Machine Learning Parallelism Could Be Adaptive, Composable and Automated

Hao Zhang

CMU-RI-TR-20-54

October 8, 2020

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Greg R. Ganger
Jinyang Li (NYU)
Deva Ramanan
Christopher Ré (Stanford)
Eric P. Xing, Chair

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Copyright © 2020 Hao Zhang.

The research in this thesis was jointly sponsored by: the National Science Foundation awards IIS-1447676 and CCF-1629559, the Defense Advanced Research Project Agency award FA872105C0003, and Petuum Inc. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Scalable Machine Learning, Parallelization, Machine Learning Parallelism, Distributed Machine Learning, Machine Learning System, Compiler, Automatic Parallelization, SysML, AutoML, Parameter Server, Composability

To my fiancée, Luna.

Abstract

In recent years, the pace of innovations in the fields of machine learning (ML) has accelerated, researchers in SysML have created algorithms and systems that parallelize ML training over multiple devices or computational nodes. As ML models become more structurally complex, many systems have struggled to provide all-round performance on a variety of models. Particularly, ML scale-up is usually underestimated in terms of the amount of knowledge and time required to map from an appropriate distribution strategy to the model. Applying parallel training systems to complex models adds nontrivial development overheads in addition to model prototyping, and often results in lower-than-expected performance. This thesis identifies and addresses research challenges in both usability and performance in parallel ML techniques and system implementations.

The first part of this thesis presents a simple design principle, adaptive parallelism, that applies suitable parallelization techniques to model building blocks (e.g., layers) according to their specific ML properties. Following it, we derive a series of optimizations and implementations optimizing different aspects of ML parallelization. We examine them and show that they significantly boost the efficiency or scalability of ML training on clusters 2-10x in their applicable scenarios.

Generalizing this methodology, this second part of this thesis formulates the ML parallelization as an end-to-end optimization problem, and seeks to solve it automatically, for two broad paradigms of ML parallelization tasks: single-node dynamic batching and distributed ML parallelisms. We present principled representations to express the two classes of ML parallelisms, along with composable system architectures, Cavs and AutoDist, respectively. They enable rapid compositions of parallelization strategies for unseen models, improve parallelization performance, and simplify parallel ML programming.

On top of them, the third part of this thesis presents an automatic parallelization framework, AutoSync, to automatically optimize synchronization strategies in data-parallel distributed training. AutoSync achieves high performance "out-of-the-box" – it navigates the space spanned by the proposed representation, and automatically identifies synchronization strategies that report 1.2 - 1.6x speedups over existing hand-optimized systems, lowering the technical barrier of distributed ML and helping make it accessible to a larger community of users. Collectively, the set of techniques and systems developed in this thesis lead to the proof of the concept and the prototype implementation of an end-to-end compiler system for large-scale ML training on distributed environments.

Acknowledgments

First and foremost, I thank my advisor Eric Xing, for giving me a huge amount of freedom and trust during my graduate studies. In my early years, Eric was instrumental in teaching me to identity the right problems to work on. Eric also offered me the chance to work at Petuum during my PhD, during which I have received tremendous support to address new challenges that I would've otherwise never been able to see in a blue-sky research environment. This academic entrepreneurship experience has helped me identify my lifelong goal.

I am extremely grateful to my thesis committee members: Greg Ganger, Deva Ramanan, Jinyang Li, and Christopher Ré, for their feedback to make this thesis happen. As a member of the BigLearning group, Greg has always been a good mentor and collaborator – during the BigLearning meetings, Greg's sharp questions always helped me refine the characterizations of the problems, solutions, and systems. During my 1-day visit at NYU, Jinyang inspired me a lot to maintain an ML-system balanced perspective. While Deva is not a SysML insider, his scientific insight and intuition have been extremely important in shaping this thesis. Chris' path of transforming ML research into useful technologies and values have been an example, and I hope to collaborate with him in the future.

Research is a team effort and I felt super grateful to be a part of amazing teams at CMU and Petuum. I appreciate the helpful discussion and the camaraderie with my friends and collaborators at CMU: Henggang Cui, Wei Dai, Zhijie Deng, Qirong Ho, Zhiting Hu, Gunhee Kim, Jin Kyu Kim, Christy Li, Aurick Qiao, Jinliang Wei, Shizhen Xu, Zhicheng Yan, Zeyu Zheng, Xun Zheng, Yuntian Deng, Qi Guo, Willie Neiswanger, Hector Li, Xiaodan Liang, Bin Zhao. I was extremely lucky to lead a small team at Petuum but do BIG things during my PhD. I thank my colleagues Peng Wu, Hong Wu, Trevin Gandhi, Zeya Wang, Tairui Wang, Xin Gao, Sean Chen, Luke Lu, Jayesh Gada, Henry Guo, Liz Yi, Cathy Serventi for their helps in various cases ranging from debugging a line a code to standing at the ICML EXPO.

During my graduate studies, I also had the opportunity to interact with many other faculty members at CMU: Phillip Gibbons, Graham Neubig, Garth Gibson, Kris Kitani, Ruslan Salakhutdinov, Min Xu, Yaser Sheikh, Abhinav Gupta, David Wettergreen, Martial Hebert. Their nice comments and feedback have been always helpful in shaping me as an independent researcher. I want to give a special acknowledgement to Graham Neubig, Kris Kitani, and Min Xu for helping me with my speaking skill requirements.

Finally, I would like to thank my family for always supporting and believing in me. Especially, I am deeply indebted to my dear girlfriend Luna Yang for her unconditional love, understanding, and encouragement during my graduate years. I can clearly remember every moment in the past 5 years of her accompany during all my ups and downs – when we were skiing the Rockies, diving Cenotes, watching sea otters. This thesis would not be possible without your continual support. Thank you,

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The pace of innovation in machine learning (ML) has accelerated, with noteworthy advances such as analysis of unstructured data (images [80, 117, 233], video [104], audio [77], human-written text [51] and graphs [130, 243]), synthesis of audio [160], video [217] and reports [131], and programs that partially automate complex operational systems, such as auto-piloting vehicles [14] or game-playing AI [190]. The common denominators tying these applications are their use of deep, complex, and hierarchically structured models, and their reliance on large volumes of data. The largest ML models may involve 100s of millions to billions of parameters [117, 187, 191] if they are dense models, or even trillions [17, 51, 171, 237] for sparse models.

To cope with the computational cost of training such large models on large datasets, software systems have been created to parallelize ML programs over multiple CPUs or GPUs (multi-device parallelism) [116, 165, 244], and multiple computing devices over a network (distributed parallelism) [1, 40, 186, 245]. These software systems exploit ML-theoretic properties to achieve speedups scaling with the number of devices. Ideally, such parallel ML systems should *democratize* the parallelization[1] of large models on large datasets, by reducing training time to allow quicker development or research iteration – For example, we note that the 2020 cost of 4 workstations, each with 4 top-end consumer GPUs, is under $\$40,000$; with the right parallel ML software system, this setup should enable practitioners to train *most* ML models over 10 times faster, if not more, than a single-GPU workstation. In practice, however, the vast majority of ML developments, regardless of in research or production environment, report experiments taken on single workstations. What are the obstacles that prevent a nimble adoption of parallel ML, putting aside the cost of hardware? This thesis identifies and exposes research challenges in both *usability* and *performance* in parallel ML techniques and system implementations.

On usability, achieving satisfying speedups requires selecting or assembling the *right* set of parallization techniques, then tuning them so to suit the ML or computational properties of the given model and the topology of the clusters – either step demands strong ML and system savvy, or intensive trial-and-error. Existing systems [1, 40, 127, 186, 222, 245] are mostly built around a single parallelization technique or optimization, exhibit separate implementations or distinct

[1]For clarity, we define ML parallelization to include both single-machine multi-core parallelization, as well as distributed parallelization (multiple devices over a network). We also use the phrases "parallel ML strategy" and "ML parallelism" interchangeably.

interfaces. Trying out a specific strategy might require (re)familiarizing with the codebase and APIs of its system counterpart, then converting the single-machine ML code into a parallel version, which is again skill-intensive, and adds nontrivial overheads to ML prototyping.

Performance-wise, different parallelization strategies (such as parameter server [127, 222, 241], all-reduce [186], peer-to-peer [245], quantization and prefix coding [132], etc.) exhibit sharp differences in performance when applied to different ML building blocks (Bayesian, tree, shallow, deep, etc. [107, 245]). Since each technology might only work on a set of models, applying DL parallel training systems to complex models "out-of-the-box" (i.e., with default recommended settings) often results in *lower-than-expected* performance, when compared to reported results on typical benchmark models. For some ML models, none of the available strategies may be a good fit at all. Contemporary ML models exhibit rich varieties and composability – as will be shown in Section 1.1 of this thesis, because elements of a composite distributed-system strategy may apply deferentially to different sub-model building blocks (e.g., a parameter server [32, 83, 127] for layers of sparse gradients, and collective allreduce [68, 186] for convolutional layers), one can exploit the structures of complex models or design novel algorithms that co-optimize multiple parallelization aspects[2] and assign to every sub-model building block, so to provide the greatest parallel throughput. As a hidden dimension, composing multiple strategies is underexplored by existing systems due to their narrowly-specialized design, but is technically demanding for new systems and programming interfaces.

In summary, although modern ML with large data and complex models almost always requires massive multi-core and multi-node systems for training and inference, writing efficient parallel ML programs remains formidable for most ML researchers and developers, even with tools like TensorFlow [1], PyTorch [165], Horovod [186], Bösen [222], because of its requirement of deep knowledge on parallel systems, ML theories, and their complex interplay, which remains *largely artisanal rather than principled*.

The goal of this thesis is to lower the barrier of parallel machine learning and bring it into masses by offering solutions to the aforementioned challenges. The thesis studies distributed parallel ML on programmability, representations of parallelisms, performance optimizations, system architectures, and automatic parallelization techniques, and contends that *distributed parallel machine learning can be made both simple and performant*. In particular, this thesis shows that the performance in parallel ML can be significantly improved by generating strategies *adaptive* to the structures of the ML model and cluster resource specifications, while the usability challenges can be addressed by formalizing the core problem "how to parallelize" as an end-to-end optimization objective, and building *composable* distributed ML systems to *automatically* optimize for such adaptive, tailor-made strategies. The set of techniques collectively form the following thesis statement:

Thesis Statement: *The structured and composable nature of nowadays ML programs allows for the building of optimized parallelization strategies that are adaptive to model and cluster specifications, composable from different ML parallelization aspects, and auto-generated via learning-based methods*.

The rest of this chapter is planed as follows:

[2]We use the term "aspects" to refer to orthogonal spaces of design choices in a parallel ML program.

- Section 1.1, using the unsupervised pretraining of language models as an example, explains the rationale, key ideas, formulations, and methodologies adopted in this thesis.
- Section 1.2 summarizes the outline, contributions, and key results of this thesis.

1.1 Example: Distributed Pretraining of Language Models on Web-scale Text

Pretrained language representations has become the most ubiquitous and critical component in NLP systems. Using a task-agnostic language model architecture, it can be unsupervisedly trained on unlabelled text crawled from the web, simply by predicting the next word or sentence. The pretrained representations can be applied flexibly on a diverse set of downstream tasks, by either slightly finetuning against a task-specific loss and dataset [51, 198], or via few-shot in-context learning [17]. Given it versatility, tremendous progress has been witnessed in recent years on developing more powerful task-agnostic LM architectures, starting from single-layer word vector representations [146, 147], to recurrent neural nets with multiple layers of representations and contextual states [198], and the latest recurrent transformer-based architectures [17, 51, 209]. Figure 1.1 illustrate a notable example – the Deep Bidirectional Transformers (BERT) – that belongs to the third category. Regardless of the architectures, language models often contain many parameters trained on very large-scale text corpora, due to their modeling capacity growing proportionally with its size and the amount of text scanned [17].

Suppose we are interested in training a BERT, implemented using frameworks such as TensorFlow on an AWS-based GPU cluster. We might begin with data-parallel training using the most popular open-source training system – Horovod [186]. Applying Horovod to convert single-machine BERT training code involves wrapping the original, framework-built-in optimizer with a Horovod-patched optimizer; then Horovod will average and apply gradients using collective allreduce or allgather across cluster nodes. While we might manage to deploy the training on the target cluster, the obtained scale-up is unlikely to grow proportionally with more resources added (ideally, linear scale-ups with the number of accelerators): all language models have embedding layers that possess a large portion of model parameters but are sparsely accessed at each training iteration on each device, allreducing or allgathering their gradients result in unnecessary network traffic; the intermediate transformers in BERT are matrix-parameterized and computationally intensive, chunking their gradients within a reduction ring, as routinely done in Horovod, might easily saturate the Ethernet bandwidth or device FLOPs on heterogeneous clusters (such as on AWS). In either case, the setup is prone to communication or computation stragglers – unlikely the training time is satisfactorily short nor is the cost of computation resources spent on training economically acceptable – the usual goal of parallelization is not achieved.

Adaptive Parallelism

In contrast to existing systems that narrowly specialize to a monolithic strategy or system technique, this thesis proposes and follows a simple methodology, called *adaptive parallelism*, to derive appropriate optimizations for parallelization performance. Adaptive parallelism suggests

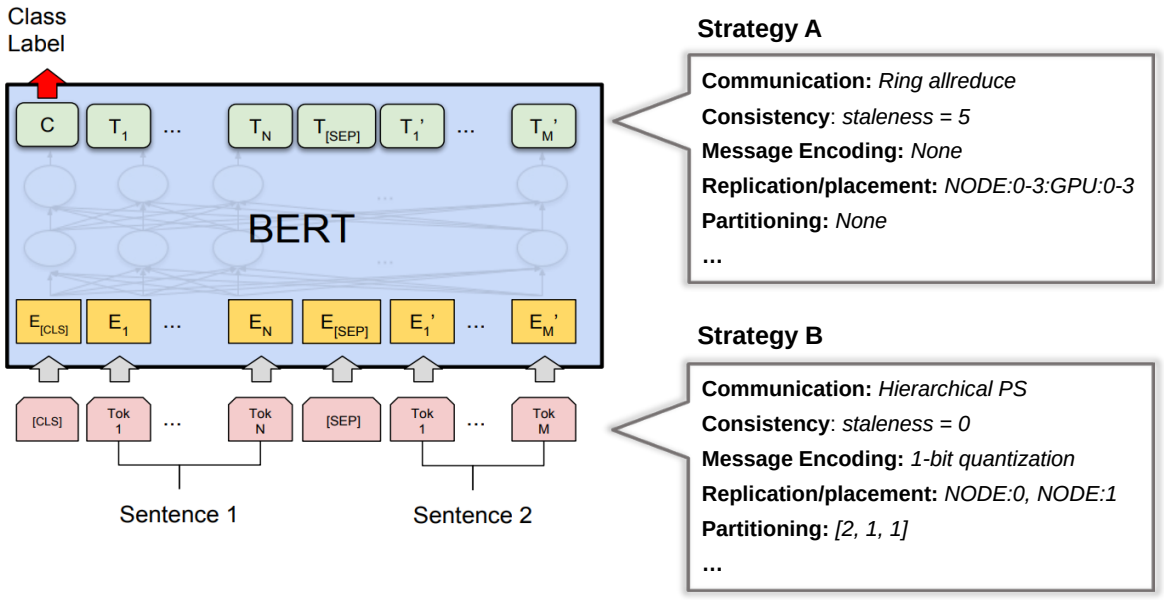

Figure 1.1: Consider the distributed training of the bidirectional transformer language model (BERT) illustrated on the left, which is composed of multiple model building block, with varying structures and computational patterns. This thesis contends to *automatically* generate an appropriate distribution strategy for each model building block, by *jointly optimizing* multiple parallelization aspects (communication, consistency, placement, partitioning, etc.). Composing these sub-strategies altogether renders a distribution strategy for the overall model, as illustrated on the right.

to generate *tailor-made* parallelization strategies, only conditioned on the model and cluster resource specifications. Using BERT as an example, we summarize the rationale and three key ideas that allow us to achieve such adaptiveness:

Sub-model strategy composition. Contemporary ML models exhibit increasing *complexity* and *composability*. For example, the BERT model in Figure 1.1 is composed of several sub-models: word and positional embedding, intermediate transformer encoders with residual connections, and a few auxiliary layers corresponded to different training losses; each component exhibits distinct computational or mathematical characteristics. It is possible to customize a suitable parallelization strategy for each sub-model, and the overall distribution strategy that achieves the greatest performance ends up with being the tailor-made composition of fittest strategies of all sub-models. The first part of this thesis presents extensive evidence to validate this hypothesis.

Co-optimization of multiple parallelization aspects. When parallelizing a model or a sub-model, optimizations from different parallelization aspects might influence each other, and the interplay dictates their effectiveness against different ML components or cluster topology, hence must be considered holistically. For instance, it might be plausible to divide huge embedding layers of BERT into smaller partitions, and use a bipartite parameter server to synchronize their gradients (other than allgather); but how to *partition* the embedding and *place* the servers must

be optimized accordingly to ensure its effectiveness. These interplay leads to an intricate space of system design considerations, which are not immediately obvious from the ideal mathematical or system-design view, and might be opaque even for veteran distributed ML practitioners. This thesis explores this space and presents results that, while deviated from common wisdom, but improve parallelization performance substantially.

Resource awareness.

This thesis, along with many concurrent research results [38, 107, 222], show that these guidelines can be instantiated to *significantly improve parallelization performance* in almost every aspect of ML parallelization, such as scheduling, communication, memory management, consistency model, to name a few.

An End-to-end Formulation

Despite various optimizations on performance, ideally, the ML developers, whose focuses are on ML prototyping, should not worry about the question *how to parallelize* – this includes: how parallelization performance shall be optimized; how their code, running on a single device, shall be modified precisely to benefit from certain optimizations. Instead, to make parallel ML training more attractive and easier to use for a wider audience, we believe the capability of distributed ML – adding more computing resources to achieve higher performance – shall be taken for granted and achieved "out-of-the-box". While there are several approaches and system implementations which ease parallel ML programming and adoption, none of them can effectively achieve all-round performance on a wide range of models while still maintaining simplicity and ease of use.

To bridge this gap, other than developing specialized systems or performing optimizations on an aspect-by-aspect basis, this thesis takes a more holistic perspective, and formalizes "how to parallelize" into a more broadly-scoped optimization problem as follows:

$$\mathcal{S}^* = \arg\max_{\mathcal{S}} f(\mathcal{G}, \mathcal{D}, \mathcal{S}) \tag{1.1}$$

in which, the ultimate goal is to maximize certain metric f characterizing the goodness of parallelization (e.g., system throughput, wall-clock time to convergence, cloud cost) with respect to a parallelization strategy $\mathcal{S}$, conditioned on (as suggested by adaptive parallelism) the input model $\mathcal{G}$ and the resource specification $\mathcal{D}$. Developing systems to solve this end-to-end optimization and automatically synthesize $\mathcal{S}^*$ would naturally bridge the gap between usability and performance. We next inspect each element in Equation 1.1 and see what prevents a trivial solution to this problem.

Composable Representations and Backends for ML Parallelisms

Fortunately, with the rapid development of ML frameworks in the past decade, dataflow graph [1, 154] has established as a universal, principled representation to express diverse ML programs and their computations in $\mathcal{G}$. It is also straightforward (as we will show in Chapter 8) to create ways of expressing the cluster configuration $\mathcal{D}$ and the performance of interest f. In order to approach this optimization with off-the-rack solvers, we need to rigorously define and implement the two

remained missing pieces in Equation 1.1: (1) an explicit, optimizable representation of $\mathcal{S}$[3], that instructs how the model $\mathcal{G}$ being parallelized over $\mathcal{D}$ in an expressible form; (2) a generic system backend that allows quickly assembling and exercising an arbitrary $\mathcal{S}$, potentially composed from multiple orthogonal parallelization aspects, such as those in Figure 1.1.

To fill this void, the second part of the thesis introduces unified representations of ML parallelisms for two distinct parallelization scenarios: dynamic batching on CPU/GPU accelerators, and distributed parallelization on computing clusters. Associated with the representations are new, modular system abstractions that decompose the design of existing DL training systems into composable and re-purposable units. Based on these units, model- and resource-adaptive parallelization strategies can be mechanically assembled, then exercised, via higher-level programming languages, simpler interfaces, hidden from low-level multi-core or distributed system sophistication.

Automatic Strategy Optimization

With the installation of a representable $\mathcal{S}$ and a composable backend, the third part of this thesis concretizes Equations 1.1 in the context of data-parallel training, and provides the first solution to automatically deriving $\mathcal{S}^*$ given $\mathcal{G}$ and $\mathcal{D}$. In particular, by instantiating the performance metric f as the *system throughput*, and narrowing down the strategy space as synchronous data-parallel ones, we present an end-to-end solver that automatically discovers synchronization strategies 1.1x-1.6x better than state-of-the-art hand-optimized systems, within reasonable optimization budget.

To conclude this section, this thesis provides basic methodologies and practical implementations for distributed ML performance optimizations, as well as novel perspectives and reformulations of the bigger problem beyond performance (Equation 1.1), along with tractable solutions to improving distributed ML ease-of-use. The set of techniques developed in this thesis lead to the *first prototype implementation of a compiler system for large-scale ML training on distributed clusters*.

We believe the ideas, formulations, systems, algorithms, models, and lessons learned in this thesis will enable more ML developers to take advantage of parallel computing to scale up ML workloads, and shed light on future research in large-scale ML to close the gap between performance and usability in distributed ML. The code and systems developed as parts of this thesis are made publicly available or being commercialized in some startup company, and have been used by various researchers in distributed ML to verify or compare their solutions.

1.2 Thesis Outline, Contributions, and Key Results

We support the thesis statement with three research components following the rationale explained in Section 1.1. They correspond to the three parts of the thesis:

[3]From this point, we use "strategy" to denote the general meaning of how the model is parallelized, as well as the explicit representation $\mathcal{S}$ we want to optimize in Equation 1.1

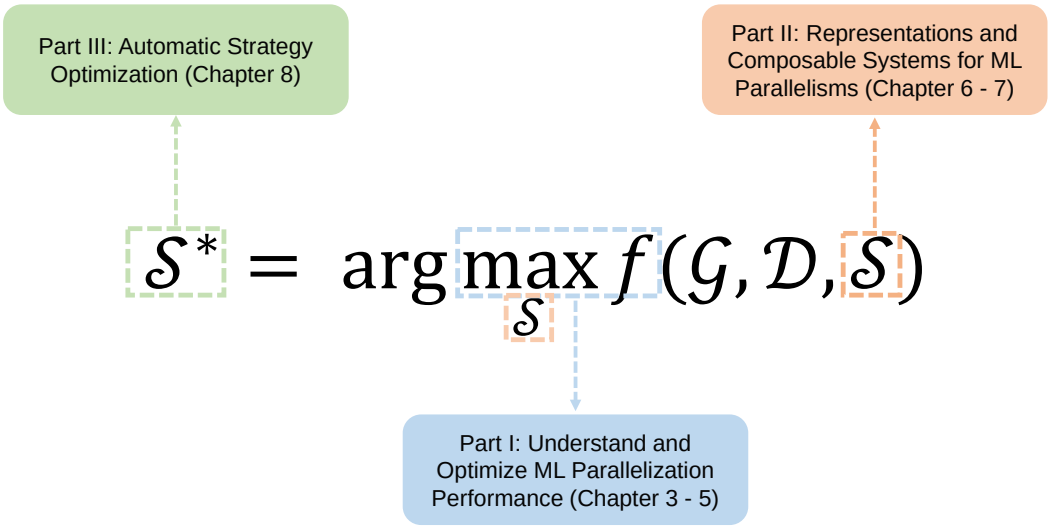

Figure 1.2: A big picture view of the three parts in this thesis.

- **Part I (Chapter 3 - Chapter 5):** Understand and optimize parallel ML performance on individual aspect of ML parallelization with adaptive parallelisms;
- **Part II (Chapter 6 - Chapter 7):** Develop unified representation and composable systems for ML parallelisms;
- **Part III (Chapter 8):** Automate ML parallelization.

The three components are *progressively structured*: Using adaptive parallelism as the guiding principle, in the first part, we build the basic understanding or performance optimizations on each individual aspect of ML parallelization covering scheduling, communication, consistency model, memory management, addressing the performance challenges or lack of understanding faced uniquely at each aspect. Generalized from them, in the second part, we establish the end-to-end formulation shown in Equation 1.1, and develop the two missing components needed to solve this objective: unified representations and composable systems of ML parallelisms. Once all the components are in place, we develop solvers to solve this optimization, bridging the gap between usability and performance.

Figure 1.2 illustrates the big picture placing each part of the thesis and its corresponded chapters under the umbrella of the unified formulation provided in Section 1.1. The results developed in this thesis were in collaborations with many co-authors: Henggang Cui, Wei Dai, Zhijie Deng, Greg Ganger, Phillip Gibbons, Qirong Ho, Zhiting Hu, Gunhee Kim, Jin Kyu Kim, Christy Li, Graham Neubig, Aurick Qiao, Jinliang Wei, Peng Wu, Eric Xing, Shizhen Xu, Zhicheng Yan, Zeyu Zheng, Jun Zhu.

Below, we outline the detailed contents, contributions, and key results in each chapter.

- Chapter 2 introduces the basic concepts in large-scale machine learning, hardware infrastructures of ML, techniques, frameworks, and software systems for ML parallelization,

and the developing trends in each of mentioned areas.

- Chapter 3 improves the scheduling and communication of distributed ML. It introduces an adaptive scheduling algorithm, wait-free backpropagation (WFBP) (Section 3.2), and an adaptive communication architecture (Section 3.3). Based on the two techniques we build Poseidon (Section 3.4), one of the state-of-the-art data-parallel DL systems, and *boost the scalability for 5 - 10x across multiple CNN models, environment and task workloads*.

  Application-wise (Section 3.5.5), this chapter presents *comparable* image classification results on the largest image classification dataset ImageNet22K (as of 2017) to state-of-the-art systems, but spends *7x less computing resources*.

- Chapter 4 studies the memory management of distributed ML. It presents a distributed shared-memory management mechanism, memory swapping (Section 4.2), to address the memory inadequacy in limited-memory devices. It then presents a specialized parameter server implementation, GeePS (Section 4.3), that adopts memory swapping for distributed DL on GPU clusters. GeePS allows *training large models that is only bounded by the largest layer rather than the overall model size*.

  As an application (Section 4.5), we enhance a typical CNN backbone with more hierarchical layers to model the category hierarchy. The augmented network, HD-CNN, scaled with *2x more parameters and 7x more memory footprints*, has boosted the image classification accuracy on ILSVRC2012 for *up to 3 percentage*, and *achieved state-of-the-art results* in 2016.

- Chapter 5 studies the consistency model of distributed ML. We perform controlled empirical studies on how staleness impacts the parallel ML training, *for the first time*, across *6 model families and 7 algorithms*, spanning comprehensively across optimization, sampling, and blackbox variational inference. It reveals that the consistency model must *flexibly adapt to model specifications to best trade-off between system throughput and statistical efficiency*.

- Chapter 6 addresses the programmability, representation, and performance for parallelizing dynamic neural networks in an end-to-end fashion, per Equation 1.1. It first presents a programming model called dynamic declaration. Based on it, it contributes to an open source DL framework, DyNet, specialized for dynamic neural networks. DyNet has become a *pioneer framework for programming and parallelizing neural networks with dynamically varying architectures*, and provided a testbed for future dynamic DL frameworks such as PyTorch [56].

  To further improve, this chapter introduces a new vertex-centric representation for dynamic neural networks (§6.4), and a corresponded, parallelization-friendly programming model (§6.4.2) for dynamic batching. Based on these techniques, the system Cavs is implemented and *improves both performance and programmability* over existing systems for dynamic NNs. It has achieved *state-of-the-art performance on both training and inference* on a variety of dynamic neural networks.

- Chapter 7, following the formulation in Equation 1.1 and rationale, builds the representa-

tion and the composable system backend for ML parallelization in distributed training. In particular, it develops a unified representation that allows parallelization strategies to be explicitly expressed by instance choices from seemingly incompatible parallelization aspects (Section 7.3). Based on this representation, we develop AutoDist (§7.4), a composable system backend using dataflow graph rewriting. AutoDist enables quickly assembling and exercising distribution strategies using a few atomic, composable dataflow graph rewriting kernels, each of which maps to an aspect-specific optimization developed in distinct existing systems. AutoDist significantly simplifies distributed ML programming, reports *matched or even better performance* than existing systems on their specialized benchmark models or cluster configurations, and offers all-round performance on unseen models.

- Chapter 8, based on the groundwork laid in previous chapters, proposes an end-to-end solver of Equation 1.1, AutoSync, to automate ML parallelization. More precisely, it constructs a novel subspace of synchronization strategies on top of the representation proposed in Chapter 7. By navigating this space, AutoSync is able to find synchronization strategies up to 1.6x better than those manually optimized, on a variety of models, even with only 200 trial run budget, achieving automatic ML parallelization "out-of-the-box".

  Furthermore, it develops transfer-learning mechanisms to reduce the auto-optimization cost – In AutoSync, trained strategy simulators can transfer among similar model architectures, among similar cluster configurations, or both. A dataset containing 10000 strategy, model, cluster, and runtime tuples is made publicly available to facilitate future research in automating the parallelization of ML programs.

- Chapter 9 characterizes the formulations and approaches taken in this thesis by discussing their limitations, and suggesting future potential research directions, and finally concludes this thesis.

Chapter 2

Background

This thesis studies the intersection area of machine learning, optimization, and distributed systems. In this chapter, we introduce basic concepts and notations used in this thesis, with a focus on highlighting the most relevant work and development trends in each of the areas.

2.1 Machine Learning: A Computational Perspective

2.1.1 "The Master Equation"

Machine learning programs are distinguished from other data center applications in that they feature intensive and diverse mathematical computations exhibited in an *iterative-convergent* form. While there are a rich variety of ML models and learning algorithms with different applicable scenarios, we can summarize the computational pattern for almost all of them (including deep learning) using just a few general equations and algorithms.

Normally, an ML program aims to fit N data samples, denoted as $\boldsymbol{x} = \{x_i, y_i\}_{i=1}^{N}$, to a model represented by $\boldsymbol{\Theta}$, by minimizing an objective function $\mathcal{L}$, as shown in the following equation:

$$\min_{\boldsymbol{\Theta}} \mathcal{L}(\mathbf{x}, \boldsymbol{\Theta}) \quad \text{s.t.} \quad \mathcal{L}(\mathbf{x}, \boldsymbol{\Theta}) = f\left(\{x_i, y_i\}_{i=1}^{N}; \boldsymbol{\Theta}\right) + r(\boldsymbol{\Theta}). \tag{2.1}$$

The objective $\mathcal{L}$ captures the relationship between the ML model parameters (also denoted as $\boldsymbol{\Theta}$) and training data $\boldsymbol{x}$. It often contains two components: (1) a loss function, $f(\cdot)$, that describes how data should fit the model; (2) a structure-inducing function, $r(\cdot)$, that places constraints or penalties for the intended application on the values that $\boldsymbol{\Theta}$ can take. Most ML programs solve the optimization in Equation 2.1 using an iterative-convergent algorithm, described in Algorithm 1.

In Algorithm 1, $\Delta_{\mathcal{L}}(\cdot)$ is an *update function* that computes model updates at step t based on a chunk of data, such as the pth mini-batch $\boldsymbol{x}_p$ in $\boldsymbol{x} = \{\boldsymbol{x}_p\}_{p=1}^{P}$, and the current parameter values $\boldsymbol{\Theta}^{(t)}$. How to calculate the updates depends on the definitions of the model (i.e., $\mathcal{L}$ and $\boldsymbol{\Theta}$), and often involves a variety of mathematical operations, such as matrix multiplcation or convolution. The model updates are then applied to update $\boldsymbol{\Theta}^{(t)}$ to $\boldsymbol{\Theta}^{(t+1)}$, on which the loss value $\mathcal{L}$ is hoped to decrease. F represents the function that applies the updates – it is simply addition when the updates $\Delta_{\mathcal{L}}(\boldsymbol{\Theta}^{(t)}, \boldsymbol{x}_p)$ are *gradients*. The training algorithm repeats the above process for a

Algorithm 1: The iterative-convergent algorithm in ML programs

```
Initialize t ← 0
for epoch = 1 ... K do
    for p = 1 ... P do
        Θ^(t+1) ← F(Θ^(t), Δ_L(Θ^(t), x_p))
        t = t + 1
```

maximal number of K epochs (in each epoch it scans the entire dataset $\boldsymbol{x}$ once), or until some convergence criteria is met.

2.1.2 Notable ML Models and Applications

Equation 2.1 and Algorithm 1 serve as the "master euqation" upon which our proposed approaches in this thesis will be grounded. Based on them, we introduce several notable classes of ML models and associated applications.

