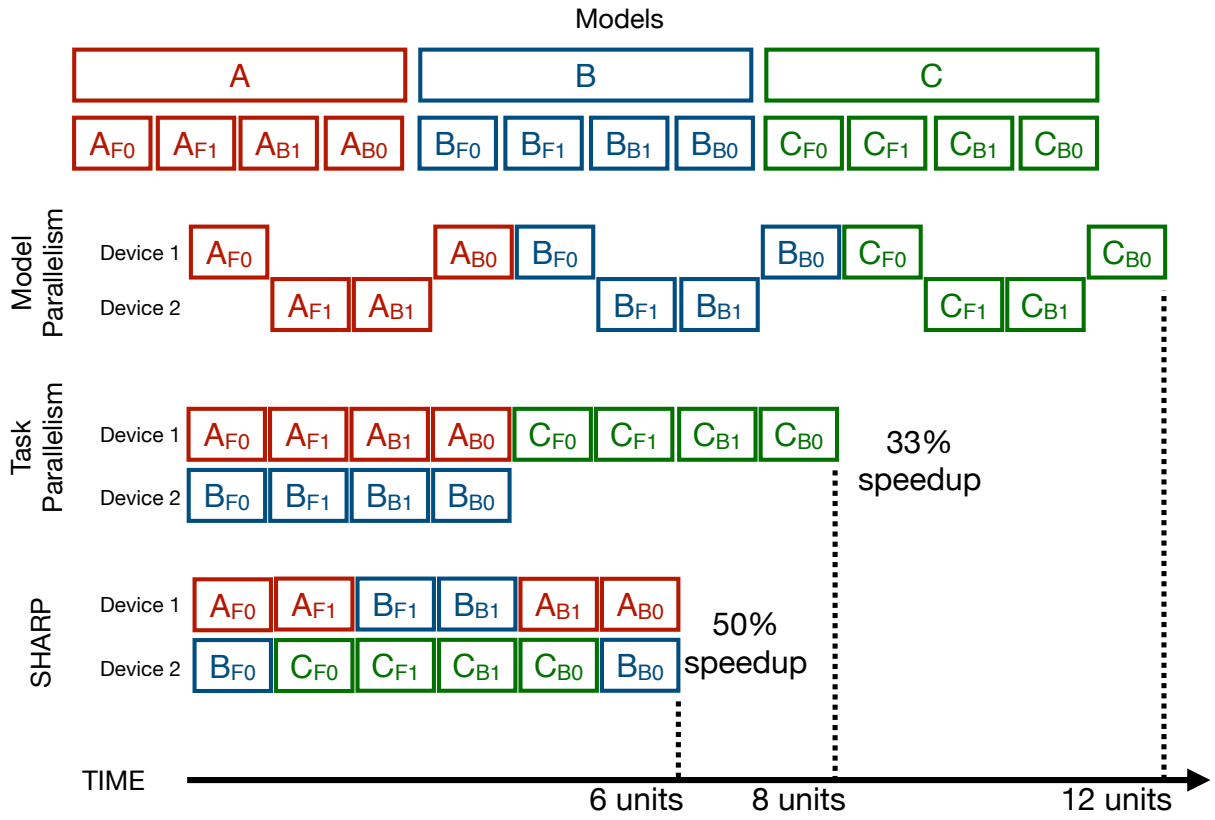

Figure 3.4. Demonstrative illustration of SHARP and contrasting with regular task parallelism and model parallelism for training 3 models A, B, and C, each with 2 shards. Real-world schedules tend to be more complex, but this simplified diagram shows SHARP's capacity for optimization.

after one another, i.e., they may be staggered over time. This is the key reason for SHARP's higher efficiency–it breaks things down and puts them back together better.

Notice that SHARP is only possible due to the flexibility of model spilling. Model spilling's shards can be used as semi-independent sub-models, as contrasted with parameter-spilling systems where there is no real notion of independence. Only with shard-spilling can we effectively support blended execution schedules across models.

While the idea of SHARP is fundamentally simple (but novel), realizing it in a working system faces two bottlenecks: (1) the sheer number of shard units and (2) the latency of swapping shards between device memory and DRAM. First, note that the number of shard units to be handled by HYDRA is multiplicative in four quantities: number of models given by the user,

number of shards per model, number of mini-batches per epoch, and number of epochs per model training run. In realistic DL scenarios, one can easily hit *tens of millions of shard units*! Thus, next we answer two questions regarding the above bottlenecks: How to automate the orchestration of large numbers of shard units? Is it possible to reduce latency of swapping shard units

3.3.5 Automated Shard Orchestration

To realize SHARP in an system, we must handle 3 kinds of "data" before, during, and after a shard unit: (1) training data mini-batch, (2) model parameters, and (3) intermediate data/outputs of a shard unit. Thankfully, DL tools like PyTorch offer APIs that enable data to be transferred from GPU memory to DRAM and vice versa. We use those APIs in HYDRA under the hood to automate shard orchestration.

Each model is defined as a "queue" of shards in DRAM, ordered according to the neural computational graph. Each shard is associated with its necessary data, such as an input mini-batch, intermediate data *between* shards, and/or gradients sent backward. The shard at the front of the queue is transferred to GPU memory along with its associated data to begin running that shard unit.

After execution completes, the shard parameters (possibly updated) are returned to DRAM. In addition, the shard's intermediate outputs, say, a gradient vector or a loss value, are also written to DRAM and attached to the model. They will be used as inputs for the model's next shard. The last shard of a model concludes a full mini-batch training pass; after that, the old mini-batch is discarded and the next mini-batch of the prepared data will be used.

3.3.6 Double-Buffering

A common trick used in RDBMSs, e.g., for external merge sort, is double-buffering [174]. The basic idea is this: the processor's memory (higher in the memory hierarchy) is split into two regions: one for active processing and the other as a "loading zone" for the next task. We

bring this trick to the DL systems world for the first time. HYDRA uses it to mask shard loading latencies. We protect a "buffer space" in GPU memory during model partitioning (Section 3.3.3) to guarantee that so much buffer memory will be available during training.

Prior analyses [192] demonstrate that intermediate activations produced during training form the largest proportion of GPU memory consumption, as much as 99% in some cases! Double-buffering need not transfer intermediate activations — those will be produced by checkpointing inputs between shard groups during training. Therefore, the double-buffer need only be large enough to hold the model state, optimizer state, and input data, which only forms only a small proportion of overall memory consumption. In practice, a buffer size consisting of 5% of total memory is more than sufficient, though users can adjust this value as needed.

When our Scheduler picks the next shard to be run, we transfer it to that GPU's buffer space *even as the previous shard unit is running there*. Interestingly, our double-buffered DL training in HYDRA also offers a serendipitous new bonus: we can avoid spilling (to DRAM) altogether in some cases. When a model's current shard unit is active, if its next shard is double-buffered on the same GPU, intermediates need not move at all, eliminating latency. While we focus on the GPU memory-DRAM dichotomy, our above techniques are general enough to be applicable across the entire memory hierarchy: between DRAM and local disk, local and remote disk, etc.

3.3.7 Scheduling Formalization of SHARP

The sheer number and variable runtimes of shard units across models necessitates a rigorous automated Scheduler. We immediately face two technical challenges. First, different models may train for different numbers of epochs due to convergence-based SGD stopping criteria or early stopping in AutoML heuristics. Second, devices may disappear over time, say, due to faults, or get added, say, due to elasticity. For these reasons, we choose a *dynamic scheduling* approach to place shard units on devices as and when a device becomes available over time. This design decision tackles all three challenges above in a unified way and also simplifies

system implementation.

We treat each model to be trained as a *queue of shard units* unifying reasoning of division within a mini-batch, across mini-batches within an epoch, and across epochs.

3.3.7.1 Formal Problem Statement as MILP

The scheduling problem is as follows. When a device (GPU) becomes available, a shard unit must be selected from one the model's queues to be placed upon that device. Shard units become *eligible* for scheduling if they have no pending dependencies, i.e., they are at the front of their queue and no other shard unit of that same model is still running on another device. The Scheduler's job is to pick a shard unit from the set of eligible shard units. Double-buffered training is already factored into this formulation: the Scheduler is actually picking shard units for double-buffering, and they get promoted from the buffer to compute.

All shard unit runtimes are given as input. Recall from Section 3.3.3 that the partitioner records this data during its pilot run. We now present the formal scheduling problem as an MILP. Table 3.1 explains our notation.

$$\texttt{Objective:} \quad \min_{X,Y} C \tag{3.1}$$

The MILP objective, as defined in Equation 3.1 is to pick a shard unit ordering that can minimize makespan (completion time of the whole workload at this granularity).

$$\forall t, t' \in [1, \ldots, |T|] \quad \forall p, p' \in P \forall j \in [2, \ldots, M_t] \; X_{t,p,j} \geq X_{t,p',j-1} + S_{t,j-1} \tag{3.2}$$

Equation 3.2 simply enforces the *sequential ordering of shard units* within a model. Note that this set per model here is unified within a mini-batch, across mini-batches within an epoch, and potentially across epochs too–they are all sequentially dependent.

Table 3.1. Notation for our MILP scheduling formalization. We denote the symbols used in our constraints as well as their plain-English definitions.

Symbol	Description
T	List of models specified by the user to be trained
P	List of devices (GPUs) available for training.
$M_i \in \mathbb{Z}^+$	M_i is the total number of shard units for model $T_i \in T$. Note that this covers all mini-batches (and potentially epochs).
$S_i \in \mathbb{R}^{M_i}$	S_i is a variable-length list of shard unit runtimes for model T_i. The runtime of shard unit j is denoted as $S_{i,j}$.
$X_i \in \mathbb{R}^{\lvert P\rvert \times \lvert M_i\rvert}$	X_i is a variable-shape matrix of start times of shard units of model T_i across workers. The start time of shard unit j on worker p is denoted as $X_{i,p,j}$. Note that this linear ordering covers not just the model's forward and backward passes but also ordering across mini-batches (and potentially epochs).
$Y_i \in \{0,1\}^{\lvert P\rvert \times L \times L}$	L is the total number of shard units across all models, i.e., $L = \sum_i M_i$, indexed cumulatively by the index of model i and its shard unit j (denoted i_j). $Y_{p,i\_j,i'\_j'} = 1 \Leftrightarrow X_{i,p,j} < X_{i',p,j'}$.
U	An extremely large value used to enforce boolean logic.

$$\forall j \in [1,\ldots,M_t] \quad \forall j' \in [1,\ldots,M_{t'}]\ X_{t,p,j} \geq X_{t',p,j'} + S_{t',j'} - (U \times Y_{p,t\_j,t'\_j'}) \tag{3.3}$$

$$\forall j \in [1,\ldots,M_t] \quad \forall j' \in [1,\ldots,M_{t'}]\ X_{t,p,j} \leq X_{t',p,j'} - S_{t,j} + (U \times (1 - Y_{p,t\_j,t'\_j'})) \tag{3.4}$$

Equations 3.3 and 3.4 enforce *model training isolation*, i.e., only one shard unit can run on a device at a time.

$$\forall j \in [1, \ldots, M_t]\ X_{t,p,j} \geq 0 \tag{3.5}$$

$$\forall j \in [1, \ldots, M_t] C \geq X_{t,p,j} + S_{t,j} \tag{3.6}$$

Equation 3.5 is just non-negativity of start times, while Equation 3.6 defines the makespan.

Using a MILP solver such as Gurobi [65] enables us to produce an "optimal" schedule in this context. But the above task is a variant of a general job-shop scheduling problem described in [206], and it is known to be NP-complete. Given that the number of shard units can span thousands to tens of millions, solving it optimally will likely be impractically slow. Thus, we look for fast and easy-to-implement scheduling algorithms that can still offer near-optimal makespans.

3.3.7.2 Intuitions on Scheduling Effectiveness

We observe that there are 2 primary cases encountered by a scheduler in our setting:

1. The number of models is equal to or greater than the number of available devices.
2. The number of models is less than the number of available devices.

In case (1), there will always be at least one eligible shard unit for each device at every scheduling decision. Any shard-parallel scheduling algorithm that accounts for all devices can easily keep all devices busy most of the time, i.e., *busy waiting* is unlikely. In case (2), all models can be trained simultaneously. Since each model's shard unit uses at most one device in SHARP, and since there are more devices than models, there is no contention for resources here. In this case, regular task parallelism-style scheduling suffices and the makespan will just be the runtime of the longest "task."

Algorithm 2: The Sharded-LRTF scheduling algorithm.

Struct {
  Remaining epochs e
  Minibatches per epoch b
  Remaining minibatches in current epoch ce
  Minibatch training time t
  Remaining train time in current minibatch cm
}
Input: Idle Models $[M]$
Output: Model $MaxModel$
$MaxTrainTime = 0$
for Index i, Model m in $[M]$ **do**
  $ModelTrainTime = ((m_e - 1) \times m_b + m_{ce} - 1) \times m_t + m_{cm}$
  if $ModelTrainTime > MaxTrainTime$ **then**
    $MaxTrainTime = ModelTrainTime$
    $MaxModel = m$
  end if
end for

In both the cases above, even basic randomized scheduling might yield reasonable makespans. However, what it will not take into account is that case (1) is not static. Over time, as models finish their training, our setting may "degrade" from case (1) to case (2). Thus, two different schedulers that operate on a workload in case (1) may differ in their effectiveness based on how gracefully they degrade to case (2). As noted before, the makespan in case (2) scenario is determined solely by the longest-running remaining model. This gives us an intuition for a simple scheduler that can often do better than randomized: *minimize the maximum remaining time among the remaining models*.

If degradation to case (2) occurs early on, and if there is a substantial differences in task runtimes post-degradation, the overall completion times can differ significantly based on the scheduling. Such degradation can arise in model selection workloads that use early stopping for underperforming models, e.g., Hyperband [111], or by manual user intervention. Thus, we aim for a scheduling algorithm that can address such cases too in a unified way.

We propose a simple and practical greedy heuristic we call Sharded Longest Remaining Time First (LRTF) based on our above intuitions. Algorithm 2 explains our algorithm. Sharded-

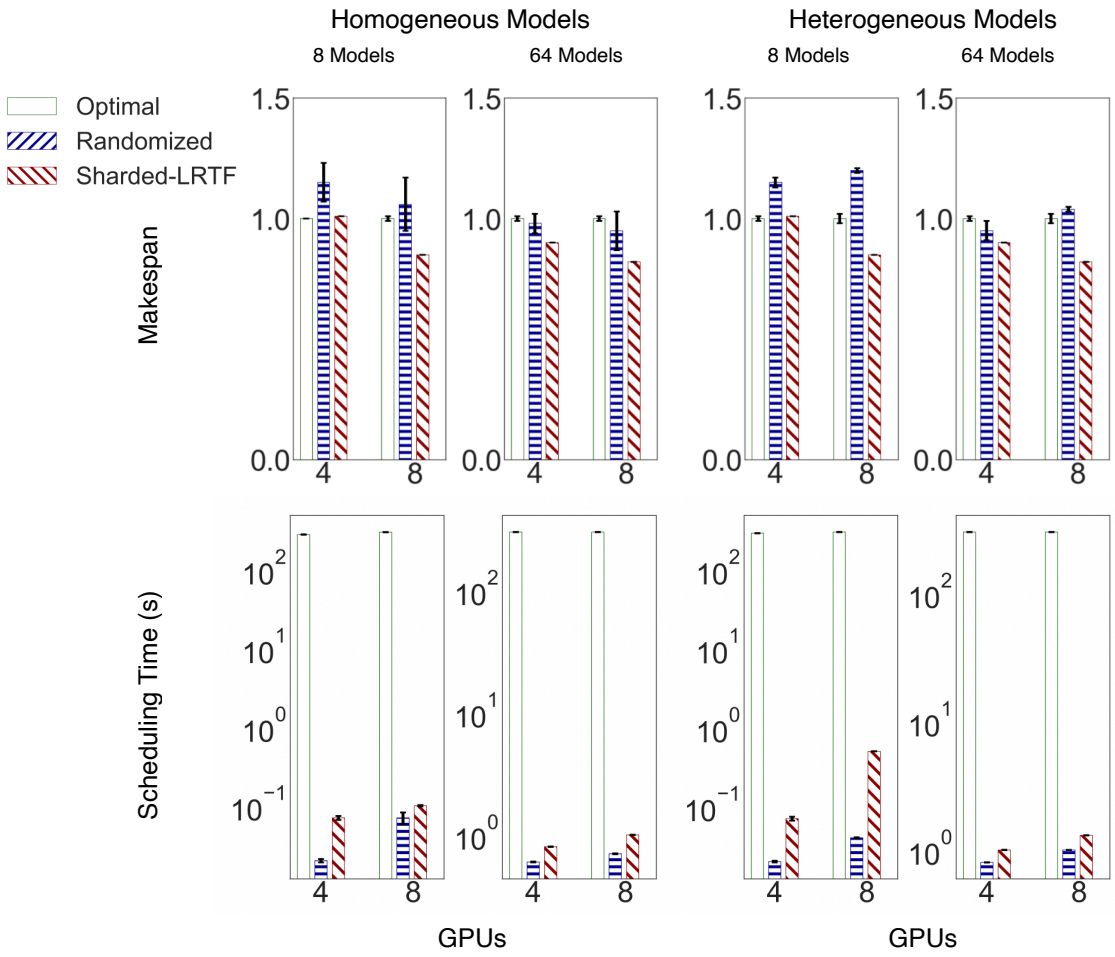

Figure 3.5. Comparison of various scheduling algorithms. Makespans are normalized to Optimal.

LRTF selects the model training task with the *longest total remaining train time* at every possible scheduling decision time. Since a new scheduling choice must be made at the completion of every shard unit, the selection will update its choice of longest task at a very fine-grained level. Note that the selection procedure runs efficiently with *linear time complexity*.[3]

3.3.7.3 Our Scheduling Algorithm: Sharded-LRTF

To quantitatively understand the effectiveness of Sharded-LRTF, we compare it using simulations against a basic randomized schedule and a Gurobi-output "optimal" schedule. For tractability, we set a timeout of 100s for Gurobi. We who both a homogeneous setting (all neural architecture are identical) and a heterogeneous setting, wherein they differ significantly. We assume all GPU devices are identical for simplicity, but that is also common in practice. Per-epoch runtimes of a model in the homogeneous setting are all fixed to 2 hours each, with 2000 shard units each. For the heterogeneous setting, per-epoch model runtimes are set between 30 minutes to 4 hours; number of shard units are set between 100 to 10,000. We randomly sample an initial set and report the average and standard deviations of 3 runs on the fixed set. Variance occurs due to non-deterministic scheduling behaviors from random selection and Gurobi timeout. Figure 3.5 shows the results.

We note that MILP "optimal" has higher makespan in some cases because Gurobi did not converge to the global optimal in the given time budget. The randomized approach matches it or performs worse in many cases. But Sharded-LRTF matches or significantly outperforms the other approaches in many cases, especially in the heterogeneous setting. This is in line with the intuition we explained that being cognizant of longer running models in the mix is helpful. Also note that the runtime of Sharded-LRTF is in the order of tens of milliseconds, ensuring it is practical for us to use in HYDRA repeatedly for scheduling shard units on devices dynamically. Note that the actual mini-batch training computations on the device are the dominant part of the overall runtime.

[3] In fact, an alternate data structure to record shard references can enable even constant-time selection.

3.4 Experiments

We now compare HYDRA against state-of-the-art open source and industrial tools for large-model DL training: PyTorch Distributed, Microsoft's DeepSpeed, FlexFlow from Stanford/CMU, and Google's GPipe. GPipe's microbatch count and partition count is equal to the GPU count. FlexFlow requires some manual guidance by editing the system-generated parallelism strategy file to ensure memory errors do not occur. We also show multiple variants with DeepSpeed, including superimposing a hybrid task parallelism (note that regular task parallelism is not applicable) and a hybrid data parallelism offered by DeepSpeed. We then dive into how HYDRA performs when various workload and system parameters are varied.

Table 3.2. Details of end-to-end workloads. *Architectures similar to BERT-Large and ViT, scaled up for demonstration.

	NLP Workload	Computer Vision Workload
Model Architectures	BERT-Large*	Vision Transformer (ViT)*
Model Sizes	1B	300M, 600M, 800M, 1B, 1.5B, 2B
Batch Size	8, 16. 32	512, 1024
Learning Rate	$10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}$	10^{-3}
Epochs	4	5

Workload Details. We use two popular DL model selection scenarios: hyperparameter evaluation and neural architecture evaluation. Table 3.2 lists details. For hyperparameter evaluation, we focus on masked-language modeling with the Transformer architecture BERT-Large [47], trained on the WikiText-2 dataset. The neural architecture is fixed and we vary batch size and learning rate as key hyperparameters to create a total of 12 models to train, each with 1B parameters. For neural architecture evaluation, we focus on a computer vision task with variants of the Vision Transformer (ViT) model [49] and the CIFAR-10 dataset. We create models with sizes between 300M parameters and 2B parameters. We also vary batch sizes, leading to a total

of 12 models again.

Machine Setup. We focus on single-node multi-GPU training, anecdotally the most common among DL practitioners. We use 8 Nvidia RTX 2080Ti GPUs with 11GB memory, NVLink-enabled. The machine runs Ubuntu 18.04. and the node has 500GB DRAM and 80 CPU cores.

3.4.1 End-to-End Workloads

Figure 3.6 presents overall runtimes and GPU utilization results. We find that the baseline off-the-shelf PyTorch Distributed and DeepSpeed model parallelism report massive resource under-utilization. Thus, their runtimes are the highest. The basic hybrids with data- or task- parallelism do provide higher utilization and some modest speedups, but the fundamental limitations of model parallelism persist with such approaches, such that they still fall substantially short of ideal linear speedup (8x in this case). GPipe-style pipeline parallelism is much better, with about a 4x speedup against regular model parallelism. But HYDRA is the most efficient approach overall, reaching about 7.5x, close to the physical upper bound. The average GPU utilization of HYDRA is also the highest at over 80%.

3.4.2 Drill Down Analysis

We now dive deeper into the behavior of HYDRA when varying key parameters of interest from both ML and system standpoints.

3.4.2.1 Impact of Model Scale.

We vary the scale of the models to see the impact on relative performance of HYDRA. We fix the number of GPUs at 8 and the number of models to 12. Figure 3.8 shows the results. We see that HYDRA's speedups over regular model parallelism *is fairly consistent* even as the model scale grows. This is because our partitioning approach (Section 3.3.3) and the dynamic Sharded-LRTF algorithm (Section 3.3.7.1) together ensure that shard unit times are similar

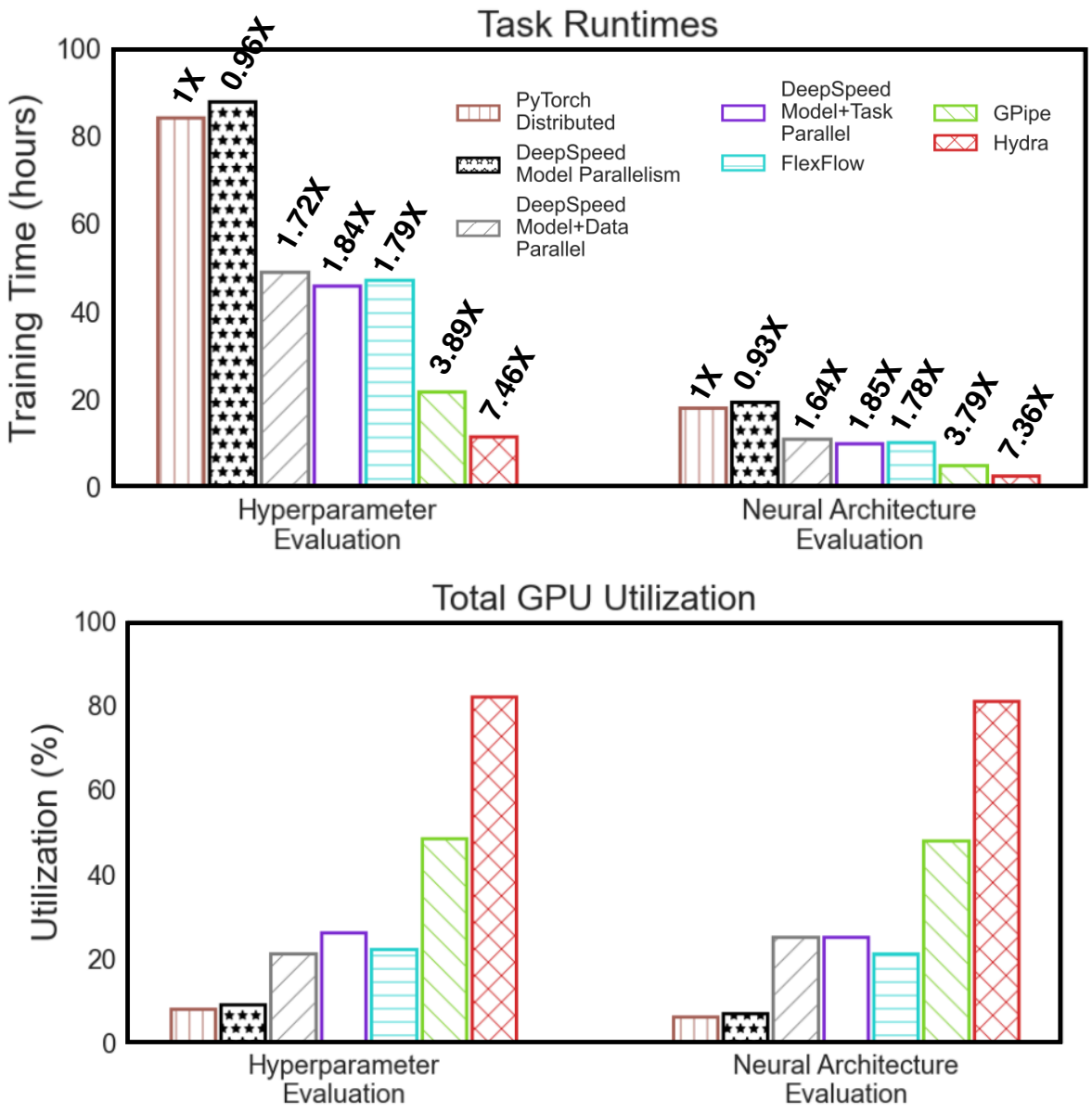

Figure 3.6. End-to-end workload results: Runtime speedups relative to the baseline PyTorch Distributed and GPU utilization. All evaluations were closely monitored to ensure none suffered hardware failure or GPU disconnects.

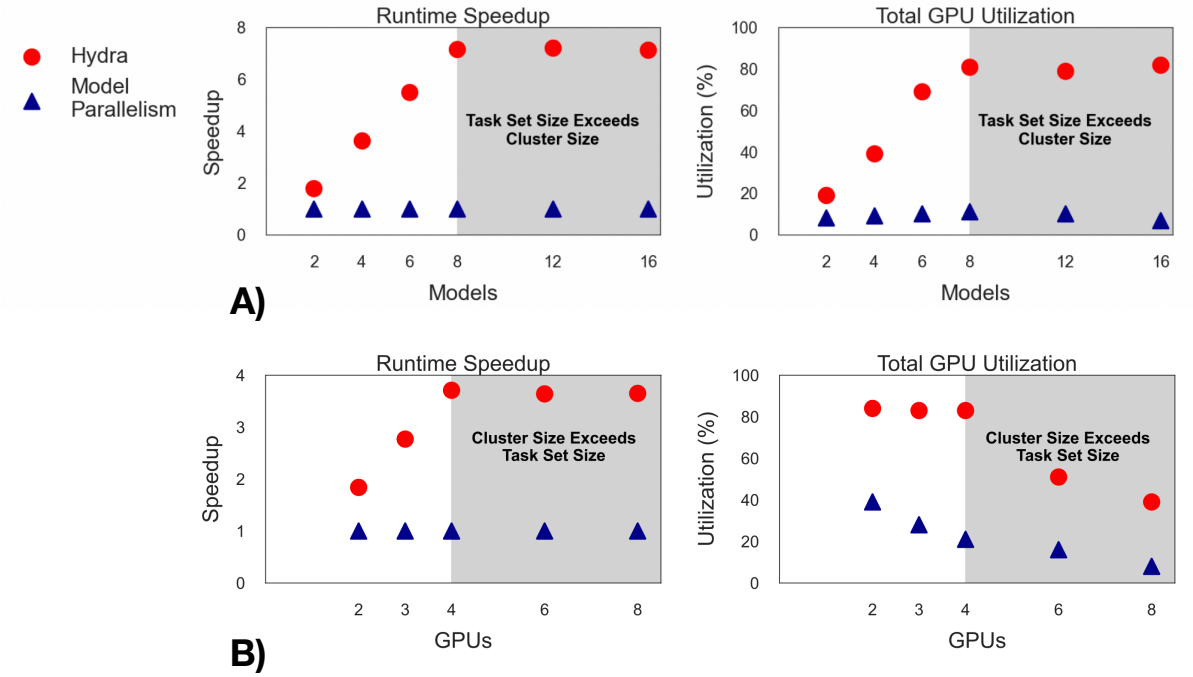

Figure 3.7. System microbenchmarks. A) demonstrates the impact of task set size on performance while resources are fixed, while B) demonstrates the impact of cluster size on performance while the task set size is fixed.

even as model scale grows; basically, it just leads to more shard units to run. Our SHARP and double-buffering techniques further ensure that having more shard units do *not* cause relatively more resource idling on average.

3.4.2.2 Impact of Number of Models.

We now vary the number of models that are trained together. The number of GPUs is set to 8; all models have are uniformly large, at 250M parameters (same Transformer workload as before). Figure 3.7A shows the results. We see that HYDRA exhibits close to 8x speedups when the number of models is 8 or more but lower than that, the speedup is capped close to the actual number of models. This flattening in the fewer-models regime is inherited from task parallelism by SHARP. The GPU utilization numbers vary proportionally to the speedups seen.

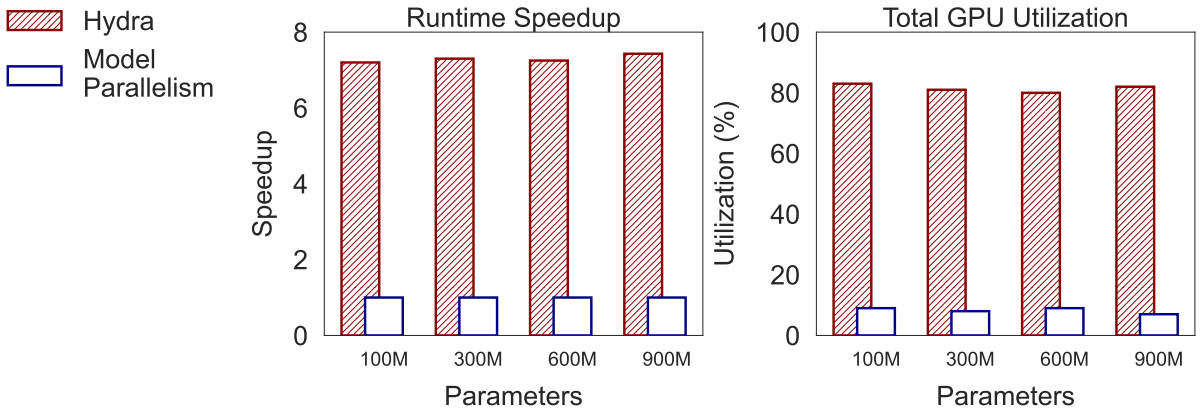

Figure 3.8. Impact of model scale. Runtimes normalized to the first instance of regular model parallelism for clarity.

3.4.2.3 Impact of Number of GPUs.

We now study how varying the number of GPUs affects HYDRA's speedup behavior. We fix the workload to 4 Transformer models, each with 250M parameters. We choose only 4 models to showcase both regimes: when the number of devices is less than models and vice versa. Figure 3.7B shows the results. We see that HYDRA exhibits a roughly linear speedup when there are more models than devices. And when that flips, since HYDRA runs out of models to schedule, the speedup flattens as the degree of parallelism is limited. As before, this is due to SHARP inheriting the degree of parallelism from task parallelism. We believe further hybridization of SHARP with data parallel training can help boost speedups and resource utilization in this regime; due to its complexity, we leave it to future work.

3.4.2.4 Ablation Tests.

In this experiment, we explore the effect of system components on framework performance. The number of devices is fixed to 8, with 16 Transformer models. All optimization levels include model spilling as a baseline, as this technique is critical to HYDRA's basic operations. Table 3.3 demonstrates the results. Pure model spilling dramatically slows down model training. This is only to be expected, given that it introduces a dependency on DRAM. SHARP's throughput improvements dominate the slowdowns of model spilling, but it is important to note that

SHARP's speedups are workload-dependent. Double-buffering largely eliminates the cost of model spilling, enabling further speedups.

Table 3.3. Runtimes and slowdowns of HYDRA when our two key optimizations are disabled one by one.

Optimization Level	Runtime (hrs)	Aggregate Speedup
HYDRA without SHARP or double-buffering	24.14	1X
HYDRA without double-buffering	4.3	5.61X
HYDRA	**1.85**	**13.05X**

3.5 Conclusion

Training larger-than-GPU-memory DL models is an increasingly critical need for DL users. Yet existing "model parallelism" tools are sub-par on scalability and parallelism, are often hard to use, and massively underutilize GPUs. Moreover, no existing system inherently supports both multi-model training and larger-than-GPU-memory model training. We present HYDRA, a new system for large-scale multi-model DL training inspired by the design and implementation of RDBMSs that uses the memory hierarchy consciously and exploits multi-task execution to optimize performance. We identify a judicious mix of data systems techniques–some novel and some classical RDBMS ideas adapted to DL (such as sharding, spilling, and double buffering)–to enable large-model training even on a single GPU. By further exploiting the high degree of parallelism in multi-model training, we devise a novel hybrid parallel execution technique inspired by multi-query optimization. Our work shows that the DL systems world can benefit from learning from the RDBMS world on data systems techniques that enable more seamless scalability and parallelism for DL users.

Chapter 3 contains material from "Hydra: A System for Large Multi-Model Deep Learning" by Kabir Nagrecha and Arun Kumar, and "Model-Parallel Model Selection for Deep Learning Systems" by Kabir Nagrecha, which appears in Proceedings of the 2021 International

Conference on Management of Data. The dissertation author was the primary investigator and author of this paper. The code for our system is open source and is available on GitHub: https://github.com/knagrecha/hydra.

Chapter 4

SATURN: Joint Optimization for Parallelism, Resource Allocation, and Scheduling on Multi-Large-Model Workloads

In this chapter, we dive deeper into our joint optimization techniques, which unify the problem spaces of parallelization, resource allocation, and scheduling on memory-intensive multi-model workloads. Such large-model architectures typically demand complex parallelization schemes to both distribute memory demands and accelerate training, e.g., fully-sharded data parallelism or pipeline parallelism. But the recent popularity of large-model architectures has spurred on the development of a vast number of competing parallelization schemes — each with their own advantages in different scenarios. Combined with this are the challenges of resource allocation and scheduling, which in turn affect the optimal parallelization choices. Thus, in order to both simplify the burden placed on practitioners and also to maximize end-to-end efficiency, a system that can optimize over all three interconnected dimensions at once is needed.

4.1 Introduction

Case Study: Consider a data scientist, Alice, building an SQL autocomplete tool to help database users at her company. She has a (private) query log that contains her company's database schemas, common predicates, etc. She downloads two LLMs from HuggingFace — GPT-2 and GPT-J — both of which are known to offer strong results for textual prediction

tasks [171, 213]. She finetunes multiple instances on her dataset, comparing different batch sizes and learning rates to raise accuracy. She uses an AWS instance with 8 A100 GPUs. She launches the DL tuning jobs in parallel, assigning one GPU each. Alas, all of them crash with out-of-memory (OOM) errors. She is now forced to pick a large-model scaling/parallelism technique and assign multiple GPUs to each job. But to do so she must answer 3 intertwined systems-oriented questions: (1) Which parallelism technique to use for each model? (2) How many GPUs to assign to each model? (3) How to orchestrate such complex parallel execution for model selection workloads?

In this paper, we tackle precisely those 3 practical questions in a unified way to make it easier, faster, and cheaper for regular DL users like Alice to benefit from such state-of-the-art large DL models.

4.1.1 Motivation

We start by first explaining why prior art for large-model and parallel DL systems is insufficient to tackle the problem. Table 4.1 lists a conceptual comparison of our setting with prior art on several key aspects. Section 8.2 discusses related work in greater detail.