Image Classification and Convolutional Neural Networks

Image classification [49] is a core task in computer vision (CV) that assigns an input image one label from a fixed set of categories, such as $\{$*car, aircraft, cat, dog*$\}$. Many other seemingly distinct CV applications, such as object detection [59, 174], semantic segmentation [135, 177], use an image classifier as a backbone. Today's image classifiers are mostly built on top of convolution neural networks (CNNs) [80, 87, 117, 122, 191], a class of NNs that contain 10s to 100s of convolutional layers. The training of CNN-based image classifiers fits perfectly with Equation 2.1 and Algorithm 1: $\mathcal{L}$ reduces to the cross-entropy loss and $\Delta_{\mathcal{L}}$ calculates the gradients via backpropagation over a batch of training images $\boldsymbol{x}_p$; the gradients are then applied to update the parameters following stochastic gradient descent or other gradient-based optimizers (F).

Recommender System and Collaborative Filtering

A recommender system seeks to predict the preference of a user on an item, given the user's past browsing, clicking, rating, or purchasing history. It is widely applied in nowadays commercial systems such as Amazon online store, Google Ads, or Netflix streaming service, to help their users find potential products of their interest. A recommender system can be realized using different ML models, such as matrix factorization (MF) [113, 123] or neural collaborative filtering (NCF) [82, 213].

In MF, a large but sparse user-item rating matrix $\boldsymbol{D}$ in the size of $u \times r$ is observed, and the goal is to decompose $\boldsymbol{D}$ into two latent lower-dimensional matrices $\boldsymbol{U}$ of $v \times u$ and $\boldsymbol{I}$ of $v \times r$ so that $\boldsymbol{D} \approx \boldsymbol{U}^\top \boldsymbol{I}$ while a loss (e.g., $\ell 2$) is minimized. the problem can be solved using iterative algorithms, such as gradient descent or alternating least square (ALS) – both fitting into Algorithm 1. In NCF, two matrices, *user embedding* and *item embedding*, are explicitly introduced as hidden representations for users and items, respectively. They are fed through a

neural network which is trained to capture the user-item interaction, and predict the user-item match score. The NCF model is trained in the same way as CNNs by replacing the cross-entropy with an appropriate loss. NCF reduces to MF when the user-item interaction function is linear.

Thus, both NCF and MF involve learning two embedding matrices with their numbers of columns equal to the number of users (u) and the number of items (r), respectively. In real-world application, these two numbers u, r are at the scale of millions to billions. When following Algorithm 1, the two embedding matrices $\boldsymbol{U}, \boldsymbol{I}$ are *sparsely accessed* at each training iteration on a batch of data $\boldsymbol{x}_p$ – only a few columns are activated and updated per step.

Topic Modeling

Topic models aim to discover the hidden, abstract "topics" that occur in a collection of documents. They are frequently used in text mining [2, 242] or bioinformatics [133]. The most popular topic model is Latent Dirichlet Allocation (LDA) [12]. Given a text corpus, LDA learns parameters $\boldsymbol{\Theta} = \{\boldsymbol{\alpha}, \boldsymbol{\beta}\}$, where $\boldsymbol{\beta}$ is the parameter of K multinational distributions over the words in the vocabulary, representing the discovered K topics; $\boldsymbol{\alpha}$ is the parameter of a Dirichlet distribution over the K topics. The learning is unsupervised and alternates between two steps following the EM algorithm [48]: in the E-step, the parameters $\boldsymbol{\Theta}$ are fixed, and the topic-word associations are inferred via posterior inference methods, such as variational inference [13] or Gibbs sampling [71]; in the M-step, the parameters are updated to maximize the log-likelihood of the model on training data. Naturally, topic models fit into Equation 2.1 and Algorithm 1.

Deep Generative Models

Deep generative models (DGM) [181] refer to a class of models that use deep neural networks with a finite set of parameters to approximate the probabilistic density of training data. Once trained, they are powerful tools to generate raw images [63, 109], text [85], or even chemical structures [101]. Notable DGMs include variational autoencoders (VAE) [109], generative adversarial networks (GAN) [63], autoregressive models [206], and normalizing flow models [110]. Most DGMs are trained either by maximizing the likelihood (MLE) over observed data (as with LDA), or by solving a minimax adversarial game. Regardless of the training objective, all DGMs are constructed by neural networks, so their training process falls under Algorithm 1, following backpropagation and gradient-based optimization.

Other ML Models and Applications

Besides the the language modeling introduced in Section 1.1 and those described above, successful applications of ML extend to autonomous driving [14], speech recognition [77, 160], game-playing AI [190], dialogue systems [15, 23], to name a few. While the model architectures range from convolutional to recurrent and transformers, the learning algorithms vary from gradient-based methods to sampling and reinforcement learning [199], they all exhibit an iterative-convergent nature, which can be parallelized using a same set of techniques, introduced next.

2.1.3 Parallel Machine Learning

Parallel machine learning comes into the picture when either $\boldsymbol{\Theta}$ or $\boldsymbol{x}$ is large, that the computation in Algorithm 1 cannot be completed by a single *worker* in acceptable time, or the contents generated at computation cannot be held within a single device. In the following text, we will use *worker* as a virtual concept to represent the entity that owns an independent set of computational resources and performs serial computation such as those in Algorithm 1, e.g., a worker might map to a CPU thread, or a GPU device, or a GPU stream if there are parallelisms on the GPU. We use *node* to denote the physical computational node on a cluster, which might own multiple GPU devices or CPU threads.

Depending on which element of $F(\boldsymbol{\Theta}^{(t)}, \Delta_{\mathcal{L}}(\boldsymbol{\Theta}^{(t)}, \boldsymbol{x}_p))$ is parallelized across workers, parallel ML can be realized by two broad paradigms of approaches: *data parallelism* and *model parallelism*.

Data Parallelism

Data parallelism refers to a class of strategies that partition the dataset $\boldsymbol{x}$ into independent splits, and dispatch them to multiple workers for parallel processing. When mapped to Algorithm 1, data parallelism alters the inner loop (Line 3-6) to the following parallel version:

$$\boldsymbol{\Theta} \leftarrow F(\boldsymbol{\Theta}, \Delta_{\mathcal{L}}(\boldsymbol{\Theta}, \boldsymbol{x}_{p_1}), \Delta_{\mathcal{L}}(\boldsymbol{\Theta}, \boldsymbol{x}_{p_2}), \ldots, \Delta_{\mathcal{L}}(\boldsymbol{\Theta}, \boldsymbol{x}_{p_M})), \tag{2.2}$$

where the update function $\Delta_{\mathcal{L}}$ is replicated across M workers, and multiple data batches $\{\boldsymbol{x}_{p_1}\}_{m=1}^{M}$ are processed in parallel on M workers $p_1, \ldots, p_M$. Data parallelism assumes the data points in $\boldsymbol{x}$ are *i.i.d.*, and it achieves the goal of parallelization by *scanning more data in unit time*.

Data parallelism requires *synchronization* support to collect the updates generated at each worker $\Delta_{\mathcal{L}}(\boldsymbol{\Theta}, \boldsymbol{x}_{p_m})$, then applies them together to update the model parameter. Note we deliberately omit the superscript t in Equation 2.2, as in data parallelism, the parameter update does not necessarily follow a strictly serial structure, and usually allows for asynchrony, i.e., the local parameter state of each worker at step t does not necessarily observe the updates from all other workers at the previous step. The degree of asynchrony is controlled by a *consistency model*, introduced in Section 2.4.1. Since workers might map to devices located across a cluster, synchronization involves communication across devices or via a network, and might be very expensive. Various optimizations and systems have been proposed to address this communication problem, and are reviewed in Section 2.4.

Many ML training algorithms process multiple samples at each training step instead a single sample (i.e., $|\boldsymbol{x}_p| > 1$), data parallelism hence appears ubiquitously in nowadays ML programs, even when the computation happens only on a single device, e.g., in the form of processing a mini-batch of samples in parallel on an accelerator device.

Model Parallelism

Model parallelism, on the other hand, refers to strategies that parallelize different parts of computation in $\Delta_{\mathcal{L}}$ (on one $\boldsymbol{x}_p$) using multiple workers. It aims to achieve the goal of parallelization,

either by accelerating the computation of $\Delta_{\mathcal{L}}$ using more resources, or by enabling more complex structures in $\Delta_{\mathcal{L}}$ that a single device memory does not suffice to support.

Depending on the computational structure presented in $\Delta_{\mathcal{L}}$, model parallelism approaches can exhibit rich varieties in a case-by-case basis. We list three common examples: (1) when the model $\mathbf{\Theta}$ is a multi-layer neural network, one can dispatch the forward-backward computation of different layers onto different workers; (2) when $\Delta_{\mathcal{L}}$ contains large computational operations, such as the multiplication of two huge matrices (`matmul`), that cannot fit in the memory of a single device, one can partition the big `matmul` into multiple `matmul`s over smaller submatrices, and let each worker take care of one part therein; (3) Some ML programs, such as matrix factorization [182], exhibit patterns that the update equation $\Delta_{\mathcal{L}}$ only accesses and updates a small part of $\mathbf{\Theta}$ per step, which allows different parallel workers to work on disjoint subsets of $\mathbf{\Theta}$ and update them simultaneously.

Like data parallelism, model parallelism also requires expensive cross-device communication to collect and synthesize the partial results generated at each parallel worker. Unlike data parallelism, the types of messages communicated between parallel workers depend on the specific parallelization approach (whereas in data-parallelism, they are always model updates), and the effectiveness of model parallelism (i.e., strong scaling) approaches largely reply on the problem size and how much parallelization opportunities are present in $\Delta_{\mathcal{L}}$ – usually they are more effective on models that are sufficiently large and complex.

While data parallelism and model parallelism focus on parallelizing different elements in $F(\cdot)$, we emphasize that they are *not mutually exclusive*, and can be combined depending on the requirements of the parallelization workload. We discuss systems that combine them in Section 2.4. How to optimally combine them is also a focus of this thesis.

2.1.4 Trends in Machine Learning

ML development in the past decade flourishes more rapidly than ever before. Due to the growing demands on modeling more complex structures in ever-increasing data, state-of-the-art ML models and algorithms evolve with more and more complexity (i.e., more complex $f(\cdot)$ or $r(\cdot)$ in Equation 2.1) reflected in four dimensions: more computations, more memory footprints, more structures, and more composability.

More Computations

OpenAI published an online article [161] that studies the correlation between AI and computation in the past decade. The analysis concludes with two key observations: (1) the amount of compute used in the largest AI training runs has been increasing exponentially with a *3.4 month doubling time*; (2) Since 2012, this Petaflop/s-days meric for state-of-the-art AI models has grown by more than 300,000x (a 2-year doubling period would yield only a 7x increase).

Figure 2.1 illustrates this trend by mapping the "models of the year" to their Petaflop/s-days metric – it clearly indicates that the computation spent on ML represents an increase by roughly a factor of 10 each year.

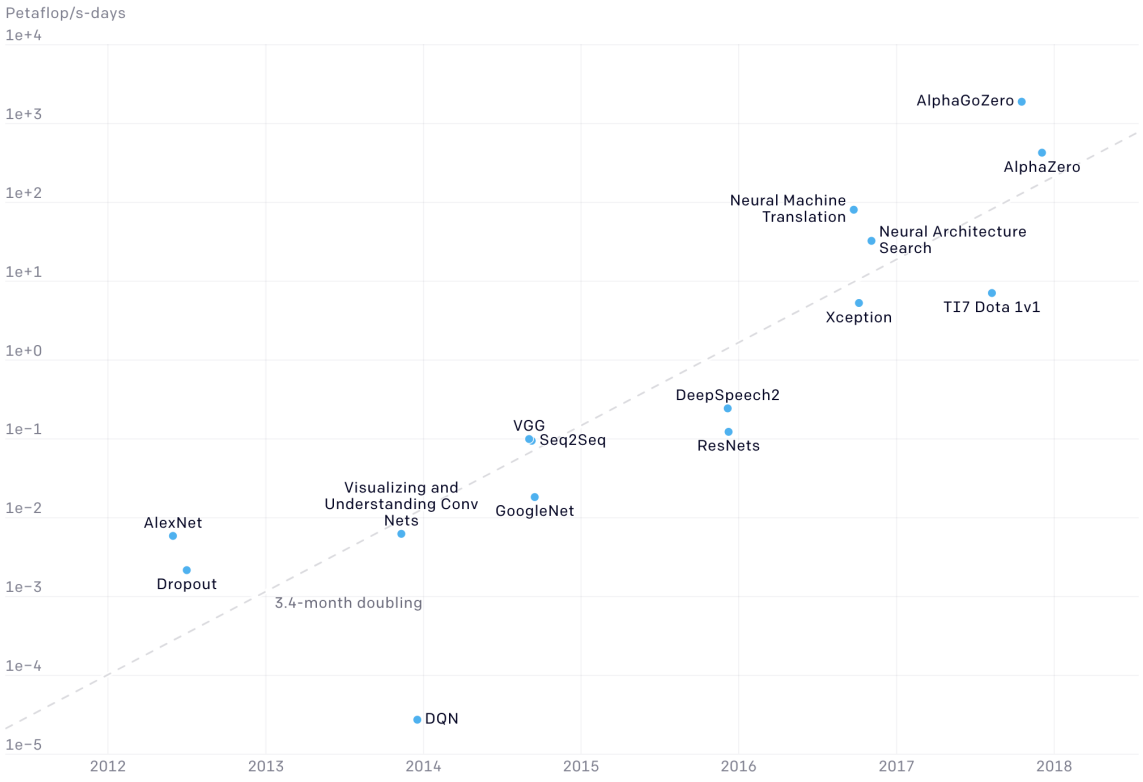

Figure 2.1: The illustration of the computational cost for the most notable ML models developed in each year from 2012 to 2018. The Y-axis is in $\log_{10}$ scale. This figure measures the computational cost using the metric *Petaflop/s-days*, which consists of performing 10^{15} neural net operations per second for one day, or a total of about 10^{20} operations. The compute-time product serves as a mental convenience, similar to kW-hr for energy (figure generated by OpenAI [161]).

More Memory Consumption

More computations naturally come with more memory footprints. Figure 2.2 [11] illustrates notable image classification models developed in the past decade, measuring their top-1 classification accuracy (Y-axis) on the ImageNet dataset [49], computation G-FLOPs taken at a single forward pass (X-axis), and number of float parameters (the size of the colored ball). Figure 2.3 [5] plots the number of parameters of models that have achieved breakthroughs in language modeling in the past 5 years. From both figures, we observe a consistent correspondence between the performance on CV/NLP applications and the number of model parameters: more parameters are likely to achieve higher performance, for example, in language modeling, the number of parameter increase *10x every 18 months*.

Oftentimes, more model parameters result in more intermediate states generated at training or inference. Consequently, holding them on accelerator devices requires more memory.

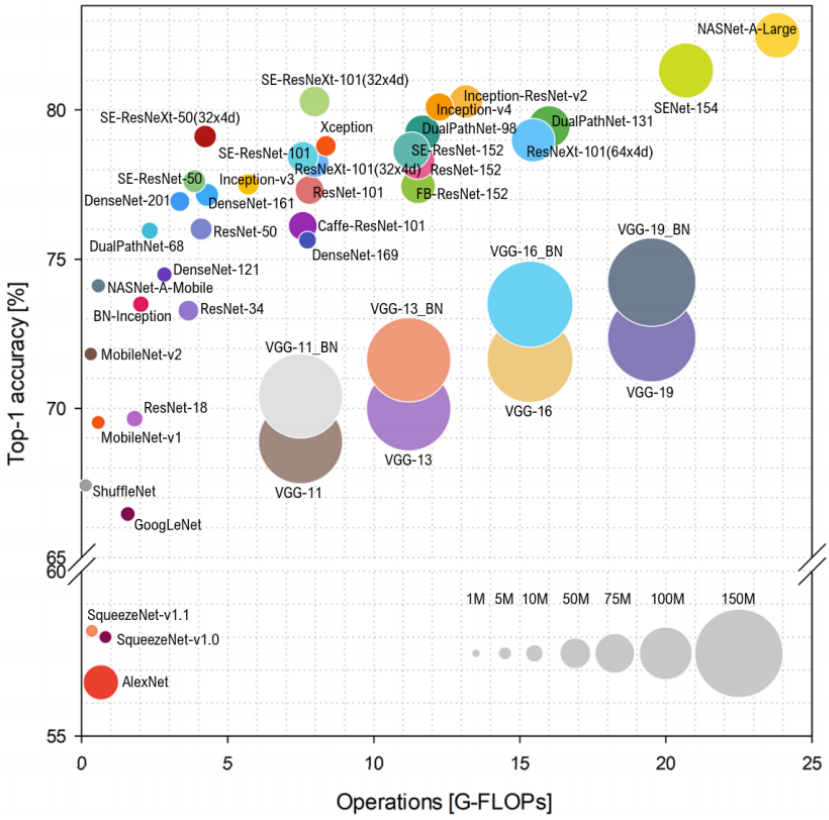

Figure 2.2: The number of parameters of notable convolutional neural networks for image classification developed in the past 10 years. The top-1 accuracy (Y-axis) is reported on ILSVRC 2012 [49], and the G-FLOPs (Y-axis) are measures on one forward pass of computation. The figure is created by Bianco et al. [11].

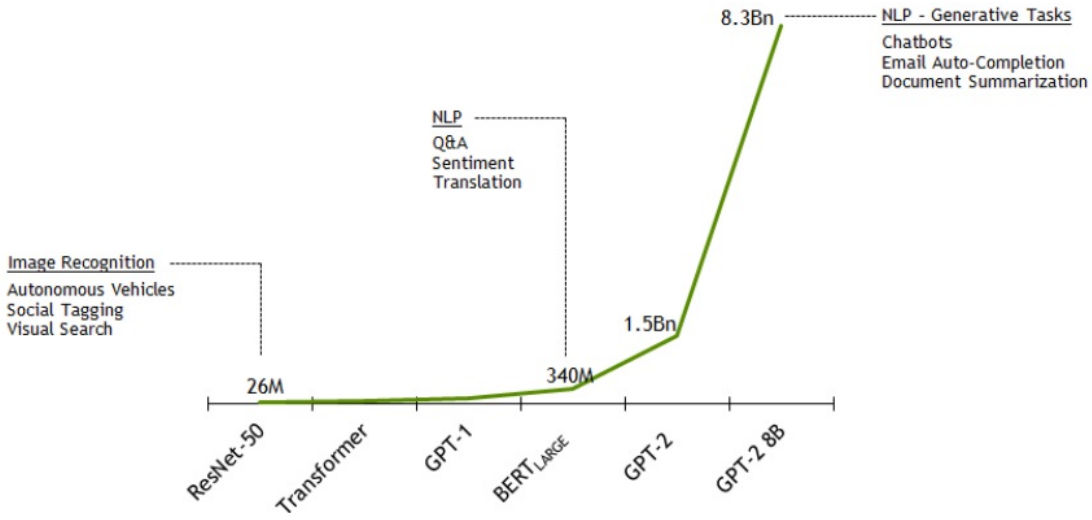

Figure 2.3: The number of parameters of state-of-the-art language models developed in the past 5 years. ResNet-50 [80] is used as a baseline. This figure was generated by NVIDIA [5].

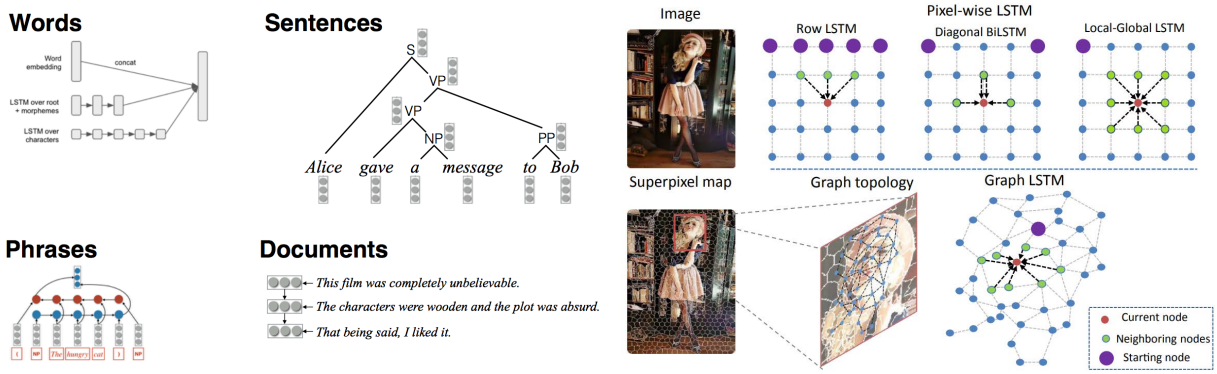

Figure 2.4: Contemporary ML models tend to exhibit more structures. (**Left**): in many NLP tasks, the model architecture changes with the structure of data it tries to fit with, e.g., words, sentences, phrases, and documents. (**Right**): in semantic segmentation, an image can be modeled as dense pixels using convolutions [135], or as a fully connected graphs using Graph-LSTM [130] where each node of the DAG corresponds to a superpixel. The figure is compiled based on figures from Neubig et al. [158] and Liang et al. [130].

More Structures

The basic ML model building blocks (e.g., convolution, attention, transformers) emerged in the past decade demonstrate increasing structures compared to classic, plain ones twenty years ago (e.g. linear regression [207] or support vector machines [35]. This is not surprising considering the demands of ML have significantly grown, and modern ML tasks aim to uncover the hidden structures in all kinds of data from various modalities. For ML models to be effective, it is necessary to cope the model design with such domain-specific structures.

For examples, convolution-based layers that encode data layer-by-layer from fine-grained details to coarse representations is suitable to extract high-level semantics from thousands of image pixels; transformers stack multiple self-attention layers so to capture the long-range dependencies presented in sequences. More examples are visualized in Figure 2.4.

Depending on these inherent structures, different ML building blocks exhibit distinct computational patterns or algorithmic characteristics, which dictate the effectiveness of existing parallelization support, while issuing requirements for newer parallelization strategies.

More Composability

Machine learning today shifts more toward deep learning and neural networks. In the past decade, ML engineering has been significantly standardized as a process of *composing a novel architecture using basic neural network building blocks*, in analogy to Lego. As a consequence, nowadays ML models present a high degree of composability. As an example, Figure 2.5 visualizes the state-of-the-art object detector, DETR [20], is composed of at least five existing NN building blocks: the CNN backbone (from image classification), the positional embedding (from language modeling), the transformer encoder, the transformer decoder, and feed-forward prediction heads.

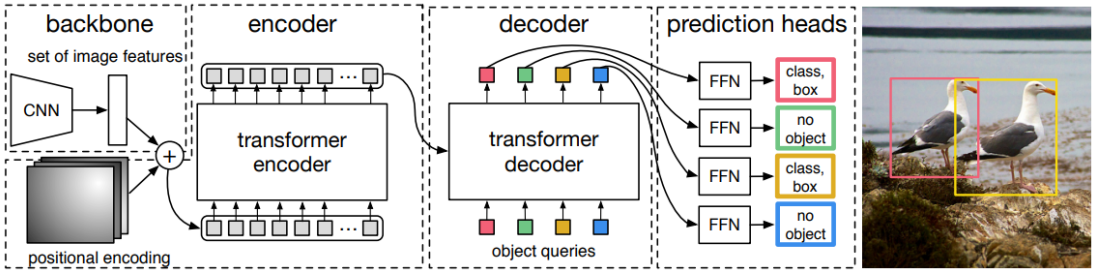

Figure 2.5: The architecture of the latest end-to-end transformer-based object detector (DETR). The diagram is created by Carion et al. [20].

This trend allows the ML community to continuously make progress through two ways: (1) composing novel model architectures from existing building blocks to achieve better application results; (2) developing and making available novel ML building blocks and algorithms. Figure 2.6 shows a typical evidence suggesting that simply composing model architectures using *existing* building blocks can cause a catalytic effect – by composing more transformer decoders [209], the GPT model [17] gradually acquires predictability in low (one or zero) shot settings (i.e., setting with very few training data) on par with models trained with massive labeled data.

To summarize this section, we anticipate ML development in the next decade will continuously grow following the trends described above, which drastically necessitates ML programs to run on massive distributed parallel environment, but also poses new challenges on the design of both hardware and software systems to cope with these trends.

2.2 Hardware Environment for Parallel Machine Learning

Typically, the hardware environment for parallel machine learning is classified into two categories: the *shared-memory computing environment* [189], and the *distributed cluster computing environment*. In a shared memory environment, computational processors (e.g., CPU cores) physically locate in close proximity, and have uniform access to a single memory space. Processors can communicate with each other through the shared memory, which has high bandwidth (400Gbps) and low latency (100ns). By contrast, a distributed cluster consists of multiple distributed nodes. Each node has one or more processors (CPU cores and GPUs) and its private memory space. Accessing the contents in the memory of a remote node must be done by distributed communication via network, which has limited bandwidth (10Gbps) and high latency (0.1ms), the latency even scales with the number of processors.

This thesis mainly focuses on developing ML parallelization techniques for distributed cluster computing environment, except in Chapter 6, where we concern the parallelization (batching) of dynamic neural networks on GPU devices.

ML computing clusters normally have two types of setups: private ML computing clusters on campus, labs or companies, and public cloud service provided by Amazon, Google and Alibaba. Figure 2.7 illustrates a shared cluster used at CMU Parallel Data Lab for research. This cluster contains 42 nodes, each with one 1 Titan X Maxwell GPU. Table 2.1 lists the recommended

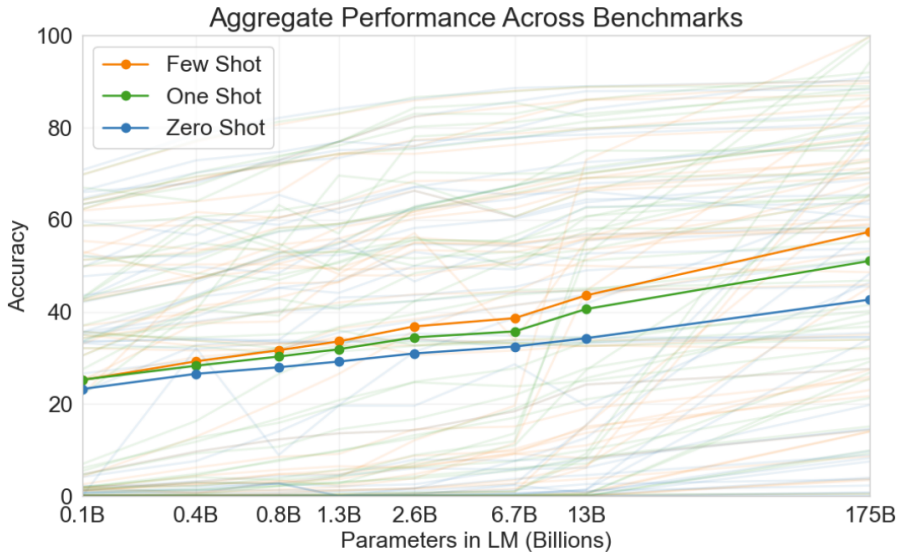

Figure 2.6: By simply composing deeper architectures with more transformer encoders, the GPT-3 model steadily improves performance in few-shot, one-shot and zero-short settings. The Y-axis reports the aggregate performance for all 42 accuracy-denominated NLP benchmarks. The figure is by Brown et al. [17].

types of instances on Amazon AWS for ML jobs, with their hardware specifications and price. Most cloud providers charge the usage of such clusters at a per-hour-node basis.

For both setups, the following characteristics are where special cares need to be taken when designing ML software systems on top of them.

Heterogeneity

Instead of having the same type of processors and hardware for all nodes (i.e., homogeneous), commodity clusters for ML mix nodes with different configurations, and processors with different computational capabilities (e.g., CPUs and GPUs) for cost effectiveness and energy efficiency. Such heterogeneity poses challenges on the design of software systems, and causes a lot of factors for considerations, for examples: moving contents between GPU memory and RAM within one node is much faster than moving contents between two separate nodes, yet they are both slower than moving between two colocated GPUs connected via NVLink[1] (e.g., AWS p3.16xlarge node (Table 2.1)); some nodes in the cluster might have higher Ethernet bandwidth while others have higher computational FLOPs, rendering their appropriateness for different types of workloads; as the number of nodes increases, stragglers and failures, which rarely exist in homogeneous settings or at a small scale, become more common.

Successful software systems need to gain awareness of such heterogeneity, make trade-offs precisely depending on the cluster specifications, in order to maximize the usages of all types of nodes and handle stragglers and faults to avoid slowdowns.

[1] https://www.nvidia.com/en-us/data-center/nvlink/.

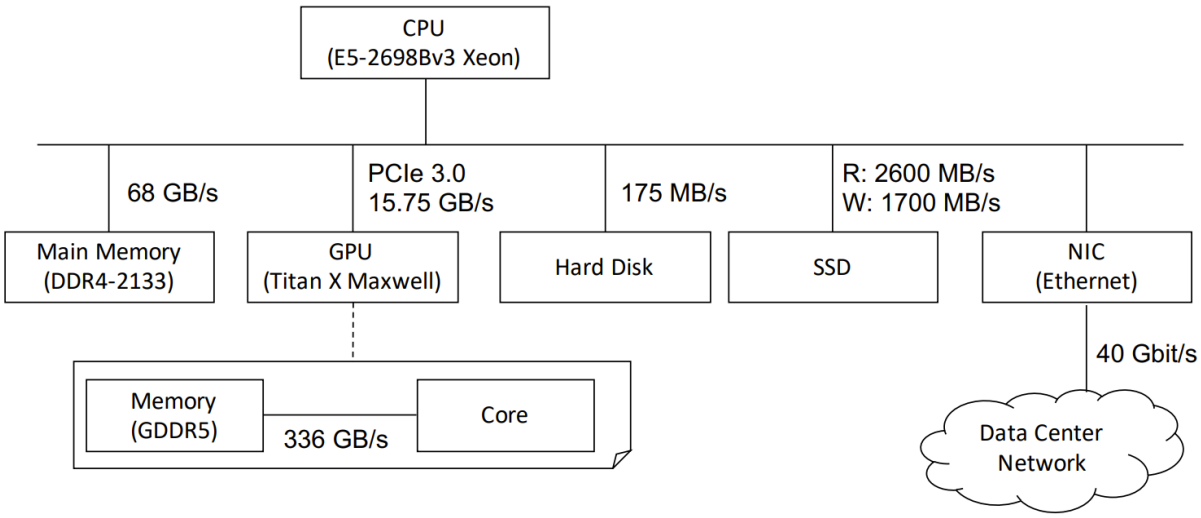

Figure 2.7: The hardware specifications of the ORCA cluster (https://orca.pdl.cmu.edu/) deployed at CMU Parallel Data Laboratory since 2016 (picture by Wei et al. [220]) for ML and SysML research.

GPUs

GPUs have become the dominant accelerator for ML workloads because their SIMD nature matches well with matrix-intensive linear algebra in ML programs. A commodity Pascal NVIDIA TITAN X provides 11x more TFLOPs than the peak performance of the most high-end CPU, such as Intel Xeon E5-2699 v4 [126]. While GPUs are more computationally powerful, the bandwidth between GPUs and RAM is limited compared to that between CPU and RAM (see Figure 2.7). In order for a GPU program to be efficient, the contents needed for computation must reside on the GPU's *dedicate memory*. Compared to RAM, GPU memory is limited and extremely expensive – due to its design, GPU memory cannot achieve high bandwidth and high capacity simultaneously; while GPUs with more FLOPs are being released every year, their capacity barely changes – GPUs installed in private clusters or on public clouds have either 12GB or 16GB memory.

These properties of GPUs have many implications for software systems. First, the improved computational capability of GPUs stresses other hardware components in the cluster that might be less a problem for non-ML workloads. Second, GPU memory is a scarce resource that needs to be carefully managed by ML software systems. Third, the dedicate memory of GPUs complicates the communication between parallel workers, as the messages need to be additionally moved between RAM and GPU memory before inter-node communication.

Limited Network Bandwidth

It can be clearly observed from Figure 2.7 and Table 2.1 that the inter-machine communication bandwidth (NIC) on both ORCA and AWS is much lower than the bandwidth between other components. In fact, most private clusters install Ethernet with $<$10 Gbps bandwidth, whereas public cloud instances have 1 to 50 Gbps bandwidth. Upgrading the bandwidth is economi-

Instance	GPUs	# CPU threads	RAM (GiB)	GPU memory (GiB)	Bandwidth (Gbps)	Price / hr / node ($)
p2.xlarge	1x K80	4	61	12	≈ 1	0.9
p2.8xlarge	8x K80	32	488	12	10	7.2
p3.2xlarge	1x V100	8	61	16	<10	3.06
p3.8xlarge	4x V100	32	244	16	10	12.24
p3.16xlarge	8x V100	64	488	16	25	24.48
p3dn.24xlarge	8x V100	96	768	16	100	31.218
g4dn.xlarge	1x T4	4	16	16	<25	0.526
g4dn.12xlarge	4x T4	48	192	16	50	3.912
g4dn.metal	8x T4	96	384	16	100	7.824

Table 2.1: Nine types of instances recommended by Amazon AWS (https://docs.aws.amazon.com/dlami/latest/devguide/gpu.html) for ML workloads. Their detailed specifications are listed on each column, including: instance name, number of GPUs and GPU type, number of CPU threads, RAM capacity, per-GPU memory capacity, Ethernet bandwidth on each node, and their per-hour-node price.

cally prohibitive, and is bounded by the highest available bandwidth in the market – the highest 100Gbps NIC switch costs at least $8800, depending on vendors, brands, and specifications. The same cost can afford nearly 12 NVIDIA RTX 3080 GPUs (released in 2020).