(1) Which parallelism technique to use for each model? There are a multitude of techniques in the ML systems world to parallelize/scale large models across GPUs. Some common techniques are: sharding the model, spilling shards to DRAM [79, 132], pipeline parallelism as in GPipe [82], fully-sharded data-parallel (FSDP) as in PyTorch [2] and ZeRO [173], hand-crafted hybrids as in Megatron [190], as well as general hybrid-parallel approaches such as Unity [207, 89] and Alpa [241]. But no technique dominates all others in all cases. Relative efficiency depends on a complex mix of factors: hardware, DL architecture specifics, even batch size for stochastic gradient descent (SGD). Figure 4.1(B) shows two empirical results on real workloads to prove our point. Even between just pipelining and FSDP, complex crossovers arise as GPU counts and batch sizes change. Furthermore, many techniques expose knobs that affect runtimes in hard-to-predict ways [118], e.g., pipelining requires tuning partitions and

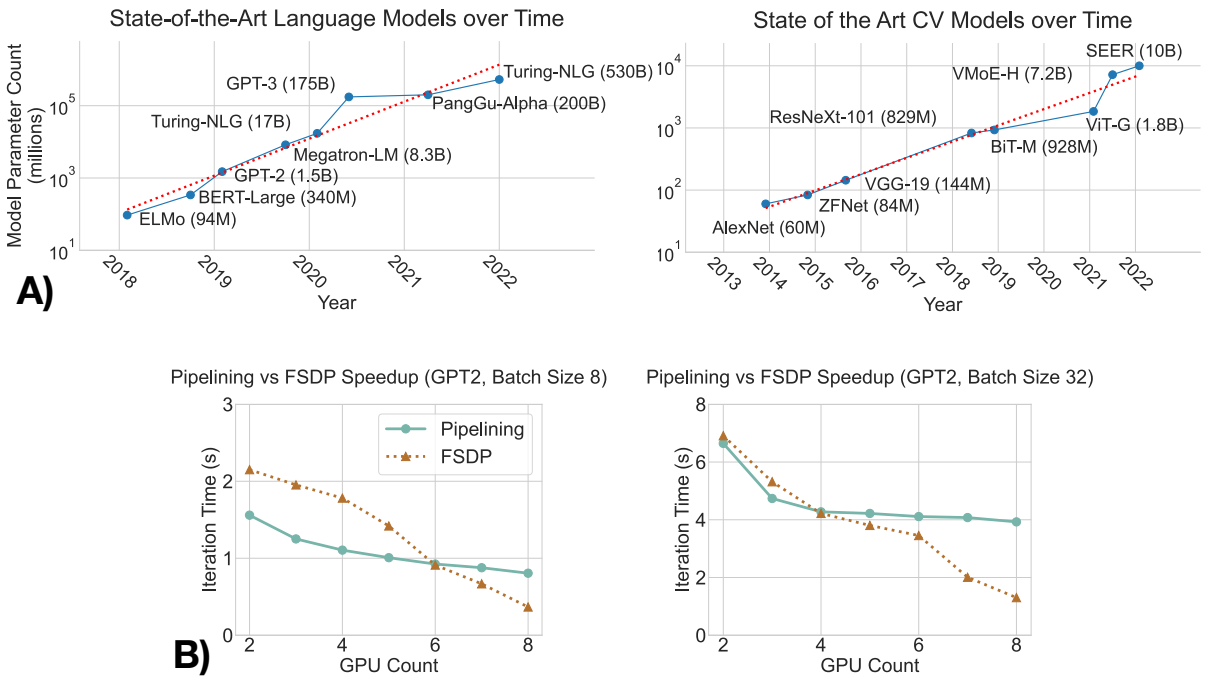

Figure 4.1. (A) Trends of the sizes of some state-of-the-art DL models in NLP and CV (log scale), extrapolated from a similar figure in [190]. (B) Our empirically measured runtime crossovers between FSDP and pipeline parallelism, with knobs tuned per setting.

"microbatch" sizes, while FSDP requires tuning offloading and checkpointing decisions. *Thus, we need to automate parallelism technique selection for large-model DL training.*

(2) How many GPUs to assign to each model? Many DL practitioners use fixed clusters or have bounded resource budgets. So, they are either given (or decide) up front the number of GPUs to use. But in multi-model settings like model selection, there is more flexibility on apportioning GPUs across models. The naive approach of running models one after another using all GPUs is sub-optimal as it *reduces model selection throughput* and adding more GPUs per model yields diminishing returns. Alas, the scaling behaviors of large-model parallelism techniques are not linear and often hard to predict, as Figure 4.1(B) shows. Prior art has studied data-parallel resource allocation (e.g., Pollux [170] and Optimus [163]) and model selection optimization (e.g., Cerebro [100] and ASHA [109]). But none of them target large-model DL, which alters the cost-benefit tradeoffs of GPU apportioning in new ways due to interplay with parallelism selection and complex scaling behaviors. *Thus, we must automate GPU apportioning*

Table 4.1. Overview of prior art. Column desiderata are described in Sections 4.1.1 and 4.1.2.

	Fidelity	Multi-Model	Resource Allocation	Parallelism Selection	Out-of-the-Box Large Model Support
Hybrid Parallelism					
Alpa [241]	✓	✗	✗	✓(limited)	✓
FlexFlow [89]	✓	✗	✗	✓(limited)	✗
Unity [207]	✓	✗	✗	✓(limited)	✓
Performance Evaluation					
Paleo [168]	✓	✗	✗	✓(limited)	✗
Model Selection					
Cerebro [100]	✓	✓	✗	✗	✗
ASHA [109]	✓	✓	✓	✗	✗
Scheduling					
Gandiva [226]	✓	✓	✗	✗	✗
Antman [227]	✓	✓	✗	✗	✗
Tiresias [62]	✓	✓	✗	✗	✗
Resource Allocation					
Pollux [170]	✗	✓	✓	✗	✗
Optimus [163]	✗	✓	✓	✗	✗
SPASE					
SATURN (ours)	✓	✓	✓	✓	✓

for large-model model selection.

(3) How to orchestrate such complex parallel execution for model selection? This is a scheduling question, i.e., deciding which jobs to run when. Two naive approaches are to run models in a random order or to use a generic task scheduler. Both can lead to GPU idling due to

a lack of awareness of how long models actually run. Prior art has studied runtime-aware DL scheduling, e.g., Gandiva [226] and Tiresias [62], but none target large-model DL. The complex interplay of parallelism selection and GPU apportionment can affect runtimes in a way that alters the tradeoffs of scheduling. The model selection setting adds more considerations: we must optimize end-to-end *makespan* rather than just a throughput objective [170, 163]. Specific desiderata must be met: *fidelity* on ML accuracy and *generality* on specification. We expand on these in Section 4.1.2.

Overall, there is a pressing need for a unified and automated way to tackle these 3 systems concerns of model selection on large models: select parallelism technique per model, apportion GPUs per model, and schedule them all on a given cluster. No prior art — including all those described in Table 4.1 — can address this novel setting that has emerged with the rise of large-model DL. We call this new joint problem SPASE*: Select Parallelism, Apportion resources, and SchedulE.*

4.1.2 System Desiderata

To help democratize large-model DL and ease practical adoption, we seek a data system that tackles SPASE with the following desiderata:

(1) Extensibility on parallelism selection. Given the variety of large-model parallelism techniques (henceforth called "parallelisms" for brevity), the system must support and select over multiple parallelisms and also make it easy for users to add new parallelisms in the future (e.g. for model-technique codesign [9, 190, 52]). Without support for user extension, parallelism selectors/hybridizers are limited in scope, as noted in Table 4.1.

(2) Non-disruptive integration with DL tools. The system must natively support popular DL tools such as PyTorch [114] and TensorFlow [8] without modifying their internals. This can offer backward compatibility as those tools evolve.

(3) Generality on multi-model specification. The system should support multiple model selection APIs, e.g., grid/random search or AutoML heuristics. We assume the system is given a

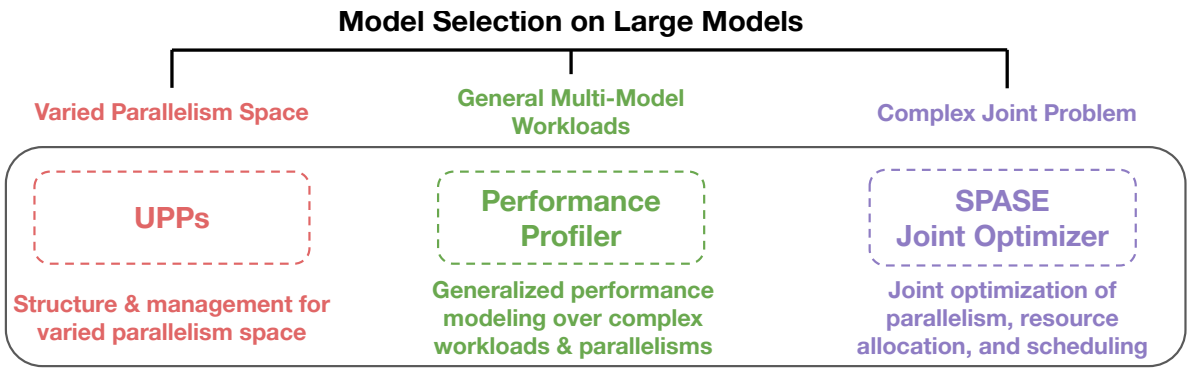

Figure 4.2. Overview of how SATURN's components tackle the SPASE problem for multi-large-model DL workloads.

set of model training jobs with known epoch counts. Evolving workloads can be supported by running all models one epoch at a time.

(4) Fidelity on ML accuracy. The system must not deliberately alter ML accuracy when applying system optimizations. Approximations such as altering the model, training algorithm, or workload parameters are out of scope because they can confound users.

4.1.3 Our Proposed Approach

To meet all of the above desiderata, we design a new information system architecture to tackle SPASE that is inspired by some techniques in database systems. We call our system SATURN. Our current focus is on the common fixed-cluster setting rather than autoscaling [181]. As Figure 4.2 shows, our approach is three-pronged:

(1) Parallelism Selection and UPPs. We translate high-level ("logical") model training specifications into optimized "physical" parallel execution plans based on instance details, inspired by physical operator selection in RDBMSs, e.g., selecting hash-join vs. sort-merge join for a given join operation. To meet the first desideratum of extensibility, we introduce the abstraction of User-Pluggable Parallelisms (UPPs). UPPs can be used to specify existing parallelisms in standard DL tool code, or enable users to add new parallelisms as blackboxes for SATURN to use. This also ensures the second desideratum of non-disruptive integration. We create a default UPP library in SATURN to support 4 major existing parallelisms: pipelining,

spilling, distributed data parallelism (DDP), and FSDP. Each UPP can support knob-autotuning, similar to auto-tuning of physical configuration parameters of a data management system [208, 74].

(2) Performance Profiling. To apportion GPUs and select parallelisms in a way that ensures the fourth desideratum, we need accurate estimates of job runtimes *as is*. We exploit a basic property of SGD: since minibatch size is fixed within an epoch, we can typically project epoch times accurately from runtime averages over a few minibatch iterations. This is similar to prior works (e.g. the Clockwork inference system [63]) that exploit the deterministic and predictable performance behaviors displayed by DNNs to proactively plan out high-quality execution schemes. Coupled with the offline nature of model selection, we can create a general and effective solution: profile all jobs using the full "grid" of options for both GPU counts and parallelisms based on only a few minibatches. The overhead of this approach is affordable due to the long runtimes of actual DL training. This also ensures our second and third desiderata as all DL tools offer data sampling APIs that we can just use on top of the user-given model specifications. Of course, we use the full training data for the actual DL jobs to ensure the fourth desideratum.

(3) Joint Optimization and Scheduling. Given the above system design choices, we can now tackle SPASE using joint optimization. We formalize this problem as a mixed-integer linear program (MILP). Using realistic runtime estimates, we perform a simulation study to compare an MILP solver (we use Gurobi [65]) to a handful of strong scheduling heuristics. The solver yields the best results overall even with a timeout. Thus, we adopt it in SATURN as our SPASE optimizer. Actual model training, not the optimizer, heavily dominates overall runtimes in DL workloads, so we view this design decision as reasonable because it ensures *both efficiency and simplicity*, easing system maintenance and adoption. Finally, we augment our Optimizer with an "introspective" scheduling extension known in prior art to further raise resource utilization.

We intentionally design SATURN to be a simple and intuitive system to tackle SPASE in a way that can help ease practical adoption. Figure 4.3 in Section 3 shows our system architecture. SATURN is implemented in Python and exposes high-level APIs for (offline) specification of UPPs and model selection APIs for actual DL training usage. Under the hood, SATURN has 4 components: Parallelism Plan Enumerator, Performance Profiler, Joint Optimizer, and Executor. The runtime layer builds on top of the APIs of the massively task-parallel execution engine Ray [136] for lower level machine resource management, e.g., placing jobs on GPUs, as well as to parallelize our profiling runs. Using two benchmark large-model workloads from DL practice, we evaluate SATURN against several baselines, including an emulation of current practice of manual decisions on SPASE. SATURN reduces overall runtimes by 39% to 49%, which can yield proportionate cost savings on GPU clusters, especially in the cloud. We perform an ablation study to isolate the impacts of our optimizations. Finally, we evaluate SATURN's sensitivity to the sizes of models, workloads, and nodes.

Novelty & Contributions. To the best of our knowledge, this is the first work to unify these three critical requirements of large-model DL workloads for end users: parallelism selection, resource apportioning, and scheduling. By casting the problem this way, we judiciously synthesize key system design lessons to craft a new information system architecture that can reduce user burden, runtimes, and costs via joint optimization in this important analytics setting. Overall, this paper makes the following contributions:

- We formalize and study the unified SPASE problem, freeing end users of large-model DL from having to manually select and tune parallelisms, apportion GPUs, and schedule multi-jobs.
- We present SATURN, a new information system architecture to tackle SPASE that is also the first to holistically optimize parallelism selection and resource apportioning for multi-large-model DL. SATURN employs a generalized profiler to estimate parallelism runtimes and an MILP solver for joint optimization.

- To enable generalized and extensible support for parallelisms, we create the abstraction of User-Defined-Parallelisms (UPPs). UPPs can be used to specify parallelisms as blackboxes in SATURN.

- We perform an extensive empirical evaluation of SATURN on two benchmark large-model DL workloads. SATURN reduces model selection runtimes by up to 49% in some cases. We make our code publicly available on GitHub [1].

4.2 Background and Preliminaries

We provide a brief background on parallelization techniques to describe the fundamentals relevant to our problem space. For the interested reader, we provide a broader overview of the ML Systems space in the Appendix of our tech report [145].

Multi-GPU parallelism is now common in large-model DL training [86]. Several parallelization schemes already exist, and researchers continue to routinely devise and propose new techniques. A comprehensive review of all such approaches is out of scope for this paper; we refer interested readers to the relevant surveys [141, 192]. Instead, we only highlight a few common approaches here for reference. We also mention the tunable knobs for each parallelism that complicate scaling behaviors and theoretical performance analyses.

Data Parallelism replicates a given DL model across multiple accelerators. Each is fed a different minibatch partition for parallel processing. Replica synchronization can be done in two ways — either via a central parent server, for Parameter Server (PS)-style data parallelism [112, 179], or through peer-to-peer communication, for all-reduce data parallelism [185, 114] with synchronization at SGD boundaries.

Model Parallelism partitions the *model* rather than the data. The model graph is sharded and partitioned over GPUs to distribute the memory footprint. The speedup potential of model parallelism depends on the partitioning scheme and model architecture. Hand-crafted, architecture-

[1] https://github.com/knagrecha/saturn

specific approaches can perform well [1], while simple and generic partitioning schemes tend to be slower [140].

Pipelining [229, 82, 94, 118] & Fully-Sharded Data Parallelism (FSDP) [173, 114] are more advanced hybridizations of model parallelism with data parallelism. Each presents its own tradeoffs and optimization knobs (e.g. "microbatches" for pipelining [118], and "offloading" and "checkpointing" [36] for FSDP). For brevity, we elaborate on the specifics of these techniques in the Appendix of our tech report [145].

Spilling is not a parallelism technique in itself but is often used in combination with a parallelism technique to reduce GPU memory pressure. It swaps model shards between GPU memory and DRAM for piece-wise GPU-accelerated execution [140, 16]. This adds DRAM-GPU communication overheads, but it can enable large models to be trained with even just one GPU. Spilling exposes a *partition count* knob, to select the number of DRAM spills during execution.

Model selection is the process of training and comparing model configurations. Two popular procedures are *grid search*, in which all combinations of sets of values of hyper-parameters (e.g., batch size, learning rate) are used, and *random search* [23], in which random hyper-parameter combinations from given intervals are used. Early stopping can reduce the set of configurations during training [109, 111, 176]. The high resource demands of model selection on large models can sometimes force a DL user to settle for a smaller search space, but this risks missing out on higher accuracy [57, 99]. Faster execution of such workloads empowers users to run larger searches, in turn helping accuracy. Many users expect *fidelity* in this setting, as we explain in Section 4.1.2.

4.3 System Overview

We now describe SATURN's architecture that meets the desiderata in Section 1.2. SATURN has 4 main modules, as Figure 4.3 shows. For workload specification, it exposes a high-level

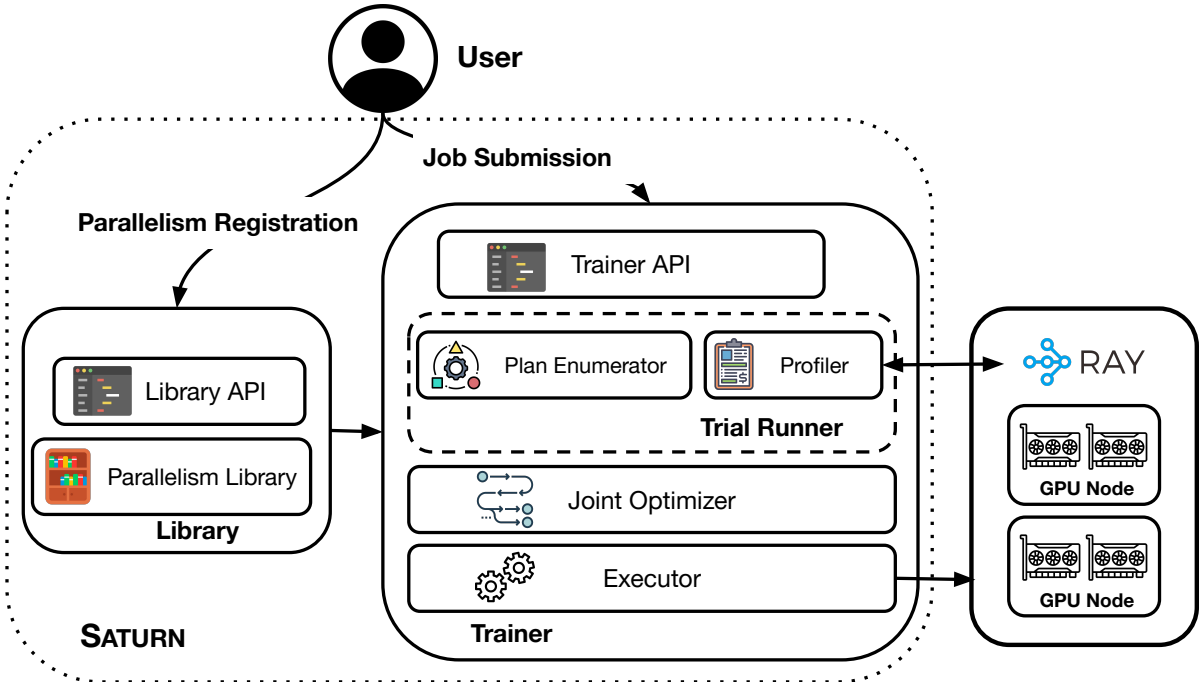

Figure 4.3. System architecture of SATURN and the interactions between the components.

API and the Parallelism Library. The Trial Runner handles runtime estimation. The Joint Optimizer and Executor tackle the SPASE problem. SATURN uses Ray [136]'s low-level APIs as the runtime layer that places jobs on GPUs. Next, we describe each of SATURN's components.

4.3.1 Workload Specification

The first phase, workload specification, is handled by our API and the Parallelism Library component.

API. SATURN's API provides an easy-to-use interface for both registering parallelisms (for developers) and submitting large-model training jobs (for end users of DL). We now provide a brief overview; due to space constraints, we provide the full example pseudocode in the technical report [145]. There are two parts to the API: the Library API and the Trainer API. Users create "Tasks" through the Trainer API by specifying functions for model initialization and data loading, along with any hyper-parameters. This is sufficiently general to cover most model selection workloads. Listing 4.1 illustrates.

```
from saturn.trainer import Task, HParams, execute, profile

t_1=Task(get_model,get_data,HParams(lr=1e-3,epochs=5,optim=SGD))
t_2=Task(get_model,get_data,HParams(lr=3e-3,epochs=5,optim=SGD))
```

Listing 4.1. Specifying tasks through SATURN's API.

Training procedures are defined by "User-Pluggable Parallelisms" (UPPs), which implement the parallel execution approach for SGD. These parallelisms can be registered with our Library by a developer (e.g., ML engineer) or a system-savvy end user of DL. The registration process is shown in Listing 4.2.

```
from saturn.library import register

register("parallelism-a", ParallelismA)
register("parallelism-b", ParallelismB)
```

Listing 4.2. Parallelism registration.

Once all parallelisms and tasks are specified, DL users can invoke the Trial Runner to produce runtime estimates in a single line of code, followed by invoking the whole training execution in another single line of code. Listing 4.3 illustrates these.

```
profile([t_1, t_2, t_3])
execute([t_1, t_2, t_3])
```

Listing 4.3. Profiling and execution invocations.

Parallelism Library. The design of this library is inspired by functional frameworks, user-defined function templates in RDBMSs, and DL model hubs [223]. We follow a define-once, use-anywhere design, wherein registered UPPs can be reused across models, execution sessions, and even different cluster users. This is achieved by managing library-registered parallelisms as a database of code files. The Library allows developers to register new parallelisms by implementing an abstract skeleton, shown in Listing 4.4.

```
class BaseParallelism:
  def search(task:Task,gpus:List[int])->Dict,float:
    pass
  def execute(task:Task,gpus:List[int],knobs:Dict)->None:
    pass
```

Listing 4.4. Parallelism specification skeleton.

The *search* function should use the task and GPUs to provide (1) execution parameters (e.g., microbatch count, partition count) and (2) a runtime estimate. Knob-optimization can also optionally be tackled here. Failed searches (e.g., OOMs) can be handled by returning null values. The *execute* function trains the provided task to completion using the allotted GPUs. It also uses any execution parameters produced during the search phase to optimize execution.

Developers can implement a UPP with standard DL tool code (e.g., TensorFlow or PyTorch) without restrictions. This enables easy integration of pre-existing parallelisms. Indeed, we validate that functionality by adding 4 major parallelisms in our default Parallelism Library: DDP [114], GPipe-style pipeline parallelism [94], FSDP, and model spilling via the FairScale package [16]. These out-of-the-box parallelisms in SATURN are maximally general in that they can be automatically applied to any DL model supported by them. Implementing UPPs for each took $100-250$ lines of Python code. Once defined, UPPs can be registered with the Library under a user-set name (e.g. "pytorch-ddp").

Our design can help developers retain a familiar environment without low-level code changes or extraneous workflows to, say, translate their parallelism implementation into a new configuration file format, a custom domain specific language, etc. Our Parallelism Library serves as an organized roster for registering and using large-model DL parallelisms. While it is a key part of SATURN, it can potentially also be useful as its own standalone tool.

4.3.2 Performance Estimation

The Trial Runner estimates the runtime performance of models with different parallelisms and GPU apportionments. The Trial Runner is *not* a parallelism selector: it simply generates the statistics needed to solve SPASE. It is our empirical substitute for the complex parallelism-

specific theoretical models used in prior art [170, 163]. Such empirical profiling helps "future proof" SATURN to an extent: by not tightly coupling SATURN to specific parallelisms' theoretical models, we can directly support future DL tool compilers and/or accelerator hardware as they evolve. As we highlight in Section 4.1.2, extensibility is one of our key desiderata. The Trial Runner has two submodules: Plan Enumerator and Profiler.

Plan Enumerator. This sub-module constructs a "grid" across all supported parallelisms and GPU apportionment levels for each model. That represents the space of "physical plans" for every model that will then be profiled to obtain runtime performance estimates.

Profiler. This sub-module takes the outputs of the Plan Enumerator to produce runtime estimates for the optimization phase. We exploit a property of SGD: since it is iterative and consistent, we can accurately extrapolate epoch runtimes from averaged performance over a just few minibatches [63]. We use Ray to parallelize these profiling runs and reduce the Profiler's runtime. In our experiments, profiling 12 multi-billion-parameter models for 4 parallelisms took $<$ 30min. This overhead is affordable because the actual DL model selection, on the full training data, can take hours or even days.

4.3.3 Joint Optimizer and Executor

We now use the Trial Runner's statistics to tackle the SPASE problem in a unified manner via holistic optimization.

Joint Optimizer. The Joint Optimizer is invoked transparently when the user invokes the *execute* function. It uses the runtime estimates produced by the Trial Runner and cluster details to produce a full execution plan. This plan bakes in all of *parallelism selection*, *GPU apportionment*, and *schedule construction*. To construct the plan, the Joint Optimizer automatically determines the following for all model configurations given by the user: (1) which parallelism to use, (2) how many GPUs to give it, and (3) when to schedule it.

Our Optimizer is implemented in two layers. First, an MILP solver to produce makespan-optimized execution plans. Second, an introspective, round-based resolver that runs on top of the

MILP solver to support dynamic reallocation. Section 4.4 goes into the technical details of the MILP, why we chose to use an MILP solver instead of heuristics, and additional techniques in the Joint Optimizer.

Executor. This module handles the running of the full execution plan generated by the Joint Optimizer. The Executor runs on top of the lower level APIs of Ray to leverage its task-parallel processing. By default, Ray uses its own task scheduler, and swapping that out for a custom scheduler is challenging. So, for the Executor we implement our plan *over* Ray's scheduler. We achieve this by "tainting" Ray-owned GPUs so that they can only be used by the corresponding jobs from our pre-calculated schedule. Thus, the Executor ensures that Ray's scheduler cannot deviate from our SPASE solution. This scheme lets us faithfully recreate the optimizer-designed plan without overheads or induced inefficiencies, even though the design goes beyond Ray's intended usage.

4.3.4 Current Limitations

SATURN supports both single-node and multi-node training across different models, but in the current version we focus on the case where each model fits in aggregate node memory (i.e., total GPU memory + DRAM). Since we focus on the large-model case, we do not consider GPU multi-tenancy (e.g., as in ModelBatch [156]). We also focus on the homogeneous GPU cluster setting and leave to future work adding support for heterogeneous hardware clusters, hardware type selection, and elastic provisioning (e.g., like in [154, 124]). Anecdotally, we find that many DL users in domain sciences and enterprises do indeed fit this setting. Furthermore, many of the parallelisms in our existing Library do not yet support cross-node training for a single model out-of-the-box. So, we defer support to a future extension as those parallelisms evolve. Despite these assumptions, SATURN can already train 10B+ parameter models on even just one node. These limitations can be mitigated in the future as follows: (1) adjust the MILP in Section 4.4 for hardware selection, (2) give the Trial Runner a larger space to explore, and (3) add multi-node parallelisms to the Library [236].

Two other relevant extensions are support for autoscaling support and elastic changes of jobs mid-execution. An obvious and straightforward way to incorporate these extensions would be to submit workloads to SATURN one-epoch-at-a-time, then induce environment/workload changes at a higher level, in between SATURN's invocations. Future work could look to support more fine-grained integrations, e.g., where SATURN controls the autoscaling decisions. We discuss some possible adaptation points in Section 4.4.4, but leave these extensions to future work.

4.4 SPASE Joint Optimizer

We now describe the SPASE problem and dive into our MILP formalization. Using a simulation study, we evaluate an MILP solver (Gurobi [65]) against baselines and heuristics from standard practice and prior art. We explain our introspective mechanism that enables SATURN to adaptively reassess its MILP solution over time.

4.4.1 Problem Basics

SPASE unifies parallelism selection, resource allocation, and schedule construction. Typical schedulers can set task start times, while resource schedulers can select a GPU apportionment as well. But with SPASE, our joint optimizer must consider a third performance-critical dimension: select the parallelism to use for each model on the allotted GPUs. To the best of our knowledge, ours is the first work to unify and tackle this joint problem.

In model selection workloads, it is common for all jobs to be given up front. So, we focus on that setting. Using the Trial Runner module, we generate the necessary runtime statistics for all given jobs. But even with that information, the joint problem is intractable; prior work on network bandwidth distribution [12] has shown that even the basic resource allocation problem is NP-hard. SPASE is a more complex version of that problem that also handles parallelism selection and makespan-optimized scheduling; so it is also NP-hard. Brute-forcing the search space is also impractical due to its sheer size. The number of schedule orderings alone grows

super-exponentially with the number of jobs [205]. As such, solving it optimally is ruled out. Thus, we choose to formulate SPASE as an MILP and use an industrial-strength MILP solver (Gurobi [65]) to leverage its time-tested optimization power. Later, in Section 4.4.3, we justify this decision further using a simulation study. We find that the MILP solver significantly and consistently outperforms known baselines and strong heuristics despite its time limit. We rely on Gurobi's sophisticated techniques to avoid pitfalls such as poor local optima [66] in this highly non-convex optimization space. Even if the solver does only reach a local optimum, the solution should be of reasonably high quality. We describe and evaluate these risks further towards the end of Section 4.4.3.1 and the Appendix of our tech report [145]. To the best of our knowledge, ours is the first MILP formulation to unify DL parallelism selection, resource allocation, and scheduling. Not only does it enable us to state the problem with mathematical precision, it also enables us to explore the problem space's intricacies via the simulation study.

4.4.2 MILP Formulation

Inputs. Our MILP input consists of a full grid of models, their valid configurations, as well as the corresponding runtime estimates generated by the Trial Runner. Table 4.2 lists our notation, and Figure 4.4(A) illustrates an example. As noted in Section 4.3, our empirical runtime estimates already bake in the communication overheads of each parallelism.

Summary. To summarize the MILP's function in plain-English: we ask the solver to assign to each task: (1) GPU IDs with associated node IDs, (2) an execution configuration (determining the parallelism and resource apportionment), and (3) a float start time. Each task should only be assigned one node and one configuration, and the number of GPUs assigned should agree with the specifications of the chosen configuration. The task should not block *any* GPUs on a node it is not using. The start time for a given task should align all assigned GPUs (i.e., gang scheduling), and the assigned start times should not cause task overlaps on the same GPUs. Ultimately, the solution should minimize the makespan.

Formulation. We now go into each constraint in depth. To make the formulation easier

Table 4.2. MILP Notation used in Section 4.4.2. We provide the symbols as well as their plain-English definitions.

Inputs to the MILP	
Symbol	Description
N	List of nodes available for execution.
T	List of input training tasks.
U	Large integer value used to enforce conditional constraints.
GPU_n	The number of GPUs available on node n.
S_t	Number of configurations available to task t. A configuration consists of both a parallelism and a GPU allocation.
$G_t \in \mathbb{Z}^{+S_t}$	Variable length list of requested GPU counts for each configuration of task t.
$R_t \in \mathbb{R}^{+S_t}$	Variable length list of estimated runtimes for each configuration of task t.
MILP Selected Variables	
Symbol	Description
C	Execution schedule makespan.
$B_t \in 0,1^{S_t}$	Variable length list of binary variables indicating whether task t uses the corresponding configuration from S_t.
$O_{t,n} \in 0,1$	Binary indicator of whether task t ran on node n.
$P_{t,n,g} \in 0,1$	Binary indicator of whether task t ran on GPU g of node n.
$A_{t1,t2} \in 0,1$	Binary indicator of whether task $t1$ ran before task $t2$. If $A_{t1,t2}$ is 1, $t2$ must have run after $t1$.
$I_{t,n,g} \in \mathbb{R}^+$	Start time of task t on GPU g of node n.

to comprehend, we illustrate our constraints using a running example workload in Figure 4.4. The figures are purely demonstrative, and do not represent a realistic model selection job — which may be considerably larger and more complex.

$$\texttt{Objective:} \quad \min_{B,O,P,A,I} C \tag{4.1}$$

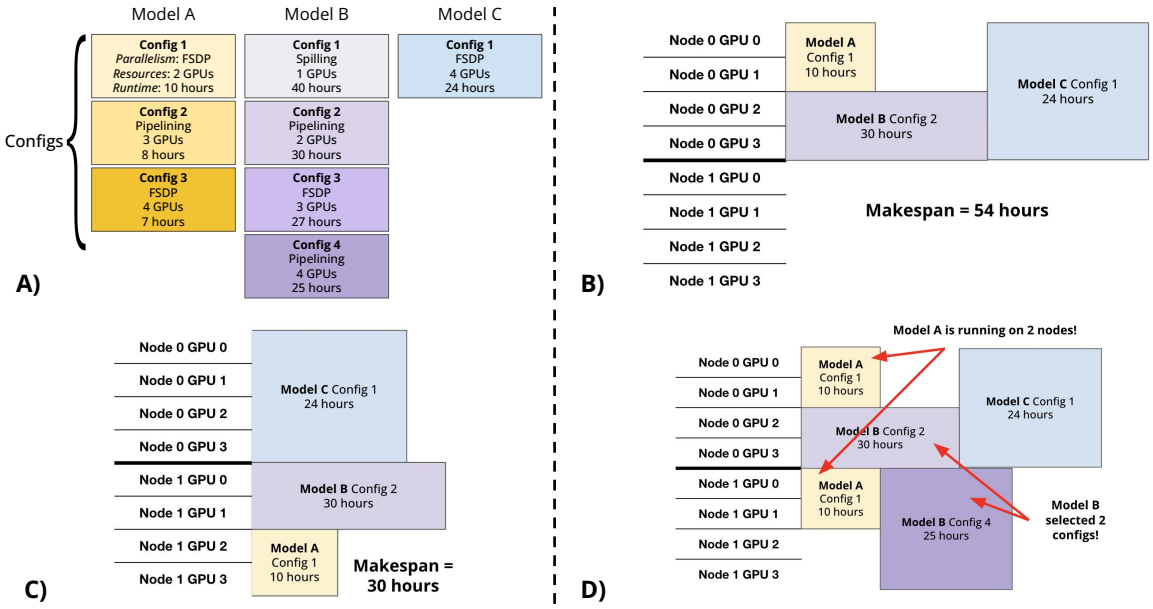

Figure 4.4. (A) depicts the configs (i.e., variables G & R) used throughout our examples; (B) illustrates a feasible but suboptimal SPASE solution and the corresponding makespan; (C) illustrates an optimal SPASE solution; (D) illustrates violations of the constraints in Equation 4.3.

We now define the constraints. Equation 4.2 defines the makespan; it is the latest task's start time plus the runtime of that task's selected configuration. Figure 4.4(B) & (C) illustrate some example SPASE solutions and their corresponding makespans.

$$C \geq I_{t,n,g} + R_{t,s} - U \times (1 - B_{t,s})$$
$$\forall s \in S_t \forall t \in T, \forall n \in N, \forall g \in G \tag{4.2}$$

Next, for each task, there should only be one selected configuration and only one selected node. Figure 4.4(D) illustrates this constraint.

$$\sum_{x \in B_t} x = 1; \sum_{y \in O_t} y = 1 \tag{4.3}$$

Next, we enforce the GPU requests of the solver onto the execution schedule. Each task must be assigned the number of GPUs corresponding to its selected configuration. Since direct equality comparisons are not possible in an MILP formulation, Equations 4.4 and 4.5 in combination ensure this constraint by enforcing both $\leq$ and $\geq$ inequalities. Figure 4.5 illustrates.