In iterative-convergent distributed ML workloads, massive float values are generated and needed to be exchanged between nodes per iteration. The unmatched development between Ethernet NICs and GPU accelerators result in a *communication bottleneck*. Therefore, reducing communication, avoiding this bottleneck, and making the best use of available network bandwidth have been a main theme in the development of distributed ML systems.

Programming on Parallel Environment

Writing efficient code on top of a heterogeneous cluster requires carefully orchestrating the resources across the cluster, and poses substantial barriers for ML practitioners due to its required domain expertise. The barrier is amplified when coupled with ML code. Desirably, the software systems should provide APIs that can hide irrelevant low-level system sophistication, while maintain the flexibility to support various types of ML algorithms and cluster setups, and debuggability to quickly diagnose problems during prototyping. The ideal software interfaces should enable users to write distributed ML code as if they were programming on their laptop, but with computational capability empowered by clusters.

2.2.1 Trends in ML Hardware Infrastructure

The rapid development of ML has also driven significant evolution of accelerator hardware. The most notable trend on ML hardware is to *trade generality for performance*. People have built increasingly complex but specialized accelerators designed for regular parallel computation

(such as linear algebra). Example accelerator features include "warps", blocks, and grids of threads, very wide vector arithmetic units (ALUs), and systolic array multipliers (MXUs) [8], as observed in the emergence of more powerful GPUs every year, TPUs [103], and other AI-specialized chips.

In the past decade, newer generation of GPUs had been released every year and prolonged *the Moore's law*, a long-held notion for the development of computer processors. However, comparing Moore's law to Figure 2.1 reveals a substantial gap between the demand of ML computation and the reality of ML processors: ML computation increases *35x every 18 months*, yet GPU computational power increases only *2x every 18 months* following Moore's laws. Unfortunately, the upper limit of GPU computational power is constrained by physical laws. In 2019, Nvidia CEO announced that "Moore's Laws isn't possible anymore" at the release of NVIDAI RTX2060, indicating the computational power increase of GPUs will flatten in the next a few years. Despite computational accelerators, new interconnect technologies, such as NVLink, NVDirect, 100 Gbps Ethernet, RDMA, and Infiniband, are deployed to meet the communication demands of various ML applications.

Arguably, these trends sign that the only possible way to supporting ever-growing demands of ML workloads is by massively distributed parallel computing.

2.3 Machine Learning Frameworks

The rapid advancement of ML owes much to the development of software frameworks that enable ML developers to quickly prototype their ideas on hardware accelerators and large datasets. In this section, we discuss a few notable software systems for ML prototyping. We call these systems *ML frameworks*, to distinguish them from those distributed systems dedicated for scaling up ML onto parallel environments (discussed in Section 2.4). We summarize their common characteristics and some ongoing development effort.

2.3.1 Earlier ML systems

Before the general ML community moved to deep learning, several families of ML models had been deployed as essential components in industry use cases. A few systems hence have been designed to allow quickly prototyping and training these models, such as LibSVM [21] for support vector machines, XgBoost [26] for gradient boosting trees, DistBelief [46] for deep neural networks, CUDA-ConvNet [117] for GPU-based CNNs, etc. The design of these frameworks specializes to the classes of ML models they correspond to, and their exposed interfaces cannot flexibly extrapolate to other models, newer training algorithms, and may be based on very low-level programming languages, such as CUDA.

Several earlier, more general ML prototyping systems originate as extensions to some well-known batch or graph processing systems, such as Hadoop [45], Spark [239], and GraphLab [61, 137]. Notable examples include Spark MLlib [144], an attempt to support many machine learning models on Spark. Since ML programs exhibit very different characteristics than traditional data processing programs (e.g., frequently updated parameter states), running ML workloads on top of these systems is usually found not ideal regarding performance and programmablibity.

2.3.2 Modern DL Frameworks

The main theme of ML development in recent years turns to deep learning. As the corner stone of DL, neural networks stand out from other classes of models for a few *engineering* reasons. First, NNs stack many atomic computational functions to express a more complex function, offering a natural abstraction of "layers" as composable and repurposable units, and a simple programming model of constructing a neural network as composing atomic layers. Second, no matter how complex the model grows, the training algorithm follows backpropagation, which can be standardized as a single set of auto-differentiation rules but applied on a variety of models. Third, the computation in NNs involves dense linear algebra that happens to match well with GPU accelerators, and is by nature amendable to data-parallelism. These reasons motivate the development of a series of deep learning frameworks that have significantly lowered the barrier of DL programming and brought DL closer to industry production. We group these frameworks into two categories, earlier pioneer DL frameworks such as Caffe [97], Torch, and Theano [204], and contemporary frameworks such as TensorFlow [1], PyTorch [165], and MXNet [27].

Earlier frameworks mostly abstract neural networks as a stack of *layers*, and provide expert-optimized implementations for these layers on CPUs and GPUs. Users program NNs by composing these layers in predefined scripts such as ProtoBuf [64], or using preliminary interfaces in the programming language the framework is built on, such as Lua or C++. The NNs are then trained using built-in hard-coded auto-differentiation algorithms. Given "layer" as the basic entity, this abstraction makes it difficult for ML developers to define new layers or new training algorithms, or refine existing ones. These frameworks also have limited or no support for devices beyond CPUs and GPUs.

Evolved from earlier frameworks, contemporary frameworks overcome these obstacles by using an improved architecture and a more principled representation, summarized below.

Dataflow Graph as the Principled Representation

These frameworks adopt *dataflow graphs* [154] as the principled representation for DL models. In a dataflow graph, the most atomic entity is a primitive mathematical operation, such as addition, matrix multiplication, or convolution, corresponding to a node in the graph. The edges indicate the direction of the data being passed between the nodes. The computation of a DL program is thus expressed as batches of data flowing through the computational graph. Besides stateless computation operations, the dataflow graph also contains stateful operations such as *variable* and *constant* operations. Besides data inputs and outputs, the computation graph may also contain *control dependency* edges to enforce ordering among operations.

Dataflow graphs make it easy for users to define new layers and new training algorithms using primitive operations in high-level language (e.g., Python). The backpropagation algorithm for user-defined layers can be auto-derived following the backpropagation chain rule. In addition to programming convenience, dataflow graph opens substantial space for performance optimizations. In frameworks such as TensorFlow and MXNet, the dataflow graph is declared at once by the user using symbolic interfaces, right before execution (a.k.a. *deferred execution*). Hence the framework can obtain a global view of the ML computation, and incorporate various optimizations, such as node fusion and graph reduction, based on static graph analysis.

On the other hand, this declarative programming mode demonstrates inflexibility when the ML computation patterns change with data samples. Frameworks such as DyNet and PyTorch hence break this rigidness, offer imperative programming interfaces, and let the graph construction and execution happen simultaneously. Besides flexibility, imperative programming improves debuggbility, as users can evaluate an expression and see the results immediately after they declare it, and make more informed decisions (on what to do next) depending on the results. These interfaces, though being more "Pythonic", miss substantial performance-improving opportunities in declarative mode.

Prior to PyTorch, Chapter 6 of this thesis extensively studies the problem of programming and training dynamic neural networks, and develops pioneer system implementations such as DyNet [158] and Cavs [244].

A Three-layered Architecture

Vaguely speaking, all modern DL frameworks follow a consistent three-layer architecture, with small variations. From frontend to backend, they are: programming API, runtime, operator definitions and implementations. We briefly introduce the functionalities belonging to each layer.

The programming API layer provides APIs for composing models in various high-level programming languages, such as Python, R, Swift, Scala, JAVA, Julia, etc. These APIs are mostly implemented as wrappers of the framework's original C++ interfaces. They cover calling methods to a diverse set of primitive operators, and methods on building graphs. Through these APIs, users compose their model as a computational graph, and the graph is taken by the runtime layer for a few (optional) optimizations, such as graph pruning, compilation, until it is ready to be evaluated. The runtime layer also includes the necessary graph-level functionalities for triggering and coordinating the execution of each node or edge following graph structures, specifically concerning memory management, scheduling, or distributed communication, etc. The implementations of various supported operators are maintained as a library in the third layer. Each operator is usually abstracted as two parts: a definition of the operator stating its device-agnostic semantics, such as input/output requirements, shape inference functions, etc., and device-specific implementations of the operator on CPUs, GPUs, TPUs, mobiles, FPGAs, etc.

With this abstraction, modern frameworks can offer simple high-level programming interfaces, easily extrapolate to newer models or algorithms, while support various types of hardware accelerators, all achieved in one single runtime system.

2.3.3 Trends in ML Frameworks

DL frameworks are fast evolving as well when DL is expanding to new application domains. While there used to be diverse effort on developing different frameworks or programming interfaces for ML, we are observing several trends of convergence: (1) Python is becoming the dominant programming language for ML, due to its simplicity and many third-party libraries on scientific computing. (2) TensorFlow and PyTorch hold the most market shares compared to other frameworks, as they offer the most ML-related resources such as template implementations, pretrained model weights, and community support.

Given these convergence, the current frameworks are being developed with two main focuses: (1) to develop better backends to satisfy the increasing demands on computational efficiency for training and inference at *production* phases, (2) to offer more friendly Python interfaces to fasten the research and development of newer ML models and algorithms at *prototyping* phases.

DL Compilers

One way to improve the runtime performance is to manually optimize the computational graph using compilation techniques such as kernel fusion, constant folding, common subexpression elimination, and other graph optimizations. Systems such as TensorFlow XLA [66], Grappler [119], Glow [178] perform these optimizations on dataflow graphs and significantly improve the execution performance. The other way is to manually optimize the implementation of specific operator kernels, such as those provided in newer versions of Intel DL Boost, and NVIDIA cuDNN[2].

These optimizations are expert-designed based on heuristics, and targeting at well-established models and training algorithms. When new operators, models, algorithms, or accelerator devices emerge, they need to be re-designed, which again needs strong expertise and expensive human time. Thus, compiler systems such as the latest XLA, TVM [29], TASO [100], and TensorComprehensions [208], propose to automate the optimization of DL graph execution and operator implementations on various types of devices, using search or ML-based methods. They mostly target at inference time when variable states in the graphs are trained and fixed. A contribution of this thesis is to propose the first compiler-like architecture and auto-optimization solution for distributed training, which is elaborated in Chapter 7 and Chapter 8.

Reconciling Programmability and Performance

Achieving programmability and performance seem to be two conflicting goals: optimizing DL performance requires a static representation of the DL model, which is however not available in imperative style programms in Python. In fact, TensorFlow and PyTorch represent the two frameworks on the two extremes of the spectrum between declarative and imperative programming. Ongoing development on DL frameworks tries to find sweetspots in between, and reconcile these two goals in one framework.

TensorFlow 2.0 migrates from declarative mode to *eager mode* [3], and allows imperative-style programming using TensorFlow graphs and operations. In addition to eager mode, it also introduces a `tf.function` Python decorator, that performs static analysis [151] of the TensorFlow code written in eager mode in a function, and creates static representations for downstream optimizations or just-in-time (JIT) compilation. Similarly, PyTorch introduces TorchScript that traces the execution of imperative PyTorch programs, and generates static computational graphs corresponded to the programs. Based on the same set of techniques (tracing and static code analysis), some emerging new frameworks, such as JAX [16], allow automatic differentiation and JIT over the native Python code (e.g., Numpy).

How to precisely extract static descriptions for imperative Python code is still an open problem. While TensorFlow, PyTorch and JAX partially achieve it by combining tracing and code

[2] https://developer.nvidia.com/cudnn.

analysis, they still exhibit many limitations. For example, tracing only provides a snapshot view of the program, thus cannot capture the dynamics presented in a Python program, such as dynamic control flow [236] and impure functions. A noteworthy line of work try to bridge this gap by building more generic intermediate representations for ML program compilation, such as the MLIR stack [120], XLA HLO [66], and TVM Relay [176]. They are still at a very preliminary stage.

Despite these limitations, these efforts are gradually shaping the ML development into two phases: prototyping phase and production phase. In prototyping phase, ML code is developed in imperative mode, with enhanced debuggability and flexibility. In production phase, they are optimized in declarative mode, and deployed for massive training and inference at larger scale – based on these interfaces, the code conversion could be done by simply altering a few lines of code within one framework.

To summarize this section, most techniques developed in this thesis try to cope with the characteristics of contemporary frameworks described above (e.g., using dataflow graphs as the main representation), and aim to optimize the distributed system design by drawing insights from the trends in the development of DL frameworks.

2.4 Distributed Systems for Large-scale ML: A Review

In this section, we provide an overview of existing ML parallelization techniques and optimizations. We organize the contents as follows: we first discuss distribution strategies, optimizations, and system implementations for ML parallelization in Section 2.4.1. We then compare existing programming interfaces for distributed ML in Section 2.4.2, followed by a summarization of recent development trends on these topics in Section 2.4.3. For some topics closely related with the research in this thesis, we defer a more comprehensive discussion of related works to each corresponding chapter.

2.4.1 Strategies for ML Parallelization

Strategies for parallelizing ML need to tackle the typical problems in each aspect of a conventional distributed system. These aspects include: communication, consistency, memory management, scheduling, and their complex interplay. We organize our review loosely following this structure.

Communication Architecture

Overcoming communication bottleneck is a holy grail problem in distributed machine learning. In data parallelism, the communication follows a regular pattern that involves synchronizing parameter updates across distributed devices at every update iteration. This synchronization support is usually provided by two types of architectures: a bipartite parameter server architecture, and collective AllReduce/AllGather architectures with predefined topology.

Parameter server architectures. A parameter server (PS) is a distributed shared memory system. It provides a systematic abstraction of the synchronization step in iterative-convergent data-

parallel ML algorithms. Precisely, it supports synchronization by holding the model parameter $\mathbf{\Theta}$ centrally in a virtual server, mapped to multiple physical nodes with distributed shared memory. To proceed one step of computation, each worker first requests access to the most update-to-date parameter states from servers via communication. Once obtained, each worker computes the parameter updates on their local split of data following data parallelism, and then sends the updates to servers. Servers holding the shared parameter states are responsible for applying the updates (F in Equation 2.2) and keeping the *freshness* of the parameter states. Following this abstraction, a parameter server is flexible to support diverse iterative-convergent ML workloads using a simple set of client APIs, including: methods to pull parameter states from remote servers, and methods to push updates to servers.

However, in order to achieve high system performance, the detailed implementation of a parameter server can go very sophisticated, and is an active field of research. The earliest PS prototype, Yahoo!LDA [4], was implemented as a specialized system to support the distributed inference of LDA, and does not provide explicit APIs for pulling parameter states or pushing parameter updates. DistBelief [46] is the first general-purpose parameter server system that scales the training of deep neural networks to 10000s of CPU cores. Li et al. [127, 128], Ho et al. [83], and Cui et al. [38] concurrently proposed three open-sourced PS-based systems, namely parameter server, SSPTable, and LazyTable, respectively, that support flexible consistency models (introduced later) for synchronization. Their implementations exhibit different pros and cons. Later parameter server systems such as Bösen [222] and IterStore [39] significantly improve PS performance by introducing managed communication [220] and leveraging the iterative access patterns of ML programs.

Existing parameter server systems usually assume CPUs as the major devices for computation, and target at a few classic ML models and training algorithms. A core focus of this thesis is to develop new and efficient techniques to support more contemporary DL workloads, and heterogeneous GPU clusters.

AllReduce/AllGather architectures. As an alternative, collective communication primitives are brought from HPC to ML recently [6, 186], attributing to their existing mature libraries and implementations. Ring *AllReduce* and *AllGather* (AllGather is the sparse version) are the two most adopted algorithms to synchronize gradients in data-parallel training.

In the Ring AllReduce algorithm [186], a virtual ring connecting all workers is constructed. Each worker in the cluster communicates with two of its neighboring peers on the ring $2\times(N-1)$ times. During this communication, a node sends and receives partial parameter updates for its peers. In the first $N-1$ iterations, received values are reduced to the local values in the node's buffer. In the second $N-1$ iterations, received values replace the values held in the node's buffer – up until this point, all workers obtain a globally consistent parameter updates – reduced from the local updates of all workers. Then, each worker applies the updates to its local parameter states and proceed to the next iteration. As long as the parameters $\mathbf{\Theta}$ are initialized from the same state on all workers, synchronous consistency is guaranteed. Depending on the device and network topology, the collective communication can be further altered to follow different communication or reduction ordering, such as tree-structured [166] and butterfly AllReduce [248], or to communicate different types of messages other than gradients using appropriate collective primitives, such as *Broadcasting sufficient factors* [226, 241].

AllReduce or AllGather-based communication architectures feature bandwidth optimality, and superior performance on high-performance devices such as GPU-to-GPU communication, due to the availability of specialized libraries on such hardware (e.g., NCCL [156]). Compared to PS, they are however less flexible: they cannot offset the communication tasks to dedicate "server" machines as every worker simultaneously accounts for computation and communication; they assume fully synchronous consistency, which sometimes can be relaxed to improve system performance, discussed in the next section.

Horovod [186] is one of the most popular open-source systems based on collective AllReduce and AllGather. Later works on this line mostly focus on training dense models such as CNNs, and push the limit of ResNet training time from several days to one hour [68], minutes [95, 234], and even seconds [232].

Hybrid architectures. Several recent works propose to hybridize different communication architectures. The core observation made in these works is that different communication architectures lead to different communication loads, depending on which model component they are applied on and the ML characteristics of these components. Leveraging the composability (e.g., layered structure) presented in ML programs, a best-of-both-worlds architecture can be derived by selectively applying one communication architecture to one part of the model, whenever it yields less communication. Two notable systems that implement such strategies are Poseidon [245] and Parallax [107], which combine the PS architecture with collective broadcasting and AllReduce, depending on the type, shape, and the parameter sparsity of model layers.

Consistency

Machine learning programs exhibit one unique characteristic: the training of many ML models can tolerate errors, such as faulty or delayed updates $\Delta_{\mathcal{L}}(\boldsymbol{\Theta}, \boldsymbol{x}_p)$ occasionally happened at some worker p in Equation 2.2, as long as the error is appropriately bounded. This *bounded-error tolerance* allows trading the synchronous consistency of the distributed system for system performance.

Bulk synchronous parallel. Bulk synchronous parallel (BSP) is the most commonly used synchronization consistency model for data-parallel training. Under BSP, each worker alternates between computation and synchronization following the same pace. Mapped to Equation 2.2, each worker p_m performs a step of computation at t, then enters the synchronization phase and propagates its local updates $\Delta_{\mathcal{L}}(\boldsymbol{\Theta}, \boldsymbol{x}_{p_m})$ to all other workers, and receives updates $\Delta_{\mathcal{L}}(\boldsymbol{\Theta}, \boldsymbol{x}_{p_1}), \ldots, \Delta_{\mathcal{L}}(\boldsymbol{\Theta}, \boldsymbol{x}_{p_M})$ from others as well (which could be realized by any communication architecture described above). Before all workers observe the updates from the other workers, they *will not* proceed to the next iteration $t + 1$. Hence, the core notion in BSP is that the synchronization phase does *not end* until all workers finish communicating updates.

Such a parallelization model is simple, and preserves the sequential semantic of the original single-node program if every worker computes on an independent set of training data – under BSP, adding more workers is equivalent with increasing the batch size. However, it is prone to stragglers, since all workers proceed at the speed of the slowest worker.

Totally asynchronous parallel. Totally asynchronous parallel (TAP) [41] is a consistency model

on the other extreme opposite to BSP. In order to alleviate the waiting for stragglers, in TAP, a worker propagates parameter updates and fetches new parameter values without waiting for other workers. Hence, a worker is not guaranteed to observe the updates of other workers of a previous step. This mechanism can be easily implemented under a parameter server architecture, by letting each worker hold locally cached version of parameter states, and update with parameter servers only when appropriate [38, 173, 222]. TAP achieves high system throughput by sacrificing the freshness of parameter states. When the sacrificed freshness (a.k.a. *staleness*) is unbounded and grows large, statistical efficiency cannot be guaranteed any more, and the model training diverges.

Stale synchronous parallel. Stale synchronous parallel (SSP) explores the sweetspots in the middle between BSP and TSP. In particular, under SSP, the fastest worker cannot be more than s steps ahead of the slowest worker, where s is an auxiliary parameter called *staleness*. SSP can be realized by incorporating a relaxed synchronization barrier on distributed workers, so that a worker cannot proceed to the next step of computation if its local parameter state is more than s steps stale, until it synchronizes with other workers. Previous work [41, 83] prove that, under mild assumptions, convergence is guaranteed for certain models when s is properly bounded.

In this thesis, we refer staleness as a parameter that controls the degree of consistency in parameter synchronization – when staleness equals 0, we achieve BSP; the consistency model is TAP when staleness is infinitely large. We extensively study the effect of staleness in Chapter 5.

Memory

The growing memory footprints of more complex DL models pose new challenges on how to enable their training on limited-memory accelerators. Besides model parallelism approaches (discussed later), we review a few memory optimizations for DL training, which is also a main topic in Chapter 4 of this thesis.

Existing memory optimizations for DL training or inference can be classified into three categories: gradient accumulation [97], memory swapping [40, 86, 143, 175, 215], and rematerialization [28, 93]. They are all based on one critical observation: the main memory consumption of DNN computation stems from storing the *intermediate activation* of each layer generated at the forward pass. All three classes of methods try to minimize the memory footprints of these intermediate states.

Gradient accumulation. Since the size of the intermediate activations grows linearly with the batch size, gradient accumulation reduces the memory footprints by simply splitting a mini-batch into many smaller *micro-batches*. It then performs many forward-backward passes over each micro-batch. Since the gradients are additive over samples, they take constant memory to store, and can be accumulated across all micro-batches to recover the gradients estimated on the original mini-batch. In the extreme case, gradient accumulation sets a micro batch size of 1, and takes as many passes as the original batch size, to afford large model training. As a major disadvantage, gradient accumulation might not be able to fully utilize accelerator FLOPs due to using smaller batches. However, because of its simplicity, gradient accumulation is built-in with almost every DL framework.

Rematerialization. Rematerialization trades extra computation to save memory. Instead of storing all intermediate states, rematerialization-based methods only retain the intermediate activations of a few critical layers as checkpoints, and discard the rest, which can be recomputed, when needed, from its closest previous checkpoint layer. A series of rematerialization methods have been proposed; they usually focus on optimally selecting which and how many layers to checkpoint based on the model structure, so to best trade-off between memory usage and recomputation cost. Chen et al. [28] propose a heuristic for checkpointing idealized unit-cost linear n-layer NN graphs with $O(\sqrt{n})$ memory usage. Jain et al. [93] propose CheckMate to generalize rematerialization as a mixed integer linear program.

Memory swapping. The memory swapping method (originally developed in Chapter 4 of this thesis) reduces device memory consumption by offloading parameters and activation values, from expensive device memory to cheaper host memory, and loading them back to device memory only when they are needed. In addition to reducing the memory usage of intermediate states, memory swapping can also reduce the memory footprint of parameters. We defer the technical details of memory swapping to Chapter 4. This method however triggers overhead due to swaping contents between device memory and host memory. Existing research on memory swapping mainly focuses on optimizing the swapping schedule based on the ML program structure, to minimize this overhead.

GeePS [40] first brings memory swapping to distributed ML, and implements it on top of a parameter server and Caffe [97]. GeePS manages to large DNNs that require 7x more device memory than a normal GPU has. TensorFlow Grappler [143] implements memory swapping for optimizing the memory on TensorFlow graphs. SwapAdvisor [86] performs joint optimization on operator scheduling, memory allocation, and swap decisions, and manages to train models up to 12 times the GPU memory limit while achieving 53-99% of the throughput of a hypothetical baseline with infinite GPU memory.

Scheduling Computation and Communication

The *more structures* in ML programs present opportunities on scheduling the computation and communication following these structures. For instance, in many models with layered structures, only a part of the model parameters (layer), but not all of them, are accessed at a time during computation. Instead of following a sequential computation-then-communication agenda, one can reschedule the computation and communication steps at the granularity of layers, while maintaining the original program semantics, so the computation and communication steps of individual layers are pipelined, to reduce bursty communication at a communication step.

Chapter 3 of this thesis develops a *wait-free backpropagation* approach to improve the computation communication scheduling in CNNs. Jayarajan and Wei et al. [94] present *priority-based parameter propagation (P3)*, that synchronizes parameters at a finer granularity and schedules data transmission in such a way that the training process incurs minimal communication delay. Later, Peng et al. generalize this idea, and present ByteScheduler [168] to combine scheduling with tensor partitioning to achieve optimal scheduling results in theory and good empirical performance with a parameter server architecture.

Compression

Compression is a ubiquitous way to reduce memory usage in many conventional data-center applications. ML workloads have many nice properties where new compression methods can be developed to *simultaneously* reduce computation, memory and communication.

Mixed precision. Modern deep learning systems use single-precision (FP32) format. Opportunities arise if the DL model accuracy would not be comprised when using a lower-precision representation for parts of its weights, gradients, or intermediate states. Following this idea, Narang et al. [145] present a mixed precision training approach, and show that a number of CNNs can be trained with mixed-precision floating point numbers without loss of accuracy. More aggressively, Courbariaux et al. [36] try to represent the NN weights as 8-bit integers (INT8) and perform fixed-point arithmetic for training, and show acceptable performance on maxout networks [62]. They later even propose to binarilize [37] the weights and activations. All these methods enjoy the benefits of accelerated computation and lower memory footprints, but at the cost of potential decreased statistical performance.

Gradient compression and quantization. A more relevant class of approaches to reduce communication in distributed ML is to compress or quantize the gradient messages transferred across the network. They can be classified into lossless and lossy compression methods.

Xie et al. [226] discover that in many ML programs, matrix-formed gradients often have rank 1, hence can be expressed as the outer product of two vectors. In the context of distributed communication, low-rank gradient matrices can be first *losslessly* decomposed into vectors, which are broadcasted across all workers and reconstructed as gradients. Putting together with collective broadcasting, one can derive a peer-to-peer based communication architecture, but with quadratic communication overhead with the number of workers. Jain et al. notice that the output of ReLU activation layers can be losslessly compressed as well, for lighter communication [92].

On the other hand, in a similar rationale with relaxed consistency, *lossly* compressing gradients can also reduce distributed communication, as long as the loss of the information is within the error-tolerance bound of the ML program. For examples, Gaia is a geo-distributed ML system that dynamically skips communicating insignificant gradients between data centers, while still guaranteeing the correctness of ML algorithms. Wen et al. [223] demonstrate that the gradients of AlexNet [117] and GoogLeNet [200] can be quantized using only 3 bits while maintaining satisfactory image classification accuracy. Seide [184] show that in a DNN model trained on speech data, one can, at no or nearly no loss of accuracy, quantize the gradients aggressively to one bit per value, to reduce distributed communication, as long as the quantization error is carried forward across mini-batches (a.k.a. *error feedback*). Lin et al. propose *deep gradient compression (DGC)*, which combines momentum correction, local gradient clipping, momentum factor masking, and warm-up training, to compress gradients while maintaining accuracy. Wang et al. [214] propose ATOMO to sparsify gradients via budgeted atomic decomposition. All these lossy compression methods sacrifice statistical performance for system efficiency, and is usually very suitable in settings where bandwidth is extremely limited, such as federated learning [111].

Model Parallelism Approaches

Most model parallelism approaches aim to break the computation or memory limit of a single device in order to train larger and more complex models. Depending on the computation structure in $\Delta_{\mathcal{L}}$, model-parallelism approaches have diverse patterns, and are generally more difficult to implement. We introduce several notable model parallelism approaches below.

Model partitioning and device placement. Krizhevsky [116] manually partitions the fully-connected layers of a CNN and places parts on multiple GPUs for parallel computation. Modern frameworks such as TensorFlow designate a *device* attribute in the declaration of an operation or a variable in the dataflow graph. At runtime, the operation will be placed on its specified device for execution. By specifying a target device for each operation in the graph, one can flexibly place different parts of the dataflow graph on a cluster of devices for execution. Mirhoseini et al. [148, 149] formulate this device placement as a combinatorial optimization problem, and develop reinforcement learning optimizers to automatically find good placements. This device placement approach enables training very large models that a single device memory cannot hold (even when batch size is 1), but is *not necessarily* computationally efficient, as devices might compute sequentially if rare parallelizability exists in the model graph, e.g., on sequential NNs.

Pipeline parallelism. In addition to model partitioning, pipeline parallelism [78, 88] improves efficiency by introducing additional pipelining. In systems like GPipe [88] and PipeDream [78], each model graph is partitioned into sequential cells, and each cell is then placed on a separate device. To perform computation over a batch, they first split the batch of training examples into smaller micro-batches, then pipeline the execution of each set of micro-batches over cells. Gradients are accumulated across all micro-batches in a mini-batch and applied at the end of a mini-batch. Pipeline parallelism allows devices to compute in parallel on mannually-spitted micro-batches, hence improves efficiency. It however assumes the model has sequential layered structure, and layers can be grouped into multiple *load-balanced* cells, which is not true for extremely large layers, such as Mixture-of-Expert layers [187].

Layer and operator partitioning. Sometimes, a single computational layer or operator might be too large to fit in one device, such as Mixture-of-Expert layers [187] or the multiplication of two gigantic matrices. A class of model parallelism approaches propose to partition layers or operators, and dispatch them to different devices for parallel execution. Mesh-TensorFlow [188] partitions large operations by rewriting the TensorFlow dataflow graph, then places parts on a mesh of devices specified by users. FlexFlow [98, 99] proposes four possible parallelism dimensions "SOAP", and uses a stochastic MCMC-based algorithm to search for the optimal partitioning strategy for each layer of a NN. Tofu [216] considers a similar problem, but at a finer granularity – it generates a partitioning strategy for each operator in the graph. These approaches tend to over-partition the graph, and their search space of partitioning strategies usually grows exponentially with the number of device in the cluster, and the number of layers/operators in the graph.

2.4.2 Distributed ML Programming APIs

Since one goal of this thesis is to simplify distributed ML programming on parallel environment, we discuss and compare some available programming interfaces exposed by existing distributed ML systems. These APIs are designed to help convert user's original, single-node code to a distributed version. Note that many systems [98, 116, 226] do not necessarily offer explicit APIs, as they are either pure research project, or are specialized for targeted models.

Manually Synchronize Updates

As previously mentioned, earlier parameter server based systems expose a set of `Push`, `Pull`, `Clock` APIs. They need to be precisely inserted into the training loop (Algorithm 1) after the gradient calculation, right before the application of the gradients, as shown in Figure 2.8. Collective communication libraries such as MPI or OpenMPI and some modern frameworks such as PyTorch adopt this set of interfaces. Despite their intuitive definitions, using these APIs involves modifying low-level optimization code, which might require some system expertise, and is very error-prone.

```
for epoch in range(10):
    for data, traget in train_set:
        loss = loss_function(output, target)
        gradients = loss.backward()
        push(gradients) # Push gradients
        updates = pull()  # Pull synchronized updates
        optimizer.apply(updates)
```

Figure 2.8: A Python code snippet illustrating how to manually synchronize gradients.

Specifying Tasks and Placements

Frameworks such as TensorFlow use a task-based distributed runtime [65] – users deploy TensorFlow on a cluster of nodes as a set of *tasks*, which are named processes that can communicate over a network. Each task contains one or more accelerator devices. This design allows manually placing an operation or a variable on a `task:device` tuple, shown in Figure 2.9, which can be done via high level language interfaces. It also offers great flexibility to realize other parallelization strategies. For example, by placing a trainable variable on a high-bandwidth node with

```
with tf.device("/job:local/task:1/cpu:0"):
    first_batch = tf.slice(x, [0], [50])
with tf.device("/job:local/task:1/cpu:0"):
    second_batch = tf.slice(x, [50], [-1])
    mean = (first_batch + second_batch) / 2
with tf.Session("grpc://localhost:2222") as sess:
    result = sess.run(mean, feed_dict={x: data})
```

Figure 2.9: A TensorFlow code snippet illustrating how to specify tasks and placements.

task name `ps:cpu:0`, and sharing it across all worker tasks, one can form a parameter server architecture.

On the downside, it requires substantially modifying the original code to append placement assignments, which, again, assumes that ML developers understand the distribution details and is able to make correct assignments of graph elements to distributed devices.