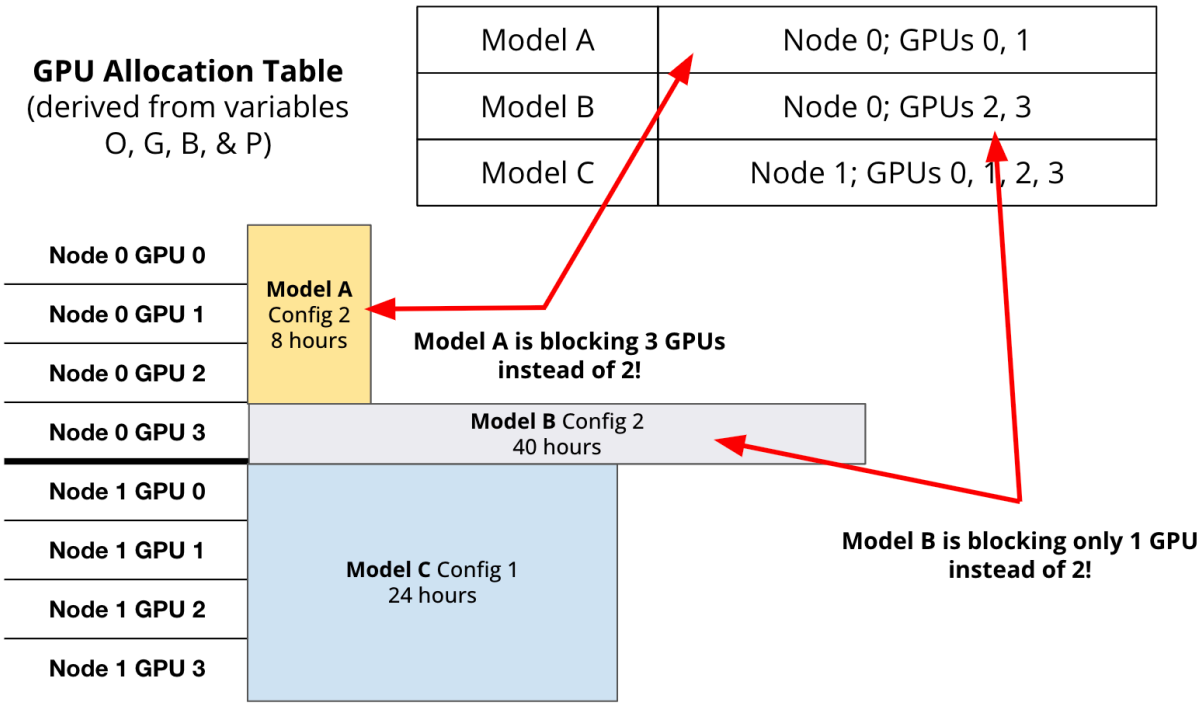

Figure 4.5. Illustration of a SPASE solution where tasks select too few or too many GPUs, violating constraints 4 & 5.

$$\sum_{t \in P_{t,n}} t \geq G_{t,s} - U \times (2 - O_{t,n} - B_{t,s}) \forall s \in S_t, \forall t \in T, \forall n \in N \tag{4.4}$$

$$\sum_{t \in P_{t,n}} t \leq G_{t,s} + U \times (2 - O_{t,n} - B_{t,s}) \forall s \in S_t, \forall t \in T, \forall n \in N \tag{4.5}$$

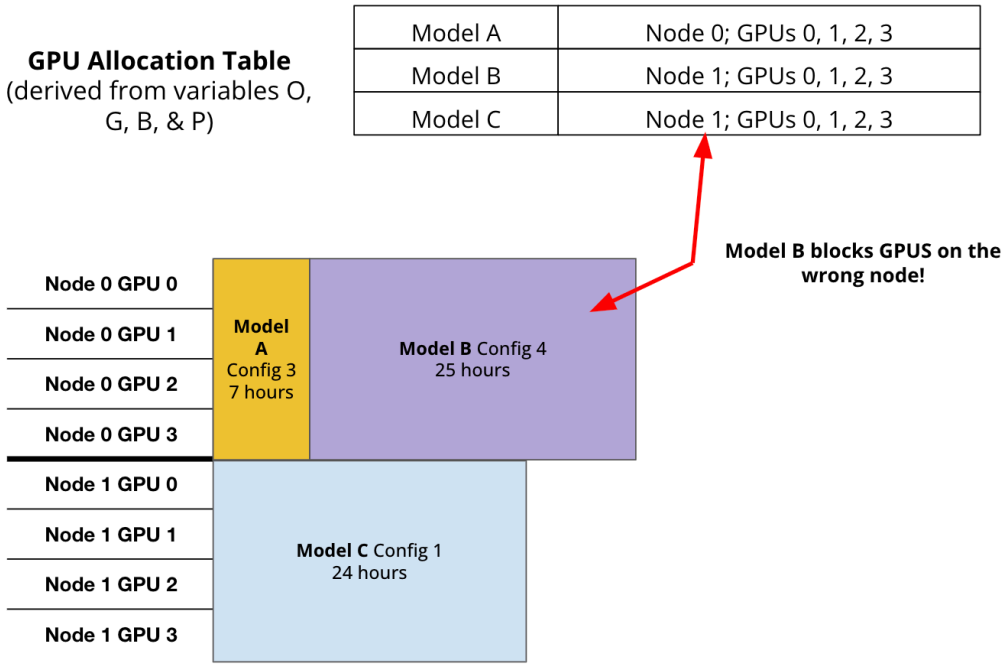

Figure 4.6. Illustration of a SPASE solution where a task blocks GPUs on a node it has not selected, violating constraints 6 & 7.

We must also ensure that the task uses *0* GPUs on any nodes it is not executing on. Equations 4.6 and 4.7 combine $\leq$ and $\geq$ inequalities to enforce this requirement. Figure 4.6 illustrates.

$$\sum_{t \in P_{t,n}} t \leq 0 - U \times (O_{t,n} + B_{t,s}) \forall s \in S_t, \forall t \in T, \forall n \in N \tag{4.6}$$

$$\sum_{t \in P_{t,n}} t \geq 0 + U \times (O_{t,n} + B_{t,s}) \forall s \in S_t, \forall t \in T, \forall n \in N \tag{4.7}$$

Next we apply a *gang scheduling* constraint, i.e. for each task, all assigned GPUs must initiate processing simultaneously. Formulating this constraint is challenging — we need consistency over a set of MILP-selected values, on a set of MILP-selected indices, across an MILP-selected gang size. Our solution is to take a fixed start-time target — the sum of MILP-selected start times over *all GPUs*, divided by the number of allocated GPUs. By ensuring each selected time is thus equal to the *average* of the times, the times must by definition be equal to one another. This constraint also naturally encourages the solver to fix start times on unused GPUs to 0 without explicit enforcement, since non-zero values bloat the numerator of the left hand side. Equations 4.8 and 4.9 in combination enforce this constraint. Figure 4.7 illustrates.

$$\begin{aligned} \frac{\sum_{x \in I_{t,n}} x}{G_{t,s}} &\leq I_{t,n,g} + U \times (3 - P_{t,n,g} - B_{t,s} - O_{t,n}) \\ &\forall s \in S_t, \forall t \in T, \forall g \in GPU_n, \forall n \in N \end{aligned} \tag{4.8}$$

$$\begin{aligned} \frac{\sum_{x \in I_{t,n}} x}{G_{t,s}} &\geq I_{t,n,g} - U \times (3 - P_{t,n,g} - B_{t,s} - O_{t,n}) \\ &\forall s \in S_t, \forall t \in T, \forall g \in GPU_n, \forall n \in N \end{aligned} \tag{4.9}$$

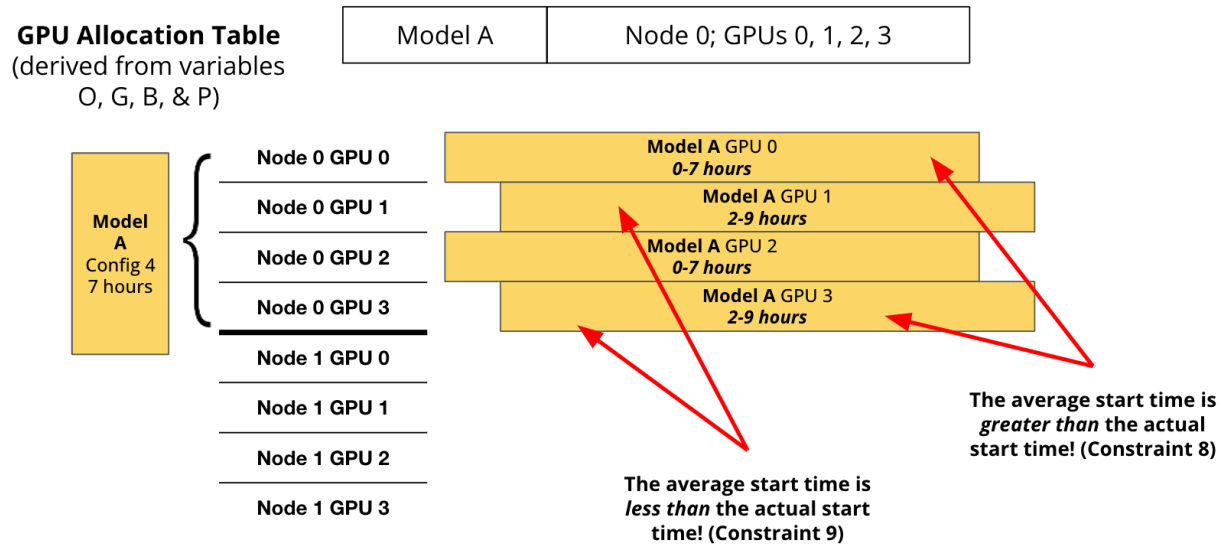

Figure 4.7. Illustration of a specific model violating gang scheduling requirements, thus breaking Constraints 8 & 9.

Finally, we encode a task isolation constraint, so that no tasks overlap on the same GPU. Equation 4.10 applies if task $t1$ came before task $t2$, while equation 4.11 guarantees no overlap if task $t1$ came after task $t2$. Variable A acts as a before-or-after selector, determining which constraint is relevant for each pair of tasks. Figure 4.8 illustrates.

$$I_{t1,n,g} \leq I_{t2,n,g} - R_{t,s} + U \times ((3 - P_{t1,n,g} - P_{t2,n,g}) - B_{t,s} + A_{t2,t1}) \\ \forall s \in S_t, \forall t1 \in T, \forall t2 \in (T - \{t1\}), \forall g \in GPU_n, \forall n \in N \tag{4.10}$$

$$I_{t1,n,g} \geq I_{t2,n,g} + R_{t,s} - U \times ((4 - P_{t1,n,g} - P_{t2,n,g}) - A_{t2,t1} - B_{t,s}) \\ \forall s \in S_t, \forall t1 \in T, \forall t2 \in (T - \{t1\}), \forall g \in GPU_n, \forall n \in N \tag{4.11}$$

Schedule & Allocation Table
(derived from variables O, G, B, & P, BoA)

Model A	Node 0; GPUs 0, 1, 2, 3	**Before B**
Model B	Node 0; GPUs 0, 1, 2, 3	**After A**
Model C	Node 1; GPUs 0, 1, 2, 3	

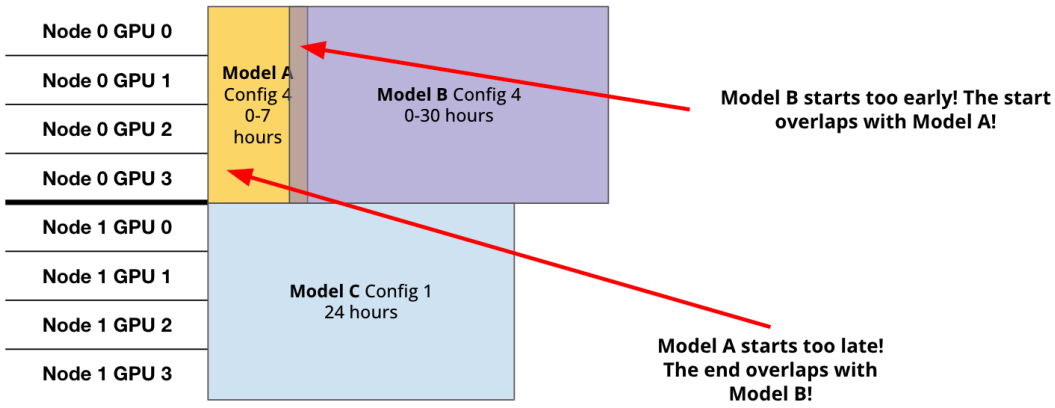

Figure 4.8. Illustration of a SPASE execution plan violating task isolation requirements, thus breaking Constraints 10 & 11.

This MILP formulation is complex because it spans and unifies three different system decisions in our setting. Our Joint Optimizer constructs all the constraints automatically for a given instance and provides them to Gurobi [65]. We use the PuLP interface for Gurobi to keep all variables within a single Python process space.

4.4.3 Simulation-based Comparisons

We now evaluate our MILP-solver approach. We begin by discussing baselines from current practice and heuristics in prior art. Then, we run evaluations on simulated workloads and find that the MILP-solver outperforms the other approaches by a significant margin.

4.4.3.1 Baselines

As the case study in Section 7.1 highlighted, large-model users must currently tackle the SPASE problem manually. So we can define the initial baseline based on current best practices. A common heuristic is to just maximize each task's allocation. Each task is given all GPUs in a

node; then the best parallelism for that particular setting is applied. The models are run one after another. This optimizes *local* efficiency and maximizes available GPU memory for each task. This heuristic becomes a suboptimal degenerate case of the apportioning and scheduling parts of the SPASE problem. We call this baseline "Max-Heuristic", and anecdotally we find this is common in current practice.

The opposite extreme would be to minimize the number of GPUs assigned to each task to maximize task-parallelism [140]. We call this baseline "Min-Heuristic." While it runs many models in parallel, this approach suffers a lot of DRAM spilling for large models.

Finally, we devise a strong algorithmic heuristic that incorporates our runtime estimates to produce non-trivial solutions. It extends an idea from Optimus, a DL resource scheduler in prior art that proposes a greedy resource allocator that uses an "oracle" to provide runtime estimates [163, 170]. Optimus iteratively assigns GPUs to whichever model that will see the greatest immediate benefit. The original Optimus implementation used a throughput-prediction oracle for PS-style data parallelism, but subsequent works [170] have made it standard to provide an alternate oracle to adapt Optimus for different parallelisms. Our Trial Runner statistics serve as our oracle, thus allowing us to manually configure optimal parallelism selections for Optimus' benefit. Since this is not part of the base offerings of Optimus, we denote this strengthened modification of Optimus as Optimus*. Optimus* serves as a strong baseline SPASE solver, tackling problems of resource allocation and model selection natively, and parallelism selection through our augmentation. SATURN's main advantage over this baseline is its use of *joint* optimization. For our simulation study, we call this baseline algorithm Optimus*-Greedy. Algorithm 3 presents its pseudocode, reusing variables from Table 4.2.

The Optimus*-Greedy algorithm yields resource allocations per task. We transform that into a SPASE solution by selecting the best parallelism for each task's allocation post-hoc. In the multi-node case, we run this algorithm one node at a time. Like many iterative greedy algorithms, this approach relies on consistent scaling behaviors. It has only a local greedy view, rather creating a one-shot global resource distribution.

Algorithm 3: OPTIMUS*-GREEDY(Tasks T, GPUs G)

$L = [1|t \in T]$
while sum(L) < G **do**
  $CR = [R_{t,s}|t, l \in (T, L), s \in S_t \text{ where } G_{s,t} == l]$
  $PR = [R_{t,s}|t, l \in (T, L), s \in S_t \text{ where } G_{s,t} == l + 1]$
  $GAIN = [c - p|c, p \in (CR, PR)]$
  $L[ArgMax(GAIN)] + +$
end while
return L

Apart from the above three approaches to cover standard practice and prior art extensions, we also include a simple randomization-based baseline. In summary, we compare with 4 approaches:

1. **Max-heuristic:** All GPUs within a node are given to one task at a time.

2. **Min-heuristic:** A single-GPU technique (spilling) is given to each task to maximize task parallelism. If additional GPUs are available, they are divided evenly.

3. **Optimus*-Greedy:** A greedy algorithm inspired by the one used in the Optimus [163] resource scheduling paper.

4. **Randomized:** Parallelisms and allocations are randomly selected for every task, then tasks are randomly scheduled.

For each of the above approaches, we use our Profiler results to select the best possible parallelism+allocation for each model. For instance, if a baseline determines that Model A should receive 8 GPUs, we refer to the Profiler to determine which parallelism gives Model A the best runtime at 8 GPUs. This same best-check procedure is used to determine the gain values for Optimus*-Greedy.

Since our MILP is complex, Gurobi is unlikely to converge to an optimal solution in a practical timeframe. Thus, we set a reasonable timeout — from our trials [145], we set it to 5mins — for the solver to produce a solution. We rely on Gurobi's industry-strength techniques

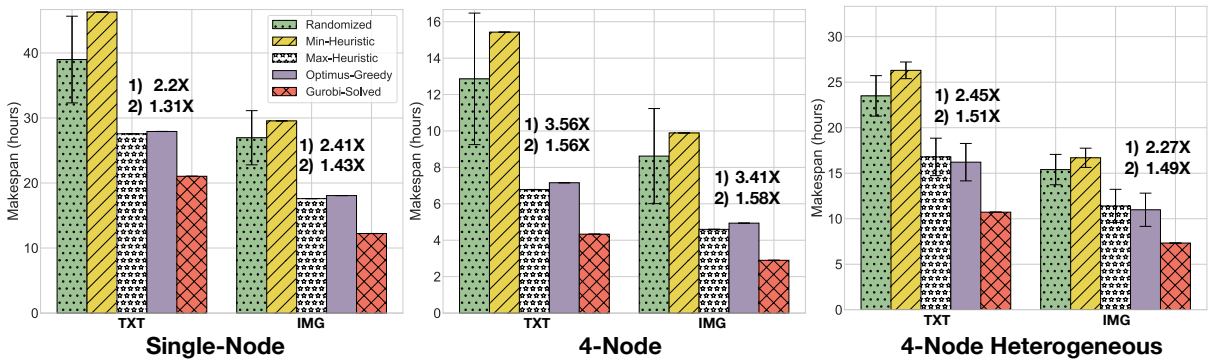

Figure 4.9. Simulation results comparing our MILP to two key baselines. For each group, we list SATURN's speedup versus (1) the weakest and (2) the second-best performer.

to find a high-quality (though possibly suboptimal) solution even within the allotted time. The Appendix of our tech report [145] shows the diminishing returns of having a larger timeout. We leave it to future work to adapt the timeout for the given workload.

4.4.3.2 Simulation Workloads

We simulate 2 benchmark workloads, described in Table 4.3. Runtime estimates for all models and configs are produced by the Trial Runner beforehand. We simulate 3 hardware settings: an 8-GPU single node, 32-GPUs over 4-nodes, and 4 heterogeneous nodes with GPU counts of 2, 2, 4, and 8 (16 GPUs in total). To adapt the baselines for the heterogeneous setting, we distribute models across nodes randomly, weighting each node's probability by its GPU count. Figure 4.9 presents the simulation results. All approaches are run 3 times and averaged, with 90% confidence intervals displayed; but only the randomized algorithm shows significant non-determinism on the homogeneous node settings. In all cases, the MILP-solver approach yields significantly better solutions than the baselines. We achieve a makespan reduction of up to 59% over the Min-Heuristic, 36% over the Max-Heuristic, 54% over Randomized, and 33% over Optimus*-Greedy. In the heterogeneous setting, the improvements are slightly lower, ranging from 18% to 42%. We attribute this to the small 2-GPU nodes, which provide less flexibility for resource apportioning or parallelism selection, thus reducing the candidate solution space. Overall, SATURN's Gurobi-solved approach consistently outperforms the alternatives.

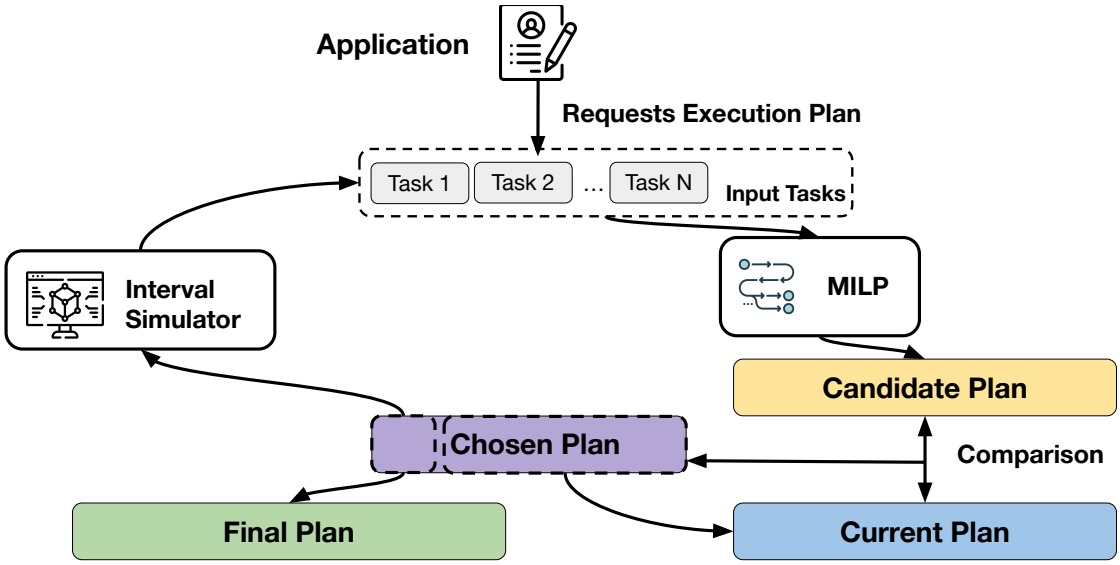

Figure 4.10. Depiction of the introspective feedback loop.

The MILP-solved approach has the highest overhead; a 5min timeout versus < 10 seconds for the baselines. But the overhead is negligible given the typical scale of the makespans.

4.4.4 Introspection

In general, one-shot up-front scheduling is suboptimal. Workloads can evolve over time, either due to online changes (e.g., an AutoML heuristic killing or adding models to train) or ongoing execution (task runtime reduce as they are trained). If the optimizer can be rerun partway through execution, it might produce a different, more performant, solution for the remainder of the workload. To achieve this, we propose the use of *introspection* [226].

A key feature in some state-of-the-art DL schedulers [226], introspection proposes that a scheduler should "learn" as it executes. There are two ways in which a schedule might be altered or adapted via introspection. First is pre-emption. Rather than blocking a GPU for a full job lifecycle, jobs can be swapped to different GPUs or paused temporarily. This enables fine-grained schedule construction and increased optimization flexibility. Second is dynamic rescaling. The initial up-front training plans could be adjusted (e.g., 6 GPUs down to 2) partway through a schedule. In SPASE, this can also involve changing the parallelism.

We now describe how we implement introspection in SATURN. Figure 4.10 illustrates our design. We treat our SPASE MILP solver as a blackbox sub-system. At periodic intervals (e.g., every 1000 seconds), we re-evaluate the underlying workload. The partial training over the previous interval may have modified the set of models. We *rerun* the solver on the interval boundaries so that it can introspectively adjust its original solution. By treating each interval-defined segment of training as effectively independent, we preserve gang scheduling semantics *within* each segment, while allowing for graceful exits and relaunches across intervals. Such sequences of independent segments are possible due to the iterative nature of SGD, as well as the ease of checkpointing models during training [170]. Global batch size consistency is respected by adjusting per-device batch sizes to account for new allocations. Since we focus on model selection with the fidelity desideratum, we cannot modify the user-configured batch size transparently. Algorithm 4 provides pseudocode for the basic algorithm. Essentially, we re-trigger the solver on a fixed interval and determine if the new solution improves performance versus just continuing with the existing plan. If the new plan is superior, we checkpoint all active jobs and re-launch with the new plan.

Algorithm 4: ROUND INTROSPECTION(Workload W, Interval I)

Schedule$S = MILP(W)$
$M = Makespan(S)$
E2ESchedule $= S[0:I]$
T $= 500$
while W not exhausted **do**
  $W = W$ after I seconds of S
  $S = S[I:]$
  $M = M - I$
  Proposal $= MILP(W)$
  if Makespan(Proposal) $\leq$ M - T **then**
    $S =$ Proposal
    $M =$ Makespan(Proposal)
  end if
  $E2ESchedule.append(S[0:I])$
end while
return L

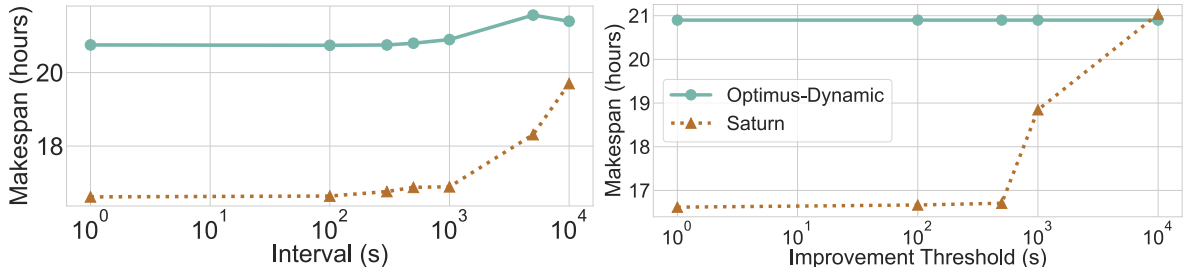

Figure 4.11. Sensitivity plots for SATURN and Optimus*-Dynamic for interval and threshold knobs. We fix the interval to 1000s for the first analysis and the threshold to 500s for the second.

To demonstrate the impact of SATURN's introspection, we compare with a new dynamic baseline, "Optimus*-Dynamic", by swapping the MILP-solver for the Optimus*-Greedy algorithm. Figure 4.11 shows the impact of the interval length and the improvement threshold knob. Since each round produces a holistically optimized solution, SATURN's performance improves monotonically (not accounting for pre-emption costs) as knobs become more fine-grained. Lower interval/threshold levels naturally subsume higher levels in this scheme. In contrast, locally-optimizing algorithms such the Optimus*-Dynamic approach have non-monotonic behaviors.

Introspection does not have to occur on interval completion; we can simulate the next-interval state based on the current solution. Then, the solving process for the next introspection round can be overlapped with execution of the current round to hide the latency of introspection. This scheme provides speedups of 15% to 20% over our one-shot MILP, as shown in Section 4.5.3. With introspection plus our MILP-solver, SATURN's Joint Optimizer is 1.5x-4.1x faster than the heuristics described in Section 4.4.3. Our introspection optimization significantly improves offline execution, but it also naturally supports online AutoML optimizations such as early-stopping [111, 109] or new job arrivals in a multi-tenant cluster through workload reassessment. We do not explicitly optimize for AutoML heuristics in the current version of SATURN; but it is easy to extend it to exploit this optimization.

We use a tolerance level, T, to describe the minimum acceptable benefit of an introspective plan switch. If the swap only provided a 5 second benefit, for example, the switching

Table 4.3. Model selection configurations of workloads.

		Workload	
		TXT	IMG
Model Selection Configuration	Model Arch. (params)	GPT-2 (1.5B), GPT-J (6B)	ViT-G (1.8B), ResNet (200M)
	Dataset	WikiText-2	ImageNet
	Batch Size	{16, 32}	{64, 128}
	Learning Rate	{1e-5, 1e-4, 3e-3}	{1e-5, 1e-4, 3e-3}
	Epochs	10	10
	# Models	12	12

overheads alone might outweigh the makespan reduction.

Our introspection optimization takes inspiration from prior art in DL cluster scheduling, e.g., Antman [227] and Gandiva [226] which demonstrated the value of pre-emption on minibatch boundaries, as well as Pollux and Optimus [170, 163], which showed the value of dynamic rescaling. Our contribution is in unifying both of those optimization ideas to craft our introspection technique, which also incorporates change-of-parallelism across introspection rounds.

4.5 Experimental Evaluation

We now run an extensive empirical evaluation. We aim to answer two questions: (1) What performance benefits does SATURN provide compared to current practice? (2) How much do each of SATURN's optimizations contribute to the overall speedups?

Workloads, Datasets, and Model Configurations: We run 2 model selection workloads with benchmark DL tasks. Table 4.3 lists the model selection configurations for both workloads. The first (TXT) is a text workload with LLMs. It uses the popular *WikiText-2* [133] dataset. WikiText-2, which is drawn from Wikipedia, has previously been used as a benchmark on landmark LLMs such as GPT-2 [171]. TXT uses two GPT models: GPT-2 (1.5B parameters),

introduced in 2019, and GPT-J (6B parameters), introduced in 2021. Both are still considered state-of-the-art for application-specific finetuning purposes[2]. The second (`IMG`) is image classification comparing a large ResNet (200M parameters) and a large-scale Vision Transformer (1.8B parameters). It uses the computer vision benchmark dataset *ImageNet* [45] (14M images and 1000 classes). `IMG`'s mix of model classes should help provide a more challenging workload to optimize.

Software Setup: All models are implemented and trained with PyTorch 2.0. We register 4 parallelisms in SATURN.

1. PyTorch Distributed Data Parallelism [114].
2. PyTorch Fully-Sharded Data Parallelism [114].
3. GPipe, adapted from an open-source implementation [94].
4. Model spilling, provided by the FairScale library [16].

We use Gurobi 10.0 for our SPASE MILP-solver; the introspection threshold and interval parameters are set to 500s and 1000s, respectively. For the underlying job orchestration, we use Ray v2.2.0. Datasets are copied across nodes upfront.

Hardware Setup: We configure 3 hardware settings: (1) 8-GPU single-node, (2) 16-GPU 2-nodes, and (3) heterogeneous 2-nodes, where one node has 8 GPUs and the other has 4 (12 GPUs in total). All settings use AWS p4d instances.

Baselines: No prior end-to-end system can solve the SPASE problem; prior art either does not support large models or else fails model selection constraints, as Table 4.1 showed. So, we compare SATURN with 4 baselines using the approaches in Section 4.4.3.

[2]Extreme-scale LLMs such as GPT-3 or BLOOM are too large (175B+ parameters) for our current scope because they need hundreds of GPUs for reasonable runtimes, which we are unable to afford. They are likely also infeasible and/or an overkill for most end users of DL, especially in domain sciences, small companies, etc., who are our main target. We leave it to future work to extend SATURN to such extreme-scale models.

(1) **Current Practice:** A heuristic without any task parallelism within nodes. It allocates 8 GPUs per task. Parallelism selection is set by a human to "optimal" choices for an 8-GPU allocation, (typically FSDP). This is perhaps most representative of current practice by end users of DL.

(2) **Random:** A randomizer tool selects parallelism and apportioning and then applies a random scheduler. This represents a system-agnostic user.

(3&4) Two modified versions of Optimus*-Greedy (Alg. 1) combined with a randomized scheduler (see Section 4.4). We name these baselines **Optimus*-Dynamic** and **Optimus*-Static**. These are the strongest baselines for large-model model-selection we could assemble from prior art.

The above approaches cover both current practices and reasonable strong heuristics for our problem setting. We note that the two Optimus*-based baselines use our Trial Runner as an oracle for their runtime estimates and parallelism selection decisions (the original Optimus paper only had runtime models for Parameter Server-style data parallelism [163]). This highlights the novelty of our problem setting — the strongest baseline from prior art needs to reuse a module of our system.

4.5.1 End-to-End Results

Model Selection Runtimes: We first compare the end-to-end runtimes versus the 4 baselines. The Trial Runner search overheads are *included* in SATURN's runtime. Figure 4.12(A) presents the results.

SATURN achieves significant speedups versus all baselines. Against Current Practice, we see makespan reductions of 39-40% on a single-node, 43-48% on 2 homogeneous nodes, and 41-45% on 2 heterogeneous nodes. Against the strongest baseline (Optimus*-Dynamic), we see makespan reductions of 30-34%, 38-40%, and 32-39% on the three hardware configurations respectively. Since the same UPP implementations are used *in all cases*, the speedups are achieved

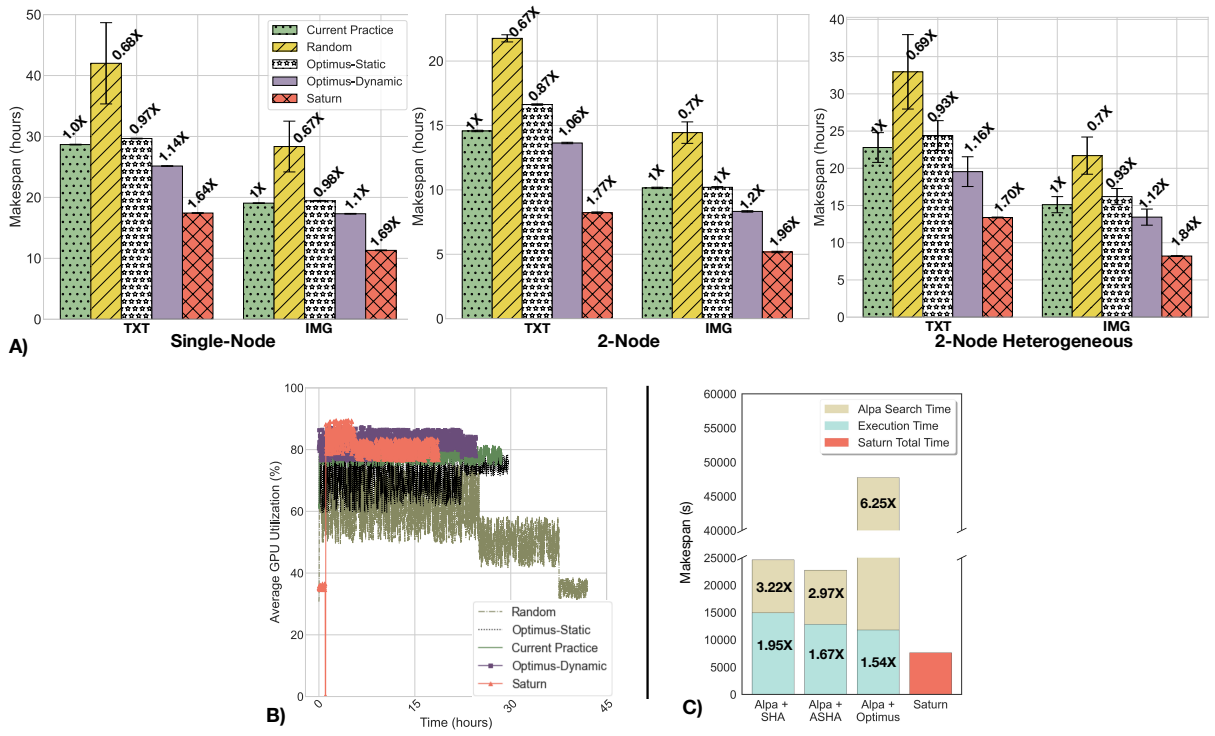

Figure 4.12. (A) End-to-end runtimes. Speedups versus current practice are also noted. Results are averaged over three trials, with the 90% confidence interval displayed. (B) Average GPU utilization over time at a 100s sampling rate on the single-node `TXT` job. (C) End-to-end runtimes of SATURN versus compositions of tools on a reduced version of the `TXT` job on 2 nodes.

purely via the better parallelism selections, resource allocations, and schedule constructions. All compared approaches (including SATURN) use logically equivalent SGD and offer the same accuracy.

Figure 4.12(B) plots GPU utilization. SATURN achieves good utilization throughout, except an initial low-utilization period for the Trial Runner's search and MILP solving period. GPU utilization alone can be misleading; tools such as nvidia-smi can artificially inflate utilization [5]. So, these results should not be taken as a measure of training performance in isolation.

Overall, SATURN reduces model selection runtimes substantially for all workloads in all evaluated settings. It also offers more qualitative benefits to end users of DL because they are freed from manually selecting parallelisms, deciding on resource allocations, or tuning system

parameters.

Intuition on Efficiency Gains. SATURN's performance improvements arise due to its *holistic* optimization approach. To the best of our knowledge, this is the first work that characterizes the parallelism performance crossovers and incorporates them into a joint optimizer. Our empirical profiler and unified SPASE formulation enable us to optimize in a parallelism-agnostic fashion. The heuristic and algorithmic baselines make assumptions about scaling behaviors (e.g., consistency, linear scaling, etc.) that do not always hold up in large-model DL practice. To prove our point further, Table 4.4 lists the parallelisms+allocations selected by SATURN for a few models from the single-node workloads. We see a non-trivial mixture of decisions across the models trained.

Table 4.4. Parallelisms and apportionments chosen by SATURN for a few evaluated models.

Model Config	Parallelism	Apportionment
GPT-2 (Batch 16, 1e-5 LR)	Pipelining	5 GPUs
GPT-2 (Batch 32, 1e-4 LR)	FSDP	4 GPUs
GPT-J (Batch 16, 1e-5 LR)	FSDP	8 GPUs
GPT-J (Batch 32, 1e-4 LR)	Pipelining	3 GPUs
ResNet (Batch 64, 1e-4 LR)	DDP	2 GPUs
ResNet (Batch 32, 1e-4 LR)	Spilling	1 GPU
ViT-G (Batch 32, 1e-4 LR)	FSDP	4 GPUs
ViT-G (Batch 16, 1e-4 LR)	FSDP	6 GPUs

SATURN's MILP-chosen SPASE solutions combine into a multi-model SPASE solution to minimize end-to-end runtimes. Our unified data systems-style approach frees DL users to focus on their goals instead of tedious low-level decisions.

4.5.2 Joint Optimization Evaluation

To better understand the value of joint optimization for SPASE, we evaluate SATURN against different compositions of tools — Alpa [241] + ASHA [109]; Alpa + Optimus* [163]; Alpa + SHA [111] — each used together but unaware of each other. A/SHA

& Optimus are designed for multi-model training and GPU allocation; Alpa tackles parallelism selection. In combination, they can be used to solve the dimensions of the SPASE problem, but in a separated fashion. We elaborate on these tools in Section 8.2.

To mimic A/SHA's early-stopping behaviors, we run SATURN and Optimus* one epoch at a time. We take the early stops produced by A/SHA and apply them to SATURN and Optimus*'s workloads on epoch boundaries. A/SHA is configured to use 3 rungs, with allocations of 1, 3, and 6 epochs respectively, so completed jobs will have run for 10 epochs. The decay factor is set to 2, so half of the jobs survive each rung. Since A/SHA was built for settings with substantially more accelerators than models, we use a smaller version of the `TXT` workload with 8 jobs (eliminating the 3e-3 learning rate option) on 2 nodes.

We report the results in Figure 4.12(C). We find that SATURN outperforms Alpa + ASHA by nearly 3X. Even if we remove Alpa's search times (e.g., if the searches were run once up-front) and directly compare SPASE solution quality, SATURN still outperforms the composite baseline by 1.67X. Against Alpa + Optimus*, the speedups are 6.25X (resp. 1.54X) when including (resp. excluding) Alpa's compilation times. The Optimus* runtime that includes the compilation times is so high because it needs to construct its throughput oracle [170] up front by running the compiler for every possible allocation for every model. SATURN's significant speedups against all 3 baselines support our view that the SPASE problem is a novel space where joint optimization has a significant role to play, rather than a simple composition of existing problem spaces.

4.5.3 Drilldown Analyses

4.5.3.1 Ablation Study

We separate our optimizations into 4 layers: scheduling, resource allocation, parallelism selection, and introspection. We apply these one-by-one as follows. First, a version without any of our optimizations. FSDP is used with checkpointing and offloading (i.e., a non-expert config), resource allocations are set manually to 4 GPUs per task, and a random scheduler is used.

Second, we use our makespan-optimized scheduler. Third, we reintroduce resource apportioning to the MILP. Fourth, we allow for automatic parallelism selection and knob tuning. Finally, we overlay introspection. This completes SATURN. We use the single-node TXT workload in our study. Table 4.5 notes the marginal speedups.

Table 4.5. Ablation study, showing how SATURN's performance changes as new optimizations are added.

Optimizations	Abs. Speedup	Extra Speedup
Unoptimized	1.0X	1.0X
+ MILP Scheduler	1.1X	1.1X
+ Resource Allocation in MILP	1.33X	1.2X
+ Auto. Parallelism Selection	1.95X	1.47X
+ Introspection	2.27X	1.16X

The scheduler-only MILP provides better packing for some initial makespan improvements. Adding in resource apportioning lets the solver reshape task runtimes and demands to produce more speedups. Automatic parallelism selection creates even more flexibility and adds in knob-tuning to improve parallelism performance. Introspection enables the solver to reassess its solution and adapt to shifts in the workload to cap off SATURN's speedups.

4.5.3.2 Sensitivity Analyses

We test SATURN's sensitivity to the size of: (1) workloads, (2) models, and (3) clusters.

For workload size scaling, we run TXT on a single 8-GPU node with the GPT-2 model, set batch size to 16, and vary the number of learning rates explored. Figure 4.13(A) presents the results. SATURN scales slightly superlinearly as larger workloads enable broader scope for optimization. This suggests strong performance on large-scale model selection workloads.

Next, we vary model size. We run TXT on a single 8-GPU node with batch size set to 16 and learning rate set to 1e-5. All models are versions of GPT-2. We vary model size by stacking encoder blocks, akin to what GPT-3 does [30]. Figure 4.13(B) presents the results.

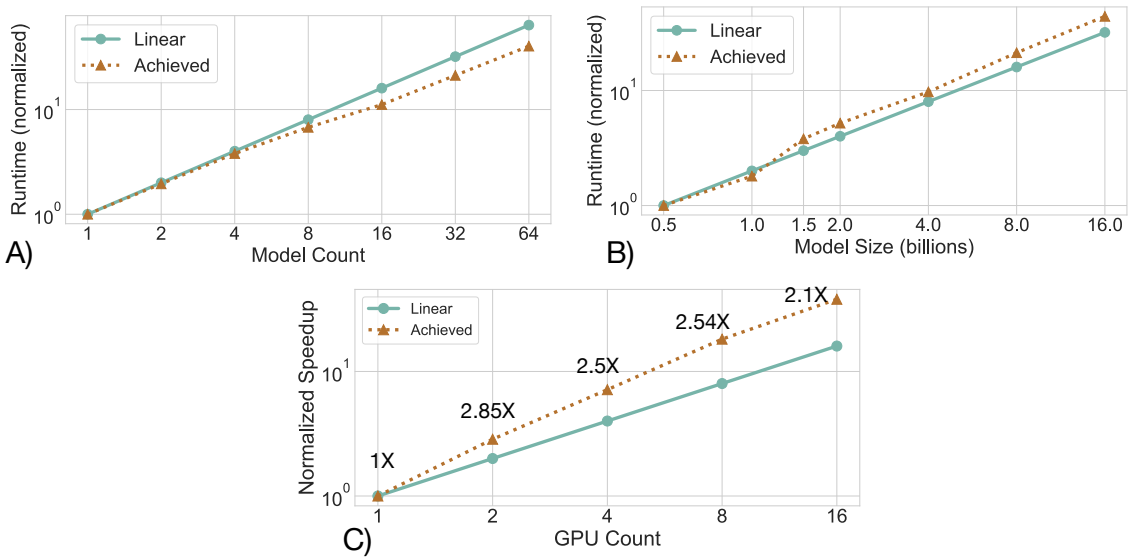

Figure 4.13. SATURN sensitivity plots on the TXT workload versus (A) workload size, (B) model size, and (C) node size. Charts are in log-log scales, normalized to the initial setting. (C) labels each point with the marginal speedup.

SATURN achieves mostly linear scaling, but with slight slowdowns on the largest model sizes. This is because the largest models force the SPASE solution to use the only viable configuration (8-GPU FSDP with checkpointing and offloading) for every model. This study shows how our SATURN's performance can be limited by the underlying UPP implementations; while it can solve SPASE effectively, it still needs performant UPP options.

Finally, we vary the number of GPUs visible to SATURN. We use TXT for this experiment. Figure 4.13(C) presents the results. SATURN achieves superlinear speedups for 2 reasons. First, the single-GPU case necessitates DRAM spilling, while larger GPU counts reduce the spilling required and open up more parallelism options. Second, higher GPU counts broaden the solution space for the MILP, enabling higher flexibility.

4.6 Conclusions

Finetuning of pre-trained large DL models is increasingly common in DL practice. But navigating the complex space of model-parallel training is unintuitive for regular DL users

even though it is needed to reduce runtimes and costs. The complex interplay of parallelism selection with model selection workloads, which requires decisions on resource apportioning and scheduling, can also lead to high resource wastage if not handled well. This work resolves these issues by formalizing the joint SPASE problem unifying large-model parallelism selection, resource apportionment, and scheduling and designing a new information system architecture we call SATURN to tackle SPASE. With user-friendly APIs, joint optimization, and a judicious mix of systems techniques, SATURN reduces large-model DL model selection runtimes by 39-49% over current practice, while freeing DL users from tedious systems-level decisions. Overall, SATURN offers maximal functionality in a critical DL setting, while promoting architectural simplicity to ease real-world adoption. Future extensions could explore alternative algorithmic approaches to the SPASE problem, extend SATURN for other scheduling objectives, and handle autoscaling clusters and dynamic job re-configurations.

Chapter 4 contains material from "Saturn: An Optimized Data System for Large Model Deep Learning Workloads" by Kabir Nagrecha and Arun Kumar, which appears in Proceedings of VLDB Endowment Volume 17, Issue 4, 2023. The dissertation author was the primary investigator and author of this paper. The code for our system is open source and is available on GitHub: https://github.com/knagrecha/saturn. This work was supported in part by Meta under a Meta Fellowship Award.

Chapter 5

INTUNE: Reinforcement-Learning-based Data Pipeline Optimization for Deep Recommender Models

5.1 Introduction

Recommendation systems now underpin many essential components of the web ecosystem, including search result ranking, e-commerce product placement, and media suggestions in streaming services. Over the last several years, many of these services have begun to employ *deep learning* (DL) models in their recommendation infrastructure, to better exploit historical patterns in their data. In turn, DL-based product recommendation has quickly become one of the most commercially significant applications of DL. Companies have begun to invest heavily in DL recommendation infrastructure, often maintaining entire datacenters and super-clusters for the sole purpose of recommender model training [9]. But in many cases, these infrastructural investments have run into critical hurdles [64]. Practitioners and cluster administrators are discovering that the *training optimization* challenges faced with DL recommender models differ significantly from those seen in historical practice with other DL model types [224]. In particular, recent studies of industry clusters have found that the unique design of recommender model architectures has left training pipelines susceptible to inefficiencies in *data ingestion* [239].

Most DL architectures are dominated by high-intensity matrix operators, and standard

tooling for DL training optimization has evolved to support models that fit this pattern [190, 69, 89, 108, 178, 82, 141]. In such cases, model execution usually dominates training times to such a degree that data ingestion procedures (e.g. disk loading, shuffling, etc) can be overlapped with and hidden underneath the matrix operation times. Unfortunately, however, DL-based recommender models (DLRMs) are atypical in this regard.

Recommender datasets are generally composed of both sparse (categorical) and dense (continuous) features, and joining information across features requires transforming these two representations into a common format. To this end, DLRM architectures use *embedding tables* to transform categorical inputs into dense embedding vectors through a hash-table lookup. These can then be combined with the dense vectors and fed through some secondary DL model to produce user-item probability ratings [221]. Figure 5.1 illustrates a typical architecture.

The embedding tables, which are the typically single largest component of the DLRM architecture, use a key-value lookup rather than dense matrix multiplication. For this reason, DLRM models are often less compute-intensive than other architectures of a comparable size. Figure 5.2 charts the differences in computational intensity between large-scale recommendation models versus language models and computer vision architectures, illustrating that DLRM models require *orders of magnitude* fewer operations than comparably-sized Transformers or Convolutional Neural Networks. This uniquely light computational footprint can lead to unexpected system optimization challenges.

Challenge & Motivation. The low computational intensity of DLRMs generally translates to low model latencies, which fail to mask the cost of data-loading and transformation. Improved GPU hardware and new model acceleration techniques have only exacerbated this issue by reducing model runtimes and increasing the requisite data-loading throughput to keep the model fed during training. The fault is not only with the models, however; the problem is aggravated further still by the generally high demands of *online data processing*, i.e. data transformation at ingestion time, for recommendation applications. In other domains (e.g. lan-

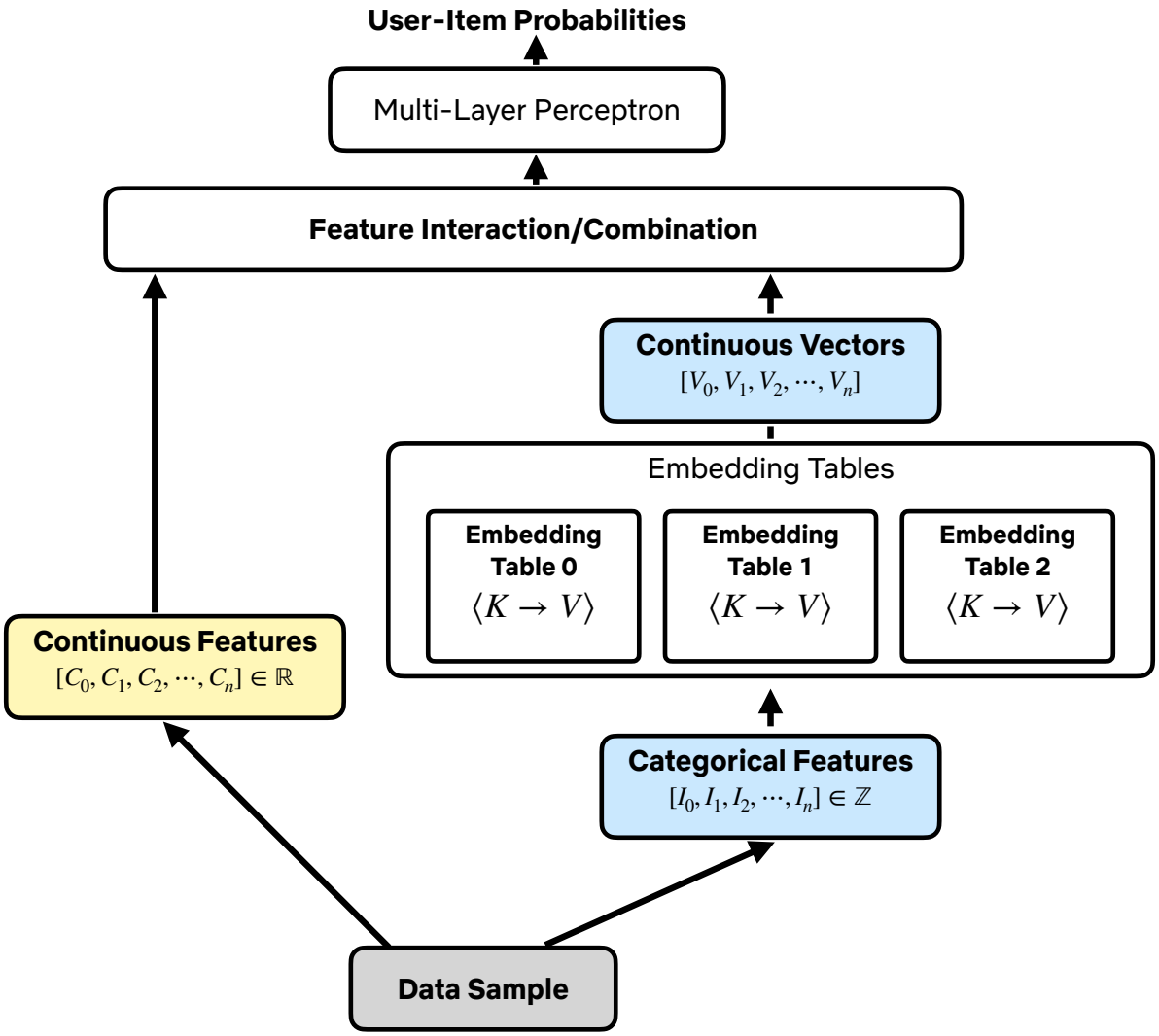

Figure 5.1. A typical DLRM architecture [38, 9]. The model uses an embedding table to convert sparse categorical data to dense vectors that can then be merged with dense features in some overlaid DNN. Adapted from a similar illustration in prior art [221].

guage modeling, computer vision) not only are training times dominated by model execution, so that data processing latencies can be more effectively hidden, but it is also practical to push the heaviest data transformation steps to an offline pre-compute phase. This greatly reduces the need to optimize data-loading. By contrast, recommendation datasets are uniquely reliant on the *online* step, which must be done alongside model execution. We attribute this to three characteristics of recommendation data: **scale**, **reusability**, and **volatility**.

First, *scale*. A recommender dataset for a popular application might span billions of

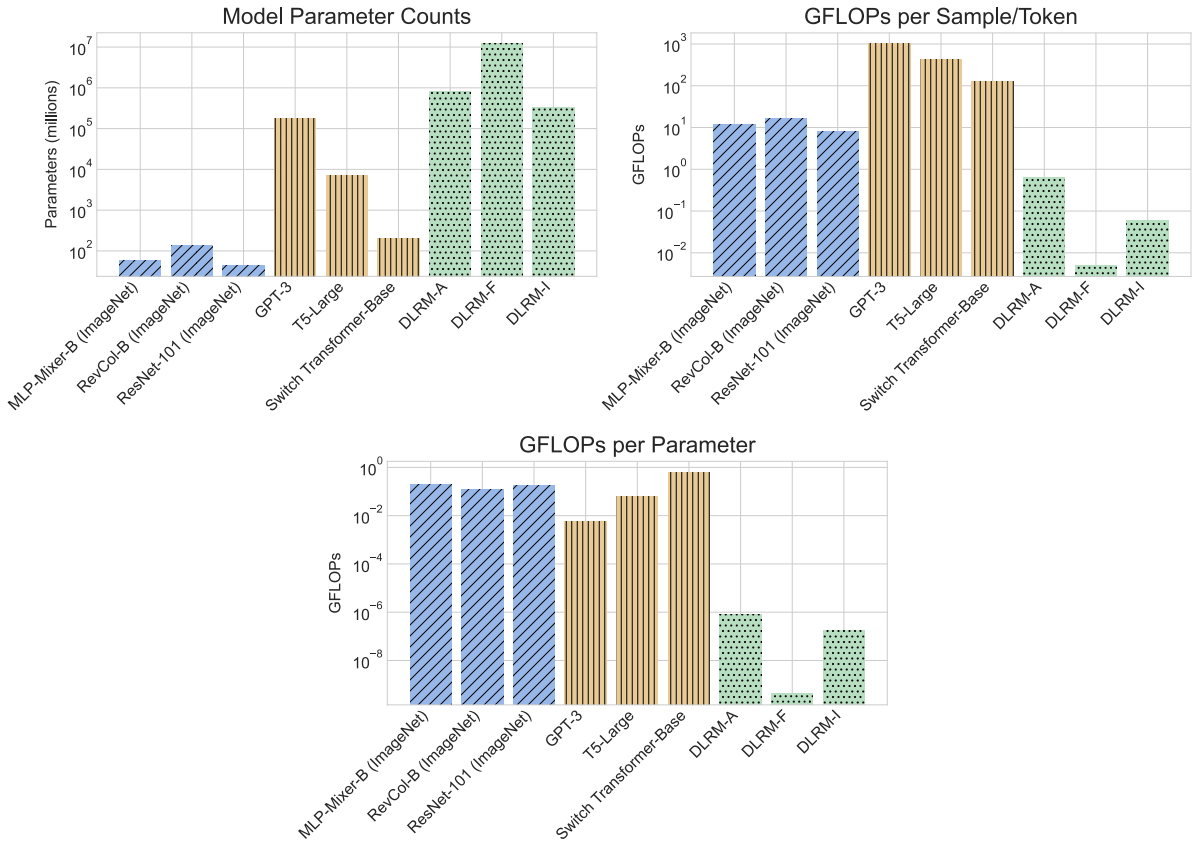

Figure 5.2. Approximate parameter & FLOP counts for popular architectures in language modeling and image recognition contrasted against DLRM models drawn from a recent paper [137]. FLOPs are reported on single-element batches (single-token for language models). We also report averaged FLOPs per parameter, derived from the previous two charts. Y-axis is set to log-scale for all charts.

interactions and require terabytes (or even petabytes!) of disk space. Offline data transformation can bloat these already high storage costs further still. Consider, for example, a common data processing operation such as *augmentation*, which randomly modifies various aspects of a data sample to produce an altogether new sample. Applying this operation offline might double or triple the size of an already massive dataset; the only practical way to run such transformations would be to do them online so that augmented samples can be discarded as soon as they are consumed. Furthermore, this scale issue often makes *caching*, which might otherwise help to mitigate processing challenges, impractical.

Second, *reusability*. A single core dataset might be reused for multiple different DLRM architectures. In a movie recommendation system, one DLRM might be used to rank rows, another might be used to rank search results, while yet another might rank genres. Each model would likely require different data transformation operations and feature extraction procedures. Pushing data transformation to the offline phase would require replicating and re-processing the original dataset dozens of times, again bloating storage and compute costs.

Third, *volatility*. Recommendation datasets are updated frequently as new interactions are recorded. In addition, *ephemeral IDs* often lead to dataset changes in domains such as e-commerce, e.g. when a product is added or removed from the platform. Any offline transformations would have to be re-run frequently as the dataset evolves. Incremental transformation is not always practical; some operations such as shuffling require the whole dataset to be present.

Prior analyses [239, 135] of DLRM training have recorded the impacts of these issues in practice, suggesting that online data ingestion optimization is critical to improving DLRM training performance. This new and emerging problem lacks a satisfactory solution. Table 5.1 provides an overview of existing tooling, but none of these prior systems can effectively tackle this issue. Generic pipeline tools such as `AUTOTUNE` & Plumber [138, 98] often lead to sub-optimal performance for DLRM jobs, or can even cause fatal out-of-memory errors. GPU data-loaders can be situationally useful, but cannot be recommended for general use due to concerns over processor cycle contention between the model and pipeline [239]. The only CPU-based DLRM data pipeline work we are aware of [239, 137] relies on a specialized cluster architecture design, and is not feasible to adopt for typical users. We expand on these in Section 4.1.1.

We seek a new system — one which can improve data-loading throughput in a general, scalable fashion without disrupting practitioner workflows or requiring large-scale cluster changes.

Approach & Contributions. In order to reason about the data-ingestion problem from first principles, we study traces taken from our internal DLRM training cluster. We focus on the

Table 5.1. Overview of existing tooling.

	Name	Description
Generic Pipelines	`AUTOTUNE` [138]	TensorFlow's built-in tool for optimizing `tf.data` pipelines, considered to be a state-of-the-art optimizer [98].
	Plumber [98]	`AUTOTUNE` alternative with roughly equivalent performance.
DLRM Pipelines	Data PreProcessing Service [239]	Meta's internal service for data ingestion. Replicates data pipelines across machines and wraps them behind a singular entry-point. Tailored for Meta's cluster; adoption would require a cluster re-design to match their architecture.
GPU Data-loading	DALI	Nvidia's tool for GPU-accelerated data-loading primitives, targets image processing operations (rotations, resizing, etc).
	NVTabular	Nvidia's tool for GPU-accelerated data-loading primitives, focuses on tabular data. Introduces GPU resource contention between the model and data pipeline; not always practical to use.

outcomes observed by real-world DLRM practitioners, and observe shortfalls in generic data pipeline optimizers. We study training times and processor utilization to better understand how poorly optimized data ingestion pipelines increase costs and reduce efficiency.

From our studies, we find that a *lack of adaptability and feedback* is the primary missing piece in generic data pipeline optimization tools. Out-of-memory errors, under-optimized user-defined-functions (UDFs), and poor responsiveness to dynamic machine re-sizing are the three main symptoms we observe. Addressing the first symptom requires incorporating feedback from the system's memory usage monitor, the second requires actively adapting the optimizer's performance model of black-box UDFs, and the third requires adaptability under changing hardware conditions.

We use this key finding to motivate the design of a new data pipeline optimization tool for

our cluster users at Netflix. We build a feedback-driven, adaptive tool to optimize data ingestion pipelines that we name INTUNE. INTUNE serves as a drop-in replacement for industry-standard optimizers such as `tf.data`'s `AUTOTUNE`, requiring no large-scale cluster redesigns or workflow disruptions. It can be applied to any data pipeline framework, including `tf.data`, PyTorch Datasets, and Ray Datasets. At the core of INTUNE is a *reinforcement learning* (RL) agent trained on historical job traces and tuned online to understand how to distribute computational resources across the data pipeline. RL provides the adaptability we need to mazimize performance. The idea of a "DL-for-systems-for-DL" loop has recently gained traction in the systems world [161]; INTUNE provides a complete example of this loop in practice.

We apply INTUNE to jobs on our real-world training cluster and see 1.18-2.29X speedups in training times versus current tooling. We observe that INTUNE can converge on an optimized resource distribution within only a few minutes, even on complex real-world pipelines. Our tests show that INTUNE is both practical and effective in improving DLRM training efficiency. We run scaling studies to test INTUNE's performance further, and find that it achieves good scaling performance with respect to both workload size and machine size.

Our contributions can be summarized as follows:

1. We provide in-depth analyses of DLRM model training job traces taken from our real-world compute cluster, highlighting the critical and unique problem of data pipeline optimization.

2. We identify and study a new gap in the DL systems landscape, and evaluate the weaknesses of state-of-the-art tooling for data ingestion during model training.

3. We propose a novel automated data-pipeline optimizer motivated by our cluster studies, INTUNE. To the best of our knowledge, INTUNE is the first system to use RL for data pipeline optimization. It is also an instance of the emerging "DL-for-systems-for-DL" loop.

4. We run comprehensive evaluations of INTUNE against state-of-the-art baselines on real-

world workloads. We find that INTUNE significantly outperforms the baselines by a factor of 1.18-2.29X, providing significant speedups and training cost reductions.

The remainder of the paper is structured as follows. Section **??** dives into the fundamentals of DL recommender training, data processing, and RL. Section 5.2 analyzes real job traces from a compute cluster to provide motivation for INTUNE's development. Section 5.3 goes into the details of INTUNE and describes how it conceptually addresses each of the challenges we identify. We show the results of our experimental studies in Section 5.4, where we benchmark our system's performance on a variety of workloads. Section 4.1.1 describes some existing tools for data processing and other related areas. Finally, we provide our concluding remarks in Section 5.5.

5.2 Cluster Study

We now analyze training jobs from our real-world DLRM compute cluster to better understand the key data pipeline challenges faced by practitioners.

5.2.1 Motivation

TensorFlow, a popular DL framework, provides the `tf.data` API for users to build input data pipelines from the primitive operations we have discussed thus far (batch, UDF, shuffle, etc). The new `torchdata` pipeline composition tool introduces similar functionality for PyTorch, though the TensorFlow data pipeline ecosystem is relatively more mature and more appropriate for our evaluations. The *de-facto* standard tool for `tf.data` data pipeline optimization is `AUTOTUNE`, which is built-in to TensorFlow [138]. `tf.data` is commonly considered to be one of the most advanced data pipeline construction tools available to practitioners, and `AUTOTUNE` is generally accepted to be state-of-the-art in pipeline optimization [98]. Due to its popularity and widespread adoption, we will take it as the standard benchmark for automated tooling in our cluster study. Historical practice on our production cluster has surfaced three issues with

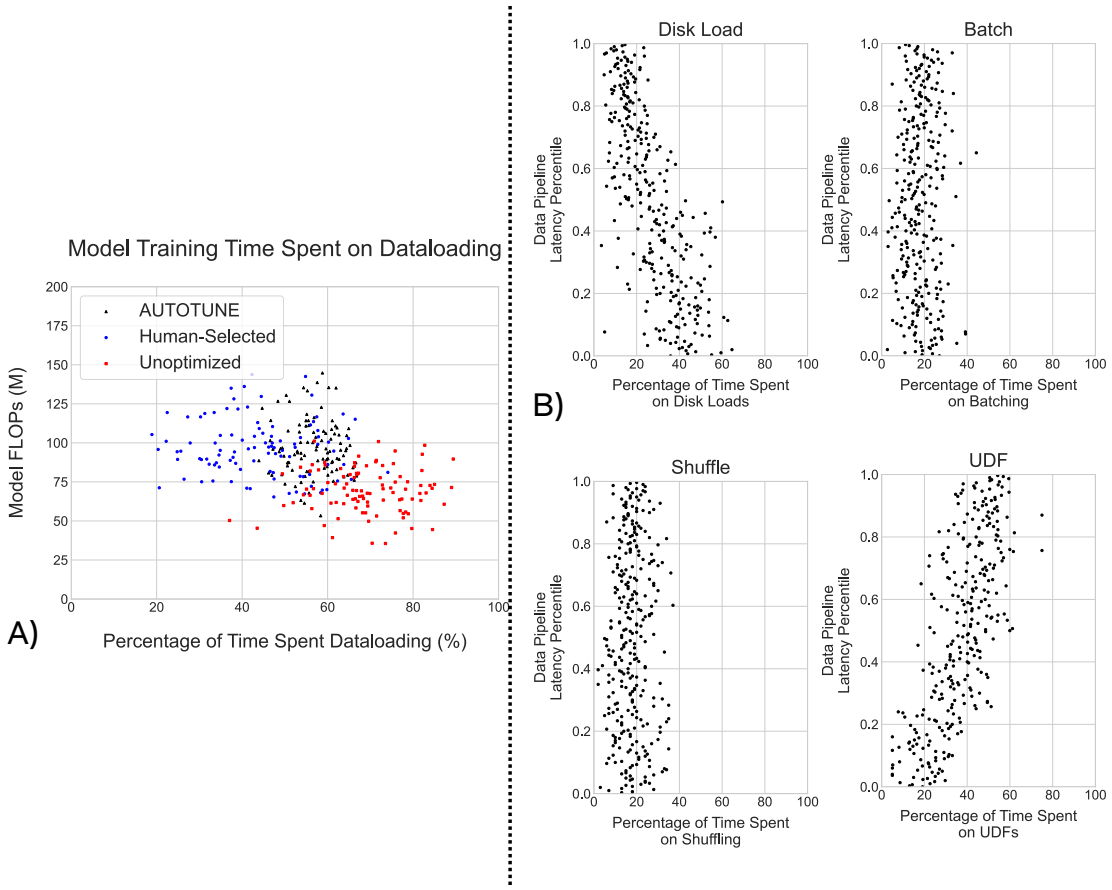

Figure 5.3. (A) Our study of real job traces shows that compute time is dominated by data processing rather than model execution, even on the most compute-intensive models. Jobs using `AUTOTUNE` are marked in black, jobs using human-selected distributions are marked in blue, and unoptimized pipelines are marked in red. (B) Breakdown of individual pipeline stage latencies when using `AUTOTUNE`. For each stage, we provide a scatter-plot the percentage of pipeline time taken up.

`AUTOTUNE`.

1. **Low efficiency on DLRM pipelines.** Tools like `AUTOTUNE` often produce suboptimal configurations in practice, bloating runtimes and costs.

2. **High failure rates.** `AUTOTUNE` has shown a tendency to trigger costly out-of-memory errors, typically caused by resource-overallocations.

3. **Poor support for rescaling.** Cluster techniques such as machine multi-tenancy or virtualization can add new resources to jobs over time. Unfortunately, `AUTOTUNE` does not take

full advantage of the new resources without human intervention.

We can validate these three points through quantitative analyses of DLRM job traces on our cluster. We recorded jobs run over a period of two weeks for our study.

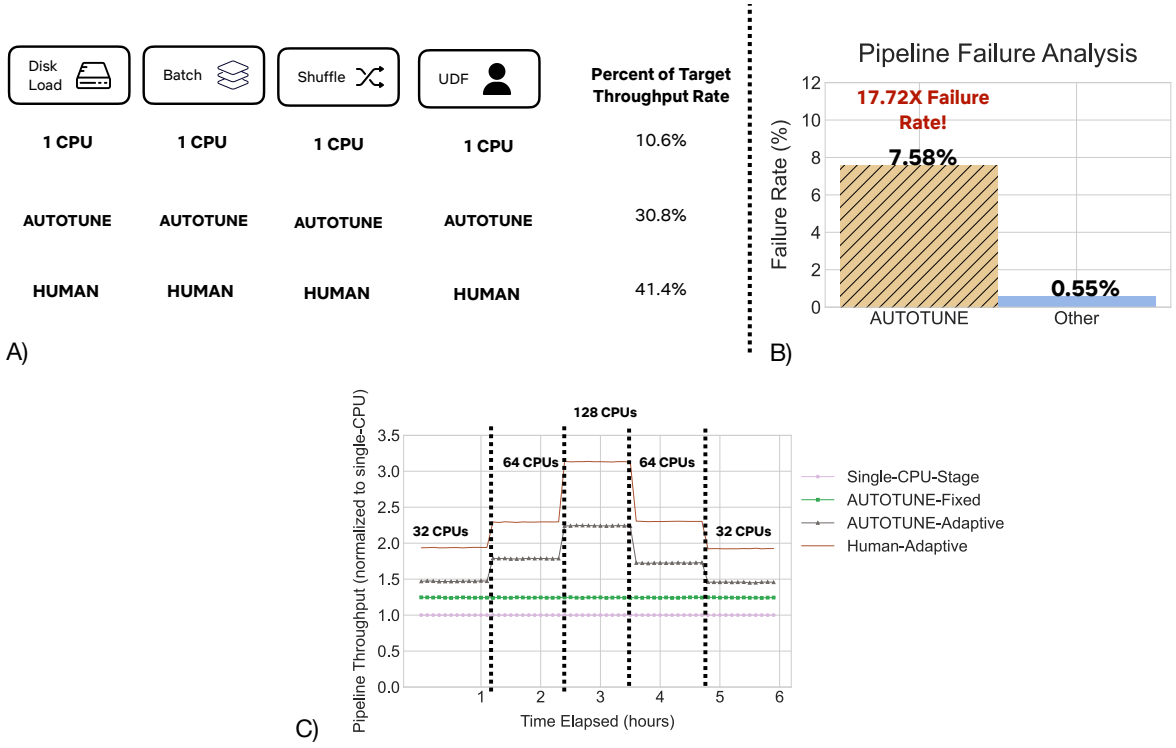

Figure 5.4. Deepdive into a case-study pipeline. (A) The percentage of target throughput achieved with different approaches. (B) Illustration of `AUTOTUNE`'s versus human-set alternatives. (C) Performance of various tools when the machine is re-sized during job execution. Approaches with manual intervention use the *-Adaptive* suffix, and results are normalized to the Single-CPU-per-stage baseline.

5.2.2 Cluster Trace Analyses

We take data from a large GPU cluster reserved only for DL recommendation model training. A broad mix of job-types are present; both exploratory ad-hoc experimentation as well as automated production pipelines. Our results show that as much as *60%* of time is spent on data ingestion rather than model execution, even when `AUTOTUNE` is applied. Manually-optimized jobs tend to perform somewhat better, while unoptimized jobs see the worst skew towards

data processing. In all cases, we see a significant opportunity for improvement — reducing data processing times would provide significant cost savings and efficiency improvements. Figure 5.3(A) presents our data.

Next, we dive into the fine-grained stages of the data pipeline, to better understand the composition of the end-to-end costs. All data pipelines in this analysis use `AUTOTUNE`, and follow a standard order of `disk load` $\rightarrow$ `batch` $\rightarrow$ `shuffle` $\rightarrow$ `UDF` $\rightarrow$ `prefetch`. These pipelines do not include a "cache" stage due to memory constraints; these jobs operate with high-dimensional features & very large datasets. Figure 5.3(B) shows the results.

Maximizing pipeline throughput requires maintaining equal per-stage latencies [118]. But `AUTOTUNE` is known to struggle with irregular stages such as UDFs or varied-size disk loads [98]. Our empirical study confirms this issue at a mass-scale. On average, UDF mappings and disk loading dominate runtimes, and the skew towards UDFs tends to grow as overall data pipeline latencies increase. The proportion of time spent on shuffling or batching tends to stay mostly consistent regardless of overall pipeline latency, further pointing to UDFs as the primary stumbling-block for `AUTOTUNE`. Unfortunately, UDFs are a key piece of most of DLRM data pipelines, covering basic tasks such as feature extraction, re-mapping, and transformation. A previous study [239] of DLRM training describes 16 common preprocessing operations; we found that *14* of these 16 required UDF implementations! Poor UDF optimization alone is sufficient to discourage `AUTOTUNE` adoption among our users.

5.2.3 Pipeline Deep Dive

To gain more detailed insights, we will now analyze a singular production training pipeline from Netflix as a case study. This job is rerun on a regular basis, multiple times per day, allowing us to collect a rich history of statistics. Training jobs are run on machines with 128 Intel Xeon 3.0GHz CPUs, and datasets are stored on a remote network filesystem. One of our primary aims in this section is to demonstrate how current state-of-the-art tooling fails to serve our needs. To illustrate this, we labeled jobs according to whether they used `AUTOTUNE`,

human-set configurations, or else no optimization at all (i.e. one CPU per stage). We contrast these approaches in our experiments.

The model is relatively small — under 10M parameters. The model latency is very low, so to avoid idle times, the data pipeline must offer a high throughput rate. The data processing pipeline requires: (1) loading data from disk, (2) shuffling it in a fixed buffer, (3) applying a UDF to extract and convert categorical features to standard mappings, then (4) batching the data before (5) prefetching it to the GPU. If only one CPU is given to each stage, pipeline throughput is 11% of the data-loading rate needed to keep the model served at all times (i.e. no idling), as shown in Figure 5.4(A). After `AUTOTUNE` distributes all these processors, pipeline throughput is increased by 2.81X to 31% of the target rate.

We contrast this against *manually* chosen allocations, which increases the pipeline throughput to 41% of the target rate. Further improvements (e.g. to 100% of the rate, with no idle times in model execution) would require scaling beyond the machine's current resources. But even within the machine, we found scope for a *1.34X* speedup versus `AUTOTUNE`! Ideally, we should be able to produce this configuration automatically, without manual intervention.

Another serious issue we observe in applying `AUTOTUNE` to this example pipeline is *overallocation*. If we allow `AUTOTUNE` to take control of the *prefetch* stage, it tries to improve performance by maximizing prefetches. This bloats memory usage, often causing OOM errors. Figure 5.4(B) illustrates the frequency of OOM errors produced by applying `AUTOTUNE` to this pipeline. Recovering from these errors requires a teardown and reset, leading to significant downtime.