Monkey-patch Optimizers

An improved interface to sidestep the manual gradient averaging in previous systems is to monkey patch[3] the optimizer. In particular, Horovod provides distributed optimizer implementations that average gradients across distributed workers using collective AllReduce. In order to minimize user code modifications, Horovod monkey-patches the host framework's (e.g., TensorFlow or PyTorch) native optimizer interfaces, and relink them to the distributed ones provided by Horovod. In this way, the users code can be conveniently converted to a distributed version in one step – switching to use the optimizer offered by Horovod, shown in Figure 2.10.

```
# Build model...
loss = ...
opt = tf.train.SGD(lr=0.01)
# Monkey-patch the TF native optimizer
opt = horovod.DistributedOptimizer(opt)
# Make training operation and train
train_op = opt.minimize(loss)
with tf.Session() as sess:
    sess.run(train_op)
```

Figure 2.10: A TensorFlow+Horovod code snippet illustrating how Horovod monkey-patches the optimizer and allows very minimal code changes for distributed training.

This interface is one of the simplest interface so far, and has gained substantial popularity in the open-source community. A few recent parameter server systems such as BytePS [167] uses the same interfaces with Horovod. However, it requires all the semantics of the distributed strategy to be implemented as an optimizer, it has limited support to strategies beyond data-parallel ones.

Python Scope or Decorator via Graph Rewriting

A new type of interfaces developed recently in the TensorFlow Distribute project adopts a python scope-based interface, shown in Figure 2.12. It offers a set of named distributed strategies, such as *ParameterServerStrategy*, *CollectiveStrategy* as python scopes, which are expected to be appended at the start of user code, to capture the model declaration as an intermediate representation. In the backend, the distributed system can rewrite the graph, and incorporate corresponded semantics depending on the chosen strategy – parameter server or collective. In newer frameworks such as JAX [16], the scoping interfaces also can appear as a python decorator, shown in Figure 2.12, to capture the model declaration and training logic in a user function.

[3] https://en.wikipedia.org/wiki/Monkey_patch.

```
strategy = tf.distributed.PSStrategy(...)
with strategy.scope():
    # Define models, loss, and optimizer
    loss, opt = ...
    grad = loss.backward()
    strategy.run(opt.apply_gradient(grad))
```

Figure 2.11: A Python code snippet illustrating the scope-based interface to convert users' single-node code to run on distributed clusters.

```
# Define the strategy and the optimizer...
strategy, opt = ...
# Define and decorate the train_step function
@strategy
def train_step(data_batch, ...):
    loss, grad = ...
    opt.apply_gradients(grad)

# Run distributed training using the decorated function
for data_batch in train_set:
    train_step(data_batch, ...)
```

Figure 2.12: A Python code snippet illustrating the decorator-based interface to run user code on distributed clusters following a predefined distribution strategy.

This set of interfaces are simple to use, and support various distribution strategies, and can extend to just-in-time compilation to automatically generate distribution strategies, which is a research topic explored in Chapter 7 of this thesis. The main drawback of this approach is that it assumes the model declarations can be fully and precisely captured via scoping or decorating, which sometimes might not be true if the code is imperative, or has a dynamically-varying structure. How to extract static representations from arbitrary ML code for downstream optimizations is a very active field of research.

2.4.3 Trends in Distributed ML Systems

In the past decade, the development of distributed ML has been focused on leveraging ML characteristics to guide the design of systems. This system-algorithm co-design method has led a a few notable systems and optimizations, such as parameter server systems, and bounded stale synchronous consistency. Continuing this methodology, we observe several new trends in the development of distributed ML systems.

Co-optimizing Multiple Parallelization Aspects

While previous systems focus on optimizing one or two aspects of parallelization, newer systems start to jointly optimize multiple aspects, to provide the greatest performance. The rationale behind co-optimization is to capture the complex interplay between different parallelization aspects, which sometimes might not be obvious.

Paliwal et al. [162] propose to jointly optimize over placement (which nodes are executed on which devices) and schedule (the node execution order on each device) to minimize runtime or peak memory usage. Jia et al. [99] put the partitioning of layers along different axis and the placement of layers into a same search space, and optimize the strategy generated from the space. Peng et al. [168] jointly optimize scheduling and tensor partitioning to achieve theoretically optimal communication schedule on a PS-based communication architecture. Huang et al. [86] jointly optimize operator scheduling, memory allocation, and swap decisions to enable training and serving DNN models with limited GPU memory size.

All these recent works, along with this thesis, show that simultaneously optimizing multiple performance-affecting factors offers new and substantial opportunities to improve distributed ML system performance.

ML-based Distributed System Autotuning

Due to the rich structures and variety of ML programs, optimizing the design of computer systems often involves complex high-dimensional combinatorial problems, and need to be dynamically adapt to the ML workload. Previous methods rely heavily on heuristics. Recent distributed system gradually incorporate data-driven autotuning capability to make the design choice. The core methodology is to train ML models using real runtime data, and predict the most likely performant choice.

We observe growing interest in applying ML-based autotuning in system design. Parallax [106] adjuts the partitioning size of variables using a learned linear model. ByteScheduler [168] determines the credit size for scheduling using Bayesian Optimization. TensorFlow optimizes the placements [148] and scheduling [162] using reinforcement learning. Horovod optimizes several adjustable "knobs" using Bayesian Optimization, too. Chapter 8 of this thesis explores the end-to-end generation of data-parallel distribution strategies using an ML-based simulator.

Part I

Aspects of ML Parallelization

Overview

The first part of the thesis studies individual aspects involved with distributed ML, and/or derive optimizations on each aspect based on the suggested approach – *adaptive parallelism*.

Chapter 3 reveals *communication* as one of the core challenges in contemporary distributed machine learning. By exploiting the layered structure of deep neural networks, the chapter develops a scheduling approach, *wait-free backpropagation*, and an *adaptive communication* algorithm, to alleviate bursty network traffics and to reduce communication loads, respectively. Based on these methods, this chapter demonstrates a distributed ML system, Poseidon, that boosts the scalability of various CNN models by 2x-10x on top of existing DL frameworks.

Chapter 4 identifies the data movement overheads, GPU stalls, and limited memory as three major hurdles on scaling big models on distributed GPUs. Once again, by adapting to the model structure, we develop the *memory swapping* technique as an effective way of training large models on limited GPU memory. Based on memory swapping, this chapter presents GeePS, a parameter server system specialized for scaling deep learning applications across GPUs. GeePS allows data-parallel training of very large neural networks, bounded by the largest layer rather than the overall model size – it is able to train DL models that would otherwise require 7x more memory (70GB) than a standard GPU normally offers (10GB).

Chapter 5 brings in the aspect of *consistency model* into the picture. It studies the impact of *staleness*, a quantitative characterization of the tolerance of delayed updates allowed in a distributed ML system, against various ML models, algorithms, and hyperparameter settings. Through simulation studies, this chapter reveals that different models or even model building blocks have different degrees of robustness against staleness. This suggests design space for future systems to smartly adapt the consistency to best trade-off between system throughput and statistical efficiency.

The results presented in this part of the thesis have appeared in the following publications:

- Zhicheng Yan, Hao Zhang, Robinson Piramuthu, Vignesh Jagadeesh, Dennis DeCoste, Wei Di, and Yizhou Yu. HD-CNN: Hierarchical Deep Convolutional Neural Network for Large Scale Visual Recognition. In *2015 International Conference on Computer Vision (ICCV 2015)*.
- Hao Zhang, Zhiting Hu, Jinliang Wei, Pengtao Xie, Gunhee Kim, Qirong Ho, and Eric Xing. Poseidon: A System Architecture for Efficient GPU-based Deep Learning on Multiple Machines. In *2016 USENIX Annual Technical Conference (USENIX ATC 2016 Poster and MLSys@ICML 2016)*.

- Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric Xing. GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-specialized Parameter Server. In *2016 European Conference on Computer Systems (EuroSys 2016)* .
- 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 2017)*.
- Wei Dai, Yi Zhou, Nanqing Dong, Hao Zhang, and Eric P. Xing. Toward Understanding the Impact of Staleness in Distributed Machine Learning. In *the 7th International Conference on Learning Representations (ICLR 2019)*.

Chapter 3

Scheduling and Communication

ML software such as TensorFlow [1], PyTorch [56], MxNet [27], Caffe [96] allow practitioners to easily experiment with DL models on a single machine. However, their distributed implementations can scale poorly for larger models. For example, we find that on the VGG19-22K network [191] (229M parameters, see Section 3.5.5) or BERT-large [51] (340M parameters, see Section 1.1), open-source TensorFlow on 32 machines can be slower than single machine (Section 3.5.2). This observation underlines the challenge of scaling DL on GPU clusters: the high computational throughput of GPUs allows more data batches to be processed per minute (than CPUs), leading to more frequent network synchronization that grows with the number of machines. Existing communication strategies, such as parameter servers (PS) for ML [127, 221], can be overwhelmed by the high volume of communication [40]. Moreover, despite the increasing availability of faster network interfaces such as Infiniband or 40GbE Ethernet, GPUs have continued to grow rapidly in computational power, and continued to produce parameter updates faster than can be naively synchronized over the network. For instance, on a 16-machine cluster with 40GbE Ethernet and one Titan X GPU per machine, updates from the VGG19-22K model will bottleneck the network, so that only an 8x speedup over a single machine is achieved.

These scalability limitations in distributed DL stem from at least two causes: (1) the gradient updates to be communicated are very large matrices, which quickly saturate network bandwidth; (2) the iterative nature of DL algorithms causes the updates to be transmitted in bursts (at the end of an iteration or batch of data), with significant periods of low network usage in between.

Following the design principle introduced in Section 1.1, in this chapter, we derive strategies to improve the performance in the scope of neural network training, specifically concerning the following two ML parallelization aspects: computation and communication scheduling (which we refer as *scheduling* in this chapter) and distributed communication.

3.1 Background

3.1.1 Distributed Training of Neural Networks

In this section, we first instantiate the master equation 2.1 in the context of *distributed training of neural networks* on *GPU clusters*, so to reveal the specific ML or computational properties

these models or training workloads possess. Based on that, we derive adaptive scheduling and communication strategies to fit with these properties.

Neural networks are a family of hierarchical models containing many layers, from as few as 5-10 [117] to as many as 100s [80]. Figure 3.1 illustrates a convolutional neural network with 6 layers. The first layer (green) is an input layer that reads data in application-specific formats, e.g., raw pixels if it is trained to classify images. The input layer is connected to a sequence of intermediate layers (cyan, orange), each of which consists of a few neurons, where each neuron applies a function transformation f on its input and produces an output. A vector output is obtained by concatenating the output of all neurons from a layer. By stacking multiple intermediate layers, the NN can transform raw input data one layer at a time, first into a series of intermediate representations, and finally into the desired output or prediction (red). DL programmers usually need to specify the computation of a layer by defining two properties of its neurons. The first is the transformation function $f(W, x)$, where x is the input to the neuron, and W is an *optional trainable* parameter. The other is the connectivity that determines how the neuron should be connected to its adjacent layer. For instance, a convolutional neural network has two types of neuron: convolutional (CONV) neuron (cyan) that are only locally connected to a subset of neurons in its previous layer, and fully-connected (FC) neurons (orange).

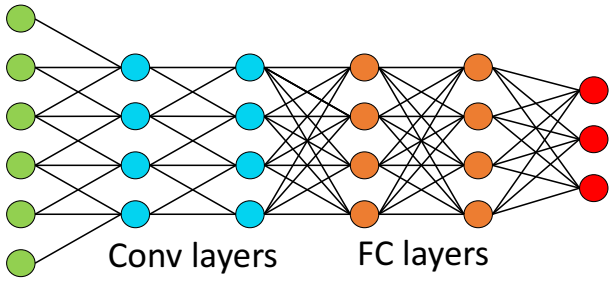

Figure 3.1: A convolutional neural network with 6 layers.

Most NNs need to be trained with data to give accurate predictions. Stochastic gradient descent (SGD) and backpropagation are commonly employed to train NNs iteratively – each iteration performs a feed forward (FF) pass followed with a backpropagation (BP) pass. In the FF pass, the network takes a training sample as input, forwards from its input layer to output layer to produce a prediction. A loss function is defined to evaluate the prediction error, which is then backpropagated through the network in reverse, during which the network parameters are updated by their gradients towards where the error would decrease. After repeating a sufficient number of passes, the network will usually converge to some state where the loss is close to a minima, and the training is then terminated.

Starting from Equation 2.1, given data $\boldsymbol{x}$ and a loss function $\mathcal{L}$, fitting the parameters $\boldsymbol{\Theta}$ of a NN can be formulated as the following *iterative-convergent* algorithm that repeatedly executing the update equation

$$\boldsymbol{\Theta}^{(t)} = \boldsymbol{\Theta}^{(t-1)} + \epsilon \cdot \nabla_{\mathcal{L}}(\boldsymbol{\Theta}^{(t-1)}, \boldsymbol{x}^{(t)}), \tag{3.1}$$

until $\boldsymbol{\Theta}$ reaches some stopping criteria, where t denotes the iteration. The update function $\nabla_{\mathcal{L}}$ calculates the gradients of $\mathcal{L}$ over current data $x_i (x_i \in \boldsymbol{x})$. The gradients are then scaled by a

learning rate ϵ and applied on $\mathbf{\Theta}$ as updates. As the gradients are additive over data samples i, i.e. $\mathbf{\Theta}^{(t)} = \mathbf{\Theta}^{(t-1)} + \epsilon \cdot \sum_i \nabla_{\mathcal{L}}(\mathbf{\Theta}^{(t-1)}, x_i)$, for efficiency, we usually feed a batch of training samples $\boldsymbol{x}^{(t)}(\boldsymbol{x}^{(t)} \subset bmx)$ at each training iteration t, as in Equation 3.1.

In large-scale training, data $\boldsymbol{x}$ are usually too large to process on a single machine in acceptable time. To speedup the training, we usually resort to *data parallelism* (Section 2.1.3), a parallelization strategy that partitions the data $\boldsymbol{x}$ and distributes to a cluster of computational worker machines (indexed by $p = 1, \cdots, P$), as illustrated in Figure 3.2. At each iteration t, every worker fetches a batch $\boldsymbol{x}_p^{(t)}$ from its data partition and computes the gradients $\nabla_{\mathcal{L}}(\mathbf{\Theta}^{(t)}, \boldsymbol{x}_p^{(t)})$. Gradients from all workers are then aggregated and applied to update $\mathbf{\Theta}^{(t)}$ to $\mathbf{\Theta}^{(t+1)}$ following

$$\mathbf{\Theta}^{(t+1)} = \mathbf{\Theta}^{(t)} + \epsilon \sum_{p=1}^{P} \nabla_{\mathcal{L}}(\mathbf{\Theta}^{(t)}, \boldsymbol{x}_p^{(t)}). \tag{3.2}$$

Data-parallelism allows data to be locally partitioned to each worker, which is advantageous for large datasets. It however requires every worker to have read and write access to the shared model parameters θ, which causes communication among workers; this communication support can be provided by a parameter server architecture [32, 221] (Figure 3.2a) or a peer-to-peer broadcasting architecture [225] (Figure 3.2b), both were originally designed for general-purpose data-parallel ML programs on CPUs.

3.1.2 Communication Architectures

We briefly review how to use a Parameter Server or a Sufficient Factor Broadcasting architecture to provide the required communication support in the distributed neural network training.

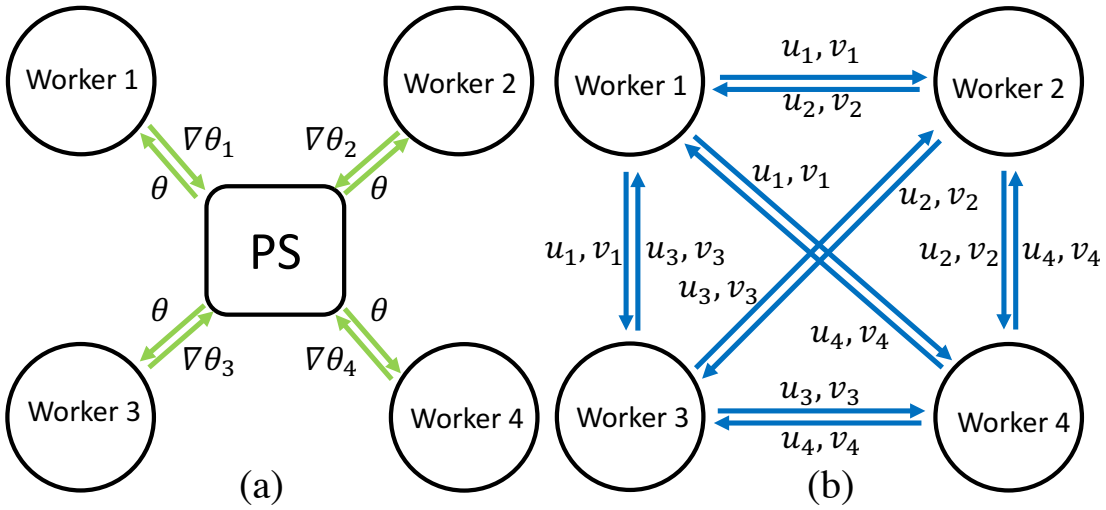

Figure 3.2: An illustration of (a) the parameter server and (b) sufficient factor broadcasting for distributed ML.

Ethernet	Rate(GBit/s)	Rate (Mb/s)	Rate (# floats/s)
1 GbE	1	125	31.25M
10 GbE	10	1250	312.5M
Infiband	40	5000	1250M
100 GbE	100	12500	3125M

Table 3.1: The maximum throughput that commonly used Ethernet can provide in terms of how many Gigabits, Megabytes and number of float parameters could be transferred per second.

Parameter Server

Typically, PS enables each worker to access the global model parameters Θ via network communications following the client-server scheme. Neural network training can be trivially parallelized over distributed workers using PS with the following 3 steps: (1) Each worker computes the gradients ($\nabla_{\mathcal{L}}$) on their own data partition and send them to remote servers; (2) servers receive the updates and apply ($+$) them on globally shared parameters; (3) a consistency scheme coordinates the synchronization among servers and workers (Figure 3.2a).

Sufficient Factor Broadcasting

Many ML models represent their parameters Θ as matrices. For example, fully-connected NNs, when trained using SGD, have their gradients $\nabla\Theta$ over a training sample as a rank-1 matrix, which can be cast as the outer product of two vectors u, v: $\nabla\Theta = uv^\top$, where u and v are called *sufficient factors* (SFs). Hence, Sufficient factor broadcasting (SFB) [225] can be employed to parallelize these models by broadcasting SFs among workers and then reconstructing the gradient matrices $\nabla\Theta$ using u, v locally. SFB presents three key differences from PS: (1) SFB uses a P2P communication strategy that transmits SFs instead of full matrices. (2) Unlike gradients, SFs are not additive over training samples, i.e., the number of SFs needed to be transmitted grows linearly with the number of data samples (not data batches); (3) the overall communication overheads of SFB increase quadratically with the number of workers.

3.1.3 Communication Challenges on GPU Clusters: An Example

Modern DL models are mostly trained using NVIDIA GPUs, because the primary computational steps (e.g., matrix-matrix multiplications) in DL match the SIMD operation that could be efficiently performed by GPUs. In practice, DL practitioners often use single-node software frameworks which mathematically derive the correct training algorithm and execute it on GPU by calling GPU-based acceleration libraries, such as CUBLAS and cuDNN. It is thus straightforward to parallelize these programs across distributed GPUs using either PS or SFB, by moving the computation from CPU to GPU, and performing memory copy operations (between DRAM and GPUs) or communication (among multiple nodes) whenever needed. However, we argue below and show empirically in Section 3.5 that these usually lead to suboptimal performance.

The inefficiency is mainly caused by parameter synchronization via the network. Compared to CPUs, GPUs are an order of magnitude more efficient in matrix computations; the production

Model	Batch size (# images)	# Parameters (# floats)	Time (s/iter)	Gradients (# floats/s)
AlexNet	256	61.3M	0.96	63.85M
VGG-16	64	128.3M	4.06	31.60M

Table 3.2: Statistics of modern CNN training, including the batch size, the number of model parameters, per-iteration computation time, and the number of gradients generated per second on a single device. The performance is evaluated on a K40 GPU with standard hyperparameters in 2016. At the time this thesis is being written (2020), GPU FLOPS have been improved (e.g. NVIDIA A100) with yet another order of magnitude – meaning that they compute even faster, and result in much heavier communication loads.

of gradients on GPUs is much faster than they can be naively synchronized over the network. As a result, the training computations are usually bottlenecked by communications.

Table 3.1 lists the standards of commonly used Ethernet and Table 3.2 shows some statistics of modern CNN training. To understand the statistics, take the training of AlexNet [117] (61.5M parameters) on Titan X with a standard batch size 256, 240 million gradients will be generated per second on each GPU (0.25s/batch). If we parallelize the training on 8 nodes using a PS, with every node also holding $1/8$ of parameters as a PS shard; then, every node needs to transfer $240\text{M}\times 7/8 \times 4 = 840\text{M}$ float parameters in one second to make sure the next iteration of computation not being blocked. Apparently, the demanded throughput (>26Gbps) exceeds the bandwidth that commodity Ethernet (i.e., 1GbE and 10GbE Ethernet) provides; the GPUs distributed across clusters cannot be fully utilized. Practically, it is usually difficult to partition the parameters completely equally, which will result in more severe bandwidth demands, or bursty communication traffic on several server nodes (as we will show in Section 3.5.4), which prevents the trivial realization of efficient DL on distributed GPUs. In addition, frequent memory copy operations between DRAM and GPU memory can cause non-trivial overheads.

We propose that a solution to these challenges should exploit the structure of DL algorithms to improve scheduling and communication: on one hand, it should identify ways in which the matrix updates can be separated from each other, and then schedule them in a way that avoids bursty network traffic. On the other hand, the solution should also exploit the structure of the matrix updates themselves, and wherever possible, reduce their size and thus the overall communication load on the network.

3.2 Scheduling

3.2.1 The Sequential Structure of DL Programs

At the core of the DL program is the BP algorithm that performs forward-backward pass through the network repeatedly. If we define a forward and a backward pass through the lth layer of a network as f_t^l and b_t^l, respectively, then a **C**omputation step at iteration t is notated as $C_t = [f_t^1, \cdots, f_t^L, b_t^L, \cdots, b_t^1]$, as illustrated in Figure 3.3. When executing on distributed GPUs, inter-machine communications are required after each C step to guarantee the synchronized replication

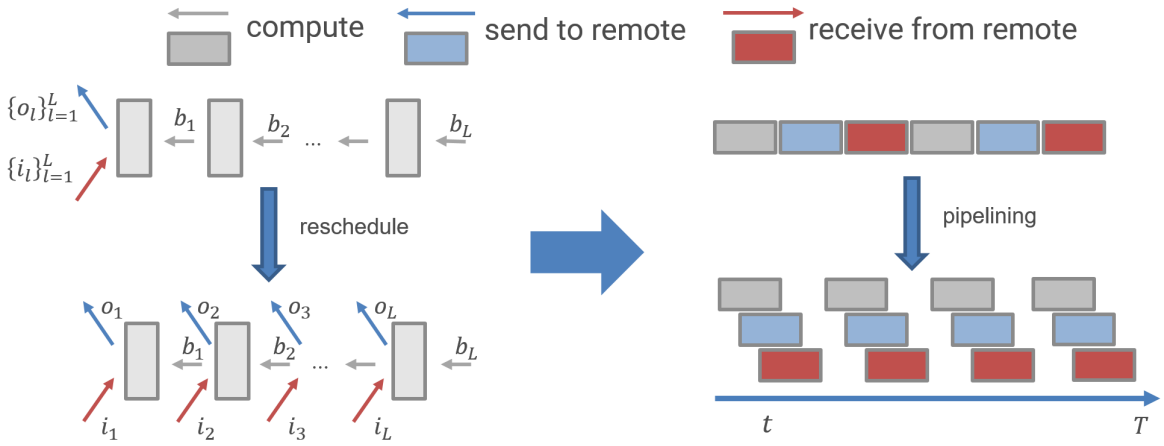

Figure 3.3: **(Top)** Traditional backpropagation and **(Bottom)** wait-free backpropagation on distributed environment.

of model parameters. We similarly define the **S**ynchronization step S_t as the process that a worker sends out locally generated updates and then receives updated parameters from remote workers at iteration t. Therefore, a naive parallelization of DL training over distributed GPUs using either PS or SFB can be expressed as alternating C_t and S_t defined above. We note that DL training is highly sequential; the communication and computation perform sequentially, waiting each other to finish (top of Figure 3.3).

We note that as every layer of a NN contains an independent set of parameters, S_t can be decoupled as $S_t = (s_t^1, \cdots, s_t^L)$, by defining s_t^l as the synchronization of parameters of layer l. If we further decompose $s_t^l = [o_t^l, i_t^l]$ as first sending out local updates of layer l (o_t^l) and reads in the updated parameters remotely (i_t^l), we can rewrite a training iteration as: $[C_t, S_t] = [f_t^1, \cdots, f_t^L, b_t^L, \cdots, b_t^1, o_t^L, \cdots, o_t^1, i_t^L, \cdots, i_t^1]$. The sequential nature of the BP algorithm presents us an opportunity to overlap the computations and communications through rescheduling: we next present *wait-free backpropagation* to overlap C_t and S_t by partially rescheduling those b_t and s_t that are independent.

3.2.2 Wait-free Backpropagation

The wait-free backpropagation (WFBP) is designed to overlap communication overheads with the computation based on two key independencies in the program: (1) the send-out operation o_t^l is independent of backward operations $b_t^i (i < l)$, so they could be executed concurrently without blocking each other; (2) the read-in operation i_t^l could update the layer parameters as long as b_t^l was finished, without blocking the subsequent backward operations $b_t^i (i < l)$. Therefore, we can enforce each layer l to start its communication once its gradients are generated after b_t^l, so that the time spent on operation s_t^l could be overlapped with those of $b_t^i (i < l)$, as shown at the bottom of Figure 3.3.

WFBP is most beneficial for training DL models that have their parameters concentrating at upper layers (FC layers) but computation concentrating at lower layers (CONV layers)[1], e.g.,

[1]Most classification models will fall into this family if the number of classes to be classified is large.

Parameters	CONV Layers (#/%)	FC Layers (#/%)
AlexNet [117]	2.3M / 3.75	59M / 96.25
VGG-16 [191]	7.15M / 5.58	121.1M / 94.42
FLOPs	CONV Layers (#/%)	FC Layers (#/%)
AlexNet [117]]	1,352M / 92.0	117M / 8.0
VGG-16 [191]	10,937M / 91.3	121.1M / 8.7

Table 3.3: Parameter and FLOP distributions of convolution and fully-connected layers in AlexNet [117] and VGG-16 [192].

VGG [192] and AdamNet [32, 40]), as illustrated in Table 3.3; it overlaps the communication of top layers (90% of communication time) with the computation of bottom layers (90% of computation time) [40, 241]. Besides chain-like NNs, WFBP is generally applicable to other non-chain like structures (e.g., tree-like structures), as the parameter optimization for deep neural networks depends on adjacent layers (and not the whole network), there is always an opportunity for parameter optimization (i.e., computation) and communication from different layers to be performed concurrently.

Some DL frameworks, such as TensorFlow, represent the data dependencies of DL programs using graphs, therefore implicitly enable auto-parallelization. However, they fail on exploring the potential opportunities of parallelization between iterations. For example, TensorFlow needs to fetch the updated parameters from the remote storage at the beginning of each iteration, while it is possible to overlap this communication procedure with the computation procedure of the previous iteration. In comparison, WFBP enforces this overlapping by explicitly pipelining compute, send and receive procedures. We describe our implementation of WFBP in Section 3.4.2 and empirically show its effectiveness in Section 3.5.2.

3.3 Adaptive Communication

While WFBP-based scheduling overlaps communication and computation, it does not reduce the communication overhead. In situations where the network bandwidth is limited (e.g., commodity Ethernet or the Ethernet is shared with other communication-heavy applications), the communication would still be unacceptably slow. To address the issue, we introduce an *adaptive communication* strategy (Algorithm 2) that combines the best of PS and SFB by adapting to both the mathematical property of DL models and the structure of computing clusters.

Our idea comes from two observations: the synchronization operations $\{S_t^l\}_{l=1}^L$ (§3.2.1) are independent of each other, meaning that we can use different communication methods for different S_t^l by specializing o_t^l and i_t^l according to the two methods described in Figure 3.2; second, a NN structure is usually predefined and fixed throughout the training – by measuring the number of parameters needed to transfer, we are able to estimate the communication overhead, so that we can always choose the optimal method even before the communication happens.

Consider training VGG19 network [192], the overheads of S_t^l could be estimated as follows (Table 3.4): assume the batch size $K = 32$, the number of workers and server nodes $P_1 = P_2 = 8$

Method	Server	Worker	Server & Worker
PS	$2P_1MN/P_2$	$2MN$	$2MN(P_1+P_2-2)/P_2$
SFB	**N/A**	$2K(P_1-1)(M+N)$	**N/A**
Adam [32]	$P_1MN+P_1K(M+N)$	$K(M+N)+MN$	$(P_1-1)(MN+KM+KN)$

Table 3.4: Estimated communication cost of PS, SFB and Adam for synchrnizing the parameters of a $M \times N$ FC layer on a cluster with P_1 workers and P_2 servers, when batchsize is K.

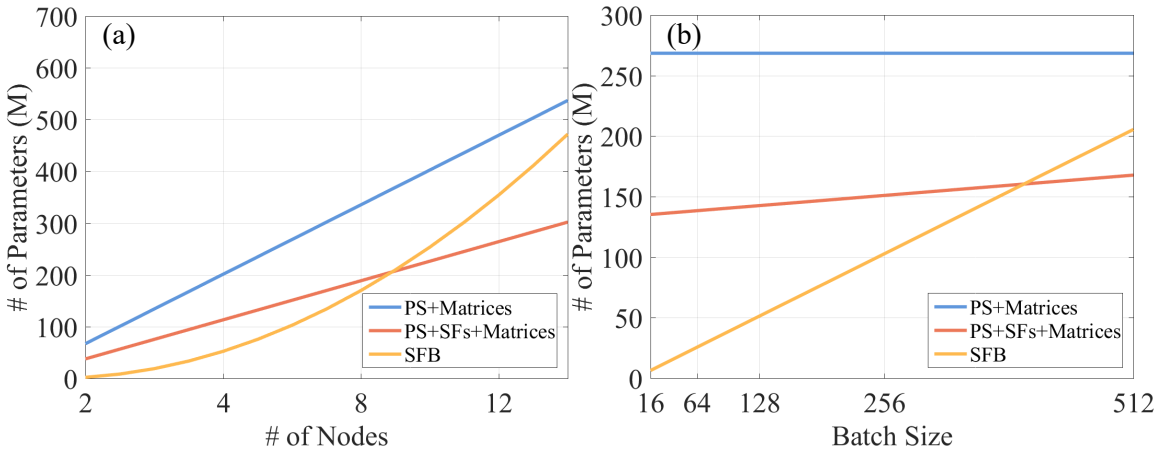

Figure 3.4: Comparisons of the three communication strategies when training AlexNet on GPU clusters. The parameters needed to be communicated between fc6 and fc7 are compared by varying (1) the number of cluster nodes P and (2) batch size K.

(assuming parameters are ideally equally partitioned over all server shards), respectively. On one hand, if l is an FC layer (with shape 4096×4096, $M = N = 4096$), synchronizing its parameters via PS will transfer $2MN \approx 34$ million parameters for a worker node, $2P_1MN/P_2 \approx 34$ million for a server node, and $2MN(P_1+P_2-2)/P_2 \approx 58.7$ million for a node that is both a server and a worker, compared to $2K(M+N)(P_1-1) \approx 3.7$ million for a single node using SFB. On the other hand, if l is a CONV layer, the updates are indecomposable and sparse, so we can directly resort to PS. Therefore, the synchronization overheads depend not only on the model (type, shape, size of the layer), but also the size of the clusters. The optimal solution usually changes with M, N, K, P_1, P_2, plotted in Figure 3.4. Determining the optimal communication architecture requires taking into account these factors, so to dynamically adjust the communication method for different parts of a model – it always chooses the best method from available ones whenever it results in fewer communication overheads.