An increasingly popular technique in large-scale compute clusters is machine resizing [154, 142, 136], either from scheduler interruption & re-assignment [226, 227], or job completions on a multi-tenant machine [142]. Rescaling the data pipeline to make use of new resources requires the optimizer to actively respond to hardware changes. Figure 5.4(C) charts the performance of various approaches in this setting. We see that out-of-the-box `AUTOTUNE` does not respond effectively to machine re-sizing, and does not increase pipeline throughput

even when new CPUs are provided to the job. If human intervention is used to reset and re-launch AUTOTUNE, the scaling improves but is still significantly under-performs the fully-human alternative. This reliance on human decision-making and intervention is impractical.

Now that we have observed DLRM data pipelining problems in a real training cluster, we seek a solution that can address these challenges.

5.3 System Design

Our studies in Section 5.2 surfaced three issues with current tooling: (1) suboptimal performance on DLRM pipelines due to UDFs, (2) inability to scale as resource caps are changed, and (3) a tendency to overallocate resources, leading to OOMs. We propose that all three of these problems would be resolved by a *feedback-driven* tool. A tool that actively evaluates its environment and collects feedback *live* would be able to (1) fine-tune its understanding of UDF performance throughout training, (2) actively respond to changing resource caps, and (3) manage memory usage.

With this in mind, we turn to RL to design our tool for data pipeline optimization — INTUNE. As we discussed in Section **??**, building an RL agent for data pipeline allocation requires us an environment, an agent, and an action space. We now discuss these RL system components.

5.3.1 Environment

The environment in our setting should reflect the data pipeline state and available hardware. Certain aspects of the environment are static (e.g. DRAM-CPU bandwidth), others are uncorrelated to the agent's actions (e.g. model latency), while others are directly impacted by its actions (memory usage, CPU usage). We model our environment with the aim of providing the RL agent with any and all information it might need to make an informed decision. Our finalized list of factors is shown in in Table 5.2.

These details are sufficient for the agent to quickly grasp its problem setting. The static

Table 5.2. RL environment factors.

	Factor	Motivation
Agent-Modified Factors	Pipeline Latency	Allows agent to understand the performance of the current configuration. May change based on agent actions.
	Free CPUs	Allows agent to see how many extra CPUs it can allocate. May change based on (1) agent action or (2) machine resizing.
	Free Memory (bytes)	Allows agent to see how much memory it can use to increase prefetch levels. May change based on agent actions.
Uncorrelated Factors	Model Latency	The actual model execution time. Updated regularly to improve estimation accuracy. Unrelated to agent actions.
Static Factors	DRAM-CPU Bandwidth (MB/s)	Interconnect speed can impact the value of prefetching. Found up front and unchanged throughout training.
	CPU Processing Speed (GHz)	CPU processing speed can impact decision-making on resource allocation. Found up front and unchanged throughout training.

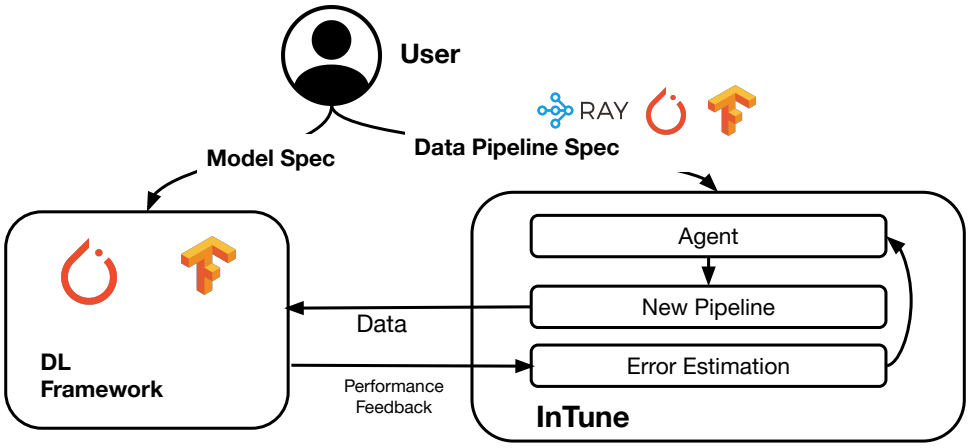

Figure 5.5. INTUNE's RL data pipeline system architecture.

factors will provide some "immediate" information while the other aspects will help it to learn how its actions impact data pipeline performance. Our agent reward is directly based on data pipeline latency and memory usage. Equation 5.1 shows the function.

$$R = throughput \times (1 - \frac{memory_{used}}{memory_{total}}) \tag{5.1}$$

If prefetch is not used excessively, then the memory usage portion of the equation is largely irrelevant. But to avoid OOM outcomes like those seen with `AUTOTUNE`, we ensure that INTUNE's reward approaches 0 as memory consumption nears 100%.

5.3.2 Agent Model

We aim to build a low-cost, lightweight model architecture for the RL agent. Since INTUNE runs in parallel with the target DL job, we do not want to over-consume resources. To minimize computational demands, INTUNE's DQN agent is a simple three-layer MLP architecture using the ReLU activation function, built in PyTorch. It can be run on either CPU or GPU resources, or even as a remote service interacting with the target job over the network.

If the action space consists of <256 possible choices, this model only requires <200FLOPs per iteration, which should not interfere excessively with the actual model training job. We train different versions of the agent in offline simulations to prepare them for live deployment/tuning. Each version is built for a different common pipeline length on our clusters (e.g. one agent for 4-stage pipelines, one for 5-stage, etc). During actual data ingestion, the model is *fine-tuned* using live feedback to adapt it for the current job. We report on convergence behaviors in Section 5.4.1.

5.3.3 Action Space

As we discussed in Section **??**, it is common practice to reshape the agent action space to improve accuracy. If we allowed our agent to directly select any distribution of resources, the size of INTUNE's action space would be $\binom{n+r-1}{r-1}$ where n is the number of CPUs and r is the number of pipeline stages. On a typical setup (e.g. 128 CPUs over 5 stages), this would

yield $1.2e7$ possible actions — which is entirely impractical. Based on the agent we described in Section 5.3.2, this would increase iteration compute costs to more than 6.1GFLOPs!

Instead, we use action-space reshaping, and design an *incremental* action space. At every step, INTUNE's agent can choose to "raise-by-one", "maintain", or "lower-by-one" the allocation of each stage. Memory-bound factors use a megabyte unit while processing-bound factors use a CPU unit. On its own, this is somewhat inefficient. In order for the system to allocate 128 CPUs, a minimum of $128/n$ iterations would be required. To improve search and convergence times, we give the agent additional options of "raise-by-five" and "lower-by-five". This yields an action space of 5^r options. Since r is typically $<= 5$, this is entirely manageable. Increasing the action space by adding new options (e.g. "raise-by-ten") could be used to further modify the convergence behaviors, but we found that increment options of 1 and 5 are sufficient for INTUNE to rapidly converge on a performant solution.

These three components — environment, model, and action space — provide the basis for INTUNE.

5.3.4 Interface & Usage

We aim to make INTUNE easy to integrate into existing user code, without disrupting workflows or requiring cluster architecture changes. Users design their data ingestion pipelines in standard framework code (e.g. PyTorch, `tf.data`), then wrap their pipeline under INTUNE, specifying any tunable performance knobs. Performance monitoring and value adjustment is all handled automatically by INTUNE. Listing 5.1 provides an example.

```
pipeline = create_pipeline()
system_pipeline = intune.pipeline_wrapper(pipeline, knobs=[pipeline.stage_1, pipeline.stage_2...])
...
train(model, system_pipeline) # replace references to pipeline with system_pipeline
```

Listing 5.1. INTUNE usage.

We illustrate the overall design in Figure 5.5. Note the generality of INTUNE's design; nothing about it is tied to a specific data pipeline framework. So long as the framework exposes

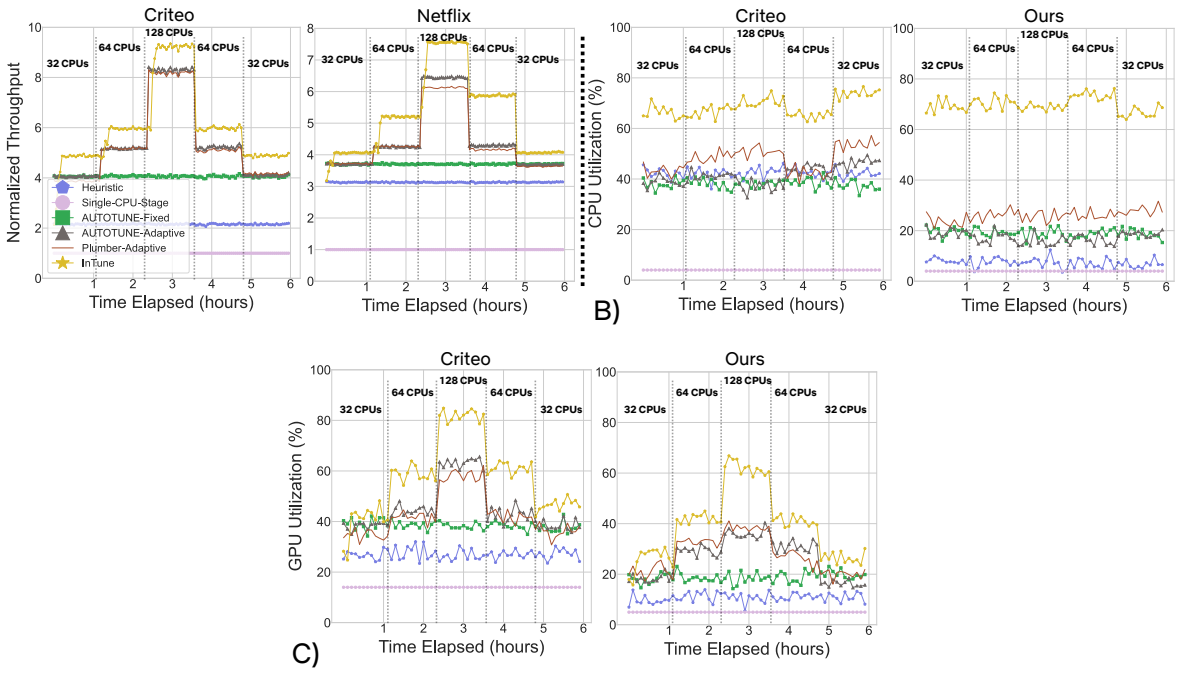

Figure 5.6. All figures use the legend in the leftmost chart. (A) Pipeline throughput over time for each approach, normalized to the Unoptimized baseline. (B) CPU utilization over time for each approach. Only active CPUs are considered, to prevent confounding system behaviors with the separate impact of rescaling. (C) GPU Utilization over time for each approach.

optimization knobs (e.g. for CPU assignment), this approach is applicable.

5.4 Evaluation

We now provide a thorough evaluation of INTUNE. Our aim is to answer the following questions.

1. Does INTUNE capable of achieving higher pipeline throughput than standard tools such as `AUTOTUNE`?

2. Is INTUNE less susceptible to issues such as out-of-memory errors than standard tooling?

3. Is INTUNE capable of responding to resource rescaling?

4. Does INTUNE converge on an optimized solution quickly?

Workloads. We use two workloads, one drawn from an internal recommender model and dataset and one built using Meta's open-source DLRM code and the Criteo dataset. The custom dataset task focuses on product recommendations while the Criteo dataset is used in a click-through-rate prediction task.

Datasets and Pipelines. The custom dataset uses dozens of sparse features, and fewer than 5 continuous features, with a batch size in the tens of thousands. The Criteo dataset consists of 26 sparse categorical features and 13 continuous features and a batch size of 24,096. We initialize the dataloader allocations with a simple "even division" of CPUs across stages. The RL agent then modifies the distribution provided by this heuristic. We do not consider a cache stage in the current version, since most of the relevant jobs on our cluster do not use one, but there is no reason INTUNE could not be extended to manage the resource allocations of cache stages as well.

Models. The model taken from our production pipeline is fairly small — <5M parameters, most of which are contained in the embedding tables. We make a large model for the Criteo dataset, with 25B+ parameters, most of which are in the embedding tables. In both cases, the model latency is sufficiently low such that training times are dominated by data processing.

Hardware Setup. The production model is trained on a single 40GB A100 GPU. We initially provide the data pipeline with 32 Intel Xeon 3.0GHz CPUs, then on regular intervals double the CPU count up to a limit of 128. Then, we halve the allocation repeatedly until we reach 32 CPUs again. Approaches other than INTUNE rescale via manual intervention and re-launching; INTUNE will naturally adapt to the environmental change and so does not require this intervention step. The Criteo model is too large for a single A100 to train, so we use a standard hybrid parallelism approach [219] to distribute memory demands and accelerate training. We use the same CPU scaling procedure applied to the custom model (i.e. $32 \rightarrow 64 \rightarrow 128 \rightarrow 64 \rightarrow 32$). The datasets are stored on a high-bandwidth network-mounted filesystem, a common approach for large-scale recommendation datasets.

Baselines: We compare INTUNE to the following baselines.

1. **Unoptimized.** In this version, only a single CPU is allocated per stage such that no parallelism is possible.

2. **AUTOTUNE.** `AUTOTUNE` is a standard TensorFlow offering, commonly used to optimize `tf.data` pipelines.

3. **AUTOTUNE-Adaptive.** We checkpoint and re-launch `AUTOTUNE` on machine rescaling intervals to allow it to adapt to the new machine resources.

4. **Plumber-Adaptive.** Plumber [98] is a more user-friendly alternative to `AUTOTUNE` that uses an MILP to distribute resources. We apply the same checkpointing approach as in `AUTOTUNE`-adaptive. Plumber offers additional auto-caching optimizations; we disable these since they are orthogonal to our resource-distribution work with INTUNE and could be integrated with INTUNE in the future.

5. **Heuristic.** This version simply distributes CPUs evenly to emulate what a human user's best guess distribution might look like. INTUNE's RL agent is initialized from this state as well.

These baselines cover most typical configurations, both manual and automated.

5.4.1 End-to-End Performance

Pipeline Performance. We compare achieved training throughput for all approaches on both the real-world and Criteo pipelines. We initiate rescales at regular intervals to evaluate how each system "responds" to the new resource availability. We normalize throughput to the *Unoptimized* baseline in all our analyses. Figure 5.6 (A) presents the results.

INTUNE provides significantly better throughput and hardware utilization than the strongest competitors on both pipelines. Accounting for flexible rescaling, the average marginal gain versus standard `AUTOTUNE` tooling increases to 2.05X and 2.29X on the custom pipeline

and Criteo pipeline respectively. Against the alternatives which employ human intervention, the marginal improvement is still significant, ranging between 10-20%. Not only does our approach eliminate the headache of manual intervention, but it also achieves lower compute times and higher utilization. In each experiment, we observe that INTUNE achieves a stable throughput rate within about 10 minutes. The *Plumber* baseline also requires some tens of iterations to converge, but this period is so short it does not register on the chart. On long-running jobs, INTUNE's 10-minute optimization time is insignificant, but it may be problematic for short ad-hoc experiments. But in such cases, fine-tuned performance optimization is rarely important.

We also observe significantly lower *failure rates* than AUTOTUNE. On average, AUTOTUNE caused an 8% OOM failure rate on both pipelines, whereas INTUNE did not cause even a single crash. This improved robustness makes INTUNE an attractive option for failure-sensitive jobs.

INTUNEalso achieves higher processor utilization, illustrated in Figures 5.6 (B) & (C) likely due to reduced idling from more effective resource allocations. Some part of the CPU utilization increase can also be attributed to the overhead of maintaining a secondary RL model; unfortunately it is difficult for us to separate the two. The improved GPU utilization follows directly from the higher data throughput, since the GPU & model are fed faster.

Intuition on Effectiveness. INTUNE's ability to map out and tune its performance estimates over time allow it to adapt and outperform other baselines on both pipelines. No other system can adapt as effectively to machine re-sizing out-of-the-box. The improvements against baselines which employ human intervention can be attributed to one of the other primary weaknesses we observed in Section 5.2, UDF performance modeling. As we will show in Section 5.4.2.1, INTUNE proves to be significantly better than the strongest baseline — AUTOTUNE — in optimizing UDF pipeline stages.

5.4.2 Drilldown Studies

We now dive into INTUNE's scaling behaviors on the real-world data pipeline. We aim to answer the following questions.

1. How does InTune's performance change as the pipeline's complexity is changed?

2. How does InTune's performance change as CPU counts are changed?

3. How does InTune's performance change as batch size changes?

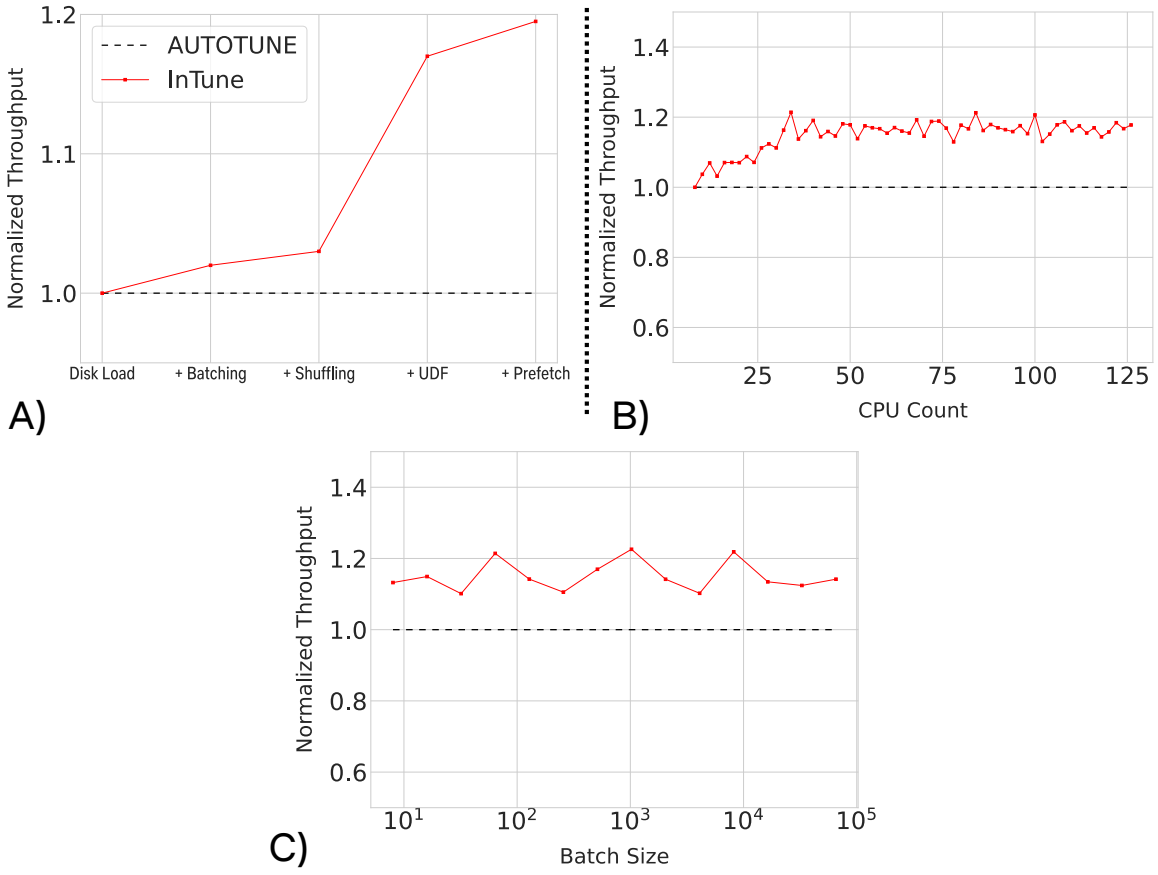

Figure 5.7. Performance scaling with respect to (A) pipeline complexity, (B) CPU count, and (C) batch size.

5.4.2.1 Pipeline Complexity Scaling

We report on performance normalized to the `AUTOTUNE` baseline on the same pipeline. All settings use the same machine, with 128 Intel Xeon 3.0GHz CPUs, and a constant model latency of 0s (to encourage maximal pipeline optimization). Pipeline "complexity" is adjusted by increasing/decreasing pipeline length (e.g. `+ batching`, `+ shuffling`). Figure 5.7(A) presents

the results. We see that our system's marginal improvement over the AUTOTUNE baseline grows as pipeline complexity increases, with a spike when UDFs are introduced. This corroborates earlier studies which found that AUTOTUNE underperforms on more complex, UDF-driven pipelines [98].

5.4.2.2 Machine Size Scaling

We now study the scaling efficacy of InTune. We increase CPU count in increments of 2, ranging from 8 to 128. AUTOTUNE is re-launched at each machine size to rebase the relative performance. Figure 5.7(B) presents the results. InTune's relative improvements over AUTOTUNE tend to grow as the valid configuration space increases, but then flattens out to a constant outperformance of roughly 20%. This flattening should be expected; AUTOTUNE is a strong baseline and even a 20% performance margin is significant [98].

5.4.2.3 Batch Size Scaling

In our final scaling study, we evaluate our system's performance with respect to batch size. Like the pipeline complexity study, we evaluate our system's ability to respond to varied workload intensity. Larger batch sizes increase demands on specific stages (i.e. the batch stage, prefetch stage, and possibly the UDF stage). Since end-system users might wish to deploy the system on any range of batch sizes, we implement this study to give a more thorough understanding of our system's offerings to users. Figure 5.7(C) presents the results. We see that our system manages to maintain (and even improve) average sample throughput even as batch size increases.

5.5 Conclusion

DLRM training costs are often dominated by online data ingestion rather than model execution. The primary throughput bottleneck in this setting is *CPU-driven* data processing rather than *GPU-accelerated* model operations. Thus, optimizing the data ingestion phase is critical to ensuring cost- & time- effective model development. Unfortunately, existing tooling

for DL data ingestion pipelines does not support DLRM setting effectively. We draw on lessons learned from analyses of real DLRM workloads from our training cluster at Netflix to motivate the design of a novel RL-based system we name INTUNE. INTUNE dynamically allocates CPUs & memory across stages of the online data ingestion pipeline, significantly improving efficiency over industry-standard baselines without requiring changes to the cluster architecture or user workflows. Benchmarks on real & synthetic training pipelines show that our system outperforms the strongest out-of-the-box tools by >2X, and human-managed baselines by up to 20%. Overall, INTUNE offers significant performance and cost benefits for recommender training pipelines and should serve to encourage further training optimization works customized for the unique needs of DLRM training. Future extensions could extend INTUNE to other decisions in the DLRM data pipeline; e.g. intermediate caching, or how to scale the pipeline across multiple machines.

Chapter 5 contains material from "InTune: Reinforcement Learning-based Data Pipeline Optimization for Deep Recommendation Models" by Kabir Nagrecha, Lingyi Liu, Pablo Delgado, and Prasanna Padmanabhan, which appears in Proceedings of the 17th ACM Conference on Recommender Systems, September 2023. The dissertation author was the primary investigator and author of this paper.

Chapter 6

INTUNEX: Reinforcement Learning for Intra- & Inter-Node Recommender Data Pipeline Optimization

In this chapter, we describe INTUNEX, an extension of INTUNE that broadens the data pipeline optimization problem to account for *multi-node* processing. INTUNEX moves beyond the problem of resource allocation and orchestration to capture the complexities of *node allocation* in a cluster, while accounting for the same fundamental challenges seen with INTUNE: complex pipelines, uncertain performance models, and high performance demands. We extend our reinforcement-learning-based meta-learning scheme to a *multi-agent* architecture, with the first level handling node allocation, and the second-level directly applying the solver used by the base INTUNE system. INTUNEX improves cluster efficiency and costs considerably, and is now used for production-scale workloads in industry.

6.1 Introduction

While the base INTUNE system is highly effective in optimizing DLRM training jobs that remain within a single node, *multi-node* processing remains a considerable challenge. This setting — which associates multiple data-processing CPU nodes with a single GPU-based trainer node — has become increasingly popular, largely due to the challenges of data ingestion we highlighted in INTUNE. But naïve replication, where the pipeline configuration is simply copied

across nodes [239] misses out on opportunities for fine-grained, cost-effective optimization. INTUNE demonstrated that such workload-aware optimization can achieve significant performance benefits.

Drawing on the same line of thinking — RL-based workload-aware optimization — we expand our optimization problem to cover a higher level of resource allocation, i.e., *node allocation*. In a given training cluster, different node instances will be present, each with their own cost & performance profiles. In such *heterogeneous* clusters, the data pipeline may be scaled over multiple *dissimilar* machines [144]. Autoscaling must also be factored in here — resized machines are effectively *new machines* with different hardware configurations [97]. The complexity of this combined problem is considerable — resource-allocation, re-sizing, & monitoring must be solved for both *intra-* and *inter-* node resources. But accurately determining which nodes to use (and how many of each are needed) can help maximize the performance and cost-efficiency of jobs. So, given some cluster resources and a DLRM training job, INTUNEX— our extended version of INTUNE — aims to identify: (1) the most cost-effective *set of nodes* for the given job, and (2) performance-optimized pipelines for *each individual node* (i.e., the base INTUNE problem).

We evaluate INTUNEX on the same toy & real-world workloads used for the original INTUNE system, but now expand the job to be run in a *multi-node* configuration on a real-world, industry cluster provided by Netflix. We observe that even in this multi-node, multi-level optimization problem, the RL principles underlying INTUNEX still enable us to quickly converge on optimized resource distributions. Ultimately, we observe that INTUNEX enables up to 15-25% higher cost-efficiency across our training cluster, potentially unlocking many millions of dollars of savings on industry-grade workloads.

6.2 System Design

Our problem combines two interlinked resource orchestration challenges: at the intra-node layer (i.e., memory & CPU resources), and at the inter-node layer (i.e., different machines & nodes as resources). So, we use a interlinked multi-agent RL design.

First, we *simplify* the problem. If two machines are identical in their resources, the pipeline solutions for those two machines can also be identical. So, rather than optimizing machines on a *single-instance* basis, we view machines as *groups*. Identical machines belong to the same abstracted group, and optimization decisions are made for the entire group simultaneously. After autoscaling events where an individual machine's resources are changed, the changed machine splits off to form a new group — since it is no longer identical to its former groupmates.

Next, we consider intra-node resource optimization. Each group's optimization can be solved independently of other groups, since there is no cross-machine dependence. So, we create a separate RL agent *for each group* to produce resource allocation solutions for each group. We refer to these agents as *group agents*. These group agents each use the design first proposed in the original InTune system [146].

Finally, we consider the problem of adding & removing nodes, i.e., inter-node optimization. A higher-level orchestrator must make these decisions — while accounting for cost efficiency (Equation 6.1). Our orchestrator must operate under uncertain conditions and handle variable per-node performance; a static solver or MILP would not be appropriate. We use a secondary RL agent — the *orchestration agent* — to tackle this problem.

$$\text{cost efficiency} = \frac{\text{throughput}}{\text{cost}} \tag{6.1}$$

Figure 6.1 illustrates the overall architecture.

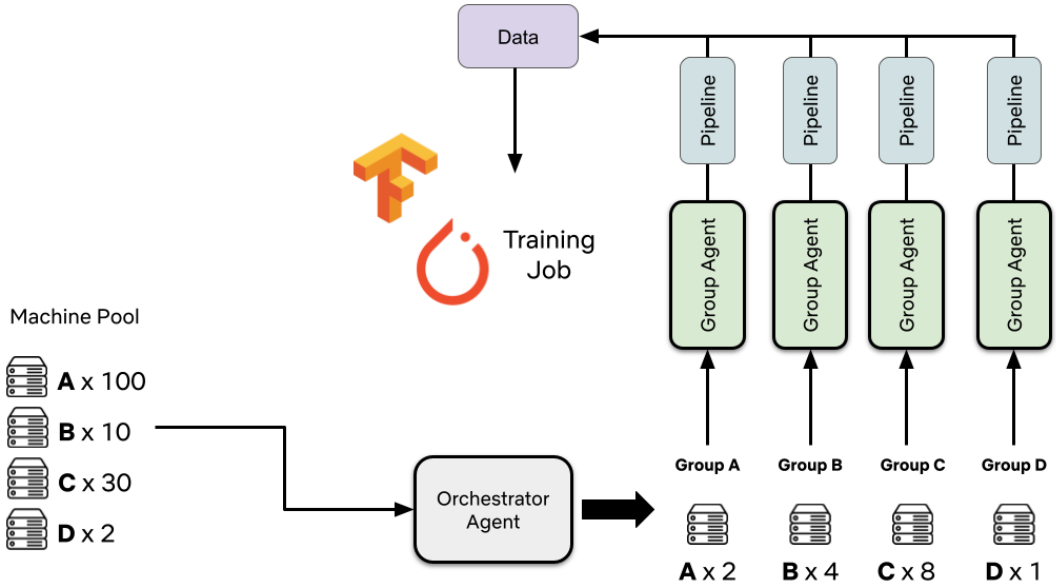

Figure 6.1. INTUNEX's multi-agent architecture.

6.2.1 Reinforcement Learning Setting

Environment. The original INTUNE paper already covers the group agent environment & solution. So, here, we focus on the environment and RL setting used for the orchestrator agent.

We design the orchestrator agent's environment to create a view of the performance of *multiple groups* simultaneously. We formulate the problem as a resource distribution challenge. The optimizer is provided with pools of different nodes, and it must draw from these pools to compose a well-optimized pipeline. Each node comes at a cost, e.g., based on cloud instance pricing or some internal budgeting scheme.

Thus, the agent must be aware of: (1) the number of instances placed in each node group, (2) the number of instances available to add to each node group, (3) the cost per instance/group, (4) the throughput of each group, and (5) as a corollary of (4), the average instance throughput for each group. Table 6.1 summarizes these environmental factors.

Autoscaling events and instance partitioning can also be factored in without difficulty, since partial nodes can be seen as "new node types" for the purposes of the orchestrator. For tractability, we assume that all possible machine sizes are known in advance.

Table 6.1. Orchestrator environment factors.

	Factor	Motivation
Agent-Modified Factors	Group Sizes.	Allows agent to understand the current number of instances per group. May change based on agent actions.
	Group Throughput	Allows agent to see the throughput of each group. Depends on group sizes, the local group agent, and the instance types.
	Free Nodes	Allows agent to see the number of machines available to add on for each group.
	Group Costs	Allows agent to see the cost associated with each group.
	Overall Costs	Allows agent to see its overall costs.
	Throughput	Allows agent to see its overall performance & throughput.
Uncorrelated Factors	Instance Throughput	The throughput of each group's base instance. Updated regularly to account for each group-agent's optimization decisions.
	Model Latency	The model's latency, which defines the required throughput to serve it.
Static Factors	Instance Costs ($)	The cost of each group's base instance.
	Per-Instance Statistics	Multiple factors covering technical details for each instance, e.g., CPU counts, memory, etc.

The actions of the orchestrator (e.g., increment/decrement group size) are then evaluated using our reward model. The reward must account for *cost-efficiency*, as shown in Equation 6.1, while accounting for application-specific biases towards cost or throughput. We define a flexible reward function to encode these requirements as shown in Equation 6.2.

$$R = \frac{\lambda(throughput)}{\phi(\text{cost})} \tag{6.2}$$

λ & ϕ encode application-specific motivations. In this paper, we use the following functions:

$$\phi(\text{cost}) = \begin{cases} \text{inf} & \text{if cost} > \text{budget}, \\ 1 & \text{if cost} < \text{budget}. \end{cases}$$

$$\lambda(\text{throughput}) = \begin{cases} 1 & \text{if } x \geq \frac{1}{\text{model latency}}, \\ \text{throughput} * \text{model latency} & \text{if throughput} < \frac{1}{\text{model latency}}. \end{cases}$$

These encode a cost budget & target throughput rate. Figure 6.2 illustrates how the reward scales with throughput & cost for a given budget & model. Of course, system users may configure λ & ϕ appropriately for their specific needs.

Agent Model. The agent architecture we use for the orchestrator is largely the same as the original architecture used for the group agents drawn from INTUNE. As before, we leave agent architecture maintenance, swapping, and deployment to the discretion of the cluster administrator. We adopt the same hybrid offline-online training paradigm, where the agent is pre-trained offline on historical traces, then updated online for incoming jobs.

Action Space. Much like the group agent, we formalize our agent's action space as a *discrete* set of increment/decrement/maintain options. This action-space shaping stage helps

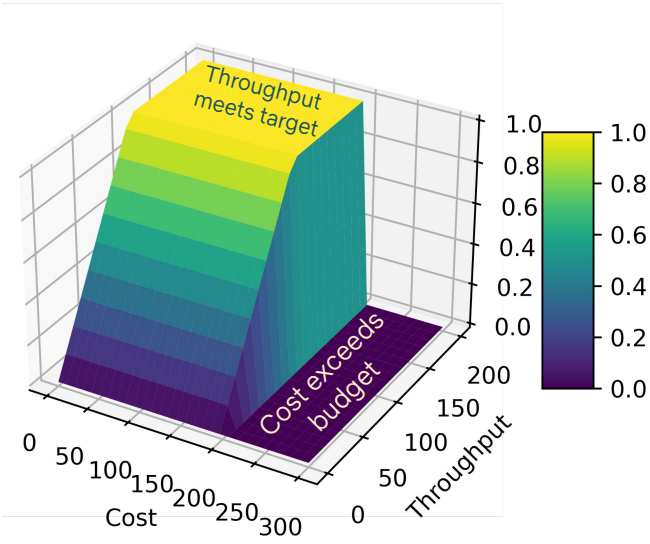

Figure 6.2. Our chosen λ and ϕ cause the reward to scale linearly with throughput unless the compute budget is exceeded or the target data serving rate has been met. We illustrate with a fixed budget of $200 and a model latency per-batch of 10ms.

minimize complexity and encourages a simpler model optimization (as opposed to operating in a continuous configuration space).

The orchestrator agent must account for *variability* in the group agents' performances. As each group agent converges on an optimized state, the throughput provided by each group might change dramatically. To mitigate this issue — which can muddy observations taken by the agent — we use an *architecturally simple* mechanism based on action space masking [48]. We simply prevent the orchestrator agent from taking a "removal" action on any group until the local group agent has at least passed some number of iterations (e.g., 100) for optimization. Previous studies [146] have shown that group agent convergence should occur relatively quickly, so reliable performance readings should be available within a short period of time.

Interface and Usage. We opt for the same interface used by INTUNE; the multi-node extension is handled transparently with background cluster queries (e.g., on Ray). We find that this is sufficient for a hands-off cluster administration approach, though adapting this interface layer should be straightforward if more transparency is needed.

6.3 Evaluation

We now provide a thorough evaluation of INTUNEX's optimization capabilities. The original INTUNE paper already makes the case for RL-based pipeline optimization on a single node, so, we focus our evaluations here on the multi-node setting.

Workloads. We use two workloads, one drawn from an internal recommender model and dataset and one built using Meta's open-source DLRM code and the Criteo dataset. The custom dataset task focuses on product recommendations while the Criteo dataset is used in a click-through-rate prediction task.

Datasets and Pipelines. The custom dataset uses dozens of sparse features, and fewer than 5 continuous features, with a batch size in the tens of thousands. The Criteo dataset consists of 26 sparse categorical features and 13 continuous features and a batch size of 24,096. We initialize the dataloader allocations for the group agent with a simple "even division" of CPUs across stages. The agents then modify the distribution provided by this heuristic.

Models. The model taken from our production pipeline is small — <5M parameters, most of which are contained in the embedding tables. We make a large model for the Criteo dataset, with 25B+ parameters, most of which are in the embedding tables. In both cases, the model latency is sufficiently low such that training times are dominated by data processing. The Criteo model is trained on 8 40GB A100 GPUs using hybrid parallelism [219], and the production model is trained on a single 40GB A100. In the multi-node experiments, we apply 8-way intra-node data-parallelism for the production model.

Hardware Setup. In addition to the trainer machine, which is provided by default as a zero-cost node, we allocate additional instances to a resource pool. This pool consists of four instance types, chosen for their diverse specifications, costs, and varied processing speeds. Each machine type comes with 32 available instances. We configure the "pipeline budget" to $1500/day. Table 6.2 describes the hardware specifications and associated costs of each instance

Table 6.2. Group hardware configurations, prior to any autoscaling events which create new groups. Costs are estimated based on AWS 1-year reservations with public pricing in the `us-east-2` region. The p4d.24xlarge is provided by default, as the trainer box.

	Group 1	*Group 2*	*Group 3*	*Group 4*	*Group 5 (default)*
CPUs	96	128	96	128	96
CPU Max Speed (GHz)	3.1	3.5	3.6	3.5	3.0
Memory (GB)	768	1024	192	512	1152
Daily Cost ($)	97.54	136.23	57.58	91.04	451.93 (0)
AWS Instance	r5d.metal	r6id.metal	c5.metal	m6i.metal	p4d.24xlarge

type.

To simulate autoscaling events, and further node heterogeneity, half of the nodes in each group undergo a re-sizing procedure, halving then doubling the available CPU & memory (e.g., 128 CPUs $\rightarrow$ 64 $\rightarrow$ 32 $\rightarrow$ 64 $\rightarrow$ 128).

Since this only affects half the nodes in each group, the evaluated systems may easily *ignore* these re-sized machines and only use the original base instances. So, the impact of autoscaling will be much less than in the previous INTUNE experiments, where the baselines were forced to respond to resource changes. Here, re-sizing is presented as an opportunity for flexibility (i.e., with more candidate instance types) rather than an enforced resource shock.

Baselines. We use the following intra-node optimization baselines, each of which implements CPU resource optimization:

1. **Unoptimized.** In this version, only a single CPU is allocated per stage such that no parallelism is possible.

2. **AUTOTUNE.** `AUTOTUNE` is a standard TensorFlow offering, commonly used to optimize `tf.data` pipelines.

3. **AUTOTUNE-Adaptive.** We checkpoint and re-launch `AUTOTUNE` on re-sized instances to allow it to adapt to the new machine's resources.

4. **Plumber-Adaptive.** Plumber [98] is a more user-friendly alternative to AUTOTUNE that uses an MILP to distribute resources. We apply the same checkpointing approach as in AUTOTUNE-adaptive.

5. **Heuristic.** This version simply distributes CPUs evenly to emulate what a human user's best guess distribution might look like. INTUNEX's RL agent is initialized from this state as well.

These baselines cover most typical configurations, both manual and automated. We combine them with a straightforward greedy knapsack solver which maximizes total available CPU cores within the cost budget. This reflects a reasonable algorithmic baseline for multi-node optimization.

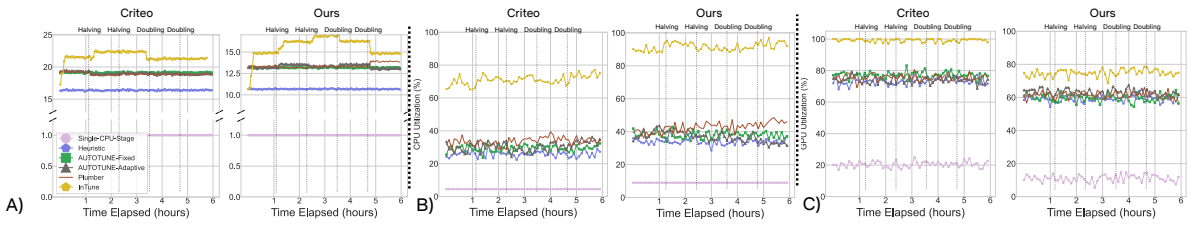

Figure 6.3. All figures use the legend in the leftmost chart. (A) Pipeline throughput over time for each approach in the multi-node experiments, normalized to the Unoptimized baseline. (B) Average CPU utilization across all budgeted nodes over time for each approach. (C) GPU utilization over time for each approach.

Pipeline Performance. Figure 6.3 (A) presents the results of our multi-node evaluations on the Criteo & Netflix pipelines. INTUNEX provides significantly better throughput and hardware utilization than the strongest competitors on both pipelines. Autoscaling events are less significant for the baselines in this setting, as they only affect half of the candidate instances. INTUNEX still benefits however, leveraging the increased flexibility offered by partial instances to optimize its cost-efficiency. It achieves peak marginal performance gains of 25% and 15% versus AUTOTUNE on the custom pipeline and the Criteo pipeline respectively.

Since the internal model runs on 8 GPUs in this setting, its throughput demands are considerable. Even so, INTUNEX sustains high GPU utilization rates across the job lifetime,

demonstrating its ability to support low-latency models & high throughput demands.

We also observe that INTUNEX still converges to a stable throughput rate relatively quickly, even with the multi-level RL formulation.

Drilldown Study. We now dive into INTUNEX's multi-node scaling behaviors on the real-world data pipeline. For single-node scaling studies, we refer the reader to the INTUNE [146] paper, which analyzes how group-agent performance scales as machine sizes change, pipeline complexity increases, and workloads are re-configured.

So, for the multi-node setting, we evaluate our system's performance with respect to the number of different instance types available in the node pool. The intention is to understand the effectiveness of our orchestration agent as the complexity of cost-efficiency problem space increases. As in the end-to-end experiments, we use AWS instances to define our instance types & budgeting. We take a selection of 20 different instances across the `c7a`, `r7i`, `r6i`, and `m7a` types. The budget is set to $1500 with 32 instances per type, as in the end-to-end experiments.

Figure 6.4 illustrates how INTUNEX's solution's cost-efficiency ratio scales as more instance types are added to the pool (in order of price). We see that INTUNEX monotonically improves its cost-efficiency ratio as the candidate space increases in size. This result implies that our RL-solver approach scales well even as the problem complexity increases, and should be capable of managing even the most diverse cluster scenarios.

6.4 Conclusion

DLRM training costs are often dominated by online data ingestion rather than model execution. The primary throughput bottleneck in this setting is *CPU-driven* data processing rather than *GPU-accelerated* model operations. Thus, optimizing the data ingestion phase is critical to ensuring cost- & time- effective model development. Unfortunately, existing tooling for DL data ingestion pipelines does not support the DLRM setting effectively. Our prior work — INTUNE — dynamically allocates CPUs & memory across stages of the online data ingestion

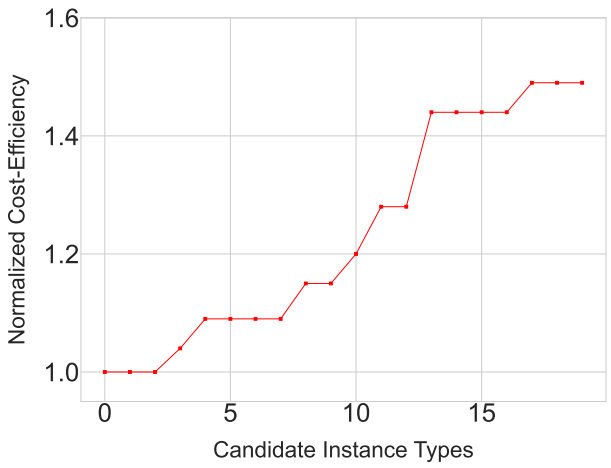

Figure 6.4. We report on cost-efficiency scaling (Equation 6.1) with respect to the degree of instance variety available in the cluster.

pipeline, significantly improving efficiency over industry-standard baselines without requiring changes to the cluster architecture or user workflows. But, for the multi-node setting, a new layer of optimization is necessary. Determining which nodes to use for a pipeline (and how many of each) is a challenging optimization problem. But, by extending INTUNE's RL formulation, we propose a new system — INTUNEX — that handles both inter-node and intra-node optimization. We find that INTUNEX can improve overall cluster efficiency on real-world workloads by as much as 15-25%, potentially unlocking significant cost savings for industry users.

Chapter 6 contains material from "Reinforcement Learning for Intra- & Inter-Node Recommender Data Pipeline Optimization" by Kabir Nagrecha, Lingyi Liu, and Pablo Delgado, which appears in ACM Transactions on Recommender Systems of the 17th ACM Conference on Recommender Systems, Volume 5. The dissertation author was the primary investigator and author of this paper.

Chapter 7

Routing Over LLMs Using Proxy Metrics for Relative Quality Estimation

In this chapter, we elaborate on our optimizations targeting LLM serving. LLMs demonstrate impressive zero-shot capabilities on many real-world tasks across a variety of domains. Unfortunately, adoption of state-of-the-art LLMs in enterprise applications is often bottlenecked by inference performance challenges. While using smaller LLMs may alleviate these bottlenecks, they usually produce lower-quality outputs. In this paper, we propose a routing procedure that uses simple lexical analysis techniques to assess the complexity of a query, then predicts the resultant quality differential between LLMs. We use the prediction to select a candidate LLM that will provide the best inference performance while respecting a user-configured quality degradation threshold. We evaluate our procedure on four different model families — LLaMA-2 [204], Qwen-1.5 [17], Vicuna [39], and Falcon [13] — over five diverse datasets representing challenging LLM settings — MT-Bench for human-like chat experiences, MMLU for broad language understanding, HellaSwag for natural language reasoning, HumanEval for code generators, and a conversational recommendation dataset from a streaming service. We also demonstrate that our approach is even effective for routing across models of different families. Our experiments show up to 6.92X improvements in inference performance while maintaining output quality close to the largest candidate LLM.

7.1 Introduction

Large language models (LLMs) have demonstrated strong performance on a variety of tasks, including chat services [29], code generation [34], and product recommendation. Recent studies have shown that overall response quality is closely correlated with *LLM scale*; larger models with more parameters typically perform better on standard benchmarks [240, 72, 34].

However, significant challenges arise when adopting such large-scale models. In particular, *inference performance* is strongly correlated with model size. Table 7.1 compares the performance achieved by differently-sized LLaMA-2 variants [204] on the MT-Bench workload.

Thus, users building performance-sensitive applications are forced into a trade-off between quality and performance. Opting to use a larger LLM may offer higher quality — but at the cost of lower serving throughput. A smaller LLM might offer the opposite trade-off.

Table 7.1. Throughput achieved by LLaMA-2-7B, -13B, and -70B on the MT-Bench workload.

MODEL	THROUGHPUT (QUERIES/SECOND)
LLAMA-2-7B	7.02
LLAMA-2-13B	2.96
LLAMA-2-70B	0.97

Many recent papers [33, 189, 196, 175, 247, 248] explore ways to navigate this trade-off by assigning different queries to different LLMs. Each of these works takes some approach to predict whether an LLM's output will satisfy a quality threshold for a given metric and domain, i.e., by training a model or function to produce a "reliability" score for the metric. To simplify prediction, most such works (barring a few exceptions, e.g., FrugalGPT [33] & AutoMix [126]) focus on tasks with straightforward evaluation mechanisms such as classification and multiple-choice Q&A.

The key to such routers is the ability to predict the quality of an LLM output on a given

metric. As such, these routers were built and evaluated for a relatively small set of straightforward, specialized tasks with simple metrics.

But LLMs have introduced a range of new applications [203, 53], e.g., content creation [107], chatbots [29], and code generators [34]. Holistic LLM evaluation for these broad tasks is non-trivial and may require complex procedures (e.g., human scoring or sandboxed code execution).

Such tasks were well-studied even before the advent of LLMs; the metrics are easy to reason about and do not need to account for the *unique capabilities* of LLMs. Predicting outcomes on such metrics and tasks with reasonable confidence may be feasible. But such tasks do not leverage the versatility, generalizability, and creativity of LLMs [53, 203], which has enabled many new real-world applications for deep learning (DL) with complex, ambiguously-defined metrics. These LLM-specific applications include (but are not limited to) interactive chatbots [29], content creation tools [107], malware analyzers [134] and detectors [58], conversational recommenders, code generators [34], and roleplay gaming assistants [246].

Building metric-predictors for such tasks is significantly more challenging. For example, if a predictor could forecast the outcome of LLM-generated code, then it could be used to address the halting problem [44]! The search for *holistic LLM evaluation* approaches has motivated a variety of complex, empirically-grounded analysis mechanisms, e.g., LLM-as-a-Judge for chatbots [240], or execution sandboxes for code generation [34]. Due to the complexity of these approaches, routers which predict the absolute quality of LLM outputs on evaluation metrics cannot branch out to support these new and emerging settings.

To better illustrate the problem, we now describe a realistic example where a routing mechanism is needed — but the results of evaluation are hard to predict.

Illustrative Use Case. Consider an ML engineer working at a large Web company. The company is building a product to help writers generate manuscripts and creative texts. They anticipate tens of thousands of monthly active users. The engineer has been tasked with hosting an LLM for the application. They are given a service-level objective (SLO) of serving 100 queries

per second. Researchers at the company have fine-tuned 3 LLaMA-2 instances, LLaMA-2-7B, -13B, and -70B. The 70B instance is "state-of-the-art"; it is the absolute best model available for this task.

Initially, the engineer hosts this state-of-the-art model. Analyses show that users typically accept about 40% of LLM-generated texts and report an 80% satisfaction rate. The serving throughput is only 30 queries per second, however.

Thus, for performance reasons, the engineer must explore the trade-off between model quality and throughput. Internal studies suggest that users will still be satisfied with a 10% degradation in output quality (i.e., a 36% acceptance rate) relative to their current experience. A/B-tests with the 7B & 13B models show 28% and 32% acceptance rates respectively. A *routing* approach may help navigate the trade-off between performance and quality more effectively.

We note two key ideas illustrated in the example that help inform our approach. First, *the quality of the user experience is bounded by the best LLM available*. In the previous example, the defined tolerable acceptance rate of 36% is not set in a vacuum; it is defined *in relation* to the best & largest LLM candidate (and the best-achievable output quality). Second, on a per-query basis, it is difficult to predict user interactions with creative texts (thus making existing LLM-routing systems inapplicable). However, it may be feasible to predict the *relative quality* of the 13B-parameter model's text versus the 70B-parameter model.

Contributions and Key Idea. To obviate the difficulty of approximating and projecting potentially complex domain-specific metrics (e.g., the acceptance rate in our content creation example), we suggest an alternate approach to routing. Rather than predicting LLM output performance or reliability, we analyze the query complexity to predict *the relative quality difference* between a candidate LLM and the best-baseline LLM, and use the estimated degradation to inform routing decisions. Thus, we shift the focus of the predictor from the *metric* (which may be ambiguous or hard to predict performance on) to assessing the *relationship* between LLMs. This re-framing lets us simplify and generalize routing to cover more complex tasks with

challenging metrics. Essentially — simplicity of method enables complexity of application.

In this paper, we derive straightforward proxy metrics and demonstrate that they are strongly correlated with quality degradation between LLM instances. We use these correlations to construct a simple threshold-based routing mechanism that assigns input queries across different candidate LLMs. We apply our approach to five diverse tasks with varied and complex metrics and demonstrate consistently strong performance across different domains and evaluation procedures. We use MT-Bench [240] with its LLM preference scoring, the MMLU [73] (5-shot) benchmark which covers a wide range of tasks including social sciences, STEM, and language comprehension, HellaSwag (10-shot) [233] for reasoning, HumanEval [34] for coding, and a conversational recommendation dataset taken from a streaming company with a Mean Reciprocal Rank (MRR) score. We compare to strong cascade baselines for inference routing optimization on complex workloads — FrugalGPT and AutoMix — as well as some predictive routers which directly select a model to use. We show that our simple predictive router is more performant than the cascade mechanisms, and also more accurate and capable than existing predictive mechanisms.

In all cases, across these diverse target metrics, workloads, and datasets, our simple and straightforward routing approach demonstrates that it can significantly improve inference performance while respecting a user-configured threshold for relative quality degradation.

1. We demonstrate that simple lexical analysis proxy metrics are strongly correlated with output quality differences between LLMs across a diverse set of tasks.

2. We exploit this insight to construct a routing mechanism that applies the proxy metrics to compute estimated accuracy degradation between model instances.

3. We apply this routing procedure to varied workloads and applications and achieve significant performance gains with minimal accuracy losses, pushing into a new region beyond the existing Pareto-frontier of quality-performance trade-offs.

7.2 Background

We provide definitions for terminology used throughout this paper. We assume that the reader is familiar with deep learning (DL) in general, and omit discussion of ubiquitous concepts (e.g. gradient descent, etc).

Query Routing is well-known in the context of P2P networks [197] & databases. These systems *distribute* content over different servers, rather than simply replicating data. Similarly, CDNs [127] route queries based on server "preferences", loads, and estimated latencies.

Throughput & Latency are used to describe the performance behaviors of serving systems. Throughput refers to the number of queries can be serviced in a fixed period of time, while latency refers to the turnaround time for a single query. Average latency scales inversely with throughput; higher serving throughput typically yields lower average latency. In the LLM context, latency may either refer to full-query latency (i.e., the time for a full response to be returned) or time-to-first-token, which is more relevant when the output is streamed to the client. For the purposes of this paper, we focus on the former case.

LLM Inference Challenges are increasingly becoming the bottleneck in real-world adoption. Transformer decoders are autoregressive [211] leading to sequential dependencies that are hard to optimize at a low-level. Thus, users are often motivated to employ smaller LLMs when they are performance-constrained.

7.3 Problem and Motivation

Objective. For a given query q, our aim is to select and apply the model m with the highest inference performance that meets some user-defined quality-degradation threshold ε on their task metric μ. Referring back to our illustrative use case in Section 7.1, we want to enable the engineer to select which LLaMA-2 model should serve a given query while maintaining output quality within 10% of the 70B-parameter instance.

Formally, we seek:

$$m_{opt}(\varepsilon) = \underset{m \text{ s.t. } \mu(m(q)) \text{ meets } \varepsilon}{\arg\max} \mathrm{P}(m,q) \tag{7.1}$$

where P refers to inference performance.

The key challenge in this optimization problem is assessing the constraint: $\mu(m(q))$ meets ε. Computing μ in the online setting is generally infeasible, since it may require access to a ground-truth label or an expensive verification process (e.g., for code generation).

The standard procedure, employed by existing frameworks, is to first train some verifier/scorer τ as a proxy for μ, then to transform the optimization problem as follows:

$$m_{opt}(\varepsilon) = \underset{m \text{ s.t. } \tau(m(q))>\varepsilon}{\arg\max} \mathrm{P}(m,q) \tag{7.2}$$

Training and defining τ is a non-trivial task. Cascade systems [175, 196, 126, 33] produce outputs and apply some verifier (e.g., a small multi-layer-perceptron or kNN) to use as τ. When μ is more complex (e.g., for chatbot workloads), correspondingly sophisticated output verifiers are necessary. FrugalGPT [33] uses DistilBERT (a 66M parameter Transformer), and AutoMix uses a self-verification procedure [120] combined with a Markov Decision Process [126, 92]. In all cases, the transformation requires invoking $m(q)$ and $\tau(m(q))$ (potentially for every single candidate m) to compute the arg max function.

One recent paper [189] suggests using a kNN to predict quality on $\mu(m(q))$ to inform routing without actually invoking $m(q)$. But as we discussed in Section 7.1, this relies on μ being easy to predict — their work focuses primarily on tasks such as classification, multiple-choice Q&A, and single-word responses. Even then, large amounts of data ($O(100K)$) are needed to train the k-NN.

Our aim is to substitute the constraint so that we no longer rely on the relationship $\tau(m(q)) \approx \mu(m(q))$ or even a predictor for μ. This lets us avoid computing $m(q)$ and sidestep challenges faced in applications with more complex and nuanced μ metrics (e.g., code generation) where the metric is difficult to predict directly.

Approach. We exploit and re-frame the semantics of ε in the objective function. Suppose we have a candidate set of models, e.g., LLaMA instances, as in our earlier example. If we select some baseline model from the candidate set m', e.g., LLaMA-2-70B, then we can define the constraint as follows:

$$m_{opt}(\varepsilon') = \underset{m \text{ s.t. } \frac{\mu(m(q))}{\mu(m'(q))} > \varepsilon'}{\arg\max} \text{TP}(m,q) \tag{7.3}$$

where ε' is now a relative degradation measure between LLaMA-2 instances rather than an absolute quality threshold. We expand on how to set ε' later in this section.

While μ might be defined in a way that makes it difficult to approximate — e.g., a human- (or LLM-) evaluated quality score, a set of unit tests for code generation — measuring relative quality degradation only requires predicting the *relationship* between two differently-sized LLMs on the given task. Indeed, previous works [76] have already found correlations between model scale and output quality; we simply exploit this relationship. In other words — while $\mu(m(q))$ and $\mu(m'(q))$ are hard to predict, $\frac{\mu(m(q))}{\mu(m'(q))}$ can be informed by the relationship between m and m'. So, we introduce an approximation function Θ for $\frac{\mu(m(q))}{\mu(m'(q))}$.

Modeling $\mathbf{q} \sim \frac{\mu(\mathbf{m}(\mathbf{q}))}{\mu(\mathbf{m}'(\mathbf{q}))}$. We now consider ways to define the approximation function $\Theta(m,q)$.

Deriving a theoretical solution to $\frac{\mu(m(q))}{\mu(m'(q))}$ is non-trivial. Instead, we look to empirically evaluate candidate metrics c such that $c(q) \sim \frac{\mu(m(q))}{\mu(m'(q))}$. These metrics can be straightforward yet still serve as useful approximations. Figure 7.1 illustrates how simply bucketing input queries by length in characters can lead to different degradation outcomes on the MT-Bench evaluation workload.

In Section 7.4, we provide an empirical study of correlations between simple lexical analysis metrics and $\frac{\mu(m(q))}{\mu(m'(q))}$. This gives us functions c such that $c(q) \sim \frac{\mu(m(q))}{\mu(m'(q))}$, providing a basis for our Θ function. In Section 7.5 we leverage these mechanisms to optimize LLM serving for a broad and diverse set of real-world applications (a chatbot task, a code generation assistant,

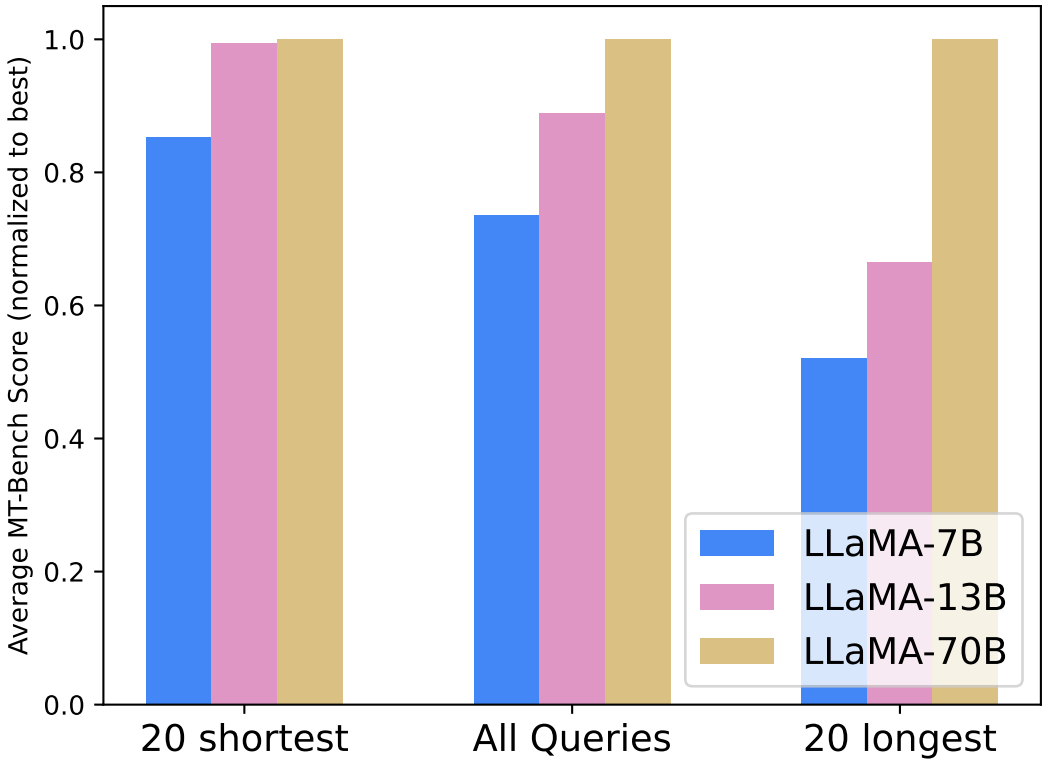

Figure 7.1. Comparison of average scores (normalized to best) across differently-sized LLaMA-2 variants on the MT-Bench task. The relative performance changes as we filter queries by length.

and a conversational recommender system) to assess whether these correlations are consistent across domains and complex metrics.

Applying Θ for routing. Deriving Θ lets us redefine our objective function as follows.

$$m_{opt}(\varepsilon') = \underset{m \text{ s.t. } \Theta(m,q)>\varepsilon'}{\arg\max} \mathrm{P}(m,q) \tag{7.4}$$

In the worst case, this procedure invokes $m'(q)$ (the biggest/slowest model). Compare this to the worst-case for cascade mechanisms, which may invoke *all* candidate LLMs. We compare the performance in Figure 7.2(A).

Quality Constraint. Our routing procedure optimizes performance, but must respect a

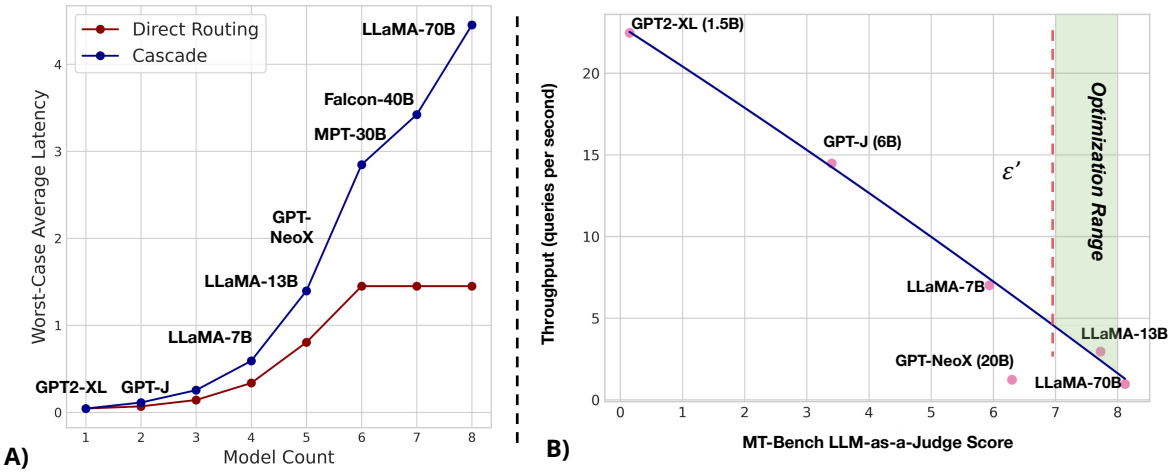

Figure 7.2. (A) Worst-case average serving latency (i.e., all queries need largest model) for a direct-routing approach versus a simple cascading approach. Models range in from size from 1.5B to 70B parameters. We do not include verifier overheads (e.g., DistilBERT invocation for FrugalGPT's cascade [33]). MPT-30B is the slowest LLM to serve, so the performance of direct routing flattens past that point. (B) Pareto-frontier between performance on MT-Bench and throughput. We plot models of sizes ranging from 1.5B parameters to 70B. Experiments run on 8 A100 GPUs. The target optimization region on this frontier is bounded by ε'.

quality constraint; application developers leveraging LLMs will likely have SLOs for both.

To illustrate, we describe a use-case taken from our experiences with the proposed routing mechanism at a large streaming company. Initially, the company ran a limited test of a large-scale LLM for a chatbot use case. They recorded the current output quality based on user interactions and feedback. Assessments showed that the throughput was insufficient for a broader deployment, but some $x\%$ of quality degradation could be tolerated without significantly affecting user satisfaction. The degradation induced by switching to a smaller LLM for all queries would have exceeded the tolerance level, however. Thus, a routing mechanism was needed.

Figure 7.2(B) illustrates how a quality degradation threshold could be overlaid to constraint our optimization over the Pareto-frontier. We expose the knob ε' from our optimization problem, so that users can configure the threshold to meet their quality needs. Thus, in our real-world example, inference performance was improved considerably while still maintaining

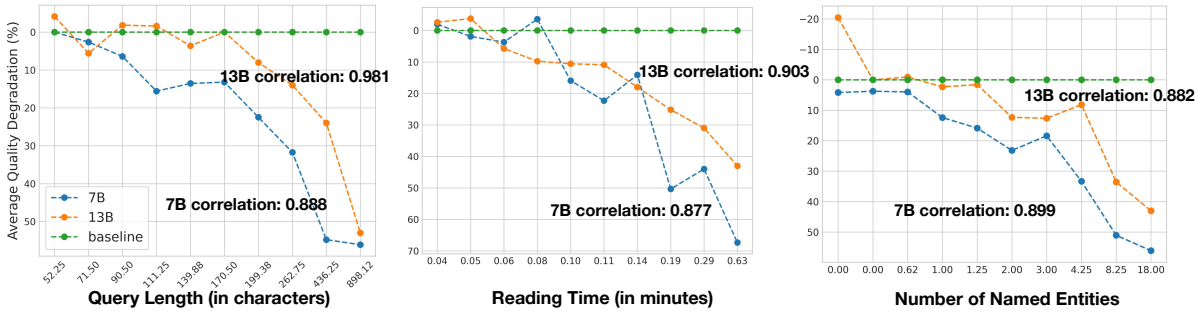

Figure 7.3. We bucket queries according to 3 different lexical analysis metrics and assess how correlated they are with accuracy degradation. We also report the Pearson correlation coefficient between the metric and the quality degradation from the 70B baseline. Y-axes are inverted to make the curve easier to understand visually.

quality degradation $< x\%$.

7.4 Approximation Metrics and Routing

Equations 7.1 and 7.4 in the previous section defined our overall objective and our proposed approximation.

Now, we look to find an appropriate function Θ that is strongly correlated with $\frac{\mu(m(q))}{\mu(m'(q))}$. For clarity, we focus our explorations in this section on the MT-Bench [240] single-turn workload, using GPT4-as-a-Judge as the evaluator. This exemplifies a metric that is challenging to approximate via some simple predictor, as explained in Section 7.1 and Section 7.3. In Section 7.5 we expand our approach to other datasets to demonstrate consistent performance across diverse domains/metrics.

We use three LLaMA-2 model instances of varying sizes (7B, 13B, and 70B) as our candidate model set M. This is reflective of typical practice in industry settings; a company might consider a few such state-of-the-art models for their serving needs.

7.4.1 Proxy Metrics

We evaluate three different simple, easy-to-compute lexical analysis metrics, implemented using the LFTK [105] library, and assess their correlation with quality degradation

between LLM instances. The metrics are presented below.

We consider the following three metrics:

1. **Query Length.** This simple character count metric is easy to compute and can serve as a simple measure of query complexity.

2. **Reading Time.** A basic reading time formula provided by the LFTK library. This can serve as a measure of lexical complexity.

3. **Number of Named Entities.** References to persons or places can imply complex logical reasoning tasks or queries that draw on general knowledge. So, this metric can help inform our understanding of query complexity.

These metrics are both straightforward and easy to compute. Applying them at routing time would induce minimal overheads. Other metrics are certainly possible, but we leave further metric exploration to future work. Our focus is demonstrating the feasibility of estimating $\frac{\mu(m(q))}{\mu(m'(q))}$ on complex tasks.

From a modeling standpoint, these metrics are well-motivated. Larger LLMs are known to be more capable of keeping track of increased logical complexity [76, 54]; metrics such as query length and the number of named entities can serve as proxies for complexity, thus estimating a potential driver of performance gaps between models.

We rank input queries according to each metric, distribute them into quantiles, then compute the Pearson correlation coefficient between the metric and quality degradation (i.e., $1 - \frac{\mu(m(q))}{\mu(m'(q))}$ where m' is the largest model) across quantiles.

Figure 7.3 illustrates the results of correlating these metrics with quality degradation between LLM instances. Note that in some edge-cases, 7B and 13B models may produce marginally better results than the 70B baseline.

We find that all three metrics — query length, reading time, and number of named entities — are strongly correlated with the response quality degradation relative to the LLaMA-2-70B

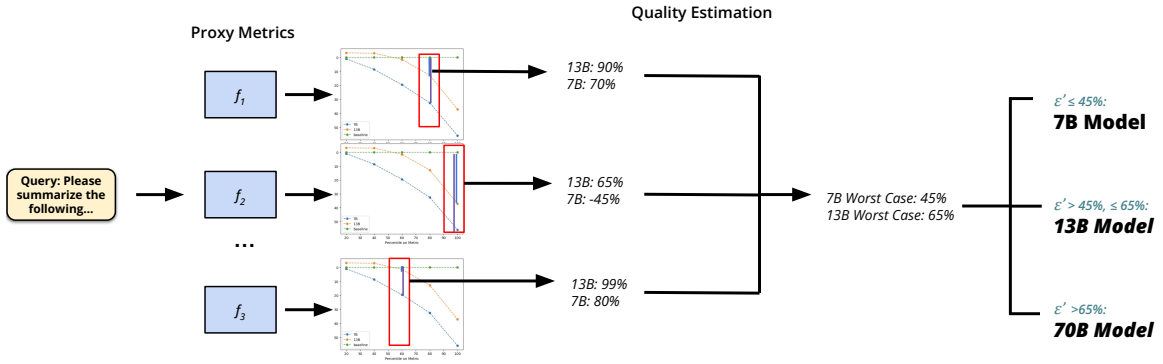

Figure 7.4. Illustration of the routing procedure applied to an incoming query to determine which candidate model to use.

model baseline.

We now define a procedure to use each metric $c(q)$ to create our approximation function $\Theta(m,q)$. Given a query q, we compute our metric $c(q)$ and map it into a quantile on our curve correlating $c(q)$ with $1-\frac{\mu(m(q))}{\mu(m'(q))}$. We then take the $\frac{\mu(m(q))}{\mu(m'(q))}$ value associated with the upper-bound of the quantile, providing an upper-bound estimate of quality degradation.

This assumes the curves for c are available in advance. We relax this assumption in Section 7.4.3, but for now, we presuppose that we have access to an oracle that can provide these curves.

7.4.2 Routing Mechanisms

We now describe and evaluate our routing scheme.

Θ-Ensemble. Previously, we demonstrated some candidate metrics c and defined a procedure to compute corresponding Θ estimators. Using an ensemble of Θ estimators can mitigate the risk of an inaccurate routing decision. Equation 7.5 presents this new formulation. Later, in Section 7.5 we provide an ablation study on the ensemble mechanism.

$$m_{opt}(\varepsilon') = \underset{m \text{ s.t. } \Theta_i(m,q)>\varepsilon' \ \forall \Theta_i \in \{\Theta_0,\Theta_1 \ldots\}}{\arg\max} \mathrm{P}(m,q) \tag{7.5}$$

Since each Θ-function is computationally light, computing 3 together should still not

induce significant overheads.

Routing Procedure. To solve Equation 7.5 we must compute the arg max function, which requires iterating over the model list M. We sort the models in order of size (which serves as a proxy for $\mathrm{P}(m,q)$). For each model, we compute $\Theta(m,q)$. If the constraints are satisfied, we compute $m(q)$ and return the response.

The largest model will yield 100% quality for every Θ function, since our degradation estimate is computed in relation to this model. We constrain the user-defined ε' threshold so that it cannot exceed 100%; thus, we guarantee that we will find *some* viable candidate model to serve the query.

Algorithm 5 presents our routing procedure in pseudocode, and Figure 7.4 illustrates an example routing process.

Algorithm 5: Θ-Proxy Routing Procedure

```
for m ∈ M do
  for Θ ∈ Θ_0, Θ_1, ... do
    if Θ(m,q) < ε' then
      skip to next m
    end if
  end for
  return m, m(q)
end for
```

In effect, this algorithm iteratively computes and moves along the Θ-approximated *Pareto-frontier* between performance and quality for the given query and proxy metric. It then returns the point along the frontier that satisfies the quality requirement across all metrics.

To verify this procedure, we evaluate on MT-Bench, setting ε' to 85%. Figure 7.5 illustrates the results. We find that our router can re-direct 64.4% of queries away from the largest model while still maintaining an output quality within 5% of the largest-model-only baseline. The actual quality degradation is significantly lower than the tolerance threshold we set; this is because our Θ-function routing procedure estimates an *upper-bound* on quality degradation.

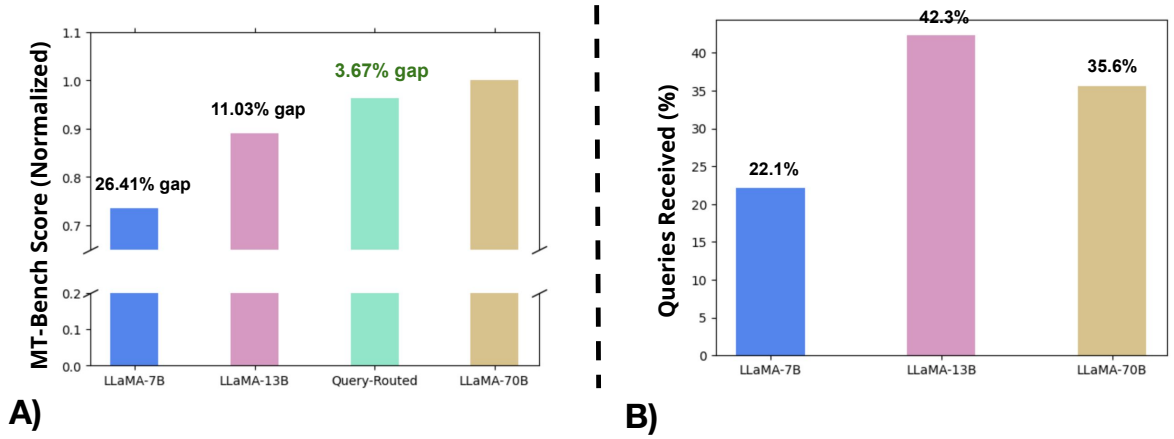

Figure 7.5. (A) Normalized MT-Bench scores of LLaMA-2 baselines model as well as our multi-model routing mechanism. (B) Distribution of query assignments made by our our router.