Microsoft Adam [32] employs a different communication strategy from those in Figure 3.2. Instead of broadcasting SFs across workers, they first send SFs to a parameter server shard, then pull back the whole updated parameter matrices. This seems to reduce the total number of parameters needed to be communicated, but usually leads to load imbalance; the server node that holds the corresponding parameter shard overloads because it has to broadcast the parameter matrices to all workers ($P_1MN+P_1K(M+N)$ messages need to be broadcasted), which easily

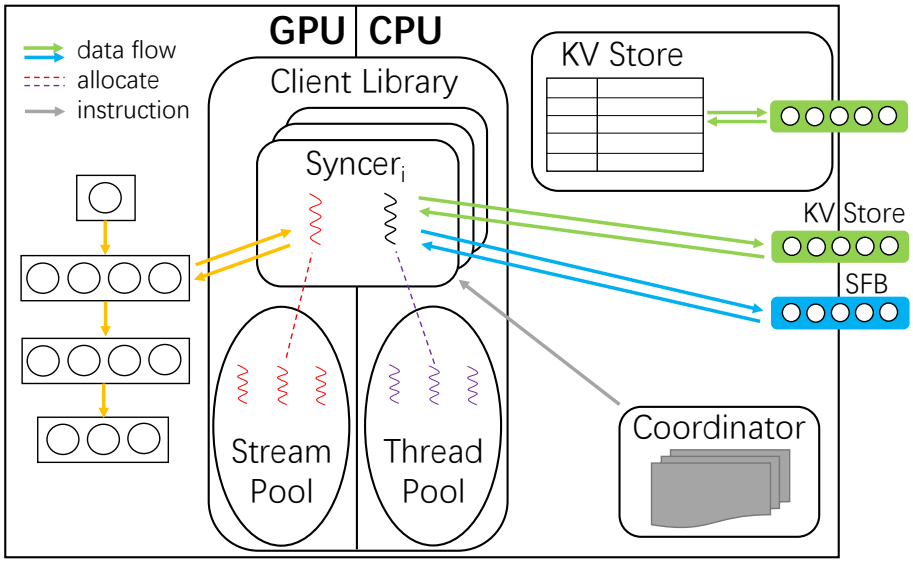

Figure 3.5: An overview of Poseidon. The diagram shows the components added and managed by Poseidon on top of the original training process (the neural network on the left) which is usually managed by the DL framework.

causes communication bottleneck (Section 3.5.4). It is noticeable that reconstructing gradients from SFs may cause extra computation cost, which however is often negligible compared to communication. We describe our implementation of adaptive communication in the next section, and assess its effectiveness in Section 3.5.

3.4 Poseidon: An Efficient Communication Architecture for Distributed DL on GPU Clusters

Based on the scheduling and communication strategies, we derive and implement Poseidon, an efficient communication architecture for data-parallel DL on distributed GPUs. Poseidon exploits the sequential layer-by-layer structure in DL programs, finding independent GPU computation operations and network communication operations in the training algorithm, so that they can be scheduled together to reduce bursty network communication. Moreover, Poseidon implements a hybrid communication scheme that accounts for each DL program layer's mathematical properties as well as the cluster configuration, in order to compute the network cost of different communication methods, and select the cheapest one – currently, Poseidon implements and supports a parameter server scheme [221] that is well-suited to small matrices, and a sufficient factor broadcasting scheme [225] that performs well on large matrices. The architecture is general to support other communicating schemes, such as collective AllGather/AllReduce, as evidenced by some later follow-up work [107].

This section first elaborates Poseidon's system architecture and APIs, and then describes how to modify a DL framework using Poseidon to enable distributed execution.

3.4.1 System Architecture

Figure 3.5 illustrates the architecture of Poseidon: a C++ communication library that manages parameter communication for DL programs running on distributed GPUs. It has three main components: coordinator, that maintains the model and the cluster configuration; KV store, a shared memory key-value store that provides support for parameter server based communication; client library, which is plugged into DL programs to handle parameter communication. Their APIs are listed in Table 3.5.

Coordinator

To setup distributed training, the client program (e.g., Caffe) first instantiates Poseidon by creating a coordinator within its process. Coordinators will first collect necessary information, including the cluster information (e.g., the number of workers and server nodes, their IP addresses) and the model architecture (e.g., the number of layers, layer types, number of neurons and how they are connected, etc.). With the information, the coordinator will initialize the KV stores and the client library with two steps: (1) allocate proper communication ports for each PS shard and peer worker; (2) determine what parameters should be transmitted via the KV store and what by SFB, and hash the parameters equally to each KV store if necessary, and save the mapping in the information book, which, throughout the whole training, is maintained and synchronized across nodes, and could be accessed elsewhere through coordinator's `Query` API. Besides, the coordinator provides another API `BestScheme` that takes in a layer and returns the optimal communication scheme for it according to the strategy described in Section 3.3 (Algorithm 2).

Algorithm 2: Get the best communication method of layer l

Input: A layer l
Output: The most suitable communication scheme

Function `BestScheme` *(l)*:
    $layer\_property$ = Query(l.name)
    P_1, P_2, K = Query(`n_worker`, `n_server`, `batchsize`)
    if $layer\_property$*.type is* `FC` **then**
        M = $layer\_property$.width
        N = $layer\_property$.height
        if $2K(P_1 - 1)(M + N) \leq \frac{2MN(P_1+P_2-2)}{P_2}$ **then**
            return `SFB`
    return `PS`

KV Store

The KV store is implemented based on a bulk synchronous parameter server [40, 221], and instantiated by coordinators on a list of user-specified "server" machines. Each instance of the KV store holds one shard of the globally shared model parameters in the form of a set of KV pairs,

of which each KV pair is stored on a chunk of DRAM. Poseidon sets the size of a KV pair to a fixed small size (e.g., 2MB), so as to partition and distribute model parameters to server nodes as equally as possible, reducing the risk of Ethernet bottleneck. Each KV store instance manages a parameter buffer on RAM, and provides PS-like APIs, such as `Receive` and `Send`, for receiving and applying updates from client libraries, or sending out parameters. It will regularly checkpoint current parameter states for fault tolerance.

Client Library

Poseidon coordinates with DL programs via its client library. Particularly, users plug the client library into their training program, and the client library will create a *syncer* for each NN layer during network assembling (so that each layer one-to-one maps to one syncer), accounting for its parameter synchronization. Each sycner is then initialized, for example, setting up connections to its corresponding PS shards or (remote) peer syncers according to the coordinator's information book, and allocating a small memory buffer for receiving remote parameter matrices or SFs, etc.

The client library manages a CPU thread pool and a GPU stream pool on the worker machine, which can be allocated by the syncer APIs when there is a syncer job created. The syncer has three main APIs, `Send`, `Receive` and `Move`, to be used in client programs. The `Move` API takes care of the memory movement between RAM and GPU memory, and performs necessary computation, e.g., the transformation between SFs and gradients, and the application of updates. It is multi-threaded using the CUDA asynchronous APIs, and will trigger an allocation from the client library's thread/stream pools when a syncer job starts (see L14 of Algorithm 3). The `Send` and `Receive` are communication APIs that synchronize layer parameters across different model replicas. The `Send` API is nonblocking; it sends out parameter updates during backpropagation once they are generated, following the protocol returned by coordinator's `BestScheme` API. The `Receive` API will be called once `Send` is finished. It requests either fresh parameter matrices from the KV stores or SFs from its peer syncers, and will block its current thread until it receives all of what it requested. The received messages are put into the syncer's memory buffer for the `Move` API to fetch.

Managing Consistency

Poseidon focuses on synchronous parallel training which is shown to yield faster convergence compared with asynchronous training in distributed DL (as measured by wall clock time) on GPUs [24, 40]. Unless otherwise specified, our discussion in this chapter assumes synchronous replication of model parameters in each training iteration, although we note that Poseidon's design can easily be applied to asynchronous or bounded-asynchronous consistency models [41, 83].

Poseidon implements the bulk synchronous consistency (BSP) model as follows. The client library maintains a binary vector C with length the number of syncers and values reset to zeros at the start of each iteration. A syncer will set its corresponding entry in C as 1 when its job finishes, and the client starts next iteration when all entries are 1. While, the KV store maintains a zero-initialized count value for each KV pair at the start of each iteration. Every time when there is an update being applied on a KV pair, its count value is increased by 1. The KV pair will be

broadcasted via its `Send` API when its count equals to the number of workers. Poseidon handles stragglers by simply dropping them. Although asynchronous models can alleviate the straggler problem in distributed ML [83], Poseidon focuses on synchronous parallel training, because synchronous execution yields the fastest per-iteration improvement in accuracy for distributed DL (as measured by wall clock time) on GPUs [24, 40] (see Section 3.5.2).

3.4.2 Integrate Poseidon with DL Frameworks

For such a system to be relevant to practitioners (who may have strong preferences for particular frameworks), we would prefer not to exploit specific traits of TensorFlow's or Caffe's design, but should strive to be relevant to as many existing frameworks as possible.

We hence implement Poseidon to be pluggable into most existing DL frameworks to enable efficient distributed execution. Algorithm 3 provides an example. Specifically, one needs to first include Poseidon's client library into the framework, then figures out where the backpropagation proceeds (L6), and insert Poseidon's syner APIs in between gradient generation and application (L7). We demonstrate in Section 3.5.2 that with slight modifications (150 and 250 LoC for Caffe and TensorFlow), both Poseidon-enable Caffe and TensorFlow deliver near-linear scalings up to 32 GPU machines. Poseidon respects the programming interfaces by the native DL library and stores necessary arguments for distributed execution as environment variables to allow zero changes on the DL application programs.

Algorithm 3: Parallelize a DL program using Poseidon

Input: The user-defined neural network net

1 **Function** `train` *(net)*:
2 **for** $iter = 1 \rightarrow T$ **do**
3 $sync\_count = 0$
4 net.Forward()
5 **for** $l = L \rightarrow 1$ **do**
6 net.BackwardThrough(l)
7 $thread\_pool$.Schedule(`sync` (l))
8 wait_until($sync\_count == net.num\_layers$)

Input: A layer l in net

9 **Function** `sync` *(l)*:
10 $stream$ = $stream\_pool$.Allocate()
11 $syncers[l]$.Move($stream$, GPU2CPU)
12 $syncers[l].method$ = $coordinator$.BestScheme(l)
13 $syncers[l]$.Send()
14 $syncers[l]$.Receive()
15 $syncers[l]$.Move($stream$, CPU2GPU)
16 $sync\_count$++

Method	Owner	Arguments	Description
`BestScheme`	Coordinator	A layer name or index	Get the best communication scheme of a layer
`Query`	Coordinator	A list of property names	Query information from coordinators' information book
`Send`	Syncer	None	Send out the parameter updates of the corresponding layer
`Receive`	Syncer	None	Receive parameter updates from either parameter server or peer workers
`Move`	Syncer	A GPU stream and an indicator of move direction	Move contents between GPU and CPU, do transformations and application of updates if needed
`Send`	KV store	updated parameters	Send out the updated parameters
`Receive`	KV store	parameter buffer of KV stores	Receive gradient updates from workers

Table 3.5: Poseidon APIs for parameter synchronization.

3.5 Evaluation

In this section, we evaluate Poseidon's performance on scaling up DL with distributed GPUs. We implement it into two different DL frameworks: Caffe [97] and TensorFlow [1]. We focus on the image classification task where DL is most successfully applied. Our evaluation reveals the following results:

- Poseidon has little overhead when plugged into existing frameworks; it achieves near-linear speedups across different NNs and frameworks, on up to 32 Titan X-equipped machines.
- The scheduling strategy (Section 3.2) substantially improves GPU and bandwidth utilization.
- The adaptive communication (Section 3.3) effectively alleviates the communication bottleneck, thus achieves better speedups under limited bandwidth.
- Poseidon compares favorably to other communication-reduction methods, such as the SF strategy in Adam [32], and the 1-bit quantization in CNTK [235].

3.5.1 Experiment Setup

Cluster Configuration

We conduct our experiments on a GPU cluster with each node equipped with a NVIDIA GeForce TITAN X GPU card, an Intel 16-core CPU and 64GB RAM, interconnected via a 40-Gigabit Ethernet switch. All cluster nodes have shared access to a NFS and read data through the Ethernet interface. We run our system on UBUNTU 16.04, with NVIDIA driver version 361.62, CUDA 8.0 and cuDNN v5.

Computation Engines

We deploy Poseidon on two DL frameworks, Caffe [96] and TensorFlow [1]. For Caffe, we use the official version at 2016/06/30 as the single node baseline, and modify it using Poseidon's client library API for distributed execution. For TensorFlow, we use its open source version r0.10, and parallelize its single-node version with Poseidon's client library, and compare to its original distributed version. Note that as the distributed runtime of TensorFlow is highly optimized (e.g., auto-parallelization of graphs [1]). Poseidon avoids leveraging any build-in optimization of distributed TensorFlow by parallelizing its single-node version instead .

Dataset and Models

Our experiments use three well-known image classification datasets:

- CIFAR-10 [115], which contains 32×32 colored images of 10 classes, with 50K images for training and 10K for testing;

- ILSVRC12 [180], a subset of ImageNet22K that has 1.28 million of training images and 50K validation images in 1,000 categories;
- ImageNet22K [180], the largest public dataset for image classification, including 14,197,087 labeled images from 21,841 categories.

We test Poseidon's scalability across different neural networks:

- CIFAR-10 quick: a toy CNN from Caffe that converges at 73% accuracy for classifying images in CIFAR-10 dataset;
- GoogLeNet [200]: a 22-layer CNN with 5M parameters.
- Inception-V3 [202]: the ImageNet winner, an improved version of GoogLeNet from TensorFlow;
- VGG19 [191]: A popular feature extraction network in the computer vision community [192] that has 16 CONV layers and 3 FC layers, in total 143M parameters;
- VGG19-22K: we modify the VGG19 network by replacing its 1000-way classifier with a 21841-way classifier, to classify images from the ImageNet22K dataset. The modified network has 229M parameters.
- ResNet-152 [80]: the ImageNet winner network with 152 layers.

We list their statistics and configurations in Table 3.6.

Metrics

We mainly focus on metrics that measure the system performance, such as speedups on throughput (number of images scanned per second). Our experiments focus on medium-scale distributed clusters with up to 32 machines, which distributed DL empirically benefits most from. Larger clusters require larger batch sizes, which hurt the convergence rate of each iteration [27, 40]. For completeness, we also report the statistical performance (time/epoch to converge) on ResNet-152. Poseidon uses synchronized replication which enables many models to converge in fewer steps [1, 24, 27, 40].

3.5.2 Scalability

To demonstrate Poseidon's scalability, we train CNNs using Poseidon with different computational engines, and compare different systems in terms of their speedups on throughput. For Caffe engine, we train GoogLeNet[2], VGG19, and VGG19-22K networks; for TensorFlow engine, we train Inception-V3, VGG-19, and VGG19-22K.

[2]As there is no official implementation of Inception-V3 in Caffe, and the performance of different unofficial implementations varies dramatically, we use GoogLeNet instead.

Model	# Params	Dataset	Batchsize
CIFAR-10 quick	145.6K	CIFAR10	100
GoogLeNet	5M	ILSVRC12	128
Inception-V3	27M	ILSVRC12	32
VGG19	143M	ILSVRC12	32
VGG19-22K	229M	ImageNet22K	32
ResNet-152	60.2M	ILSVRC12	32

Table 3.6: Neural networks for evaluation. Single-node batch size is reported. The batch size is chosen based on the standards reported in literature (usually the maximum batch size that can fill in the GPU memory is used).

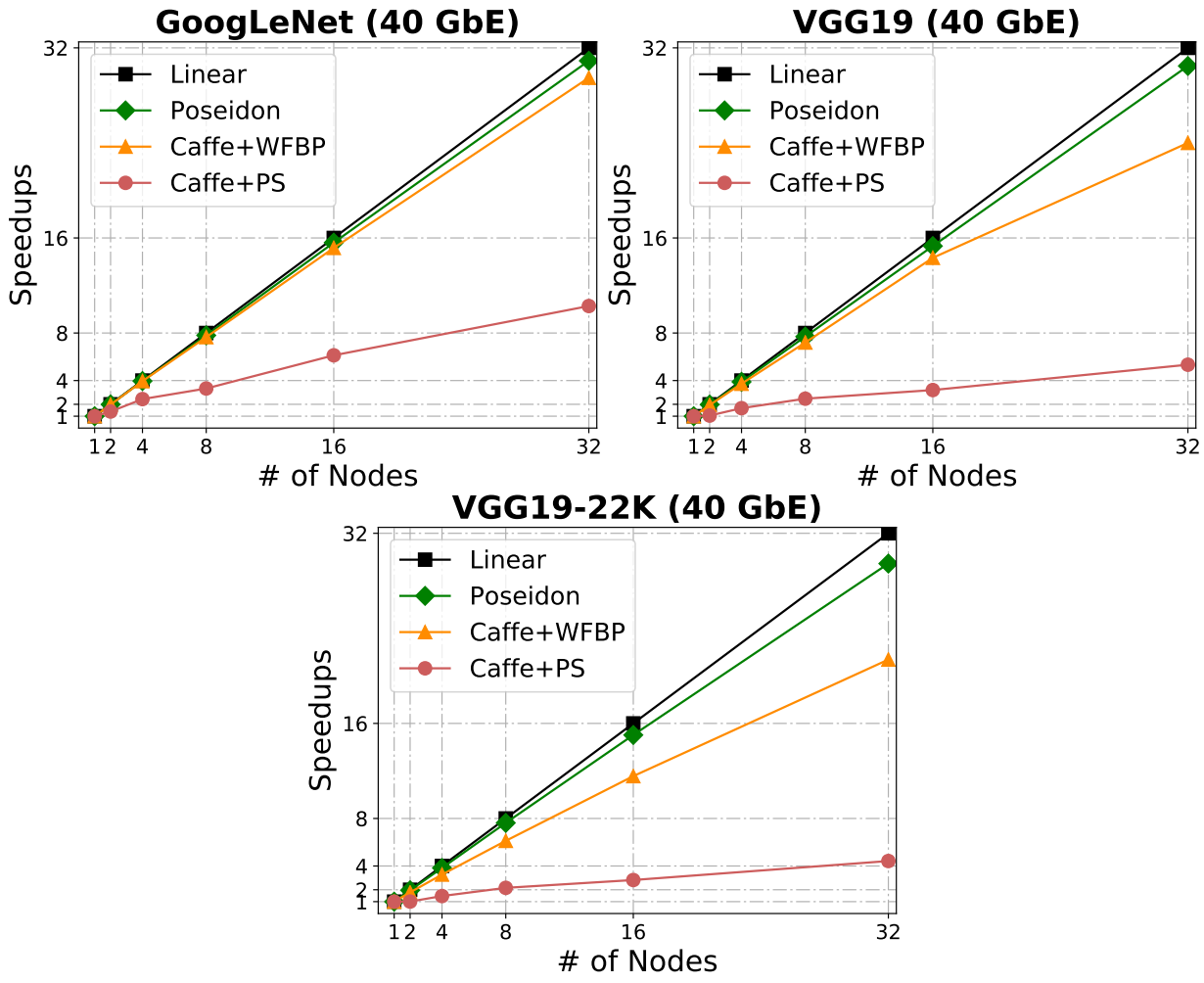

Figure 3.6: Throughput scaling when training GoogLeNet, VGG19 and VGG19-22K using Poseidon-parallelized Caffe and 40GbE bandwidth. Single-node Caffe is set as baseline (i.e., speedup = 1).

Caffe Engine

Figure 3.6 shows the throughput vs. number of workers when training the three networks using Caffe engine, given 40GbE Ethernet bandwidth available. We compare the following systems:

- **Caffe**: unmodified Caffe that executes on a single GPU;
- **Caffe+PS**: we parallelize Caffe using a vanilla PS, i.e., the parameter synchronization happens sequentially after the backpropagation in each iteration;
- **Caffe+WFBP**: parallelized Caffe using Poseidon so the communication and computation are overlapped. However, we disable the adaptive communication so that parameters are synchronized only via PS;
- **Poseidon**: the full version of Poseidon-Caffe.

Poseidon shows little overheads when combined with Caffe; running on a single node with no communication involved, Poseidon-Caffe can process 257, 35.5 and 34.2 images per second when training GoogLeNet, VGG19 and VGG19-22K, respectively, as compared to the original Caffe, which can process 257, 35.5 and 34.6 images, and Caffe+PS, which can only process 213.3, 21.3 and 18.5 images per second, due to the overheads caused by memory copy operations between RAM and GPU, which have been overlapped by Poseidon with the computation. In distributed environment, the rescheduling of computation and communication significantly improves the throughput: when training GoogLeNet and VGG19, incorporating WFBP achieves almost linear scalings up to 32 machines, and for the larger VGG19-22K network, Caffe+WFBP achieves 21.5x speedup on 32 machines. We conclude that rescheduling and multi-threading the communication and computation are key to the performance of distributed DL on GPUs, even when the bandwidth resource is abundant. Poseidon provides an effective implementation to overlap these operations for DL frameworks, to guarantee better GPU utilization.

When the available bandwidth is sufficient, the adaptive communication strategy shows small improvement on training GoogLeNet and VGG19. However, when training VGG19-22K which has three FC layers that occupy 91% of model parameters, it improves over Caffe-WFBP from 21.5x to 29.5x on 32 nodes.

TensorFlow Engine

We also modify TensorFlow using Poseidon, and compare the following systems in terms of speedup on throughput:

- **TF**: TensorFlow with its original distributed execution runtime;
- **TF+WFBP**: we modify TensorFlow using Poseidon's client library – we change the *assign* operator in TensorFlow, so that instead of being applied, the parameter updates will be synchronized via *Poseidon's PS interface with WFBP*;
- **Poseidon**: the full version of Poseidon-parallelized TensorFlow with the adaptive communication enabled.

We train Inception-V3, VGG19 and VGG19-22K models and report the results in Figure 3.7.

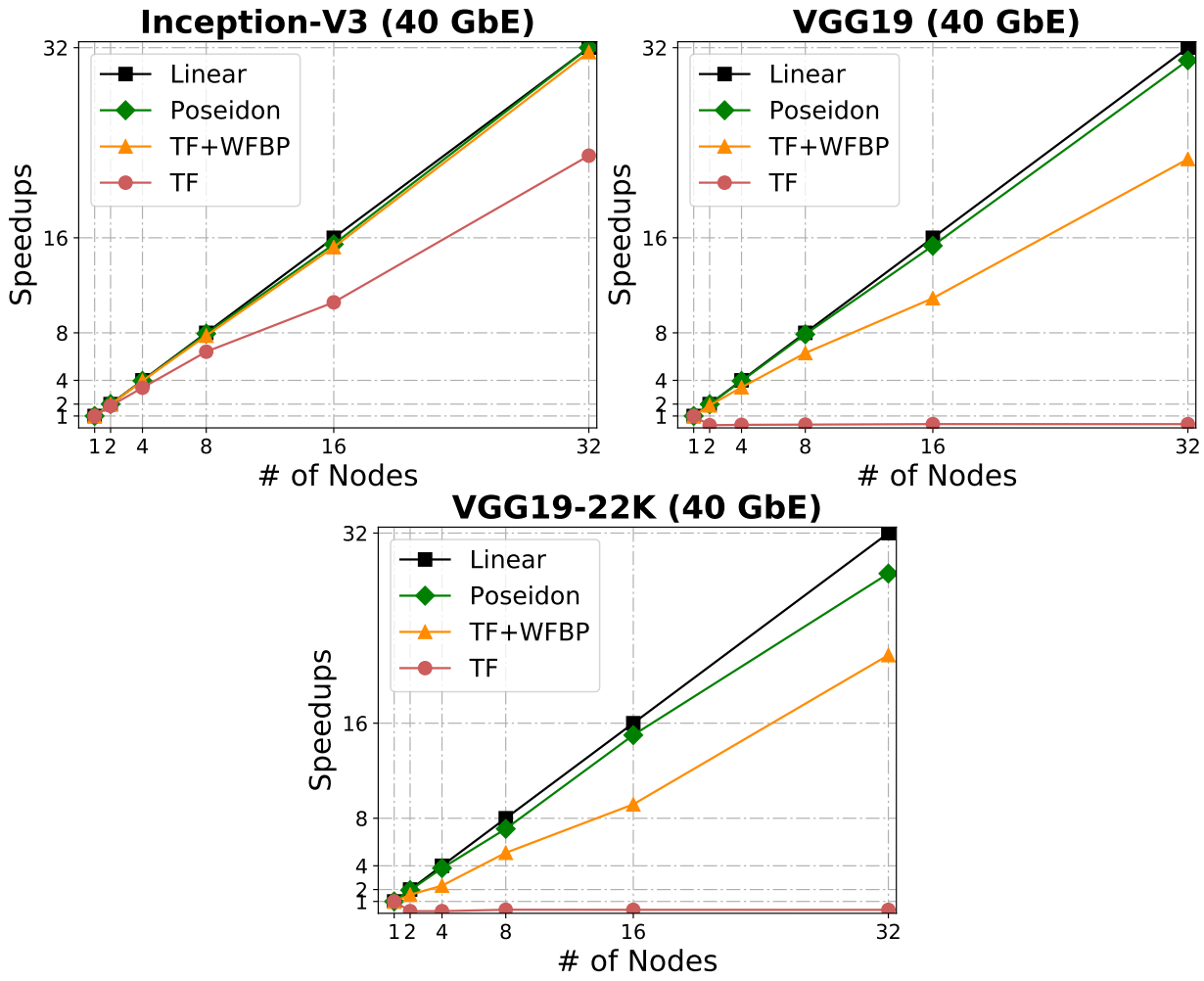

Figure 3.7: Throughput scaling when training Inception-V3, VGG19 and VGG19-22K using Poseidon-parallelized TensorFlow and 40GbE bandwidth. Single-node TensorFlow is set as baseline (i.e., speedup = 1).

Running on a single node, Poseidon processes 43.2, 38.2 and 34.5 images per second on training Inception-V3, VGG19 and VGG19-22K, while original TensorFlow processes 43.2, 38.5 and 34.8 images per second on these three models, respectively – little overhead is introduced by our modification. In distributed execution, Poseidon achieves almost linear speedup on up to 32 machines. Distributed TensorFlow, however, demonstrates only 10x speedup on training Inception-V3 and even fails to scale on training the other two networks in our experiments. To investigate the problem of TensorFlow and explain how Poseidon improves upon it, we illustrates in Figure 3.8 the (averaged) ratio of busy and stall time of a GPU when training the three networks using different systems on 8 nodes. Observe that Poseidon keeps GPUs busy in most of the time, while TensorFlow wastes much time on waiting for parameter synchronization. The inefficiency of distributed TensorFlow stems from two sources. First, TensorFlow partitions model parameters in a coarse-grained granularity – each tensor (instead of a KV pair) in the model is assigned to a PS shard. A big tensor (such as the parameter matrix in VGG19) is highly likely

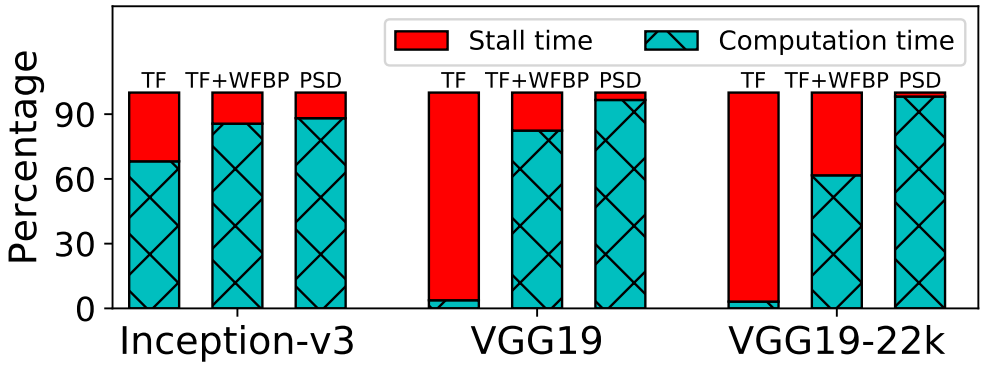

Figure 3.8: Breakdown of GPU computation and stall time when training the three networks on 8 nodes using different systems.

to create communication bottleneck on its located server node. Poseidon fixes this problem by partitioning parameters among server nodes in a finer-grained granularity using KV pairs, so that every node has evenly distributed communication load; as an evidence, TF+WFBP demonstrates higher computation-to-stall ratio in Figure 3.8. Second, TensorFlow cannot reduce the communication overheads while the suggested adaptive communication approach effectively reduces the size of messages. As a result, Poseidon further improves upon TF+WFBP from 22x to 30x on 32 nodes.

Multi-GPU Settings

The scheduling and communication strategies can be directly extended to support distributed multi-GPU environment with minor modifications. Specifically, when there are more than 1 GPU on a worker node, Poseidon will first collect the gradient updates following WFBP locally (either by full matrices or SFs) from multiple GPUs to a leader GPU using `CudaMemcpy()` API. If those updates are determined to be communicated via full matrices, Poseidon will aggregate them locally before sending out. Using Caffe engine on a single node, Poseidon achieves linear scalings on up to 4 Titan X GPUs when training all three networks, outperforming Caffe's multi-GPU version, which shows only 3x and 2x speedups when training GoogLeNet and VGG19. When running on AWS p2.8xlarge instances (8 GPUs each node), Poseidon reports 32x and 28x speedups when training GoogLeNet and VGG19 with 4 nodes (32 GPUs in total), confirming the existence of the overheads caused by memory movement between GPUs, though less substantial than network communication[3]. We will tackle this issue in Chapter 4.

Statistical Performance

For completeness, we report in Figure 3.9 the statistical performance for training ResNet-152 using Poseidon. Poseidon achieves near-linear speedups on both system throughput and statistical convergence: Poseidon delivers 31x speedup in terms of throughput, and reaches 0.24

[3]Note that the K80 GPUs on p2.8xlarge has less GFLOPS than Titan X used in our main experiments – the communication burden is less severe.

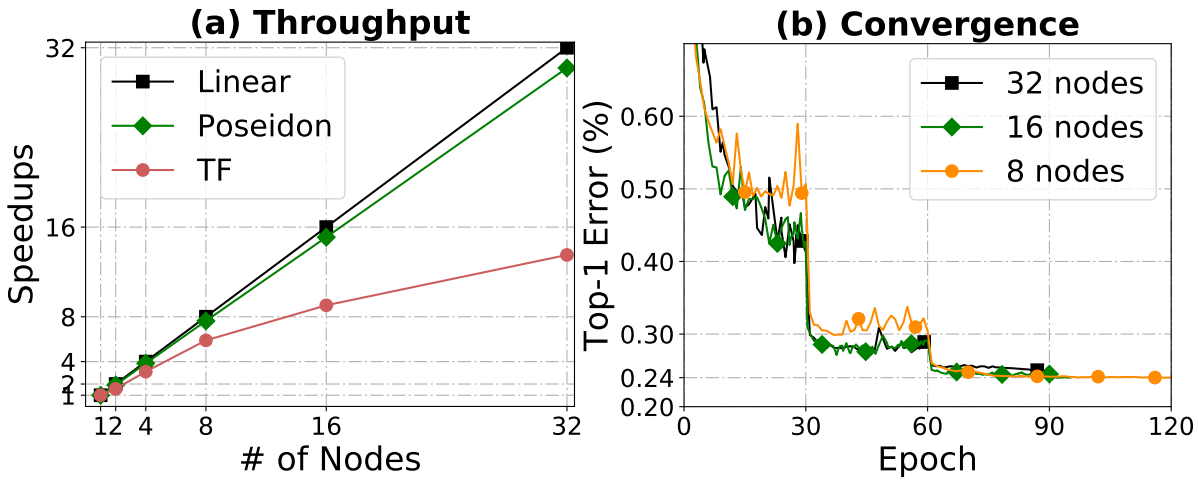

Figure 3.9: (a) Speedup vs. number of nodes and (b) Top-1 test error vs. epochs for training ResNet-152 using Poseidon-TensorFlow and the original TensorFlow.

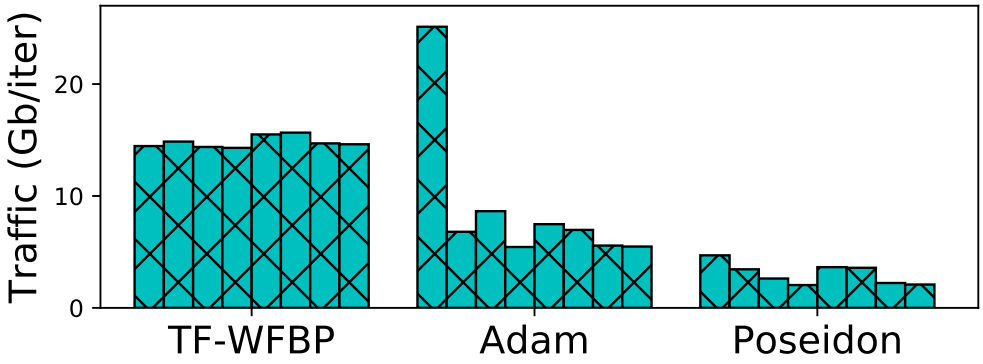

Figure 3.10: Averaged communication load when training VGG19 using *TF-WFBP*, *Adam* and *Poseidon* with TensorFlow engine. Each bar represents the network traffic on a node.

reported error with less than 90 epochs with both 16 and 32 nodes – thus linear scales in terms of time to accuracy, compared to 8 nodes with batchsize = 32×8, which is a standard setting as in [80], echoing recent results that synchronous training on distributed GPUs yields better performance than asynchronous training in terms of time to quality for most NNs [24, 40]. For other NNs in Table 3.6, Poseidon delivers the same quality of accuracy as reported in their papers [117, 192, 200, 202] on up to 32 GPUs.