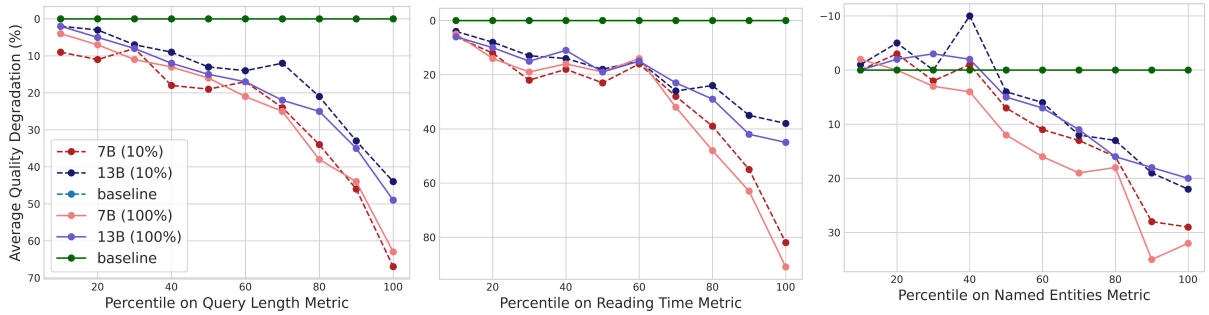

Figure 7.6. Curves observed using a random subset of the prompts versus the curves from the full dataset.

7.4.3 Sample-Based Estimation

We previously assumed access to the correlation curves between proxy metrics and quality degradation. To relax this assumption, we use a straightforward approach used in other systems works [186]: we construct the curve based on a random sample of queries that might be collected online or from some pre-existing deployment.

To validate our approach, we use a large-scale real-world dataset for a conversational recommender system at a major streaming service. The dataset is $O(10k)$ in scale. We use 10% of the prompts to construct our curve. The relevant evaluation metric is MRR. Figure 7.6

presents the curves from the subset and contrasts them against the curves from the full dataset. Qualitatively, the curves are similar — the sampling mechanism is effective. Future iterations could look to eliminate even this data-sample requirement.

7.5 Experiments

7.5.1 End-to-End Experiments

We now run a series of experiments to compare our work to existing serving options. We evaluate whether our proposed routing techniques materially shifts the Pareto-frontiers (and thus, the feasible and desirable serving options) for performance-constrained LLM application developers.

Workloads. We use four workloads taken from a diverse range of LLM-based applications. First, a chatbot workload — we use the MT-Bench evaluation scheme here. Second, a general evaluation of broad LLM capabilities, with a 5-shot MMLU evaluation. Third, an evaluation of reasoning capabilities and logic, with a 10-shot HellaSwag evaluation. Fourth, an internal conversational recommender system (CRS) workload taken from a major streaming service. We use a recommendation metric, MRR, to score this workload. A fifth workload — HumanEval, for code generation — is used in a separate experiment for code-specific models as well. This diverse set of workloads, metrics, and datasets can help assess the generalizability of our approach across domains.

MT-Bench and HumanEval are relatively small datasets, while the others include several thousand prompts. For the large-scale datasets, sampling a small subset for curve construction is relatively easy; we use roughly 10% of the datasets to define our curves or train any relevant routers/predictors used in the baselines. For the smaller datasets, we use 50% of whole data to construct the curve, since a smaller sample would induce excessive variability.

Baselines. We compare our approach to three single-model baselines and two state-of-the-art cascade baselines. We enumerate them below.

1. **Single Model X.** These simple baselines route all requests to some singular model X.

2. **FrugalGPT (Cascade).** This baseline adapts the cascading design from FrugalGPT [33] and applies it to the models used in the simple baselines. We do not include FrugalGPT's orthogonal optimizations (e.g. caching, etc) since they could also be combined with our routing mechanism. We train the verifier (and LLM cascade selector) on the same subsets of data we sample for our routing mechanism's curve construction.

3. **AutoMix (Cascade).** AutoMix [126] is a recent work that uses a mixture of self-verification [81] and a Partially Observable Markov Decision Process [92] as its verifier. AutoMix was originally benchmarked for the two-model case; it is straightforward to extend their proposed cascade chain to cover 3 models, though.

4. **K-NN (Predictive).** A recent work [189] proposed using a k-NN predictive algorithm for direct routing. This can serve as a more direct comparison to our own simple routing procedure.

5. **MLP (Predictive).** Another recent paper [78] suggests using an MLP for prediction, directly inspired by the paper mentioned in the previous baseline. This serves as another direct comparison to our procedure.

All baselines use 8-A100-40GB nodes, one per model replica (for simple baselines) or model instance. The cascade mechanisms, e.g., FrugalGPT and AutoMix were designed to navigate the trade-off between output quality and the *cost* of LLM API invocation, not inference performance. But they are the state-of-the-art for model serving selection on LLM workloads, and thus the main options practitioners can use today. So, we consider them necessary and useful baselines to show how cascade mechanisms compare to direct routers.

Models. In order to assess whether our techniques generalize across different model families, we use 4 different model families, each of which provides 3 or more differently-sized instances. We enumerate them below.

Table 7.2. Reported throughput lift achieved by our routing approach relative to the approximate Pareto-frontier of the single-model baselines.

	LLaMA-2	Qwen-1.5	Vicuna	Falcon	All	**Average**
MT-Bench	1.8X	3.28X	1.7X	3.98X	N/A	**2.69X**
MMLU	1.92X	4.77X	1.97X	6.92X	4.55X	**4.02X**
HellaSwag	2.8X	2.42X	1.67X	3.68X	2.31X	**2.58X**
CRS	2.98X	3.64X	2.08X	3.41X	3.39X	**3.31X**
HumanEval	1.65X	N/A	N/A	N/A	N/A	N/A
Average	**2.23X**	**3.14X**	**2.18X**	**4.49X**	**3.41X**	**3.15X**

1. **LLaMA-2** (7B, 13B, 70B).
2. **Qwen-1.5 [17]** (7B, 14B, 72B).
3. **Vicuna [39]** (7B, 13B, 33B).
4. **Falcon [13]** (7B, 40B, 180B).

We also assess *cross-family* routing, by including a combined baseline where all models across all 4 families are used.

Results. Figure 7.7 presents the Pareto-frontiers we observe over the simple baselines, then overlays the scores achieved by the algorithmic baselines (including our own). Table 7.2 presents the aggregated performance gains across the workloads. We assume an online application with a data stream, and report performance as average throughput/node.

Our approach moves significantly beyond the Pareto-frontier. With ε' set to 85%, we maintain the quality requirement relative to the large-model baseline on all workloads, while achieving considerable speedups on all workloads. Our approach outperforms all routing and cascade baselines as well.

To demonstrate our technique's effectiveness on non-natural-language tasks, we also run a benchmark for a code-specific model series (CodeLLaMA) on the HumanEval dataset, with

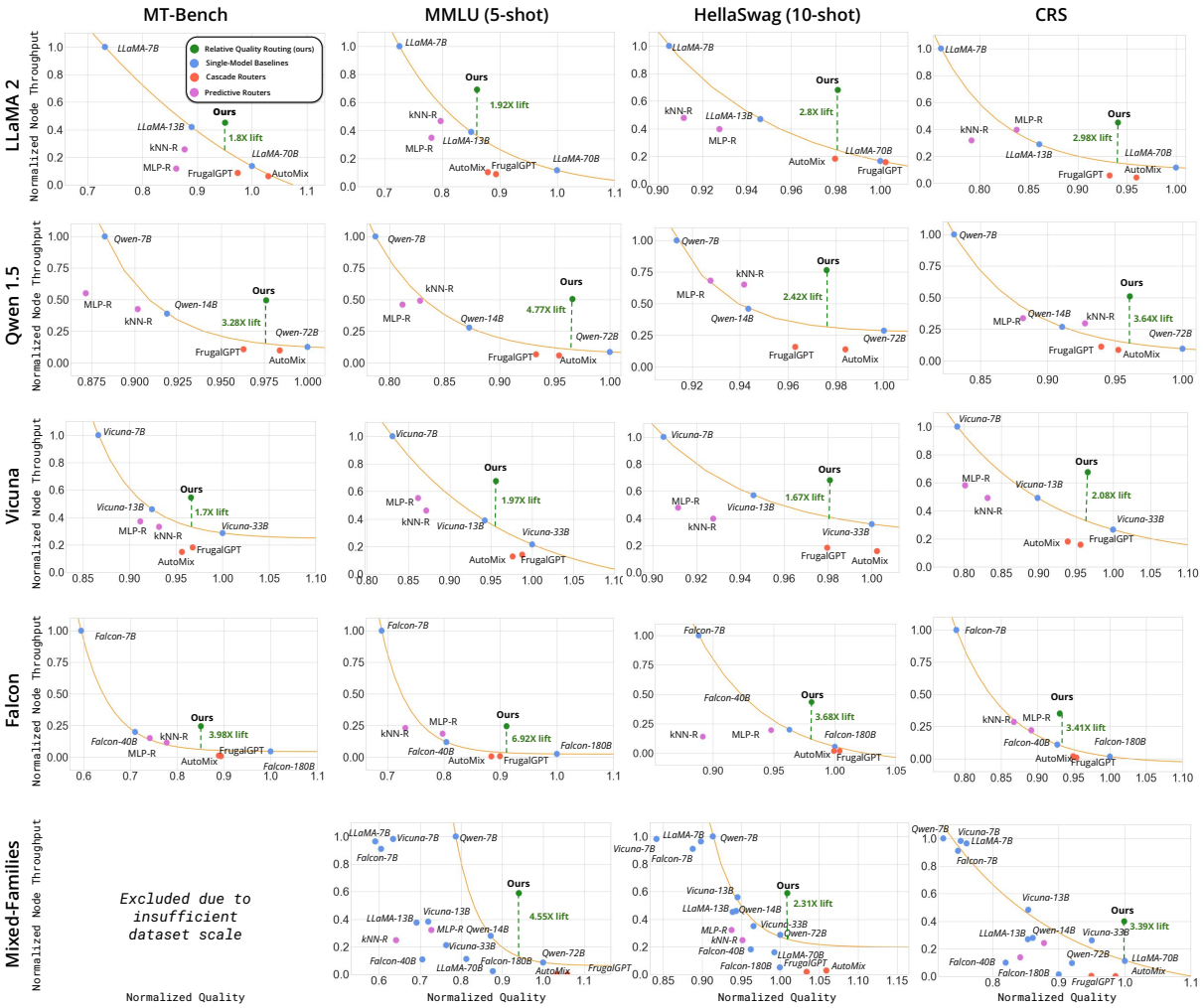

Figure 7.7. We chart how our approach compares to algorithmic baselines on the Pareto-frontier of quality-performance trade-offs in routing. We exclude the all-model baseline for the MT-Bench workload, since it has too few questions (80) to effectively train the cascade mechanisms or predictive router baselines for a mix of many different models.

the Pass@1 metric. Figure 7.8 illustrates the results.

7.5.2 Ablation & Scaling Studies

To better understand the behaviors of our approach, we run an ablation study on our Θ-ensemble and a scaling study. For simplicity, our scaling and ablation studies will focus on routing with the LLaMA-2 family [204].

Ablation Study. We drill-down into the Θ-ensemble of proxy metrics to better understand

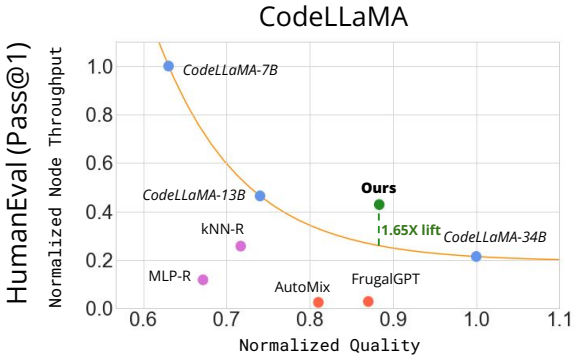

Figure 7.8. We apply our approach to a domain less strictly tied to natural language expression — coding. Our results show that relative quality estimation still works effectively for programmatic text and coding tasks.

how increasing the number of proxies affects the relationship between the ε' threshold and observed quality. We chart out ε' versus actual evaluated quality in three cases — one where ε' has to be satisfied for all three Θ-constraints, one where it has to satisfied for two, and one where it only has to be satisfied for one. Figure 7.9 illustrates.

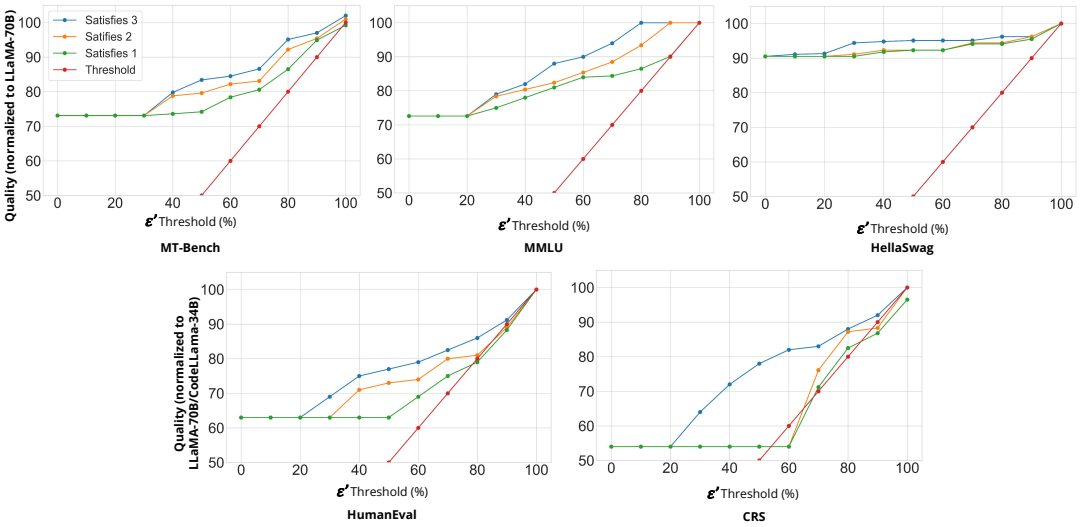

Figure 7.9. Ablation tests on our five workloads demonstrating how the relationship between the accuracy threshold and the actual observed accuracy shifts depending on the number of proxies involved.

We see that more proxies induces more conservative routing decisions to better respect the ε' threshold. The rate of accuracy improvement in relation to ε' increases considerably with

each additional proxy.

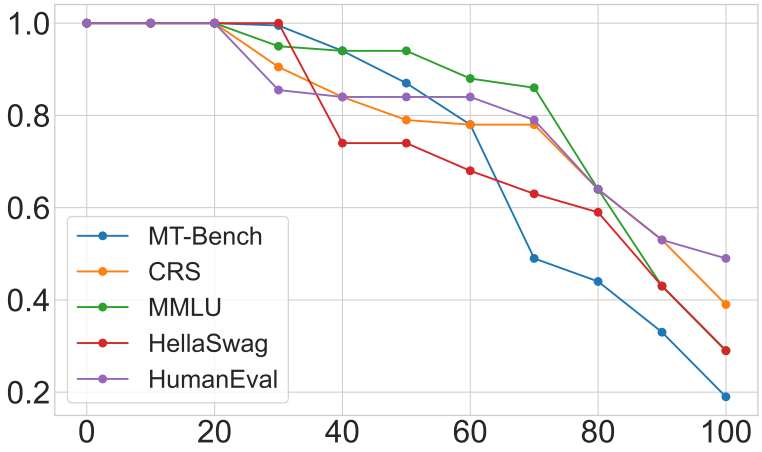

Figure 7.10. We chart normalized throughput against the ε' threshold to demonstrate how the knob can tune performance.

Scaling Study. Next, we study how altering the relative degradation threshold ε' affects the throughput of our system. We chart out throughput versus threshold on all three workloads, normalized to the fastest baseline (LLaMA-Small). We observe a downward curve; as the threshold increases and the router introduces more "large-model" invocations for higher quality guarantees, the achieved inference performance degrades. We can see that ε' serves as an effective knob for users to navigate the quality-performance trade-off.

7.6 Conclusion

LLMs have enabled many new and exciting applications of DL. But evaluating LLMs in such applications can require complex, non-standard metrics. This presents a challenge when designing *routing* mechanisms to distribute queries over LLMs; predicting output quality on such metrics is non-trivial. We present a new approach to LLM routing that predicts *relative degradation* between LLM instances rather than the actual metric scores. We demonstrate that simple lexical analysis techniques can be used to estimate this LLM-to-LLM relationship. We use

this to build a mechanism to route queries over candidate LLMs. Evaluations on three workloads show up to 6.92X throughput improvements versus a large-model baseline with $< 12\%$ quality degradation. As more users leverage LLMs in performance-sensitive applications (e.g., web search, ad serving), optimizing inference will likely be critical to broad adoption. We believe our work, which helps users navigate performance-quality trade-offs more effectively, can help address this challenge. In the future, we could look to make our system more end-to-end by *automating* proxy metric-selection based on the application, and also expand our studies to cover more emerging LLM use cases. We can also look to support more settings, e.g., offline scoring, where workload distribution and straggler management become important.

Chapter 7 contains material from "Routing Over LLMs Using Proxy Metrics for Relative Quality Estimation" by Kabir Nagrecha, Arun Kumar, and Hao Zhang, which is currently under submission. The dissertation author was the primary investigator and author of this paper.

Chapter 8

Related Work

8.1 Related Work for HYDRA

Pipeline parallelism: State-of-the-art pipeline parallel tools [82, 70] build on top of model parallelism by treating the sequence of shards as a staged pipe. They exploit the fact that DL training uses mini-batch SGD, wherein mini-batches are independent samples of the training dataset, and so stage out successive mini-batches through the pipe of shards to reduce idling. Prior work on pipelining has explored the problem of "bubble" periods during which the pipe has to be flushed to avoid collisions between mini-batches moving in opposite directions [56, 229, 115, 153].

One approach suggests mitigating the issue by creating "asynchronous" stages that use stale parameters for updates without executing backward passes of mini-batches in synchronous order, rescheduling them to reduce the number of flushes that need to occur in total [70, 153, 56, 229]. However, asynchronous pipelining introduces accuracy degradation and cannot guarantee convergence to the same degree of accuracy as the non-parallelized model [15, 119, 139, 200].

Data Parallel Hybrids: Techniques like ZeRO [173] apply parameter sharing to distribute memory costs across model instances, while still employing data parallel distribution of the input batches. This reduces the memory demands per GPU, but at the cost of frequent cross-device synchronization. In cases when the model is very large, these frameworks often need

to be hybridized with other techniques. For example, ZeRO-Offload and L2L [178, 167] propose offloading data from GPU memory to DRAM to improve scalability GPU. ZeRO-Offload runs parameter updates on the CPU instead of the GPU, and L2L executes a single layer at a time in a quasi-spilling approach. However, these systems are not scoped to address multi-model training, unlike HYDRA.

Offloading Systems: SwapAdvisor [79] is a tensor offloading system that allows data (parameters, gradients, intermediates) to be swapped out of GPU memory and onto DRAM. This enables large models to be trained on a single GPU. Unlike HYDRA's flexible shard-swapping, SwapAdvisor swaps at an individual tensor level with a complex swapping plan optimized across the entire model's execution. The degree of optimization prevents easy generalization to SHARP's multi-model blended scheduling. In addition, SwapAdvisor's swap plan simulator can introduce overheads due to the sheer size of the optimization space, and the communication latencies introduced by swapping can be costly. Another data offloading framework [132] proposed offloading only intermediates activations produced during training that would not be necessary until later in the model's execution. Such offloads are only possible along long-term residual connections that span several layers' worth of execution time, and the design also requires that model parameters fit on GPU memory. Both these frameworks could be hybridized with HYDRA to enable larger shards to be trained, but this would increase system complexity substantially.

Tensor parallelism: In contrast to inter-layer model parallelism, tensor parallelism offers the ability to parallelize compute between model shards. Rather than splitting a model into different layer groups, it splits individual layers into multiple pieces that can be run in parallel [187, 89]. However, tensor parallelism introduces substantial overhead in splitting tensors and recombining them, as well as challenges around efficient data communication. Moreover, not all layers can be split in such a fashion — convolutional layers are among the few that can be tensor-parallelized easily. In the future, HYDRA could be extended to support mixed

layer-tensor parallelism, but for now we only address inter-layer model parallelism.

Model Memory Reducers: Techniques to reduce the memory footprints of large architectures have received much attention in DL systems [36, 61, 101, 85, 87]. Model quantization [84] in particular has been a popular technique for reducing memory footprints at inference time. The goal of such systems is *orthogonal* to HYDRA, and memory reduction techniques could be integrated into the overall architecture in the future. Other work on machine teaching [215] and data distillation [217] aims to minimize the memory footprints of data, but these techniques address a different aspect of memory in DL systems.

Multi-query optimizations for DL systems: Some recent works have looked to optimize ML systems by exploiting multi-model execution, e.g., systems such Cerebro [100], ModelBatch [157], ASHA [109], and SystemML [25]. Cerebro proposes a hybrid parallelism scheme named MOP combining task- and data- parallelism, akin to (but different from) SHARP's hybrid model-task parallelism. SystemML also hybridizes task- and data- parallelism, but for classical ML workloads rather than DL. ModelBatch raises GPU utilization by altering the DL tool's internal execution kernels. None of them tackle larger-than-GPU-memory models, which is our focus. Other examples of MQO for DL systems are Krypton [148] and HummingBird [150] but they focus on inference, not training.

8.2 Related Work for SATURN

Parallelism Selectors and Hybridizers: Paleo [168] focused on performance models for data parallelism and model parallelism. But the DL parallelism landscape has changed since then (2016), with numerous new approaches. While Paleo might be extended to newer parallelisms, our empirical Trial Runner approach is more easily extensible and highly general. Alpa, FlexFlow, and Unity [241, 89, 207] focus on generating bespoke parallelism strategies for model architectures through complex search procedures. They can produce efficient *single-model plans*, but the cumulative search overheads can get high when applied repeatedly to multi-model

training. They also do not consider multiple models being trained in model selection workloads. In addition, these tools must manually be redesigned for new approaches (e.g., spilling). These tools could potentially be viewed as parallelisms for SATURN's UPP abstraction.

DL Model Selection Systems: SATURN follows a line of work on systems for model selection, including Cerebro [100], Hyperband [111], and ASHA [109]. However, none of these prior works were explicitly designed for the large-model setting, where users must navigate multiple complex and varied parallelisms, as explained in Section 7.1. Cerebro hybridizes task- and data-parallelism to train multiple DL models in parallel on sharded data. Hyperband reallocates training resources (e.g., number of epochs) across tasks based on convergence behaviors. SHA implements a rung-based promotion plan to kill off less-promising job instances and prioritize the execution of higher-value ones. ASHA extends this to execute promotions asynchronously. ModelKeeper [103] suggests warm-starting across similar model configurations. This could be used to reduce the demands of model selection up front, before SATURN executes. All these techniques exist at a higher-level of abstraction, e.g., data sharding, early-stopping, or warm-starting. Thus, they are orthogonal to SATURN and could be combined with our work in future extensions.

DL Resource Schedulers: Pollux and Optimus [170, 163, 164] tackle apportionment [146] and scheduling, two parts of SPASE. But they do not explicitly support larger-than-GPU-memory models, where complex parallelisms alter performance tradeoffs in non-trivial ways, as our work shows. In Optimus' case, we can take the core mechanisms and adapt them for large-model training, as we do in our Optimus* baselines. But such adaptations underperform native large-model tools such as SATURN. These tools also do not target model selection workloads and optimize for throughput, while makespan is better suited for our setting. They also alter model accuracy, violating our fidelity desideratum. A config submitted to Pollux (e.g., batch size X and learning rate Y) may yield different accuracies than the same X and Y without Pollux. Themis [128] studies scheduling fairness for ML jobs from different users; their goal

and setting are orthogonal to ours in that we focus on model selection jobs from the same user and optimize for makespan.

Pipelining & FSDP: Pipelining is a modification of model parallelism in which the model is sharded in a sequential fashion. It partitions a minibatch into smaller "microbatches," then shuttles the microbatches through the model partitions [229, 82, 94, 118]. This enables different model shards to concurrently run different microbatches. The speedup of pipelining is heavily tied to the partitioning scheme and the number of microbatches. Prior work has underscored the importance of tuning these knobs via either expert knowledge or automated heuristics [118].

Fully-Sharded-Data-Parallelism (FSDP) is a more recent approach that blends model parallelism with data parallelism. Originally introduced in Microsoft's ZeRO [173], it has since been integrated into the PyTorch Distributed package [114]. FSDP partitions a model graph across multiple accelerators, then sends different minibatch partitions to the accelerators. FSDP runs All-Gather on model layers in sequence as data moves through the graph. The currently executing layer group is data-parallel-replicated; the other operators are still distributed in a model-parallel way. FSDP exposes two main user-configured optimizations to reduce GPU memory pressure: (1) gradient checkpointing [36] and (2) DRAM spilling. Turning these knobs on can lower GPU memory pressure at the cost of some performance. Ascertaining when it is worth turning one or both of these techniques generally requires empirical testing.

DL Cluster Schedulers: Schedulers such as Gandiva, Apollo, Tiresias, & Antman target a different setting [227, 226, 27, 62, 18] and require manual resource specification. Gandiva does offer opportunistic rescaling for elastic [225] jobs, but without knowledge of the model's scalability. These systems and other orchestrators like Pathways [20] tackle systems challenges that arise with very large clusters. Our focus is *complementary* in that we aim to free end users of DL from needing to hand-tune systems factors. One could potentially integrate SATURN with such larger schedulers by allocating a set of nodes for SATURN to manage locally for model

selection. Gavel [154] schedules over heterogeneous resources, which is beyond our scope, but does not tackle the SPASE problem. Their metrics to handle heterogeneity could potentially be used in a future SATURN extension.

System Optimizers: KungFu [129] provides an interface for users to express various procedures for mid-training system parameter changes. Litz [169] provides a programming model for elastic parameter server data parallelism. TeraPipe uses dynamic programming to optimize the partitioning and execution of pipeline parallelism [118]. Systems like Rammer [123], GO [245], TVM [35], SystemML's query rewriter [26] and compiler autotuners [166, 210] provide similar up-front optimizations for DL workloads. These automated search procedures are orthogonal to our own work and support can be added in the future using our UPP abstraction and Library API.

Other Model Selection Systems: Nautilus [147] optimizes model selection for transfer learning. α-NAS [91] proposes a method for creating architecture search workloads. Other systems like FairRover [235] tackle human-in-the-loop model building. These works are orthogonal to our own — they create/modify the model selection workload that SATURN executes.

Other DL System Optimizations: Optimizations such as compilation [35, 6, 104, 95, 166], batching [155, 238, 121], compression [68, 182, 46], and graph substitution [87, 88, 207] are orthogonal to our work. Many systems (e.g. DeepSpeed [173, 172, 178], Megatron [158, 190, 1], Hotline [11], HugeCTR [220], RecShard [186, 9], HogBatch [125], and Switch Transformers [52]) propose new parallelisms, all expressible under our Library API. Data pipeline optimizations [177, 215, 218, 222], fairness systems [234, 59, 180], and end-to-end pipeline managers [24, 21, 193, 165, 228] are also mostly orthogonal to our own work; we do not restrict the workload/data design. Some other works have addressed large-model challenges in different ways; e.g. by using alternative, parameter-efficient architectures [194] or else by training on non-GPU hardware [41, 198]. By contrast, SATURN is intended to optimize existing large-model GPU-training settings. Still other works optimize for settings such as DL inference [230, 19, 148, 14, 32, 31] or non-DL compute [60, 212, 75, 10, 50, 237]. These works

target a different setting entirely.

8.3 Related Work for InTune and InTuneX

Data Pipeline Optimizers: `AUTOTUNE` is generally considered to be the gold-standard data pipeline optimizer for `tf.data` pipelines [138]. Built-in to TensorFlow, the tool offers users with a seamless optimization experience that can be added to existing pipelines in just a single line of code. This design philosophy of abstraction has its detractors; recent works [98] have criticized its black-box approach to optimization. `AUTOTUNE` is designed to support any `tf.data` pipeline, but its generality leaves it vulnerable to task-specific issues, such as those we outline in Section 7.1 and explore in Section 5.2. InTune is more narrowly focused than `AUTOTUNE` in target workloads, but is also more general in its support for non-`tf.data` pipelines.

Plumber was introduced as a more user-friendly alternative to `AUTOTUNE` (but still restricted to `tf.data`). It uses a linear programming solver to determine a resource allocation. However, in practice it often *underperforms* `AUTOTUNE` [98] in its allocations. It does offer the ability to automatically inject caching into the pipeline to improve performance. A future version of our system could borrow this optimization from Plumber and add caching as an action for the agent. But in the recommenders setting, where dataset sizes routinely reach several terabytes or even petabytes [137], caching optimizations may not always be feasible.

DALI & NVTabular offer GPU-accelerated data-loader primitives for image and tabular data modalities respectively In practice, we find that NVTabular is more suited to offline feature engineering, since using the GPU for online data ingestion can lead to contention over cycles between the pipeline and the model. As a result, practitioners on our cluster have generally found it impractical to adopt NVTabular for our target setting.

CoorDL proposed a set of carefully-designed techniques to eliminate data stalls, including sophisticated caching procedures [135]. We address the related, but orthogonal, problem of data pipeline parallelization and throughput optimization. We leave it to future work to combine

their results with our own. Another recent work [83] focused on analyzing various training pipelines and identifying typical bottlenecks. They characterize general pipeline design spaces; by contrast, our work focuses specifically on DLRM challenges.

Meta's Data PreProcessing Service [239] tackles a similar problem to ours — online data ingestion pipelines for large-scale DLRM training. Their work primarily focuses on understanding performance issues in Meta's cluster, and describing their "disaggregated data service" approach. The idea of the service is to place replicas of the data ingestion pipeline on additional, separate CPU servers, which feed the trainer machine the data samples over the network. The details of their auto-scaling procedure remain closed, but this resource allocation procedure only operates at a multi-node level. By contrast, we focus on both in-node and cross-node resource allocation. Since we maximize per-node performance, it offers more opportunities for restraint at the multi-node level.

Other query optimization tools [148, 100, 147, 82, 214] exist, but target settings other than data pipeline optimization.

Resource Allocation: Some works (Pollux [170], Optimus [163], and Gandiva [226]) have tackled GPU apportioning in the scheduling setting. There is some overlap in the core ideas behind these works and our own.

Pollux proposes a novel "goodput" metric, combining convergence rates with throughput — to account for the fact that performance-oriented decisions (e.g., batch-size rescaling) may simultaneously reduce convergence efficiency. Similarly, our design uses *cost-efficiency* for multi-node orchestration, balancing costs against throughput.

Optimus proposes a greedy resource allocation strategy to distribute spare compute over DL jobs — but subsequent works like Saturn demonstrated the value of jointly optimizing the allocation problem for global efficiency. InTune takes lessons from both works — both observing the value of automatic resource distribution & the importance of global optimization objectives.

Still other works [109, 100] consider general resource apportionment for hyperparameter tuning. Like INTUNE, these tools reduce the manual configuration burden in the DL training process.

Broadly, these tools — and INTUNE — are part of an emerging "systems for DL" space. Such works develop *workload-aware* optimizations [43], proposing systems that identify and exploit the unique characteristics of deep learning workloads to improve performance and scalability.

Deep Reinforcement Learning for Systems: In parallel with the emergence of systems-for-DL research, deep-learning-for-systems work has also been gaining in popularity [209, 37]. Deep RL in particular has become a tool-of-choice for various systems applications [67]. Several works have tackled resource allocation using RL [164, 71, 201]. They aim to use the flexibility of RL to tackle the complexity and dynamic nature of intractable, online problems. Similarly, our work exploits the flexibility of RL to meet the needs of recommender data ingestion pipelines that are unaddressed by existing systems. Others have applied RL for SQL optimization [242, 130, 131, 162, 96]. The use of a learned algorithm helps relax the need for exact information that may be impractical to obtain in large RDBMSs. Our work also uses RL to relax the need for exact profiling of blackbox UDFs.

8.4 Related Work for LLM Routing

Inference optimization: Techniques to improve serving performance has long been a key focus of systems for DL research [148, 149]. The popularity of LLMs has driven the development of a broad swathe of inferencing techniques & systems for this class of models. Works such as vLLM [102], AlpaServe [117], FlexGen [188], & Orca [231] propose various orchestration and GPU management techniques to boost inferencing throughput and minimize latency. Lower-level kernel optimizations [43, 42, 106, 216] have also gained widespread attention.

Multiplexing Inference Requests: Some recent works [247] explore multiplexing

inference requests across multiple models, much like our own approach. Cache & Distil [175], OCaTS [196] propose recording application queries over time and using them to train a small-scale student model. This idea is orthogonal to our own proposal.

Like our paper, FrugalGPT [33] and AutoMix [126] benchmark complex LLM applications where challenging and holistic evaluation metrics are needed. They use cascade schemes, where one or more LLMs (or APIs) are invoked in order of increasing complexity until a verifier assesses that the output quality has reached some threshold. FrugalGPT focuses on optimizing over an API marketplace with many candidate models (rather than a hosted cluster with fewer candidates), and offers mechanisms tailored to that setting — e.g., a cascade optimizer that eliminates API options not worth considering. To approximate challenging metrics, FrugalGPT uses a secondary Transformer, DistilBERT [183] while AutoMix uses a combination of self-verification [243] and a Markov Decision Process [92]. These designs are state-of-the-art for multi-LLM routing on complex tasks, but focus on optimizing API invocation *cost*. This is useful for cost-conscious users, e.g., domain scientists and small businesses. But for users hosting their own models (e.g., in mid-to-large-scale enterprises) for performance-sensitive applications, the inference performance-quality trade-off may be more important. In our experiments, we benchmarked adapted versions of FrugalGPT and AutoMix for hosted models, but found that our direct-routing procedure is better suited for performance-quality optimization than cascade mechanisms.

CELMOC [248] and some others [189, 247] consider direct-routing schemes like ours. They either limit themselves to straightforward tasks, where there is a clear correct or incorrect answer derived from the response itself, e.g., classification or multiple-choice Q&A, or simplify the task (e.g., repeated identical queries). In such cases, training a metric-predictor may show strong results. But as we emphasize in Section 7.1, the new applications that have emerged specifically with LLMs often involve complex, difficult-to-approximate metrics.

Mixture-of-Experts Routing: In recent years, Mixture-of-Expert (MoE) architectures

have become increasingly popular [122, 90]. MoE routing demonstrates some similarities to our problem setting. Architectures such as the Switch Transformer [244] and systems such as GShard [108] route input tokens across different candidate layers within the model itself. This sort of "conditional computation" can significantly increase efficiency, and enable practitioners to scale up models by several orders of magnitude without suffering significant computational cost increases. Our work introduces routing at a higher-level, i.e., across different model instances.

Query routing: The process of sending different inputs across different servers has previously been studied in the context of load balancers for CDNs [160] and distributed data stores [197]. CDN load balancers distribute requests across multiple server replicas to minimize response latencies and avoid overloading servers. These CDNs now underpin a significant portion of internet traffic as a whole — as LLM traffic grows, similar infrastructure may likely become critical to enabling broad adoption.

Chapter 9

Conclusion and Future Work

In this dissertation, we identify and explore the need for *orchestration* systems in the context of large-scale DL. We demonstrate that such orchestrators — which tackle fundamental problems such as scheduling, resource allocation, routing, technique selection, as well as the confluence of these problems — can offer considerable speedups and cost-reductions for DL workloads. To users in the domain sciences and small-to-medium enterprises, this could represent a new way to access previously too-costly state-of-the-art models and techniques. For users operating on massive compute clusters, these systems can offer improved experimentation velocity and significant cost savings. Our techniques, which combine workload-aware optimization with a data-systems-inspired lens, take a holistic view of the DL lifecycle, addressing multiple critical stages — data processing, training/fine-tuning, and serving. The resultant speedups are already being leveraged in real-world practice by both domain scientists and large technology companies, thus demonstrating that our proposed orchestration layer represents an important piece of the DL systems landscape.

9.1 Future Work Related to HYDRA

Non-Sequential Neural Architectures. HYDRA focuses on neural computational graphs that can be represented as sequences of layers or groups of layers, in keeping with more typical pipeline parallel schemes [115, 159, 229, 56]). The most popular classes of GPU-memory-

bottlenecked models in DL practice today, viz., Transformers, as well as most convolutional neural networks and multi-layer perceptrons do satisfy the assumption. Some not-fully-sequential models such as Inception, ResNets, and DenseNets are easily handled because residual or skip connections can be linearized with a single "super-vertex" in the graph specification given to HYDRA. As long as the user defines the graph in that way using the DL tool's API, HYDRA works out of the box for such models too. But for recurrent neural networks (RNNs) and graph neural networks (GNNs), HYDRA would need to be extended to account for non-trivial dependencies across shard units of a model. Backpropagation through time, maintaining memory cells, and cross-layer global connections all require non-trivial extra implementation machinery and modifications to our Scheduler. We leave such extensions to future work.

Large Model Inference. This work focused primarily on training of large models. But a trained model is then used for inference in an application. If one wants to use a GPU for inference, the same GPU memory bottleneck exists. Fortunately, HYDRA's model spilling, automated partitioning, and automated shard orchestration all suffice already for out-of-the-box large model inference too.

DL Tool Generality. HYDRA is currently implemented as a wrapper around PyTorch. But all of our techniques are generic enough to be used with, say, TensorFlow or MXNet as well. Future extensions could look to adapt HYDRA to support these other frameworks.

CUDA-level Optimization. We designed HYDRA to operate on top of PyTorch to ensure backward compatibility as PyTorch evolves. This means we did not exploit any low-level optimizations for GPU-to-GPU transfers. One could technically imagine using multiprocessing in CUDA to reduce this latency, including for our double-buffering technique. But all this will require us to write new kernels in CUDA for memory management and hook them into the DL tool. We leave such ideas to future work.

More Hybrid Parallelism. When there are fewer models than devices, HYDRA may under-utilize the devices due to the limitation it inherits from task parallelism. But one could

do better by hybridizing data parallelism and pipeline parallelism with HYDRA to raise overall resource utilization. This is feasible because both of those approaches are technically *complementary* to SHARP and HYDRA's other techniques. We leave such sophisticated hybrid-of-hybrids parallelization to future work. But we note that in cases where there are more models than devices, SHARP is already close to optimal utilization, as our empirical results show.

9.2 Future Work Related to SATURN

Alternate Scheduling Objectives. SATURN currently optimizes for end-to-end makespan on batched multi-model workloads, but in a broader compute cluster, other objectives (e.g., throughput, fairness, average completion time) might be more relevant. Adapting SATURN's optimization scheme to support such alternative metrics should be relatively straightforward, only requiring a minor adjustment of the MILP formulation. The general techniques we propose should be applicable regardless.

Cloud Autoscaling. SATURN has some limited support for "elastic auto-scaling" jobs through its introspection mechanism, which can checkpoint and re-launch jobs with alternate resource apportionments. But it is also possible that the broader cluster itself might have some autoscaling optimizations (i.e., the amount of resources on the cluster changes over time). To support such environments, SATURN would need to actively monitor for resource changes in order to adapt its execution plans on-the-fly.

Learned Profiling. SATURN's empirical profiler is both simple and effective; many prior scheduling works have used similar mechanisms as well. But in scenarios with very limited cluster resources or strict time restrictions, the profiling overheads may seem excessive. SATURN's profiler could be replaced with a predictive mechanism, similar to the throughput estimator used in Gavel [154]. This could reduce profiling overheads — at the cost of introducing some uncertainty to the performance measurements. As we emphasized in our desiderata, modern parallelization techniques often introduce complex performance behaviors that are difficult to

predict or assess. Even so, some sort of learned predictor could be a useful mechanism for specific use cases with straightforward parallelism selections.

9.3 Future Work Related to InTune and InTuneX

Other DL Settings. InTune and InTuneX primarily focus on the recommendation setting for three reasons. First, that recommender data pipelines tend to be uniquely complex, and commonly process terabytes or petabytes of data. Second, that recommender models often demand expensive GPU nodes, but are still bottlenecked by data processing. Third, that recommendation workloads make up an outsized proportion of DL practice in industry. However, the techniques proposed with InTune and InTuneX are *not* recommender-specific; rather, recommender systems are simply an ideal use case. There is no reason the same system could not be used for image processing pipelines in CNN training, or even for NLP workloads.

Caching. Works such as Plumber [98] have already proposed simple — but theoretically optimal — mechanisms to identify cache points in data processing pipelines. As we mention in our evaluation of recommender pipelines, the size of recommendation datasets often makes caching impractical, but there are certainly use cases with relatively smaller datasets where caching could be useful. Fortunately, integrating the greedy caching mechanism proposed in Plumber with InTune or InTuneX should be fairly straightforward. The caching determinations are run *up front*, based on data-sizing estimates, and can be done separately from the actual parallelization and optimization stages.

9.4 Future Work Related to LLM Routing

Expert-Routing. Our current scheme focuses on mitigating the performance of serving a large model by offloading queries to smaller models when possible. However, an alternate formulation might consider multiple *similarly-sized* models, each with their own domain specializations. Routing over such models might require a different query assessment mechanism, e.g.,

based on query topic, rather than complexity.

Automatic Metric Selection. Our current work demonstrates the effectiveness of a few straightforward complexity metrics on some given domains. These metrics are unlikely to serve as "universal" measures of complexity; instead, users must consider what measures are most appropriate for their domain and application. A future iteration of our work might look to automatically select metrics drawn from a candidate pool based on their effectiveness in predicting the quality degradation behaviors of different model candidates.

Model Swapping. We currently assume all models are hosted in GPU memory and are immediately accessible at query time. However, in a more resource-constrained cluster, it is possible that some models may be stored on main system memory and swapped back and forth between the GPU and DRAM. Incorporating the overheads of these swaps might introduce a new layer of complexity to our routing formulation, but could be beneficial for domain scientists or small-to-medium enterprises who lack sufficient GPU resources to host all candidate models at once.

Bibliography

[1] State-of-the-Art Language Modeling Using Megatron on the NVIDIA A100 GPU. https://developer.nvidia.com/blog/language-modeling-using-megatron-a100-gpu/, 2020.

[2] Fully Sharded Data Parallel: faster AI training with fewer GPUs. https://engineering.fb.com/2021/07/15/open-source/fsdp/, 2021.

[3] What was the largest dataset you analyzed / data mined?, 2021.

[4] Accelerate Large Model Training using PyTorch Fully Sharded Data Parallel. https://huggingface.co/blog/pytorch-fsdp, 2022.

[5] nvidia-smi (1) user's manual, 2023.

[6] XLA: Optimizing Compiler for Machine Learning : TensorFlow, Accessed January 31, 2021.

[7] 2023 State of Data + AI, Accessed May 24, 2023.

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[9] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. Understanding training efficiency of deep learning recommendation models at scale, 2020.

[10] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.*, 38(4), jul 2019.

[11] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J. Nair. Heterogeneous acceleration pipeline for recommendation system training, 2022.

[12] Akashdeep, Karanjeet Kahlon, and Harish Kumars. Survey of scheduling algorithms in ieee 802.16 pmp networks. *Egyptian Informatics Journal*, 15, 03 2014.

[13] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, Daniele Mazzotta, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. The falcon series of open language models, 2023.

[14] Joy Arulraj. Accelerating video analytics. *SIGMOD Rec.*, 50(4):39–40, jan 2022.

[15] Mahmoud Assran, Nicolas Loizou, Nicolas Ballas, and Mike Rabbat. Stochastic gradient push for distributed deep learning. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 344–353. PMLR, 09–15 Jun 2019.

[16] FairScale authors. Fairscale: A general purpose modular pytorch library for high performance and large scale training. https://github.com/facebookresearch/fairscale, 2021.

[17] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. Qwen technical report, 2023.

[18] Yixin Bao, Yanghua Peng, and Chuan Wu. Deep learning-based job placement in distributed machine learning clusters. In *IEEE INFOCOM 2019-IEEE conference on computer communications*, pages 505–513. IEEE, 2019.

[19] Yixin Bao, Yanghua Peng, Chuan Wu, and Zongpeng Li. Online job scheduling in distributed machine learning clusters. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 495–503, 2018.

[20] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Dan Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent El Shafey, Chandramohan A. Thekkath, and Yonghui Wu. Pathways: Asynchronous distributed dataflow for ml, 2022.

[21] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1387–1395, 2017.

[22] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *CoRR*, abs/1802.09941, 2018.

[23] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

[24] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthoer, Kevin Innerebner, Florijan Klezin, Stefanie Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqi, and Sebastian Benjamin Wrede. Systemds: A declarative machine learning system for the end-to-end data science lifecycle, 2019.

[25] Matthias Boehm, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Arvind C. Surve, and Shirish Tatikonda. Systemml: Declarative machine learning on spark. *Proc. VLDB Endow.*, 9(13):1425–1436, sep 2016.

[26] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Alexandre V. Evfimievski, and Prithviraj Sen. On optimizing operator fusion plans for large-scale machine learning in systemml, 2018.

[27] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for {Cloud-Scale} computing. In *11th USENIX symposium on operating systems design and implementation (OSDI 14)*, pages 285–300, 2014.

[28] Justin Boyan and Andrew Moore. Generalization in reinforcement learning: Safely approximating the value function. *Advances in neural information processing systems*, 7, 1994.

[29] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[30] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. *CoRR*, abs/2005.14165, 2020.

[31] Jiashen Cao, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. Thia: Accelerating video analytics using early inference and fine-grained query planning, 2021.

[32] Jiashen Cao, Karan Sarkar, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. Figo: Fine-grained query optimization in video analytics. In *Proceedings of the 2022 International*

Conference on Management of Data, SIGMOD '22, page 559–572, New York, NY, USA, 2022. Association for Computing Machinery.

[33] Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance, 2023.

[34] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021.

[35] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 578–594. USENIX Association, 2018.

[36] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training Deep Nets with Sublinear memory cost, 2016.

[37] Zui CHen, Lei Cao, Sam Madden, Ju Fan, Nan Tang, Zihui Gu, Zeyuan Shang, Chunwei Liu, Michael Cafarella, and Tim Kraska. Seed: Simple, efficient, and effective data management via large language models. *arXiv preprint arXiv:2310.00749*, 2023.

[38] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, pages 7–10, 2016.

[39] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023.

[40] John H Crawford. The i486 cpu: Executing instructions in one clock cycle. *IEEE Micro*, 10(1):27–36, 1990.

[41] Shabnam Daghaghi, Nicholas Meisburger, Mengnan Zhao, Yong Wu, Sameh Gobriel, Charlie Tai, and Anshumali Shrivastava. Accelerating slide deep learning on modern cpus: Vectorization, quantizations, memory optimizations, and more, 2021.

[42] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023.

[43] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[44] Martin Davis. *Computability and unsolvability*. Courier Corporation, 2013.

[45] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[46] Aditya Desai, Yanzhou Pan, Kuangyuan Sun, Li Chou, and Anshumali Shrivastava. Semantically constrained memory allocation (scma) for embedding in efficient recommendation systems, 2021.

[47] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[48] Sumanta Dey, Sharat Bhat, Pallab Dasgupta, and Soumyajit Dey. Imperative action masking for safe exploration in reinforcement learning. In *International Workshop on Explainable, Transparent Autonomous Agents and Multi-Agent Systems*, pages 130–142. Springer, 2023.

[49] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *CoRR*, abs/2010.11929, 2020.

[50] Vahid Faghihi, Kenneth Reinschmidt, and Julian Kang. Construction scheduling using genetic algorithm based on building information model. *Expert Systems with Applications*, 41:7565–7578, 11 2014.

[51] Babak Falsafi, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis. Unlocking energy. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, pages 393–406, 2016.

[52] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2021.

[53] Giorgio Franceschelli and Mirco Musolesi. On the creativity of large language models. *arXiv preprint arXiv:2304.00008*, 2023.

[54] Yao Fu, Litu Ou, Mingyu Chen, Yuhao Wan, Hao Peng, and Tushar Khot. Chain-of-thought hub: A continuous effort to measure large language models' reasoning performance, 2023.

[55] Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. Break the sequential dependency of llm inference using lookahead decoding, 2024.

[56] Alexander L. Gaunt, Matthew A. Johnson, Maik Riechert, Daniel Tarlow, Ryota Tomioka, Dimitrios Vytiniotis, and Sam Webster. Ampnet: Asynchronous model-parallel training for dynamic neural networks, 2017.

[57] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[58] Google Cloud. Rsa: Google cloud, security, ai workbench, and generative ai. https://cloud.google.com/blog/products/identity-security/rsa-google-cloud-security-ai-workbench-generative-ai, 2023. (Accessed: <your access date>).

[59] Stefan Grafberger, Shubha Guha, Julia Stoyanovich, and Sebastian Schelter. Mlinspect: A data distribution debugger for machine learning pipelines. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2736–2739, New York, NY, USA, 2021. Association for Computing Machinery.

[60] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM*, August 2014.

[61] Audrūnas Gruslys, Remi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-Efficient Backpropagation Through Time. 2016.

[62] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.

[63] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020.

[64] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–501. IEEE, 2020.

[65] LLC Gurobi Optimization. Gurobi Optimizer Reference Manual, 2021.

[66] Gurobi Optimization, LLC. Mixed Integer Programming Basics, 2022.

[67] Ameer Haj-Ali, Nesreen K. Ahmed, Ted Willke, Joseph Gonzalez, Krste Asanovic, and Ion Stoica. A view on deep reinforcement learning in system optimization, 2019.

[68] Song Han, Huizi Mao, and William J Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv preprint arXiv:1510.00149*, 2015.

[69] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training, 2018.

[70] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, and Phillip B. Gibbons. Pipedream: Fast and efficient pipeline parallel DNN training. *CoRR*, abs/1806.03377, 2018.

[71] Ying He, F. Richard Yu, Nan Zhao, Victor C. M. Leung, and Hongxi Yin. Software-defined networks with mobile edge computing and caching for smart cities: A big data deep reinforcement learning approach. *IEEE Communications Magazine*, 55(12):31–37, 2017.

[72] Zhankui He, Zhouhang Xie, Rahul Jha, Harald Steck, Dawen Liang, Yesu Feng, Bodhisattwa Prasad Majumder, Nathan Kallus, and Julian McAuley. Large language models as zero-shot conversational recommenders, 2023.

[73] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding, 2021.

[74] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *Cidr*, volume 11, pages 261–272, 2011.

[75] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 295–308, USA, 2011. USENIX Association.

[76] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

[77] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019.

[78] Qitian Jason Hu, Jacob Bieker, Xiuyu Li, Nan Jiang, Benjamin Keigwin, Gaurav Ranganath, Kurt Keutzer, and Shriyash Kaustubh Upadhyay. Routerbench: A benchmark for multi-llm routing system, 2024.

[79] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355, 2020.

[80] Jianyu Huang and Robert A. van de Geijn. Blislab: A sandbox for optimizing gemm, 2016.

[81] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet, 2023.

[82] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline parallelism. *CoRR*, abs/1811.06965, 2018.

[83] Alexander Isenko, Ruben Mayer, Jeffrey Jedele, and Hans-Arno Jacobsen. Where is my training bottleneck? hidden trade-offs in deep learning preprocessing pipelines. In *Proceedings of the 2022 International Conference on Management of Data*. ACM, jun 2022.

[84] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference, 2017.

[85] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E. Gonzalez. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *CoRR*, abs/1910.02653, 2019.

[86] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, unjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant gpu clusters for dnn training workloads, 2019.

[87] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 47–62, New York, NY, USA, 2019. Association for Computing Machinery.

[88] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing dnn computation with relaxed graph substitutions. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 27–39, 2019.

[89] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks, 2018.

[90] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024.

[91] Charles Jin, Phitchaya Mangpo Phothilimthana, and Sudip Roy. Neural architecture search using property guided synthesis. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):1150–1179, oct 2022.

[92] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998.

[93] Anssi Kanervisto, Christian Scheller, and Ville Hautamäki. Action space shaping in deep reinforcement learning, 2020.

[94] Chiheon Kim, Heungsub Lee, Myungryong Jeong, Woonhyuk Baek, Boogeon Yoon, Ildoo Kim, Sungbin Lim, and Sungwoong Kim. torchgpipe: On-the-fly pipeline parallelism for training giant models, 2020.

[95] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[96] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning, 2018.

[97] Kubernetes. Updating the pod's resources, 2023.

[98] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis. Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines. *Proceedings of Machine Learning and Systems*, 4:33–51, 2022.

[99] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M. Patel. Model selection management systems: The next frontier of advanced analytics. *SIGMOD Rec.*, 44(4):17–22, May 2016.

[100] Arun Kumar, Supun Nakandala, Yuhao Zhang, Side Li, Advitya Gemawat, and Kabir Nagrecha. Cerebro: A layered data platform for scalable deep learning. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021.

[101] Ravi Kumar, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua Wang. Efficient Rematerialization for Deep Networks. 32:15172–15181, 2019.

[102] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. *arXiv preprint arXiv:2309.06180*, 2023.

[103] Fan Lai, Yinwei Dai, Harsha V. Madhyastha, and Mosharaf Chowdhury. ModelKeeper: Accelerating DNN training via automated training warmup. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 769–785, Boston, MA, April 2023. USENIX Association.

[104] Tung D. Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. Tflms: Large model support in tensorflow by graph rewriting, 2018.

[105] Bruce W. Lee and Jason Lee. LFTK: Handcrafted features in computational linguistics. In *Proceedings of the 18th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2023)*, pages 1–19, Toronto, Canada, July 2023. Association for Computational Linguistics.

[106] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, and Daniel Haziza. xformers: A modular and hackable transformer modelling library. https://github.com/facebookresearch/xformers, 2022.

[107] Daniel Leiker, Sara Finnigan, Ashley Ricker Gyllen, and Mutlu Cukurova. Prototyping the use of large language models (llms) for adult learning content creation at scale, 2023.

[108] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding, 2020.

[109] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A system for massively parallel hyperparameter tuning, 2020.

[110] Liangde Li, Supun Nakandala, and Arun Kumar. Intermittent human-in-the-loop model selection using cerebro: a demonstration. *Proceedings of the VLDB Endowment*, 14(12), 2021.

[111] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Efficient hyperparameter optimization and infinitely many armed bandits. *CoRR*, abs/1603.06560, 2016.

[112] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 583–598, USA, 2014. USENIX Association.

[113] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. *Advances in Neural Information Processing Systems*, 27, 2014.

[114] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12):3005–3018, 2020.

[115] Shigang Li and Torsten Hoefler. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[116] Side Li and Arun Kumar. Towards an optimized group by abstraction for large-scale machine learning. *Proceedings of the VLDB Endowment*, 14(11):2327–2340, 2021.

[117] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, 2023.

[118] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models, 2021.

[119] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous decentralized parallel stochastic gradient descent. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 3043–3052. PMLR, 10–15 Jul 2018.

[120] Zhan Ling, Yunhao Fang, Xuanlin Li, Zhiao Huang, Mingu Lee, Roland Memisevic, and Hao Su. Deductive verification of chain-of-thought reasoning. *arXiv preprint arXiv:2306.03872*, 2023.

[121] Rui Liu, Sanjan Krishnan, Aaron J Elmore, and Michael J Franklin. Understanding and Optimizing Packed Neural Network Training for Hyper-Parameter Tuning. *arXiv preprint arXiv:2002.02885*, 2020.

[122] Yiheng Liu, Tianle Han, Siyuan Ma, Jiayue Zhang, Yuanyuan Yang, Jiaming Tian, Hao He, Antong Li, Mengshen He, Zhengliang Liu, Zihao Wu, Dajiang Zhu, Xiang Li, Ning Qiang, Dingang Shen, Tianming Liu, and Bao Ge. Summary of chatgpt/gpt-4 research and perspective towards the future of large language models, 2023.

[123] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *Proceedings of the 14th USENIX Conference*

on Operating Systems Design and Implementation, OSDI'20, USA, 2020. USENIX Association.

[124] Yujing Ma, Florin Rusu, Kesheng Wu, and Alexander Sim. Adaptive elastic training for sparse deep learning on heterogeneous multi-gpu servers, 2021.

[125] Yujing Ma, Florin Rusu, Kesheng Wu, and Alexander Sim. Adaptive stochastic gradient descent for deep learning on heterogeneous cpu+gpu architectures. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 6–15, 2021.

[126] Aman Madaan, Pranjal Aggarwal, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, Shyam Upadhyay, Mausam, and Manaal Faruqui. Automix: Automatically mixing language models, 2023.

[127] Bruce M Maggs and Ramesh K Sitaraman. Algorithmic nuggets in content delivery. *ACM SIGCOMM Computer Communication Review*, 45(3):52–66, 2015.

[128] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient {GPU} cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.

[129] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. Making training in distributed machine learning adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[130] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo. *Proceedings of the VLDB Endowment*, 12(11):1705–1718, jul 2019.

[131] Ryan Marcus and Olga Papaemmanouil. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. ACM, jun 2018.

[132] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. Training deeper models by gpu memory optimization on tensorflow. In *Proc. of ML Systems Workshop in NIPS*, 2017.

[133] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.

[134] Microsoft Corporation. Security co-pilot. https://www.microsoft.com/en-us/security/business/ai-machine-learning/microsoft-security-copilot#tabx6635d5358fed4c248b4326df262e0d93, 2023.

[135] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in dnn training, 2020.

[136] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications, 2017.

[137] Dheevatsa Mudigere et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models, 2021.

[138] Derek G Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf. data: A machine learning data processing framework. *arXiv preprint arXiv:2101.12127*, 2021.

[139] Giorgi Nadiradze, Amirmojtaba Sabour, Peter Davies, Ilia Markov, Shigang Li, and Dan Alistarh. Decentralized sgd with asynchronous, local and quantized updates, 2020.

[140] Kabir Nagrecha. Model-parallel model selection for deep learning systems. SIGMOD-/PODS '21, page 2929–2931, New York, NY, USA, 2021. Association for Computing Machinery.

[141] Kabir Nagrecha. Systems for parallel and distributed large-model deep learning training, 2023.

[142] Kabir Nagrecha and Arun Kumar. Hydra: A system for large multi-model deep learning, 2021.

[143] Kabir Nagrecha and Arun Kumar. Saturn: An optimized data system for multi-large-model deep learning workloads (information system architectures). 2023.

[144] Kabir Nagrecha and Arun Kumar. Saturn: An optimized data system for multi-large-model deep learning workloads (information system architectures). 2023.

[145] Kabir Nagrecha and Arun Kumar. Tech report of saturn: An optimized data system for multi-large model deep learning, May 2023.

[146] Kabir Nagrecha, Lingyi Liu, Pablo Delgado, and Prasanna Padmanabhan. Intune: Reinforcement learning-based data pipeline optimization for deep recommendation models, 2023.

[147] Supun Nakandala and Arun Kumar. Nautilus: An optimized system for deep transfer learning over evolving training datasets. In *Proceedings of the 2022 International Conference on Management of Data*, pages 506–520, 2022.

[148] Supun Nakandala, Kabir Nagrecha, Arun Kumar, and Yannis Papakonstantinou. Incremental and approximate computations for accelerating deep cnn inference. *ACM Transactions on Database Systems (TODS)*, 45(4):1–42, 2020.

[149] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. A tensor compiler for unified machine learning prediction serving. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 899–917, 2020.

[150] Supun Nakandala, Gyeong-In Yu, Markus Weimer, and Matteo Interlandil. Compiling classical ml pipelines into tensor computations for one-size-fits-all prediction serving. *Conference and Workshop on Neural Information Processing Systems*, 10‘9.

[151] Supun Nakandala, Yuhao Zhang, and Arun Kumar. Cerebro: A data system for optimized deep learning model selection. *Proceedings of the VLDB Endowment*, 13(12):2159–2173, 2020.

[152] Supun Chathuranga Nakandala. *Query Optimizations for Deep Learning Systems*. PhD thesis, UC San Diego, 2022.

[153] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training, 2021.

[154] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads, 2020.

[155] Deepak Narayanan, Keshav Santhanam, Amar Phanishayee, and Matei Zaharia. Accelerating Deep Learning Workloads through Efficient Multi-Model Execution. In *NeurIPS Workshop on Systems for Machine Learning*, page 20, 2018.

[156] Deepak Narayanan, Keshav Santhanam, and Matei Zaharia. Accelerating model search with model batching. In *1st Conference on Systems and Machine Learning (SysML), SysML*, volume 18, 2018.

[157] Deepak Narayanan, Keshav Santhanam, and Matei Zaharia. Accelerating model search with model batching. *Proceedings of Fourth Conference on Machine Learning and Systems (MLSys'18)*, 2018.

[158] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[159] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient Large-Scale Language Model Training on GPU Clusters. *CoRR*, abs/2104.04473, 2021.

[160] Erik Nygren, Ramesh K Sitaraman, and Jennifer Sun. The akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.

[161] Kunle Olukotun. Systems for ml and ml for systems: A virtuous cycle. MLSys, 2022.

[162] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. Learning state representations for query optimization with deep reinforcement learning, 2018.

[163] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[164] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, Chen Meng, and Wei Lin. Dl2: A deep learning-driven scheduler for deep learning clusters, 2019.

[165] Arnab Phani, Benjamin Rath, and Matthias Boehm. Lima: Fine-grained lineage tracing and reuse in machine learning systems. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1426–1439, New York, NY, USA, 2021. Association for Computing Machinery.

[166] Phitchaya Mangpo Phothilimthana, Amit Sabne, Nikhil Sarda, Karthik Srinivasa Murthy, Yanqi Zhou, Christof Angermueller, Mike Burrows, Sudip Roy, Ketan Mandke, Rezsa Farahani, Yu Emma Wang, Berkin Ilbeyi, Blake Hechtman, Bjarke Roune, Shen Wang, Yuanzhong Xu, and Samuel J. Kaufman. A flexible approach to autotuning multi-pass machine learning compilers. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 1–16, 2021.

[167] Bharadwaj Pudipeddi, Maral Mesmakhosroshahi, Jinwen Xi, and Sujeeth Bharadwaj. Training large neural networks with constant memory using a new execution algorithm, 2020.

[168] Hang Qi, Evan R Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. 2016.

[169] Aurick Qiao, Abutalib Aghayev, Weiren Yu, Haoyang Chen, Qirong Ho, Garth A. Gibson, and Eric P. Xing. Litz: Elastic framework for high-performance distributed machine learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[170] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021.

[171] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[172] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning, 2021.

[173] Samyam Rajbhari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. 2020.

[174] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. 1996.

[175] Guillem Ramírez, Matthias Lindemann, Alexandra Birch, and Ivan Titov. Cache & distil: Optimising api calls to large language models, 2023.

[176] Jeff Rasley, Yuxiong He, Feng Yan, Olatunji Ruwase, and Rodrigo Fonseca. Hyperdrive: Exploring hyperparameters with pop scheduling. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, page 1–13, New York, NY, USA, 2017. Association for Computing Machinery.

[177] Sergey Redyuk, Zoi Kaoudi, Sebastian Schelter, and Volker Markl. Dorian in action: assisted design of data science pipelines. *Proceedings of the VLDB Endowment*, 15(12):3714–3717, 2022.

[178] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training, 2021.

[179] Alexander Renz-Wieland, Rainer Gemulla, Zoi Kaoudi, and Volker Markl. NuPS: A parameter server for machine learning with non-uniform parameter access. In *Proceedings of the 2022 International Conference on Management of Data*. ACM, jun 2022.

[180] Yuji Roh, Kangwook Lee, Steven Euijong Whang, and Changho Suh. Sample selection for fair and robust training, 2021.

[181] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 500–507. IEEE, 2011.

[182] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, A Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[183] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2020.

[184] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, March 1988.

[185] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[186] Geet Sethi, Bilge Acun, Niket Agarwal, Christos Kozyrakis, Caroline Trippel, and Carole-Jean Wu. Recshard: statistical feature-based memory optimization for industry-scale neural recommendation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 344–358, 2022.

[187] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, and Ashish Vaswani. Mesh-TensorFlow: Deep Learning for Supercomputers. *CoRR*, abs/1811.02084, 2018.

[188] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Re, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. 2023.

[189] Tal Shnitzer, Anthony Ou, Mírian Silva, Kate Soule, Yuekai Sun, Justin Solomon, Neil Thompson, and Mikhail Yurochkin. Large language model routing with benchmark datasets, 2023.

[190] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2019.

[191] Abraham Silberschatz. *Operating System Concepts with Java, 6th Edition, with Student Access Card EGrade Plus 1 Term Set*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2004.

[192] Nimit Sharad Sohoni, Christopher Richard Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. Low-memory neural network training: A technical report. *CoRR*, abs/1904.10631, 2019.

[193] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics, 2016.

[194] Olivier Sprangers, Sebastian Schelter, and Maarten de Rijke. Parameter efficient deep probabilistic forecasting, 2021.

[195] SQuAD Squad. The stanford question answering dataset (2021).

[196] Ilias Stogiannidis, Stavros Vassos, Prodromos Malakasiotis, and Ion Androutsopoulos. Cache me if you can: an online cost-aware teacher-student framework to reduce the calls to large language models, 2023.

[197] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on networking*, 11(1):17–32, 2003.

[198] Yipeng Sun and Andreas M Kist. Deep learning on edge tpus. *arXiv preprint arXiv:2108.13732*, 2021.

[199] Andrew S. Tanenbaum. *Modern Operating Systems, 4th editiont*. Pearson, 2021.

[200] Zhenheng Tang, Shaohuai Shi, Xiaowen Chu, Wei Wang, and Bo Li. Communication-efficient distributed deep learning: A comprehensive survey. *arXiv preprint arXiv:2003.06307*, 2020.

[201] Gerald Tesauro, Rajarshi Das, and Nicholas K. Jong. Online performance management using hybrid reinforcement learning. 2005.

[202] Sebastian Thrun and Anton Schwartz. Issues in using function approximation for reinforcement learning. In *Proceedings of the Fourth Connectionist Models Summer School*, volume 255, page 263. Hillsdale, NJ, 1993.

[203] Oguzhan Topsakal and Tahir Cetin Akinci. Creating large language model applications utilizing langchain: A primer on developing llm apps fast. In *International Conference on Applied Engineering and Natural Sciences*, volume 1, pages 1050–1056, 2023.

[204] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[205] Alan Tucker. *Applied combinatorics*. John Wiley & Sons, Inc., 1994.

[206] J.D. Ullman. NP-Complete Scheduling Problems. *Journal of Computer , System Sciences.*, 10(3):384–393, June 1975.

[207] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 267–284, 2022.

[208] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1009–1024, New York, NY, USA, 2017. Association for Computing Machinery.

[209] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*, pages 1009–1024, 2017.

[210] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions, 2018.

[211] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[212] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[213] Ben Wang and Aran Komatsuzaki. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

[214] Guanhua Wang, Zhuang Liu, Brandon Hsieh, Siyuan Zhuang, Joseph Gonzalez, Trevor Darrell, and Ion Stoica. sensai: Convnets decomposition via class parallelism for fast inference on live data. In *Proceedings of Fourth Conference on Machine Learning and Systems (MLSys'21)*, 2021.

[215] Pei Wang, Kabir Nagrecha, and Nuno Vasconcelos. Gradient-based algorithms for machine teaching. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1387–1396, 2021.

[216] Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

[217] Tongzhou Wang, Jun-Yan Zhu, Antonio Torralba, and Alexei A. Efros. Dataset Distillation. *CoRR*, abs/1811.10959, 2018.

[218] Tongzhou Wang, Jun-Yan Zhu, Antonio Torralba, and Alexei A Efros. Dataset distillation. *arXiv preprint arXiv:1811.10959*, 2018.

[219] Zehuan Wang, Yingcan Wei, Minseok Lee, Matthias Langer, Fan Yu, Jie Liu, Shijie Liu, Daniel G Abel, Xu Guo, Jianbing Dong, et al. Merlin hugectr: Gpu-accelerated recommender system training and inference. In *Proceedings of the 16th ACM Conference on Recommender Systems*, pages 534–537, 2022.

[220] Zehuan Wang, Yingcan Wei, Minseok Lee, Matthias Langer, Fan Yu, Jie Liu, Shijie Liu, Daniel G. Abel, Xu Guo, Jianbing Dong, Ji Shi, and Kunlun Li. Merlin hugectr: Gpu-accelerated recommender system training and inference. In *Proceedings of the 16th ACM Conference on Recommender Systems*, RecSys '22, page 534–537, New York, NY, USA, 2022. Association for Computing Machinery.

[221] Yingcan Wei, Matthias Langer, Fan Yu, Minseok Lee, Jie Liu, Ji Shi, and Zehuan Wang. A GPU-specialized inference parameter server for large-scale deep recommendation models. In *Sixteenth ACM Conference on Recommender Systems*. ACM, sep 2022.

[222] Steven Euijong Whang, Yuji Roh, Hwanjun Song, and Jae-Gil Lee. Data collection and quality challenges in deep learning: A data-centric ai perspective, 2021.

[223] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[224] Carole-Jean Wu, Ramya Raghavendra, Udit Gupta, Bilge Acun, Newsha Ardalani, Kiwan Maeng, Gloria Chang, Fiona Aga Behram, James Huang, Charles Bai, Michael Gschwind, Anurag Gupta, Myle Ott, Anastasia Melnikov, Salvatore Candido, David Brooks, Geeta Chauhan, Benjamin Lee, Hsien-Hsin S. Lee, Bugra Akyildiz, Maximilian Balandat, Joe Spisak, Ravi Jain, Mike Rabbat, and Kim Hazelwood. Sustainable ai: Environmental implications, challenges and opportunities, 2021.

[225] Yidi Wu, Kaihao Ma, Xiao Yan, Zhi Liu, Zhenkun Cai, Yuzhen Huang, James Cheng, Han Yuan, and Fan Yu. Elastic deep learning in multi-tenant gpu cluster, 2019.

[226] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[227] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. {AntMan}: Dynamic scaling on {GPU} clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548, 2020.

[228] Eric P. Xing, Qirong Ho, Wei Dai, Jin-Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, page 1335–1344, New York, NY, USA, 2015. Association for Computing Machinery.

[229] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher R. Aberger, and Christopher De Sa. Pipemare: Asynchronous pipeline parallel dnn training, 2020.

[230] Shuochao Yao, Yifan Hao, Yiran Zhao, Huajie Shao, Dongxin Liu, Shengzhong Liu, Tianshi Wang, Jinyang Li, and Tarek Abdelzaher. Scheduling real-time deep learning services as imprecise computations. In *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10, 2020.

[231] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th*

USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pages 521–538, 2022.

[232] Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and Ali Ghodsi. The shift from models to compound ai systems. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/, 2024.

[233] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence?, 2019.

[234] Hantian Zhang, Xu Chu, Abolfazl Asudeh, and Shamkant B. Navathe. Omnifair: A declarative system for model-agnostic group fairness in machine learning. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2076–2088, New York, NY, USA, 2021. Association for Computing Machinery.

[235] Hantian Zhang, Nima Shahbazi, Xu Chu, and Abolfazl Asudeh. Fairrover: Explorative model building for fair and responsible machine learning. In *Proceedings of the Fifth Workshop on Data Management for End-To-End Machine Learning*, DEEM '21, New York, NY, USA, 2021. Association for Computing Machinery.

[236] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 181–193, 2017.

[237] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. Slaq: Quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 390–404, New York, NY, USA, 2017. Association for Computing Machinery.

[238] Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, and Lidong Zhou. Retiarii: A Deep Learning Exploratory-Training Framework. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 919–936, 2020.

[239] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding data storage and ingestion for large-scale deep recommendation model training. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ACM, jun 2022.

[240] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023.

[241] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning, 2022.

[242] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning, 2017.

[243] Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, et al. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification. *arXiv preprint arXiv:2308.07921*, 2023.

[244] Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Zhao, Andrew M Dai, Quoc V Le, James Laudon, et al. Mixture-of-experts with expert choice routing. *Advances in Neural Information Processing Systems*, 35:7103–7114, 2022.

[245] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter Ma, Qiumin Xu, Hanxiao Liu, Phitchaya Mangpo Phothilimthana, Shen Wang, Anna Goldie, Azalia Mirhoseini, and James Laudon. Transferable graph optimizers for ml compilers. 2020.

[246] Andrew Zhu, Lara J. Martin, Andrew Head, and Chris Callison-Burch. Calypso: Llms as dungeon masters' assistants, 2023.

[247] Banghua Zhu, Ying Sheng, Lianmin Zheng, Clark Barrett, Michael I Jordan, and Jiantao Jiao. On optimal caching and model multiplexing for large model inference. *arXiv preprint arXiv:2306.02003*, 2023.

[248] Marija Šakota, Maxime Peyrard, and Robert West. Fly-swat or cannon? cost-effective language model choice via meta-modeling, 2023.