3.5.3 Bandwidth Experiments

To understand the effectiveness of the proposed adaptive communication strategy, we create an ablation environment where network bandwidth is limited. We use Linux traffic control tool *tc* to lower the available bandwidth on each node, and compare the training throughput between with and without the adaptive communication. We focus on Caffe engine in this section because it is lighter and less optimized than TensorFlow.

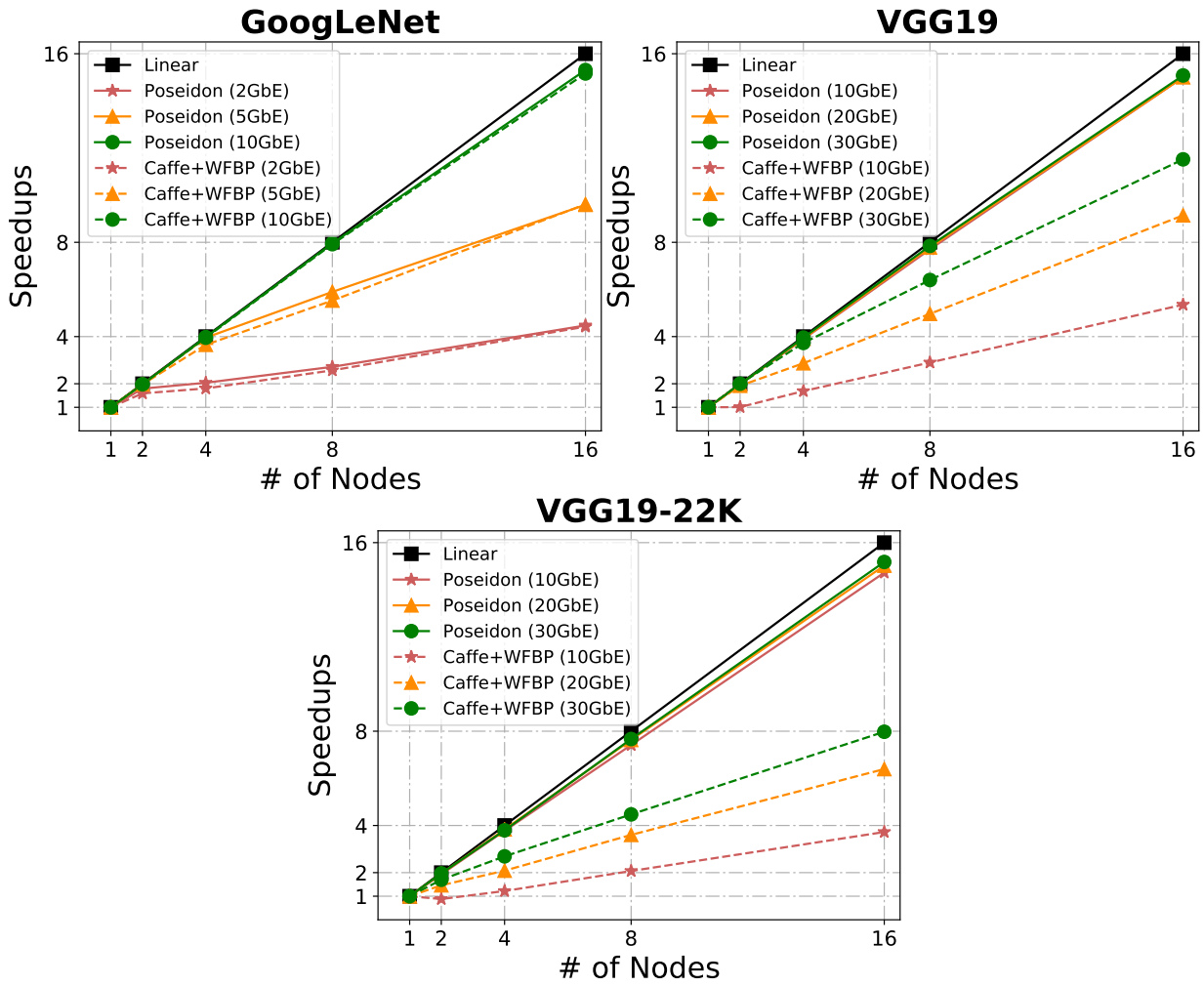

Figure 3.11: Throughput scaling when training GoogLeNet, VGG19 and VGG19-22K using Poseidon-parallelized Caffe with *varying network bandwidth*. Single-node Caffe is set as baseline (speedup = 1).

Figure 3.11 plots the speedup on throughput vs. number of workers when training GoogLeNet, VGG19 and VGG19-22K with different maximum bandwidth. Clearly, limited bandwidth prevents a standard PS-based system from linearly scaling with the number of nodes; for example, given 10GbE bandwidth (which is a commonly-deployed Ethernet configuration in most cloud computing platforms), training VGG19 using PS on 16 nodes can only be accelerated by 8x. This observation confirms our argument that limited bandwidth would result in communication bottleneck when training big models on distributed GPUs. Fortunately, Poseidon significantly alleviates this issue. Under limited bandwidth, it constantly improves the throughput by directly reducing the size of messages needed to be communicated, especially when the batch size is small; when training VGG19 and VGG19-22K, Poseidon achieves near-linear speedup on 16 machines using only 10GbE bandwidth, while an optimized PS would otherwise need 30GbE or even higher to achieve. Note that Poseidon will never underperform a traditional PS scheme because it will reduce to a parameter server whenever it results in less communication overheads;

for instance, we observe that Poseidon reduces to PS when training GoogLeNet on 16 nodes, because GoogleNet only has one thin FC layer (1000×1024) and is trained with a large batch size (128).

3.5.4 Comparisons to Other Methods

In this section, we compare Poseidon against other communication reduction methods, including Adam [32] and CNTK 1-bit quantization [235], and discuss their pros and cons.

Comparison to Adam

[32] To save bandwidth, Adam [32] synchronizes the parameters of a FC layer by first pushing SFs generated on all workers to a PS node, and then pulling back the full parameter matrices thereafter. As direct comparisons to Adam [32] are inaccessible, we implement its strategy in Poseidon, and compare it (denoted as *Adam*) to *TF-WFBP* and *Poseidon* by monitoring the network traffic of each machine when training VGG19 on 8 nodes using TensorFlow engine. As shown in Figure 3.10, the communication workload is highly imbalanced using Adam's strategy. Unlike a traditional PS (TF-WFBP) where the parameters are equally distributed over multiple shards, Adam cannot partition the parameters of FC layers because of their usage of SFs. Although the `push` operation uses SFs to reduce message size, the `pull` requires some server nodes to broadcast big matrices to each worker node, which creates bursty traffic that results in communication bottleneck on them. By contrast, Poseidon either partitions parameters equally over multiple PS shards, or transmits SFs among all peer workers, both are communication load-balanced and avoid bursty communication situations. Quantitatively, Adam delivers 5x speedup with 8 nodes when training VGG19.

Comparison to CNTK

[235] We compare Poseidon to the 1-bit quantization technique proposed in CNTK [235]. We create a baseline *Poseidon-1bit* which uses the 1-bit strategy to quantize the gradients in FC layers, and add the residual to updates of the next iteration. We then train the CIFAR-10 quick network, and plot the training loss and test error vs. iterations for two systems (both have linear scaling on throughput). As in Figure 3.12, 1-bit quantization yields worse convergence in terms of accuracy – on 4 GPUs, it achieves 0.5 error after 3K iterations, while Poseidon quickly converges to 0.3 error at iteration 1000. We conjecture this is caused by the quantization residual, which seems to be equivalent to delayed updates that may hurt the convergence performance when training NNs on images, confirmed by Cui *et al.* [40]. We also directly train VGG19 using CNTK-1bit system, and report 5.8x, 11x, 20x speedups on 8, 16 and 32 nodes, respectively, thus less scale-ups than Poseidon, and also compromised statistical performance due to approximate updates.

3.5.5 Application: Scaling Up Image Classification on ImageNet 22K

ImageNet 22K was the largest public dataset for image classification (2017), including nearly 14.2M labeled images from 21,841 categories, which was rarely touched by the research com-

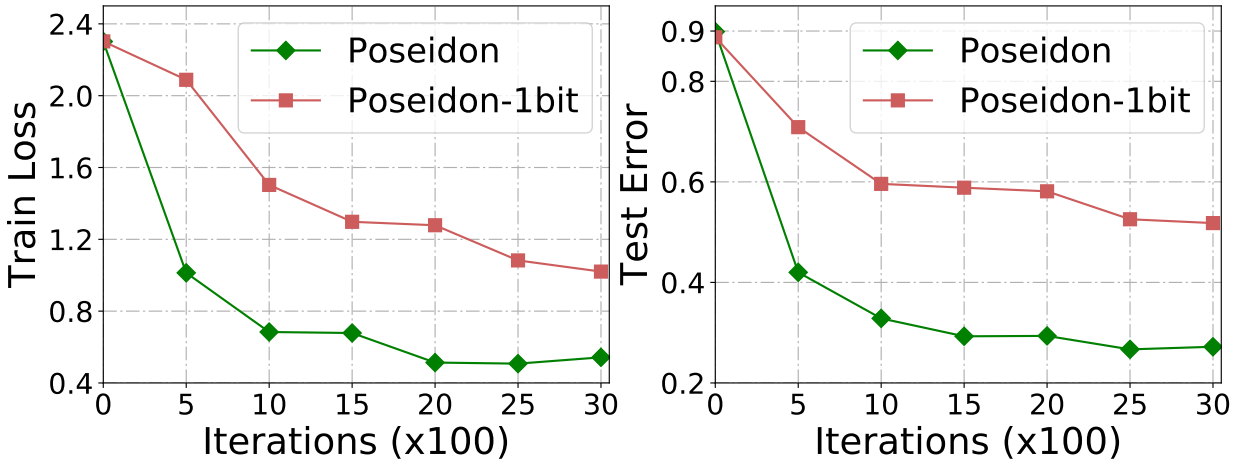

Figure 3.12: Training loss and test error vs. iteration when training CIFAR-10 quick network using *Poseidon* and *Poseidon-1bit* on 4 GPUs with Caffe engine.

Framework	Data	# machines/cores	Time	Train acc	Test acc
Poseidon	7.1M ImageNet22K for training, 7.1M for test	8 / 8 GPUs	3 days	41%	23.7%
Adam [32]	7.1M ImageNet22K for training, 7.1M for test	62 machines/?	10 days	N/A	29.8%
MxNet [27]	All ImageNet22K images for training, no test	1/4 GPUs	8.5 days	37.19%	N/A
Le *et al.* [121] w/ pretrain	7.1M ImageNet 22K, 10M unlabeled images for training, 7.1M for test	1K/16K CPU cores	3 days	N/A	15.8%

Table 3.7: Comparisons of the image classification results on ImageNet 22K.

munity due to its massive data size and complexity. We experiment on ImageNet 22K to demonstrate the scalability of Poseidon. As no official test data exists for evaluation, following previous settings in [32, 46, 121], we randomly split the whole set into two parts, and use the first 7.1 million of images for training and remained for test. Similar to ILSVRC 2012, we resize all images to 256×256 and report the top-1 test accuracy.

Settings

We design an AlexNet-like architecture; specifically, the CNN takes a random 227×227 crop from the original image, and forwards it into 5 CONV layers and 2 FC layers before prediction. The CNN has convolution filters with sizes $7 \times 7, 5 \times 5$ and 3×3. Similar to AlexNet, the first, second and fifth CONV layers are followed by max pooling layers with size 3×3 and stride 2. Two FC layers with 3,000 neurons each are put at the top of the network, followed by a softmax layer to be a 21,841-way classier with 120M parameters and 1.8 billion of connections overall.

We train the CNN with data-parallelism by equally partitioning the training data into 8 GPU nodes. The batch size and staleness are fixed at 256 and 0, respectively. The network is trained using the step learning rate policy, with base learning rate 0.005 and decreased 6 times.

Performance

Table 3.7 compares our result to those of previous work on ImageNet 22K: Adam, MxNet, and Le et al. [121]. Note that at this point fair comparisons between different frameworks might not be possible, because the experiment setup of ImageNet 22K was not standardized, all the source codes were not fully available yet, and large variations might exist in system configurations, models, and implementation details. However, it is clear that Poseidon achieves a competitive accuracy 23.7% with the state-of-the-art systems with shorter training time and less machine resources. Compared to Adam [32], we only use 30% training time and 13% machines to achieve 23.7% accuracy with a similarly sized model. Promisingly, we achieve a higher training accuracy with 3 days of training using a well-established CNN model — this compares favorably to MXNet, which uses the whole set of 14.1 million images to train an inception-BN structure [91] using 4 GPUs in a single machine without network communication, and reports 37.1% train accuracy after 8.5 days of training.

3.6 Additional Related Work

PS-based Distributed DL Systems

Based on the parameter server [127, 221] architecture, a number of CPU-based distributed DL systems have been developed, such as [46, 121, 218, 252] and Adam [32]. They are purely PS-based systems on CPU-only clusters, whereas Poseidon addresses the more challenging case of GPU clusters.

Scaling up DL on distributed GPUs is an active field of research. Coates *et al.* [34] build a GPU-based multi-machine system for DL using model parallelism rather than data parallelism, and their implementation is rather specialized for a fixed model structure while demanding specialized hardware, such as InfiBand networking. TensorFlow [1] is Google's distributed ML platform that uses a dataflow graph to represent DL models, and synchronizes model parameters via PS. It therefore cannot dynamically adjust its communication method depending on the layer and cluster information as Poseidon does. MXNet [27] is another DL system that uses PS for distributed execution, and supports TensorFlow-like graph representations for DL models. By auto-parallelizing independent subgraphs, both frameworks implicitly overlap the communication and computation. By contrast, Poseidon has a more explicit way to overlap them via its client library. Hence, Poseidon can be also used to parallelize non-graph-based frameworks. Moreover, both MXNet and TensorFlow do not address the bottleneck caused by limited network bandwidth, which undermines their scalability when training large models with dense layers (e.g., big softmax). Besides, Cui *et al.* propose GeePS [40] that manages the limited GPU memory and report speedups on distributed GPUs. While, GeePS does not address the issue of limited network bandwidth. Therefore, Poseidon's technique could be combined with them to enable better training speedups. Also of note are several efforts to port Caffe onto other

distributed platforms, such as SparkNet [152], YahooCaffe [231] and FireCaffe [89], the former reports a 4-5 times speedup with 10 machines (and hence less scalability than our results herein).

Other Distributed ML Systems

CNTK [235] is a DL framework that supports distributed executions and addresses the problem of communication bottleneck via the 1-bit quantization technique. CNTK demonstrates little negative impact on convergence in speech domains [184, 185]. However, in may scenarios (Section 3.5.4), the performance is usually compromised by noisy gradients [1, 40]. By contrast, Poseidon's adaptive communication strategy reduces the communication while always guaranteeing synchronous training. There are also growing interest in parallelizing ML applications using peer-to-peer communication, such as MALT [125], SFB [225] and Ako [219]. Poseidon draws inspiration from these works but goes one step further as it is an adaptive best-of-both-worlds protocol, which will select client-server communication whenever it would result in fewer overheads.

Recently, there is a surge of interest in bring in collective communication primitives, such as *AllReduce*, *AllGather*, from the high-performance computing areas into distributed deep learning [68, 95, 186]. Specialized systems built on top of these libraries (e.g. MPI) have successfully scaled ResNet training on thousands of GPU nodes on a homogeneous cluster, and reduced the the ImageNet training into hours or even minutes. The proposed techniques in scheduling and communication can be integrated with collective communication, similarly as we have done with parameter server and sufficient factor broadcasting architectures.

Adaptive Communication

A second successful instance of adaptive communication [107] is to mix parameter server with AllReduce [107], based on the sparsity of trainable variables in NNs – defined by how their elements are accessed in computation. For a dense variable, all elements are accessed at least once during a single training iteration, which is the case for most build blocks in CNNs. For a sparse variable, only a subset of the elements are accessed in one iteration – a pattern that is commonly found in NLP models with embedding layers, where the embedding variable could be as large as with billions of float parameters. Synchronizing such a variable across multiple GPUs requires significant network bandwidth and consumes many CPU clocks for aggregating results from GPUs, and if done in the same way as for dense variables, usually results in no scalability. While on the other hand, treating all variables as sparse variables is sub-optimal, as there are highly optimized implementations for communicating dense variables across GPUs such as the NCCL library. The idea therefore comes in naturally – estimating the communication overhead ahead of execution, and adaptively choosing from PS and AllReduce based on the sparse access patterns of different layers and estimated overheads.

Scheduling for Distributed DL

WFBP, as one of the earliest structure-aware scheduling for distributed DL, has inspired a few follow-up work, such as ByteScheduler [168] and PipeDream [78]. ByteScheduler analyzes and derives the optimal schedule in the ideal scenario with strong assumptions, and uses Bayesian

Optimization (BO) to auto-tune the partition size to work with a practical environment. PipeDream generalizes Pipeline parallelism that automatically partitions DNN training across workers, combining inter-batch pipelining with intra-batch parallelism to better overlap computation with communication, while minimizing the amount of data communicated.

Chapter 4

Memory Management

The limited size of GPU device memory was viewed as a serious impediment to parallel DL training on GPU clusters, limiting the size of the model to what could fit in a single device memory. In fact, as observed by prior work [32, 34, 116], this would seem to imply that GPU-based systems (with their limited GPU memory) are suited only for relatively small neural networks. Besides, the frequent data movements between GPU and DRAM also adds nontrivial overheads to both communication and computation.

In this chapter, we derive strategies and system implementations to improve memory management in distributed ML. We first present *memory swapping (MemSwap)*, a simple memory management mechanism inspired by a key observation in DL training – a layer's computation only depends on the outputs (and intermediate states) produced by its dependent layers, but not that by the entire model. Informed by the dependency structure presented in the model, the memory management can be adapted to hide data movement overheads, and made it possible for training large models that otherwise cannot fit in the limited GPU memory.

We then present a corresponding implementation, GeePS [40], a parameter server system specialized for scaling deep learning applications across GPUs distributed among multiple server machines. Based on MemSwap, GeePS performs a number of optimizations specially tailored to making efficient use of GPUs, such as GPU-friendly caching, data staging, and memory management techniques – GeePS overcomes the above-mentioned limitations by assuming control over memory management and placement, and carefully orchestrating data movement between CPU and GPU memory based on its observation of the access patterns at each layer of the neural network.

Combining the techniques presented in Chapter 3 and in this chapter enables designing and employing CNN models with a much greater number of parameters (which otherwise might be difficult due to the significantly increased communication loads and memory footprints). In Section 4.5 of this chapter, we briefly introduce a new CNN architecture, hierarchical deep convolutional neural networks (HD-CNN), built on top of GeePS and Poseidon. By augmenting a backbone model with 2x more auxiliary parameters and 7x more intermediate states, HD-CNN boosts the image classification accuracy on ImageNet for up to 3 percent, and had achieved state-of-the-art results in 2016.

4.1 Introduction

This section walks through deep learning on GPU and GPU clusters from the perspective of *memory usage and consumption*, and then raises the specific challenges in the aspect of memory management.

4.1.1 Deep Learning on GPUs: a Memory Management Perspective

Most high end GPUs are self-contained devices that can be inserted into a server machine, as illustrated in Figure 4.1. One key aspect of GPU devices is that they have dedicated local memory, which we will refer to as *GPU memory*, and their computing elements are only efficient when working on data in that GPU memory. Data stored outside the device, in CPU memory, must first be brought into the GPU memory (e.g., via PCI DMA) for it to be accessed efficiently.

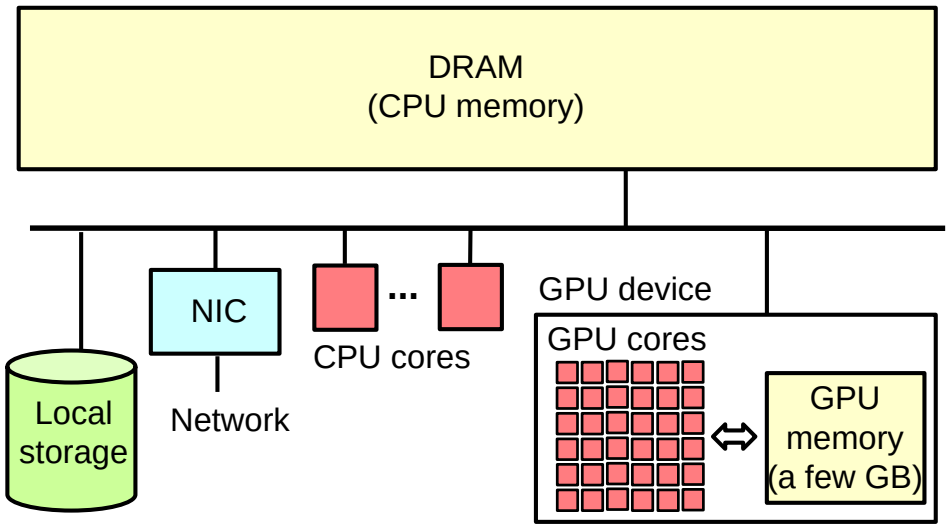

Figure 4.1: A machine with a GPU device.

Contemporary ML frameworks such as Caffe, TensorFlow, PyTorch, MxNet all follow a similar pattern to interact with GPU devices and their memory: a single-threaded worker launches and joins with GPU computations, by calling NVIDIA cuBLAS and cuDNN libraries as well as some customized CUDA kernels. Each mini-batch of training data is read from an input file via the CPU, moved to GPU memory, and then processed as described above. For efficiency, many frameworks keep all model parameters and intermediate states in the GPU memory. As such, it is effective only for models and mini-batches small enough to be fully held in GPU memory. Figure 4.2 illustrates the CPU and GPU memory usage adopted in these frameworks.

4.1.2 Memory Challenges for Distributed DL on GPU Clusters

Parameter server architecture has become a popular approach to making it easier to build and scale DL across CPU-based clusters [4, 32, 38, 46, 83, 169], particularly for data-parallel execution. Figure 4.3 illustrates the basic parameter server architecture from a memory perspective

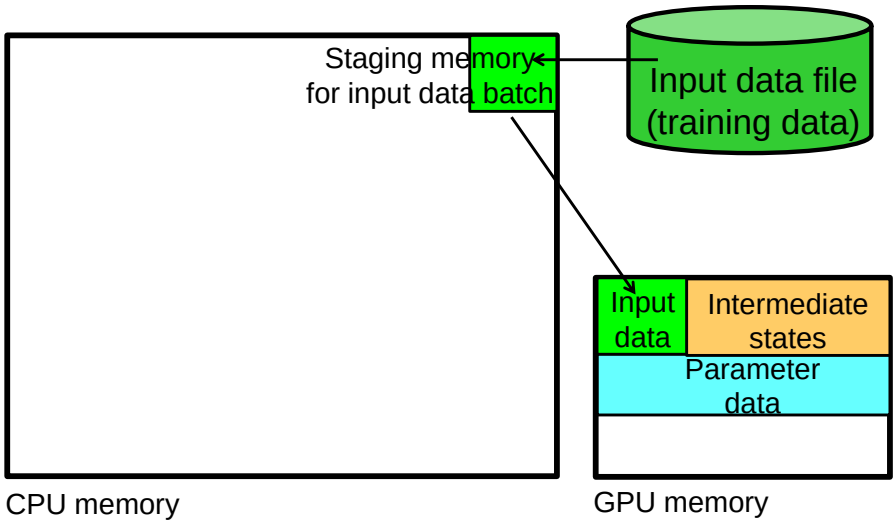

Figure 4.2: Single GPU deep learning and memory usage on contemporary ML frameworks.

– all states shared among worker threads (i.e., the model parameters being learned) are kept in distributed shared memory implemented as a specialized key-value store called a "parameter server", usually on DRAM. While the picture illustrates the parameter server as separate from the machines executing worker threads, and some systems do work that way, the server-side parameter server state is commonly sharded across the same machines as the worker threads. The latter approach is particularly appropriate when considering a parameter server architecture for GPU-based ML execution, since the CPU cores and CPU memory would be largely unused by the GPU-based workers.

Given its proven value in CPU-based distributed ML, it is natural to use the same basic architecture and programming model with distributed ML on GPUs. To explore its effectiveness, we ported two applications (based on Caffe) to a state-of-the-art CPU-based parameter server system (IterStore [38]). Doing so was straightforward and immediately enabled distributed deep learning on GPUs, confirming the application programmability benefits of the data-parallel parameter server approach. Figure 4.4 illustrates what sits where in memory, to allow comparison to Figure 4.2 and designs described later.

While it was easy to get working, the performance was not acceptable. As noted by Chilimbi et al. [32], the GPU's computing structure makes it "extremely difficult to support data parallelism via a parameter server" using current implementations, because of GPU stalls, insufficient synchronization/consistency, or both. Also as noted by them and others [224], the need to fit the full model, as well as a mini-batch of input data and intermediate neural network states, in the GPU memory limits the size of models that can be trained. While Chapter 3 has proposes solutions to address issues related with communication and scheduling, the next section describes design for overcoming the limitations in memory.

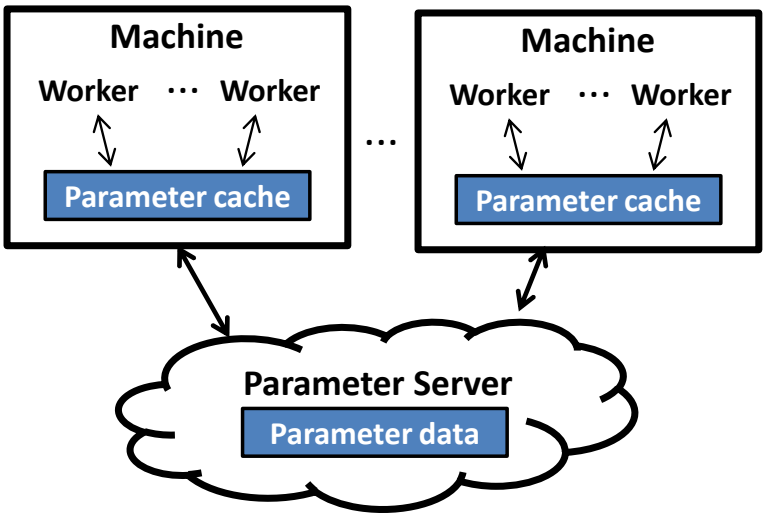

Figure 4.3: Parallel ML with parameter server. Parameter cahce or data usually locate on shared CPU memory (DRAM).

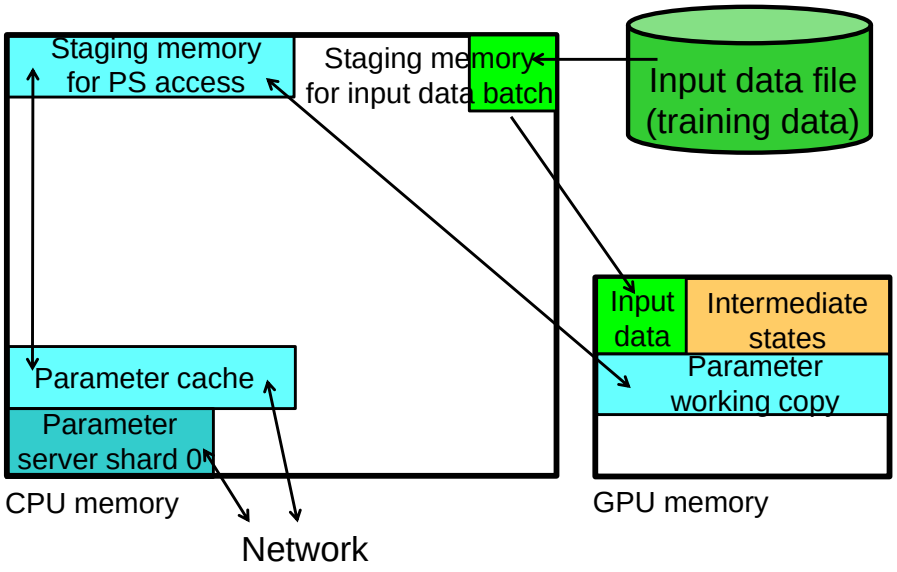

Figure 4.4: Distributed ML on GPUs using a CPU-based parameter server. The right side of the picture is much like the single-GPU illustration in Figure 4.2. But, a parameter server shard and client-side parameter cache are added to the CPU memory, and the parameter data originally only in the GPU memory is replaced in GPU memory by a local working copy of the parameter data. Parameter updates must be moved between CPU memory and GPU memory, in both directions, which requires an additional application-level staging area since the CPU-based parameter server is unaware of the separate memories.

4.2 Memory Swapping

The section describe memory swaping (MemSwap), and a series of techniques developed based on it to enable distributed training of large models on limited GPU device memory.

The key idea of memory swapping is simple – when the GPU memory of a machine is not big enough to host all data, including parameters and intermediate states, we swap parts of the data to the CPU memory. For efficiency, we restrict the ML training to still access everything through GPU memory, as before, and the memory management library will do the data movement between GPU and CPU when needed. Copying data between GPU and CPU memory could significantly slow down data access. To minimize slowdowns, a memory manager will use separate threads to perform the memcpy operations between CPU and GPU memory in the background. In order to prefetch the content from CPU to GPU, it will need to know in advance the sets of parameter data that the application will access. This could be enabled by incorporating knowledge of the neural network structure into memory management. For example, in the forward pass of CNNs, once the forward computation f_i (defined in Section 3 and Figure 3.3) is finished, the intermediate states or parameters stored before layer i could be temporally moved to CPU memory as they are no longer needed until the backward pass comes back at layer i; and this movement could be performed concurrently with the ongoing forward computation $f_j(j > i)$, upcoming backward computation $b_j(j > i)$, and communication $o_j, i_j(j > i)$. Similarly, before the backward computation approaches layer i, the data movement for the intermediate states needed for b_i, from CPU memory back to GPU memory, can be performed in advance and overlapped7d with the backward computation operations $b_j(j > i)$ and associated communications.

We next describe an implementation of MemSwap on distributed shared memory to enable efficient support of parallel ML applications running on distributed GPUs, including three primary specializations to a parameter server architecture: explicit use of GPU memory for the parameter cache, batch-based parameter access methods, and parameter server management of GPU memory on behalf of the application. The first two address performance, and the third expands the range of problem sizes that can be addressed with data-parallel execution on GPUs. But note MemSwap, as a generic memory management strategy, can be used in *any non-distributed setting or accelerator devices with limited memory*. In fact, since the proposal of MemSwap and GeePS, we have observed extensive development on using MemSwap to improve the memory management in deep learning [86, 119].

4.2.1 Maintaining the Parameter Cache in GPU memory

One important change needed to improve parameter server performance for GPUs is to keep the parameter cache in GPU memory, as shown in Figure 4.5 (Section 4.2.3 discusses the case where everything does not fit). Perhaps counter-intuitively, this change is not about reducing data movement between CPU memory and GPU memory — the updates from the local GPU must still be moved to CPU memory to be sent to other machines, and the updates from other machines must still be moved from CPU memory to GPU memory. Rather, moving the parameter cache into GPU memory enables the parameter server client library to perform these data movement steps in the background, overlapping them with GPU computing activity. Then, when the application

uses the read or update functions, they proceed within the GPU memory. Putting the parameter cache in GPU memory also enables updating of the parameter cache state using GPU parallelism.

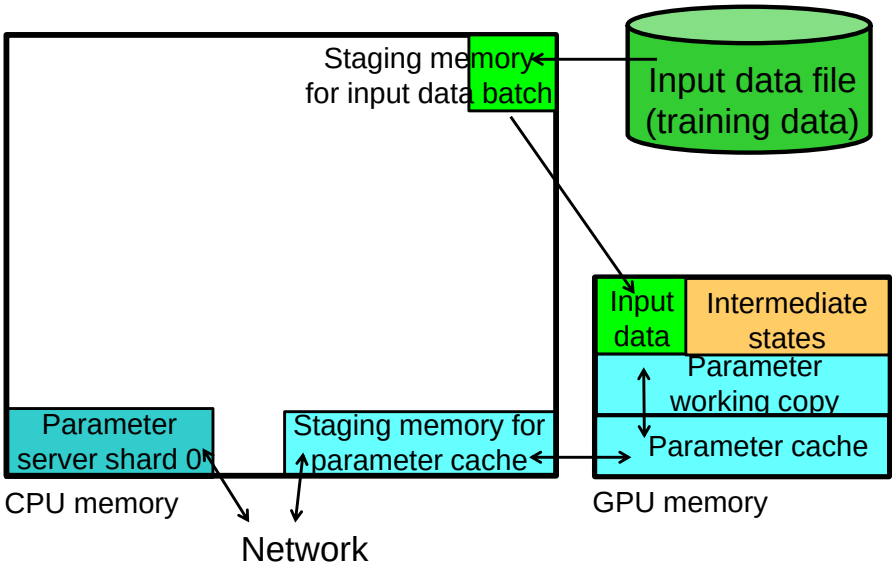

Figure 4.5: Parameter cache in GPU memory. In addition to the movement of the parameter cache box from CPU memory to GPU memory, this illustration differs from Figure 4.4 in that the associated staging memory is now inside the parameter server library. It is used for staging updates between the network and the parameter cache, rather than between the parameter cache and the GPU portion of the application.

4.2.2 Pre-built Indexes and Batch Operations

Given the SIMD-style parallelism of GPU devices, per-value read and update operations of arbitrary model parameter values can significantly slow execution. In particular, performance problems arise from per-value locking, index lookups, and data movement. To realize sufficient performance, GPU-specialized parameter server needs to support batch-based interfaces for reads (e.g., `READ-BATCH`) and updates (e.g., `UPDATE-BATCH`). Moreover, we exploit the repeating nature of iterative model training [38] to provide batch-wide optimizations, such as pre-built indexes for an entire batch that enable GPU-efficient parallel "gathering" and updating of the set of parameters accessed in a batch. These changes make parameter servers much more efficient for GPU-based training.

4.2.3 Managing Limited GPU Device Memory

As noted earlier, the core idea of MemSwap is to manage the GPU memory for the application and swap the data that is not currently being used to CPU memory. It can move the data between GPU and CPU memory in the background, minimizing overhead by overlapping the transfers with the training computation, and our results demonstrate that the two do not interfere with one another.

Managing GPU Memory Inside the Parameter Server

Incorporating MemSwap requires parameter server to provide read and update interfaces with parameter-server-managed buffers. When the application reads parameter data, the parameter server client library will *allocate* a buffer in GPU memory for it and return the pointer to this buffer to the application, instead of copying the parameter data to a buffer provided by the application. When the application finishes using the parameter data, it returns the buffer to the parameter server. We call those two interfaces `BUFFER-READ-BATCH` and `POST-BUFFER-READ-BATCH`. When the application wants to update parameter data, it will first request a buffer from the parameter server using `PRE-BUFFER-UPDATE-BATCH` and use this buffer to store its updates. The application calls `BUFFER-UPDATE-BATCH` to pass that buffer back, and the parameter server library will apply the updates stored in the buffer and reclaim the buffer memory. To make it concise, in the rest of this chapter, we will refer to the batched interfaces using PS-managed buffers as `READ`, `POST-READ`, `PRE-UPDATE`, and `UPDATE`.

The application can also store their local non-parameter data (e.g., intermediate states) in the parameter server using the same interfaces, but with a `LOCAL` flag. The local data will not be shared with the other application workers, so accessing the local data will be much faster than accessing the parameter data. For example, when the application reads the local data, the parameter server will just return a pointer that points to the stored local data, without copying it to a separate buffer. Similarly, the application can directly modify the requested local data, without needing to issue an `UPDATE` operation.

Swapping Data to CPU Memory When It Does Not Fit

By storing the local data in the parameter server, almost all GPU memory can be managed by the parameter server client library. When the GPU memory of a machine is not big enough to host all data, the parameter server will store part of the data in the CPU memory. The application still accesses everything through GPU memory, as before, and the parameter server library will do the data movement for it. When the application `READ`s parameter data that is stored in CPU memory, the parameter server will perform this read using a CPU core and copy the data from CPU memory to an allocated GPU buffer. Likewise, when the application `READ`s local data that is stored in CPU memory, the parameter server will copy the local data from CPU memory to an allocated GPU buffer. Figure 4.6 illustrates the resulting data layout in the GPU and CPU memories.

GPU/CPU Data Movement in the Background

Copying data between GPU and CPU memory could significantly slow down data access. To minimize slowdowns, MemSwap uses separate threads to perform the `READ` and `UPDATE` operations in the background. For an `UPDATE` operation, because the parameter server owns the update buffer, it can apply the updates in the background and reclaim the update buffer after it finishes. In order to perform the `READ` operations in the background, the parameter server will need to know in advance the sets of parameter data that the application will access. Fortunately, as we have explained above, this could be inferred based on the knowledge of the model structure, so the parameter server can easily predict the `READ` operations and perform them in advance

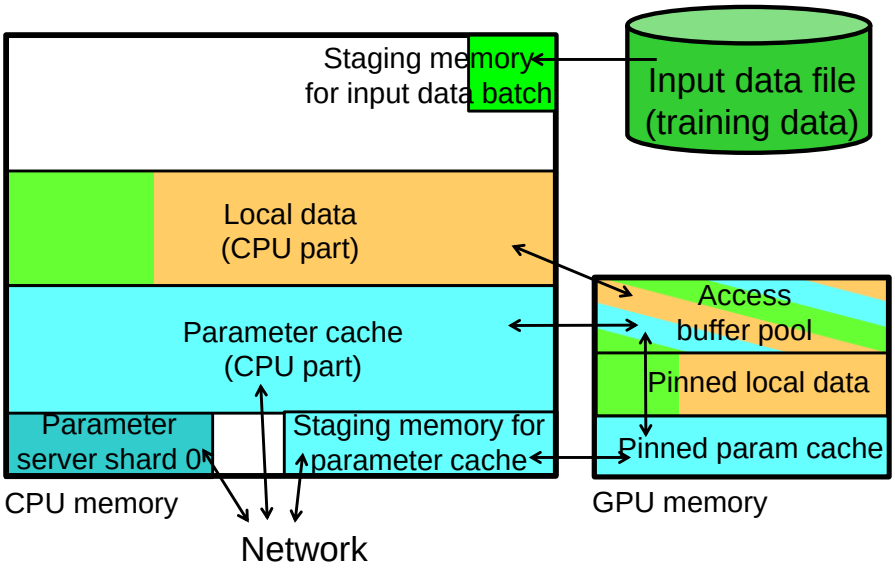

Figure 4.6: Parameter cache and local data partitioned across CPU and GPU memories. When all parameter and local data (input data and intermediate states) cannot fit within GPU memory, our parameter server can use CPU memory to hold the excess. Whatever amount fits can be pinned in GPU memory, while the remainder is transferred to and from buffers that the application can use, as needed.

in the background.

4.3 GeePS: Memory-optimized Parameter Server on GPUs

This section describes GeePS, a GPU-specialized parameter server system that optimizes memory management based on MemSwap described in Section 4.2.

4.3.1 GeePS Data Model and API

GeePS is a C++ library that manages both the *parameter data* and *local data* for machine learning applications. The distributed application program usually creates one ML worker process on each machine and each of them links to one instance of the GeePS library. For GPU-based ML applications (such as Caffe or TensorFlow), the worker often runs in a single CPU thread and launches and joins with GPU computations, which might be NVIDIA library calls or customized CUDA kernels. On initializing the GeePS library, the application will provide the list of hosts that the program is running on, and each GeePS instance will create connections to the other hosts.

GeePS manages data as a collection of *rows* indexed by keys. The rows are then logically grouped into *tables*, and rows in the same table share the same attributes (e.g., data age). GeePS implements the read and update operations with parameter-server-managed buffers for parameter data access, and we call them `READ`, `POST-READ`, `PRE-UPDATE`, and `UPDATE` for short. For

local data access, GeePS provides LOCAL-ACCESS and POST-LOCAL-ACCESS, for which the application can directly modify the accessed local data without an explicit update operation. GeePS also provides a TABLE-CLOCK function for application workers to signal the completion of per-table updates. GeePS implicitly synchronizes the application workers by letting them wait on read operations, when the data is not available yet. It supports three execution synchrony models: BSP, SSP [83], and Asynchrony.

4.3.2 Architecture

Storing Data

Each GeePS instance stores one shard of the master version of the parameter data in its *parameter server shard*. The parameter server shard is not replicated, and fault tolerance is handled by checkpointing. In order to reduce communication traffic, each instance has a *parameter cache* that stores a local snapshot of the parameter data, and the parameter cache is refreshed from the parameter server shards, such as at every clock for BSP. When the application applies updates to the parameter data, those updates are also stored in the parameter cache (a write-back cache) and will be submitted to the parameter server shards at the end of every clock. The parameter cache has two parts, a GPU-pinned parameter cache and a CPU parameter cache. If everything fits in GPU memory, only the GPU parameter cache is used. But, if the GPU memory is not big enough, GeePS will keep some parameter data in the CPU parameter cache. The data placement policies are described in Section 4.3.4. Each GeePS instance also has an *access buffer pool* in GPU memory, and GeePS allocates GPU buffers for READ and PRE-UPDATE operations from the buffer pool. When POST-READ or UPDATE operations are called, the memory will be reclaimed by the buffer pool. GeePS manages application's input data and intermediate states as *local data*, which is local to each worker. The local data also has a GPU-pinned part and a CPU part, with the CPU part only used if necessary. GeePS divides the key space into multiple *partitions*, and the rows in different partitions are managed in separate data structures, with different sets of communication threads.

Data Movement across Machines

GeePS performs communication across machines asynchronously with three types of communication threads: *keeper threads* manage the parameter data in parameter server shards; *pusher threads* move parameter data from parameter caches to parameter server shards, by sending messages to keeper threads; *puller threads* move parameter data from parameter server shards to parameter caches, by receiving messages from keeper threads.

The communication is implemented using sockets, so the data needs to be copied to some CPU staging memory before being sent through the network, and the received data will also be in the CPU staging memory. To send/receive data from/to GPU parameter cache, the pusher/puller threads will need to move the data between CPU memory and GPU memory using CUDA APIs.

Data Movement inside a Machine

GeePS uses two background threads to perform the READ and UPDATE operations for the application. The *allocator* thread performs the READ and PRE-UPDATE operations by allocating buffers from the buffer pool and (only for READ) copying parameter values from the parameter cache to the buffers. The *reclaimer* thread performs the POST-READ and UPDATE operations by (only for UPDATE) applying updates from the buffers to the parameter cache and reclaiming the buffers back to the buffer pool. These threads assign and update parameter data in large batches with pre-built indices by launching CUDA kernels on GPUs, as described in Section 4.3.3.

Locking

GeePS's background threads synchronize with each other, as well as the application threads, using mutex locks and conditional variables. Unlike some other CPU-based parameter servers that use per-row locks [38, 221], we use a coarse-grained locking design, where one set of mutex lock and conditional variable are used for a whole key partition. We make this design decision for two reasons. First, with coarse-grained locking, batched data operations can be easily performed on a whole partition of rows. Second, unlike CPU applications, where one application thread is launched for each CPU core, a GPU application often has just one CPU host thread accessing the parameter server, making lock contention less of an issue.

Operation Sequence Gathering

Some of the specializations (pre-built indices, background READ, and data placement decisions) exploit knowledge of the operation sequence of the application. GeePS implements an operation sequence gathering mechanism like that described by Cui et al. [38]. It can gather the operation sequence information either by directly analyzing the model structure, or in the first iteration or in a *virtual iteration*, during which the application just reports its sequence of operations without doing any real computation or keeping any states. GeePS uses the gathered operation sequence information as a hint to build the data structures (parameter server shard, parameter cache, and local data), build the access indices, prefetch the data (including cross-machine data fetching and background READ), and make GPU/CPU data placement decisions. Since the gathered access information is used only as a hint, knowing the exact operation sequence is not a requirement for correctness, but a performance optimization.

For most deep learning applications (including CNN and RNN), the application accesses all model parameters every mini-batch. For some applications with sparse training data (e.g., BOW representation for NLP tasks), the bottom layer of the network might just use a subset of the weights. Even for these tasks, the operation sequence of a whole epoch still repeats. The operation sequence only changes when the training data is shuffled. For this special case, we just prefetch all the parameter data.

4.3.3 Parallelizing Batched Access

GeePS provides a key-value store interface to the application, where each parameter row is named by a unique key. When the application issues a batched read or update operation, it

will provide a list of keys for the target rows. GeePS could use a hash map to map the row keys to the locations where the rows are stored. But, in order to make the batched access be executed by all GPU cores, GeePS will use the following mechanism. Suppose the application update n rows, each with m floating point values, in one `UPDATE` operation, it will provide an array of n keys $\{keys[i]\}_{i=1}^{n}$ and an array of n parameter row updates $\{\{updates[i][j]\}_{j=1}^{m}\}_{i=1}^{n}$. GeePS will use an index with n entries, where $\{index[i]\}_{i=1}^{n}$ is the location of the parameter data for $\{keys[i]\}_{i=1}^{n}$. Then, it will do the following data operation for this `UPDATE`: $\{\{parameters[index[i]][j] \mathrel{+}= updates[i][j]\}_{j=1}^{m}\}_{i=1}^{n}$. This operation can be executed with all the GPU cores. Moreover, the index can be built just once for each batch of keys, based on the operation sequence gathered as described above, and re-used for each instance of the given batch access.

4.3.4 GPU Memory Management

GeePS keeps the GPU-pinned parameter cache, GPU-pinned local data, and access buffer pool in GPU memory. They will be all the GPU memory allocated in a machine if the application stores all its input data and intermediate states in GeePS and uses the GeePS-managed buffers. GeePS will pin as much parameter data and local data in GPU memory as possible. But, if the GPU memory is not large enough, GeePS will keep some of the data in CPU memory (the CPU part of the parameter cache and/or CPU part of the local data).

In the extreme case, GeePS can keep all parameter data and local data in the CPU memory. But, it will still need the buffer pool to be in the GPU memory, and the buffer pool needs to be large enough to store all the *actively used data* even at peak usage. We refer to this peak memory usage as *peak size*. In order to perform the GPU/CPU data movement in the background, GeePS does double buffering by making the buffer pool twice as large as the peak size.

Data Placement Policy

We will now describe our policy for choosing which data to pin in GPU memory. In our implementation, any local data that is pinned in GPU memory does not need to use any access buffer space. The allocator thread will just give the pointer to the pinned GPU local data to the application, without copying the data. For the parameter data, even though it is pinned in GPU memory, the allocator thread still needs to copy it from the parameter cache to an access buffer, because the parameter cache could be modified by the background communication thread (the puller thread) while the application is doing computation. As a result, pinning local data in GPU memory gives us more benefit than pinning parameter cache. Moreover, if we pin the local data that is used at the peak usage, we can reduce the peak access buffer usage, because it doesn't need the buffer, allowing us to reserve less memory for the access buffer.

Algorithm 4 illustrates our GPU/CPU data placement policy, and it is run at the setup stage, after the access information is gathered from the application. The algorithm chooses the entries to pin in GPU memory based on the gathered access information and a given GPU memory budget. While keeping the access buffer pool twice the peak size for double buffering, our policy will first try to pin the local data that is used at the peak in GPU memory, in order to reduce the peak size and thus the size of the buffer pool. Then, it will try to use the available capacity to pin

Algorithm 4: GPU/CPU data placement policy

Input: $paramData, localData \leftarrow$ parameter data and local data accessed at each layer
Input: $totalMem \leftarrow$ the amount of GPU memory to use
 # Start with everything in CPU memory
$cpuData \leftarrow allLocalData \cup allParamData$
$gpuData \leftarrow \emptyset$
 # Set access buffer twice the peak size for double buffering
$peakSize \leftarrow$ peak data usage, excluding GPU local data
$bufferSize \leftarrow 2 \times peakSize$
$availMem \leftarrow totalMem - bufferSize$
 # First pin local data used at peak
while *more local data in* $cpuData$ **do**
    $localData \leftarrow$ CPU local data used at peak
    $\Delta(peakSize) \leftarrow peakSize$ change if move $localData$ to GPU
    $\Delta(memSize) \leftarrow size(localData) + 2 \times \Delta(peakSize)$
    if $availMem < \Delta(memSize)$ **then**
        break
    Move $localData$ from $cpuData$ to $gpuData$
    $availMem \leftarrow availMem - \Delta(memSize)$
 # Pin more local data using the available memory
for *each* $localData$ *in* $cpuData$ **do**
    if $availMem \geq size(localData)$ **then**
        Move $localData$ from $cpuData$ to $gpuData$
        $availMem \leftarrow availMem - size(localData)$
 # Pin parameter data using the available memory
for *each* $paramData$ *in* $cpuData$ **do**
    if $availMem \geq size(paramData)$ **then**
        Move $paramData$ from $cpuData$ to $gpuData$
        $availMem \leftarrow availMem - size(paramData)$
 # Use the reset available memory as access buffer
Increase $bufferSize$ by $availMem$

more local data and parameter cache data in GPU memory. Finally, it will add the rest available GPU memory to the access buffer.

Avoiding Unnecessary Data Movement

When the application accesses the local data that is stored in CPU memory, usually the allocator thread will need to allocate a piece of GPU memory from the buffer pool and copy the data from CPU memory to GPU memory. However, sometimes this data movement is unnecessary, when the application just needs an uninitialized piece of GPU memory. For example, when we train a deep neural network, the input data and intermediate data will be overwritten by a new mini-batch, so the old values from the last mini-batch can be safely thrown away. To avoid the unnecessary data movement, we allow the application to specify a `no-fetch` flag when calling `LOCAL-ACCESS`, and it tells GeePS to just allocate a piece of GPU memory, without fetching the data from CPU memory. Similarly, when the application calls `POST-LOCAL-ACCESS`, it can use a `no-save` flag to tell GeePS to just free the GPU memory, without saving the data to CPU memory.

4.4 Evaluation

This section evaluates GeePS's using three image classification models and a video classification model executed in the original and modified Caffe application. The evaluation confirms three main findings:

- GeePS provides effective data-parallel scaling of training throughput and training convergence rate, at least up to 16 machines with GPUs.
- GeePS's efficiency is much higher, for GPU-based training, than a traditional CPU-based parameter server and also much higher than parallel CPU-based training performance reported in the literature.
- GeePS's dynamic management of GPU memory allows data-parallel GPU-based training on models that are much larger than used in state-of-the-art deep learning for image classification and video classification.

A specific non-goal of our evaluation is comparing the classification accuracies of the different models. Our focus is on enabling faster training of whichever model is being used, which is why we measure performance for several.

4.4.1 Experimental Setup

Application Setup

We use Caffe [97], the open-source singe-GPU convolutional neural network application discussed earlier.[1] Our experiments use unmodified Caffe to represent the optimized single-GPU

[1]For the image classification application, we used the version of Caffe from `https://github.com/BVLC/caffe` as of June 19, 2015. Since their master branch version does not support RNN, for the video classification

Algorithm 5: GeePS-Caffe training with virtual iteration

```
L ← number of layers in the network
paramDataKeys ← decide row keys for param data
localDataKeys ← decide row keys for local data
# Report access information with virtual iteration
TrainMiniBatch (TrainMiniBatch) (null, virtual = yes)
# Real training iterations
while not done do
    TrainMiniBatch (null, virtual = false)

Function TrainMiniBatch (trainData, virtual):
    # Forward pass
    for i = 0 ∼ (L − 1) do
        paramDataPtr ← geeps.Read(paramDataKeys_i, virtual)
        localDataPtr ← geeps.LocalAccess(localDataKeys_i, virtual)
        if not virtual then
            Setup layer_i with data pointers
            Forward computation of layer_i
        geeps.PostRead(paramDataPtr)
        geeps.PostLocalAccess(localDataPtr)
    # Backward pass
    for i = (L − 1) ∼ 0 do
        paramDataPtr ← geeps.Read(paramDataKeys_i, virtual)
        paramUpdatePtr ← geeps.PreWrite(paramDataKeys_i, virtual)
        localDataPtr ← geeps.LocalAccess(localDataKeys_i, virtual)
        if not virtual then
            Setup layer_i with data pointers
            Backward computation of layer_i
        geeps.PostRead(paramDataPtr)
        geeps.Write(paramUpdatePtr)
        geeps.PostLocalAccess(localDataPtr)
        geeps.TableClock(table = i)
```

case and a minimally modified instance (**GeePS-Caffe**) that uses GeePS for data-parallel execution. As shown in Algorithm 5, GeePS-Caffe uses GeePS to manage all its parameter data and local data (including input data and intermediate data). The parameter data of each layer is stored as rows of a distinct GeePS tables, allowing GeePS to propagate the each layer's updates during the computations of other layers, as suggested by the WFBP scheduling in Poseidon [241]. In order to enable the specializations from access information hints, GeePS-Caffe performs a virtual iteration to report its access information to GeePS, before starting the real computations. The GeePS calls with a `virtual` flag are just recorded but without any further actions taken.

Cluster Setup

Each machine in our cluster has one NVIDIA Tesla K20C GPU, which has 5 GB of GPU device memory. In addition to the GPU, each machine has four 2-die 2.1 GHz 16-core AMD® Opteron 6272 packages and 128 GB of RAM. Each machine is installed with 64-bit Ubuntu 14.04, CUDA toolkit 7.5, and cuDNN v2. The machines are inter-connected via a 40 Gbps Ethernet interface (12 Gbps measured via iperf), and Caffe reads the input training data from remote file servers via a separate 1 Gbps Ethernet interface.

Image Classification Datasets and Models

The experiments use two datasets for image classification. The first one is the ImageNet22K dataset [49], which contains 14 million images labeled to 22,000 classes. Because of the computation work required to train such a large dataset, CPU-based systems running on this dataset typically need a hundred or more machines and spend over a week to reach convergence [32]. We use half of the images (7 million images) as the training set and the other half as the testing set, which is the same setup as described by Chilimbi et al. [32]. For the ImageNet22K dataset, we use a similar model to the one used to evaluate ProjectAdam [32], which we refer to as the *AdamLike* model. We were not able to obtain the exact model that ProjectAdam uses, so we emulated it based on the descriptions in the paper. Our emulated model has the same number and types of layers and connections, and we believe our training performance evaluations are representative even if the resulting model accuracy may not be. The AdamLike model has five convolutional layers and three fully connected layers. It contains 2.4 billion connections for each image, and the model parameters are 470 MB in size.

The second dataset we used is ILSVRC12 [49]. It is a subset of the ImageNet22K dataset, with 1.3 million images labeled to 1000 classes. For this dataset, we use the *GoogLeNet*, a recent inception model from Google. The network has about 100 layers, and 22 of them have model parameters. Though the number of layers is large, the model parameters are only 57 MB in size, because they use mostly convolutional layers.

application, we used the version from https://github.com/LisaAnne/lisa-caffe-public/tree/lstm_video_deploy as of Nov 9, 2015.

Video Classification Datasets and Models

We use the UCF-101 dataset [195] for our video classification experiments. UCF-101 has about 8,000 training videos and 4,000 testing videos categorized into 101 human action classes. We use a recurrent neural network model for video classification, following the approach described by Donahue et al. [52]. We use the GoogLeNet network as the CNN layers and stack LSTM layers with 256 hidden units on top of them. The weights of the GoogLeNet layers have been pre-trained with the single frames of the training videos. Following the same approach as is described by Donahue et al. [52], we extract the video frames at a rate of 30 frames per second and train the model with randomly selected video clips of 32 frames each.

Training Algorithm Setup

Both the unmodified and the GeePS-hosted Caffe train the models using the SGD algorithm with a momentum of 0.9. Unless otherwise specified, we use the configurations listed in Table 4.1. Our experiments in Section 4.4.3 evaluate performance with different mini-batch sizes. For AdamLike and GoogLeNet network, we used the same learning rate for both single-machine training and distributed training, because we empirically found that this learning rate is the best setting for both. For the RNN model, we used a different learning rate for single-machine training, because it leads to faster convergence.

Model	Mini-batch size (per machine)	Learning rate
AdamLike	200 images	0.0036, div by 10 every 3 epochs
GoogLeNet	32 images	0.0036, div by 10 every 150 epochs
RNN	1 video, 32 frames each	0.0000125 for 8 machines, 0.0001 for single-machine, div by 10 every 20 epochs

Table 4.1: Model training configurations.

GeePS Setup

We run one application worker (Caffe linked with one GeePS instance) on each machine. Unless otherwise specified, we let GeePS keep the parameter cache and local data in GPU memory for each experiment, since it all fits for all of the models used; Section 4.4.3 evaluates performance when keeping part of the data in CPU memory, including for a very large model scenario. Unless otherwise specified, BSP mode is used.

4.4.2 Scaling Deep Learning with GeePS

This section evaluates how well GeePS supports data-parallel scaling of GPU-based training on both the image classification application and the video classification application. We compare **GeePS** with three classes of systems:

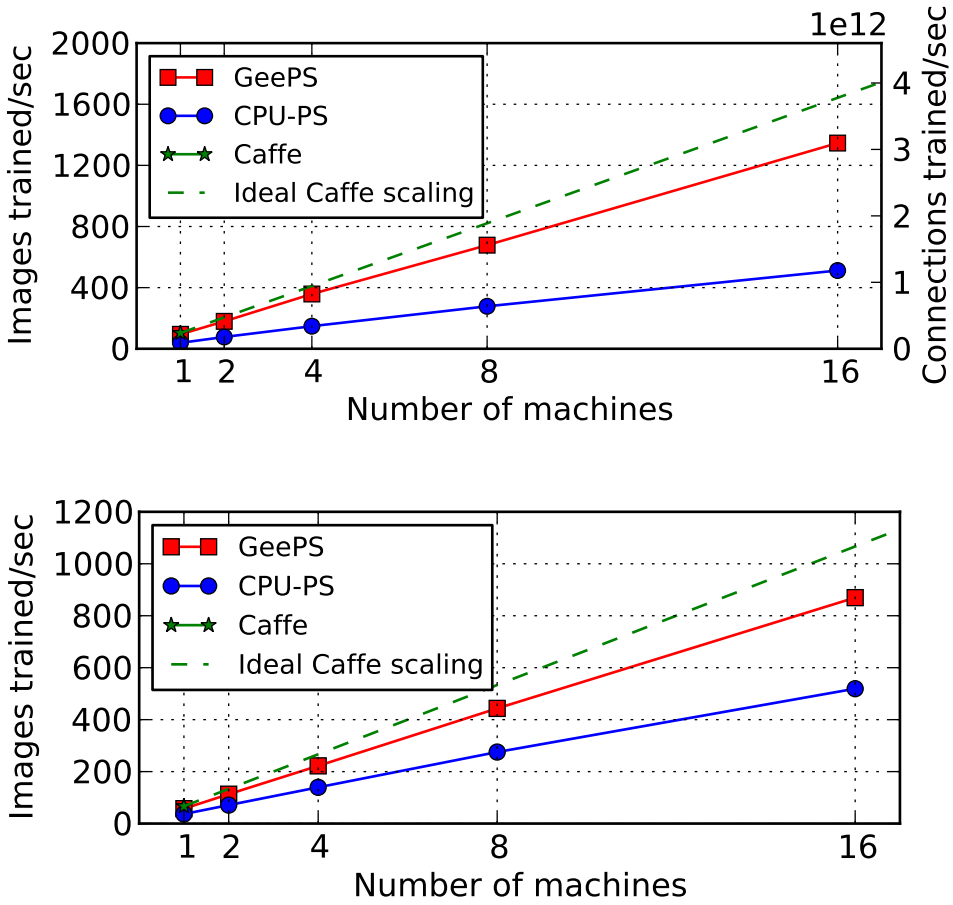

Figure 4.7: Image classification throughput scalability for (upper) AdamLike model on ImageNet22K dataset and (bottem) GoogLeNet model on ILSVRC12 dataset. Both GeePS and CPU-PS run in the fully synchronous mode.

- **Caffe**: *Single-GPU optimized training* with the original unmodified Caffe system, which represents training optimized for execution on a single GPU.
- **CPU-PS**: multiple instances of the modified Caffe running on GPU workders linked via IterStore [38], a state-of-the-art CPU-based parameter server.
- **CPU workers with CPU-based parameter server**: reported performance numbers from recent literature are used to put the GPU-based performance into context relative to state-of-the-art CPU-based deep learning.

Image Classification

Figure 4.7 shows model training throughput of the image classification application, in terms of both number of images trained per second and number of neural network connections trained

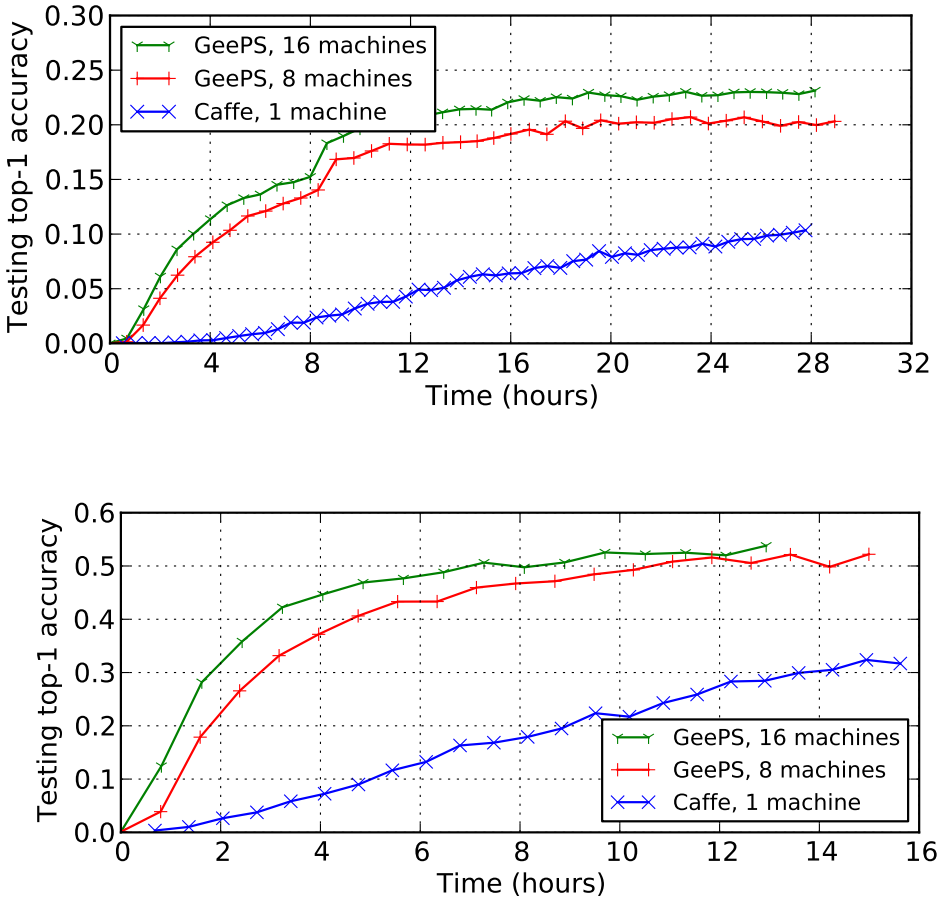

Figure 4.8: Image classification top-1 accuracies for (upper) AdamLike model on ImageNet22K dataset and (bottom) GoogLeNet model on ILSVRC12 dataset.

per second. Note that there is a linear relationship between those two metrics. GeePS scales almost linearly when we add more machines. Compared to the single-machine optimized Caffe, GeePS achieves 13× speedups on both GoogLeNet and AdamLike model using 16 machines. Compared to CPU-PS, GeePS achieves over 2× more throughput.

Chilimbi et al. [32] report that ProjectAdam can train 570 billion connections per second on the ImageNet22K dataset when using 108 machines (88 CPU-based worker machines with 20 parameter server machines). Figure 4.7(upper) shows the GeePS achieves higher throughput using only 4 GPU machines, because of efficient data-parallel execution on GPUs.

Figure 4.8 shows the image classification top-1 testing accuracies of our trained models. To evaluate convergence speed, we compare the amount of time required to reach a given level of accuracy, which is a combination of image training throughput and model convergence per trained image. For the AdamLike model on the ImageNet22K dataset, Caffe needs 26.9 hours to reach 10% accuracy, while GeePS needs only 4.6 hours using 8 machines (6× speedup) or 3.3 hours using 16 machines (8× speedup). For the GoogLeNet model on the ILSVRC12 dataset, Caffe

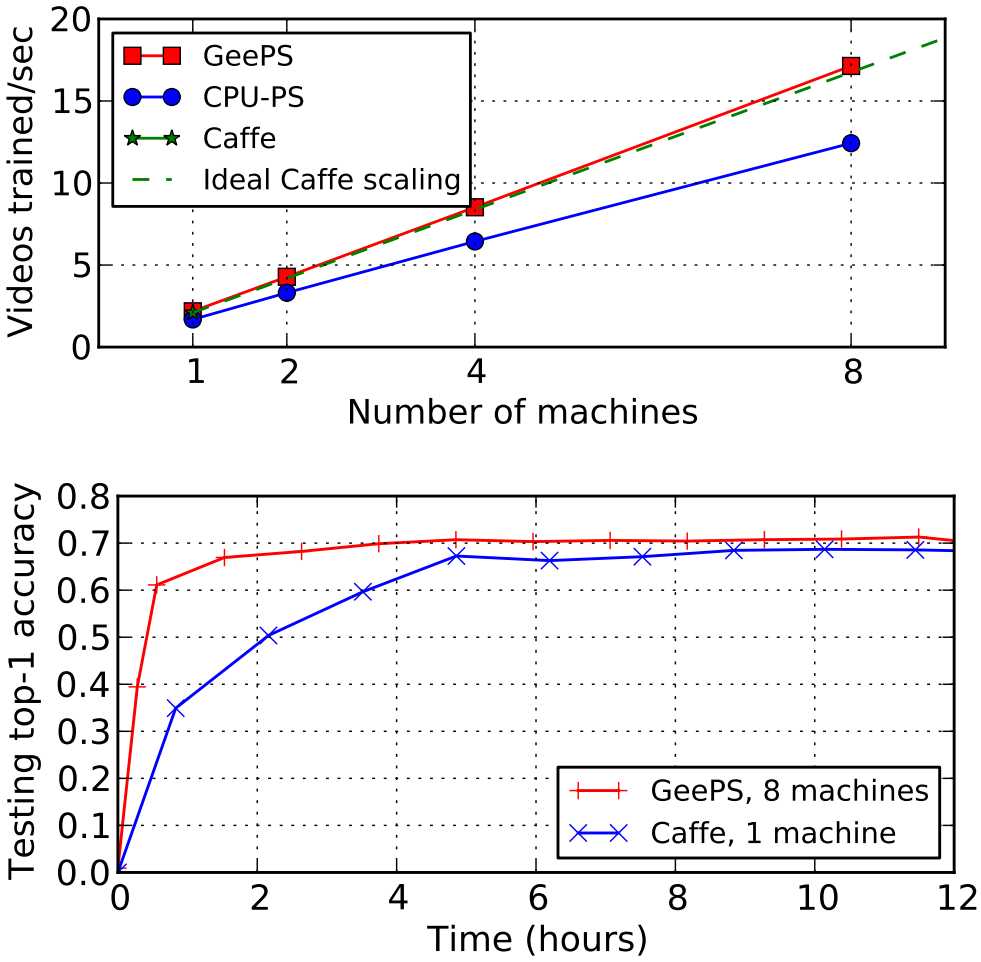

Figure 4.9: Video classification task: (upper) training throughput; (bottom) top-1 accuracies.

needs 13.7 hours to reach 30% accuracy, while GeePS needs only 2.8 hours using 8 machines (5× speedup) or 1.8 hours using 16 machines (8× speedup). The model training time speedups compared to the single-GPU optimized Caffe are lower than the image training throughput speedups, as expected, because each machine determines gradients independently. Even using BSP, more training is needed than with a single worker to make the model converge. But, the speedups are still substantial.

For the AdamLike model on the ImageNet22K dataset, Chilimbi et al. [32] report that ProjectAdam needs one day to reach 13.6% accuracy using 58 machines (48 CPU-based worker machines with 10 parameter server machines). GeePS needs only 6 hours to reach the same accuracy using 16 machines (about 4× speedup). To reach 13.6% accuracy, the DistBelief system trained (a different model) using 2,000 machines for a week [46].

Because both GeePS and CPU-PS run in the BSP mode using the same number of machines, the accuracy improvement speedups of GeePS over CPU-PS are the same as the throughput speedups, so we leave them out of the graphs.

Video Classification

Figure 4.9(upper) shows the training throughput of the video classification application. GeePS scales linearly from Caffe (8× throughput with 8 machines). Figure 4.9(bottom) shows the top-1 testing accuracies. To reach 60% accuracy, Caffe needs 3.6 hours, while GeePS needs 0.5 hours (7× speedup); to reach 68% accuracy, Caffe needs 8.4 hours, while GeePS needs 2.4 hours (3.5× speedup).

4.4.3 Dealing with Limited GPU Memory

An oft-mentioned concern with data-parallel deep learning on GPUs is that it can only be used when the entire model, as well as all intermediate state and the input mini-batch, fit in GPU memory. GeePS eliminates this limitation with its support for managing GPU memory and using it to buffer data from the much larger CPU memory. Although all of the models we experiment with (and most state-of-the-art models) fit in our GPUs' 5 GB memories, we demonstrate the efficacy of GeePS's mechanisms in two ways: by using only a fraction of the GPU memory for the largest case (AdamLike) and by experimenting with a much larger synthetic model. We also show that GeePS's memory management support allows us to do video classification on longer videos.

Artificially Shrinking Available GPU Memory

With a mini-batch of 200 images per machine, training the AdamLike model on the ImageNet22K dataset requires only 3.67 GB, with 123 MB for input data, 2.6 GB for intermediate states, and 474 MB each for parameter data and computed parameter updates. Note that the sizes of the parameter data and parameter updates are determined by the model, while the input data and intermediate states grow linearly with the mini-batch size. For best throughput, GeePS also requires use of an access buffer that is large enough to keep the actively used parameter data and parameter updates at the peak usage, which is 528 MB minimal and 1.06 GB for double buffering (the default) to maximize overlapping of data movement with computation. So, in order to keep everything in GPU memory, the GeePS-based training needs 4.73 GB of GPU memory.

Recall, however, that GeePS can manage GPU memory usage such that only the data needed for the layers being processed at a given point need to be in GPU memory. Figure 4.10 shows the per-layer memory usage for the AdamLike model training, showing that it is consistently much smaller than the total memory usage. The left Y axis shows the absolute size (in GB) for a given layer, and the right Y axis shows the fraction of the absolute size over the total size of 4.73 GB. Each bar is partitioned into the sizes of input data, intermediate states, parameter data, and parameter updates for the given layer. Most layers have little or no parameter data, and most of the memory is consumed by the intermediate states for neuron activations and error terms. The layer that consumes the most memory uses about 17% of the total memory usage, meaning that about 35% of the 4.73 GB is needed for full double buffering.

Figure 4.11 shows data-parallel training throughput using 8 machines, when we restrict GeePS to using different amounts of GPU memory to emulate GPUs with smaller memories. When there is not enough GPU memory to fit everything, GeePS must swap data to CPU memory. For the case of 200 images per batch, when we swap all data in CPU memory, we need only

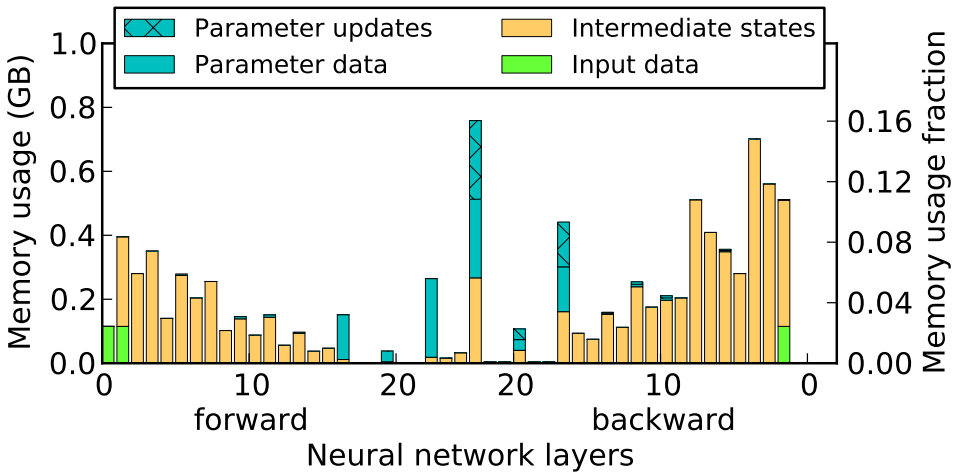

Figure 4.10: Per-layer memory usage of AdamLike model on ImageNet22K dataset.

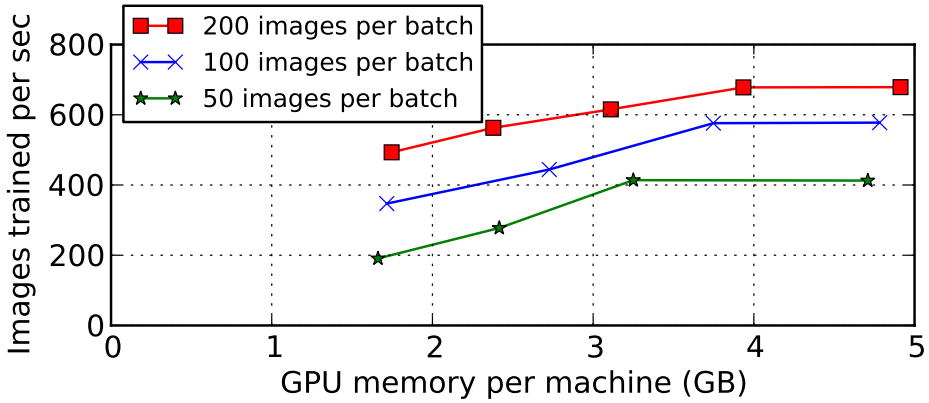

Figure 4.11: Throughput of AdamLike model on ImageNet22K dataset with different GPU memory budgets.

35% of the GPU memory compared to keeping all data in GPU memory, but we are still able to get 73% of the throughput.

When the GPU memory limits the scale, people are often forced to use smaller mini-batch sizes to let everything fit in GPU memory. Our results in Figure 4.11 also shows that using our memory management mechanism is more efficient than shrinking the mini-batch size. For the three mini-batch sizes compared, we keep the inter-machine communication the same by doing multiple mini-batches per clock as needed (e.g., four 50-image mini-batches per clock). For the case of 100 images per batch and 50 images per batch, 3.7 GB and 3.3 GB respectively are needed to keep everything in GPU memory (including access buffers for double buffering), and the rest of the available GPU memory is used to extend the access buffers. While smaller mini-batches reduce the total memory requirement, they perform significantly less well for two primary reasons: (1) the GPU computation is more efficient with a larger mini-batch size, and (2) the time for reading and updating the parameter data locally, which does not shrink with

mini-batch size, is amortized over more data.

Training a Very Large Neural Network

To evaluate performance for much larger neural networks, we create and train huge synthetic models. Each such neural network contains only fully connected layers with no weight sharing, so there is one model parameter (weight) for every connection. The model parameters of each layer is about 373 MB. We create multiple such layers and measure the throughput (in terms of # connections per second) of training different sized networks. Figure 4.12 shows the results. For all sizes tested, up to a 20 GB model (56 layers) that requires over 70 GB total (including local data), GeePS is able to train the neural network without excessive overhead. The overall result is that GeePS's GPU memory management mechanisms allows data-parallel training of very large neural networks, bounded by the largest layer rather than the overall model size.

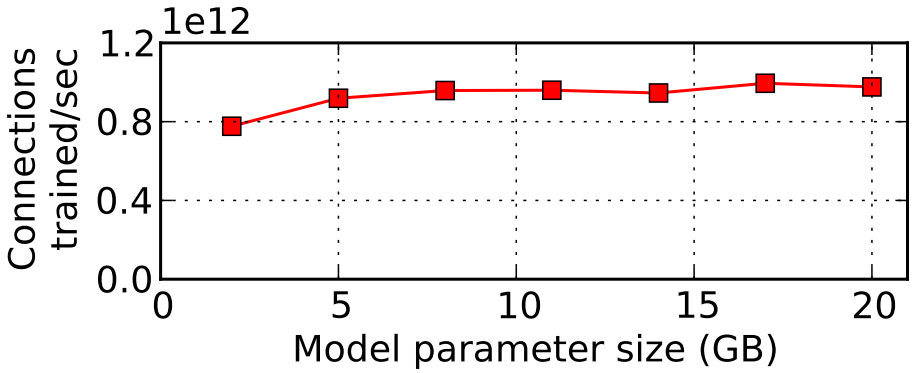

Figure 4.12: Training throughput on very large models. Note that the number of connections increases linearly with model size, so the per-image training time grows with model size because the per-connection training time stays relatively constant.

Video Classification on Longer Videos

The video classification application requires complete sequences of image frames to be in the same mini-batch, so the GPU memory size will either limit the maximum number of frames per video or force the model to be split across multiple machines, incurring extra complexity and network communication overhead. Using unmodified Caffe, for example, our RNN can support a maximum video length of 48 frames.[2] Because the videos are often sampled at a rate of 30 frames per second, a 48-frame video is less than 2 seconds in length. Ng et al. [238] find that using more frames in a video improves the classification accuracy. In order to use a video length of 120 frames, Ng et al. use a model parallel approach to split the model across four machines, which incurs extra network communication overhead. By contrast, with the memory

[2]Here, we are just considering the memory consumption of training. If we further consider the memory consumption for testing, the supported maximum video length is even lower.

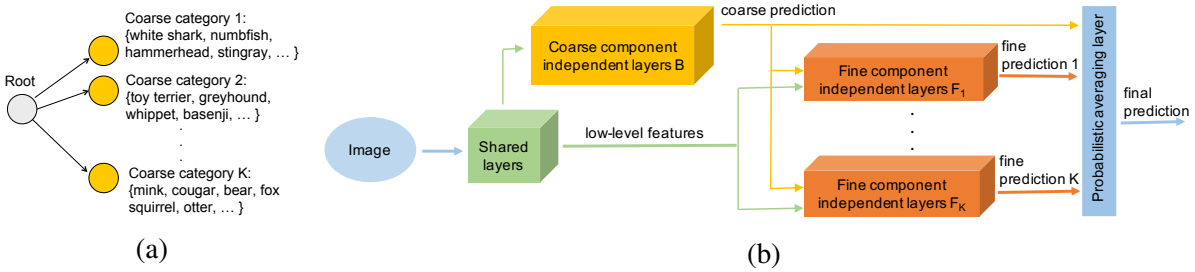

Figure 4.13: (a) A two-level category hierarchy taken from ImageNet-1000 dataset. (b) The hierarchical deep convolutional neural network (HD-CNN) architecture, where the orange and yellow blocks are layers introduced in addition to the backbone model.

management support of GeePS, we are able to train videos with up to 192 frames, with pure data parallelism.

So far, we have evaluated the efficacy of MemSwap and GeePS on standard models and tasks. Next, we show that the optimizations presented in Chapter 3 and Chapter 4 open the design space of models with extremely more parameters and memory footprints, which in turn boosts the task performance.

4.5 Application: Hierarchical Deep Convolutional Neural Networks (HD-CNN)

In image classification, visual separability between different object categories is highly uneven, and some categories are more difficult to distinguish than others. Such difficult categories demand more dedicated classifiers. However, existing deep convolutional neural networks (CNNs) are trained as flat N-way classifiers, and few efforts have been made to leverage the hierarchical structure of categories. A straightforward idea to enhance existing models is to embed multi-level category hierarchy into CNNs, and to direct the CNN to learn higher-level features to distinguish between coarse but easier categories (e.g., cars vs. plants) first, and then more subtle and dedicate features to classify finer-grained but more difficult categories (e.g., eagle vs. hawk) [50, 187, 233].

Implementing this idea (which is also called *HD-CNN* [233] shown in Figure 4.13) is however hindered by a few system challenges. Introducing multi-level category hierarchies significantly deepens the backbone CNN with more intermediate layers and coarse category classifiers. Each added layer would generate more intermediate states that need to be stored for backpropagation, which increases peak memory usage and easily exceeds the GPU memory limit; every intermediate classifier for coarse categories brings in at least one fully-connected softmax with dense matrix-shaped parameters, significantly burdening the network communication in distributed training.

Scaling Up HD-CNN

We show that scaling up HD-CNN is a perfect examination of the proposed approaches. Using VGG-19 [191] as the backbone, we incorporate a two-layered coarse-to-fine category hierarchy on ImageNet-1000 [49], resulting in 4x more parameters. We optimize the training of such a neural network (shown in Figure 4.13) using the strategies presented in the previous two chapters.

To reduce the peak GPU memory usage, we let the system swap inactive intermediate states back to host CPU memory based on the model structure, and reserve space on GPU memory for the upcoming computation. To communicate the parameters of HD-CNN, we customize the communication scheme using the adaptive communication (Section 3.3) based on the layer type and the number of neurons in the layer, effectively minimizing the size of communicated messages, caused by adding more fully-connected layers.

Method	top-1, top-5
GoogLeNet,multi-crop [200]	N/A,7.9
VGG-19-layer, dense [191]	24.8,7.5
VGG-16-layer+VGG-19-layer, dense	24.0,7.1
Base:ImageNet-VGG-16-layer, dense	24.79,7.50
HD-CNN w/ 2-layer hierarchy,dense	**23.69,6.76**

Table 4.2: Errors on ImageNet validation set.

We successfully scale out this model onto a cluster with 32 GPU nodes, each with 12Gb standard GPU memory. Note that this HD-CNN has *nearly 300M parameters, and would need 70Gb GPU memory to train otherwise*. We reported state-of-the-art image classification performance on ImageNet back in 2015 [233] in Table 4.2.

4.6 Additional Related Work

In addition to the related work introduced in Section 2.4.1, we review a few additional work on distributing DL onto distributed GPUs.

Deep Image [224] is a custom-built supercomputer for deep learning via GPUs. The GPUs used have large memory capacity (12 GB), and their image classification application fit within the memory of GPUs they used, allowing use of data-parallel execution. They also support for model-parallel execution, with ideas borrowed from Krizhevsky et al. [116]. They partition the model on fully connected layers, but not on convolutional layers. The machines are interconnected by Infiniband with GPUDirect RDMA, so no CPU involvement is required, and they do not use the CPU cores or CPU memory to enhance scalability like GeePS does. Deep Image exploits its low latency GPU-direct networking for specialized parameter state exchanges rather than using a general parameter server architecture like GeePS.

Chapter 5

Consistency Model

In this chapter, we turn our attention to the aspect of consistency model in distributed parallel machine learning.

Most distributed machine learning (ML) systems store a copy of the model parameters locally on each machine to minimize network communication. In practice, in order to reduce synchronization waiting time, these copies of the model are not necessarily updated in lock-step, and can become *stale*. Despite much development in large-scale ML, the effect of *staleness* on the learning efficiency is inconclusive, mainly because it is challenging to control or monitor the staleness in complex distributed environments.

This chapter studies the convergence behaviors of a wide array of ML models and algorithms under delayed updates, by resorting to pure empirical simulations. Our extensive experiments reveal the rich diversity of the effects of staleness on the convergence of ML algorithms and offer insights into seemingly contradictory reports in the literature.

These conclusions confirm that in terms of consistency, different models, or even building blocks of models, have different degrees of robustness against staleness, which offers design space for future systems to adapt the consistency model (which were usually implemented as fixed in existing ML systems) to best trade-off between system throughput and statistical efficiency. Unlike previous chapters, however, we defer a system implementation with adaptive consistency to future studies.

5.1 Distributed ML: Synchronous or Asynchronous?

With the advent of big data and complex models, there is a growing body of works on scaling machine learning under synchronous and non-synchronous distributed execution [46, 68, 127]. These works, however, point to seemingly contradictory conclusions on whether non-synchronous execution outperforms synchronous counterparts in terms of absolute convergence, which is measured by the wall clock time to reach the desired model quality. For deep neural networks, Chilimbi et al. [32] and Dean et al. [46] show that fully asynchronous systems achieve high scalability and model quality, but others argue that synchronous training converges faster [24, 40]. The disagreement goes beyond deep learning models: a body of work [83, 173, 247, 250] have empirically and theoretically show that many algorithms scale effectively un-

der non-synchronous settings, but another line of work [75, 142, 150] demonstrate significant penalties from asynchrony.

The crux of the disagreement lies in the trade-off between two factors contributing to the absolute convergence: *statistical efficiency* and *system throughput*. Statistical efficiency measures convergence per algorithmic step (e.g., a mini-batch), while system throughput captures the performance of the underlying implementation and hardware. Non-synchronous execution can improve system throughput due to lower synchronization overheads, which is well understood [24, 32, 38, 83]. However, by allowing various workers to use *stale* versions of the model that do not always reflect the latest updates, non-synchronous systems can exhibit lower statistical efficiency [24, 40]. How statistical efficiency and system throughput trade off in distributed systems is far from clear.

The difficulties in understanding the trade-off arise because statistical efficiency and system throughput are coupled during execution in distributed environments. Non-synchronous executions are in general non-deterministic, which can be difficult to profile. Furthermore, large-scale experiments are sensitive to the underlying hardware and software artifacts, which confounds the comparison between studies. Even when they are controlled, innocuous change in the system configurations such as adding more machines or sharing resources with other workloads can inadvertently alter the underlying staleness levels experienced by ML algorithms, masking the true effects of staleness.

Understanding the impact of staleness on ML convergence independently from the underlying distributed systems is a crucial step towards decoupling statistical efficiency from the system complexity. The gleaned insights can also guide distributed ML system development, potentially using different synchronization for different problems. In particular, we are interested in the following aspects: Do ML algorithms converge under staleness? To what extent does staleness impact the convergence?

By resorting to simulation study, we side step the challenges faced in distributed execution. We study the impact of staleness on a diverse set of models: Convolutional Neural Networks (CNNs), recurrent neural networks (RNNs), Deep Neural Networks (DNNs), multi-class Logistic Regression (MLR), Matrix Factorization (MF), Latent Dirichlet Allocation (LDA), and Variational Autoencoders (VAEs). They are addressed by 7 algorithms, spanning across optimization, sampling, and blackbox variational inference. Our findings suggest that while some algorithms are more robust to staleness, no ML method is immune to the negative impact of staleness. We find that all investigated algorithms reach the target model quality under moderate levels of staleness, but the convergence can progress very slowly or fail under high staleness levels. The effects of staleness are also problem dependent. For CNNs, DNNs, and RNNs, the staleness slows down deeper models more than shallower counterparts. For MLR, a convex objective, staleness has minimal effect. Different algorithms respond to staleness very differently. For example, high staleness levels incur more statistical penalty for Momentum methods than stochastic gradient descent (SGD) and Adagrad [53]. Separately, Gibbs sampling for LDA is highly resistant to staleness up to a certain level, beyond which it does not converge to a fixed point. Overall, it appears that staleness is a key governing parameter of ML convergence.

Model	Algorithms	Key Parameters	Dataset
CNN RNN	SGD	η	CIFAR10 (CNN) Penn Treebank (RNN)
	Momentum SGD	η, momentum=0.9	
	Adam	$\eta, \beta_1 = 0.9, \beta_2 = 0.999$	
	Adagrad	η	
	RMSProp	η, decay=0.9, momentum=0	
DNN/MLR	SGD	$\eta = 0.01$	MNIST
	Adam	$\eta = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$	
LDA	Gibbs Sampling	$\alpha = 0.1, \beta = 0.1$	20 NewsGroup
MF	SGD	$\eta = 0.005$, rank=5, $\lambda = 0.0001$	MovieLens1M
VAE	Blackbox VI (SGD, Adam)	Optimization parameters same as MLR/DNN	MNIST

Table 5.1: Overview of the models, algorithms [53, 71, 108, 117, 179], and dataset in our study. η denotes learning rate, which, if not specified, are tuned empirically for each algorithm and staleness level (over $\eta = 0.001, 0.01, 0.1$), β_1, β_2 are optimization hyperparameters (using common default values). α, β in LDA are Dirichlet priors for document topic and word topic random variables, respectively.

5.2 Method and Setup

We study six ML models and focus on algorithms that lend itself to data parallelism, which is a primary approach for distributed ML. Our algorithms span optimization, sampling, and black box variational inference. Table 5.1 summarizes the studied models and algorithms.

5.2.1 Simulation Model

Each update generated by worker p needs to be propagated to both worker p's model cache and other worker's model cache. We apply a uniformly random delay model to these updates that are in transit. Specifically, let u_p^t be the update generated at iteration t by worker p. For each worker p' (including p itself), our delay model applies a delay $r_{p,p'}^t \sim \text{Categorical}(0, 1, .., s)$, where s is the maximum delay and Categorical() is the categorical distribution placing equal weights on each integer . Under this delay model, update u_p^t shall arrive at worker p' at the start of iteration $t + 1 + r_{p,p'}^t$. The average delay under this model is $\frac{1}{2}s + 1$. Notice that for one worker with $s = 0$ we reduce to the sequential setting. Since model caches on each worker are symmetric, we use the first worker's model to evaluate the model quality. Finally, we are most interested in measuring convergence against the logical time, and wall clock time is in general immaterial as the simulation on a single machine is not optimized for performance.

5.2.2 Models and Algorithms

Convolutional Neural Networks (CNNs)

CNNs have been a strong focus of large-scale training, both under synchronous [40, 68, 245] and non-synchronous [24, 46, 75, 134] training. We consider residual networks with $6n + 2$ weight layers [80]. The networks consist of 3 groups of n residual blocks, with 16, 32, and 64 feature maps in each group, respectively, followed by a global pooling layer and a softmax layer. The residual blocks have the same construction as in [80]. We measure the model quality using test accuracy. For simplicity, we omit data augmentation in our experiments.

Deep Neural Networks (DNNs)

DNNs are neural networks composed of fully connected layers. Our DNNs have 1 to 6 hidden layers, with 256 neurons in each layer, followed by a softmax layer. We use rectified linear units (ReLU) for nonlinearity after each hidden layer [155].

Multi-class Logistic Regression (MLR)

MLR is the special case of DNN with 0 hidden layers. We measure the model quality using test accuracy.

Matrix Factorization (MF)

MF is commonly used in recommender systems and have been implemented at scale [38, 83, 105, 118, 129, 138]. Let $D \in \mathbb{R}^{M \times N}$ be a partially filled matrix, MF factorizes D into two factor matrices $L \in \mathbb{R}^{M \times r}$ and $R \in \mathbb{R}^{N \times r}$ ($r \ll \min(M, N)$ is the user-defined rank). The ℓ_2-penalized optimization problem is

$$\min_{L,R} \frac{1}{|D_{obs}|} \left\{ \sum_{(i,j) \in D_{obs}} ||D_{ij} - \sum_{k=1}^{K} L_{ik} R_{kj}||^2 + \lambda(||L||_F^2 + ||R||_F^2) \right\},$$

where $|| \cdot ||_F$ is the Frobenius norm and λ is the regularization parameter. We partition observations D to workers while treating L, R as shared model parameters. We optimize MF via SGD, and measure model quality by training loss defined by the objective function above.

Latent Dirichlet Allocation (LDA)

LDA is an unsupervised method to uncover hidden semantics ("topics") from a group of documents, each represented as a bag of tokens. LDA has been scaled under non-synchronous execution [4, 138, 237] with great success. In LDA each token w_{ij} (j-th token in the i-th document) is assigned with a latent topic z_{ij} from totally K topics. We use Gibbs sampling to infer the topic assignments z_{ij}. The Gibbs sampling step involves three sets of parameters, known as sufficient statistics: (1) document-topic vector $\theta_i \in \mathbb{R}^K$ where θ_{ik} the number of topic assignments within document i to topic $k = 1...K$; (2) word-topic vector $\phi_w \in \mathbb{R}^K$ where ϕ_{wk} is the number of topic

assignments to topic $k = 1, ..., K$ for word (vocabulary) w across all documents; (3) $\tilde{\phi} \in \mathbb{R}^K$ where $\tilde{\phi}_k = \sum_{w=1}^{W} \phi_{wk}$ is the number of tokens in the corpus assigned to topic k. The corpus (w_{ij}, z_{ij}) is partitioned to workers, while ϕ_w and $\tilde{\phi}$ are shared model parameters. We measure the model quality using log likelihood.

Variational Autoencoder (VAE)

VAE [109] is commonly optimized by black box variational inference, which can be considered as a hybrid of optimization and sampling methods. The inputs to VAE training include two sources of stochasticity: the data sampling $\boldsymbol{x}$ and samples of random variable ϵ. We measure the model quality by test loss. We use DNNs with 1∼3 layers as the encoders and decoders in VAE, in which each layer has 256 units furnished with rectified linear function for non-linearity. The model quality is measured by the training objective value, assuming continuous input $\boldsymbol{x}$ and isotropic Gaussian prior $p(\boldsymbol{z}) \sim \mathcal{N}(0, \boldsymbol{I})$.

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are widely used in recent natural language processing tasks. We consider long short-term memory (LSTM) [84] applied to the language modeling task, using a subset of Penn Treebank dataset (PTB) [140] containing 5855 words. The dataset is pre-processed by standard de-capitalization and tokenization. We evaluate the impact of staleness for LSTM with 1 to 4 layers, with 256 neurons in each layer. The maximum length for each sentence is 25. Note that 4 layer LSTM is about 4x more model parameters than the 1 layer LSTM, which is the same ratio between ResNet32 and Resnet 8.

5.3 Empirical Findings

We use batch size 32 for CNNs, DNNs, RNNs, MLR, and VAEs. For MF, we use batch size of 25000 samples, which is 2.5% of the MovieLens dataset (1M samples). We study staleness up to $s = 50$ on 8 workers, which means model caches can miss updates up to 8.75 data passes. For LDA we use $\frac{D}{10P}$ as the batch size, where D is the number of documents and P is the number of workers. We study staleness up to $s = 20$, which means model caches can miss updates up to 2 data passes. We measure the logical time in terms of the amount of work performed, such as the number of batches processed.

5.3.1 Convergence Slowdown

Perhaps the most prominent effect of staleness on ML algorithms is the slowdown in convergence, evident throughout the experiments. Figure 5.1 shows the number of batches needed to reach the desired model quality for CNNs and DNNs/MLR with varying network depths and different staleness ($s = 0, ..., 16$). Figure 5.1(b)(d) show that convergence under higher level of staleness requires more batches to be processed in order to reach the same model quality. This additional work can potentially be quite substantial, such as in Figure 5.1(d) where it takes up to 6x more batches compared with settings without staleness ($s = 0$). It is also worth pointing out

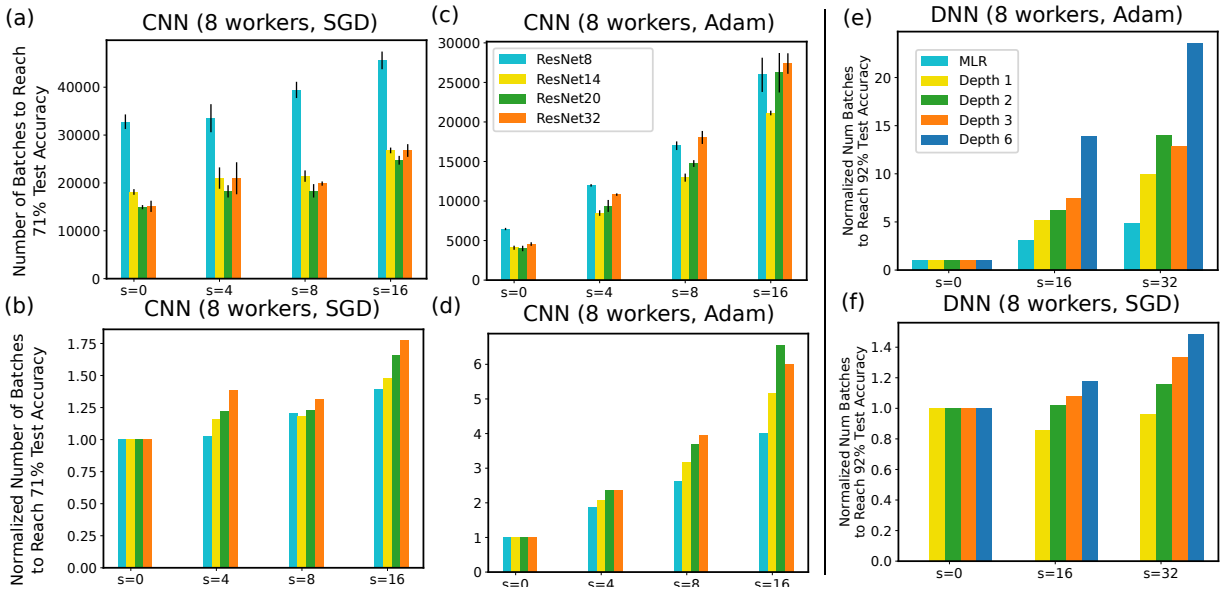

Figure 5.1: **(a)(c)** The number of batches to reach 71% test accuracy on CIFAR10 for 4 variants of ResNet with varying staleness, using 8 workers and SGD (learning rate 0.01) and Adam (learning rate 0.001). The mean and standard deviation are calculated over 3 randomized runs. **(b)(d)** The same metrics as (a)(c), but each model is normalized by the value under staleness 0 ($s = 0$), respectively. **(e)(f)** The number of batches to reach 92% accuracy for MLR and DNN with varying depths, normalized by the value under staleness 0. MLR with SGD does not converge within the experiment horizon (77824 batches) and thus is omitted in (f).

that while there can be a substantial slowdown in convergence, the optimization still reaches desirable models under most cases in our experiments. When staleness is geometrically distributed and in the presence of stragglers (Figure 5.1(c)), we observe similar patterns of convergence slowdown.

We are not aware of any prior work reporting slowdown as high as observed here. This finding has important ramifications for distributed ML. Usually, the moderate amount of workload increases due to parallelization errors can be compensated by the additional computation resources and higher system throughput in the distributed execution. However, it may be difficult to justify spending large amount of resources for a distributed implementation if the statistical penalty is too high, which should be avoided (e.g., by staleness minimization system designs or synchronous execution).

5.3.2 Model Complexity

Figure 5.1 also reveals that the impact of staleness can depend on ML parameters, such as the depths of the networks. Overall we observe that staleness impacts deeper networks more than shallower ones. This holds true for SGD, Adam, Momentum, RMSProp, Adagrad (Figure 5.1), and other optimization schemes, and generalizes to other numbers of workers (see

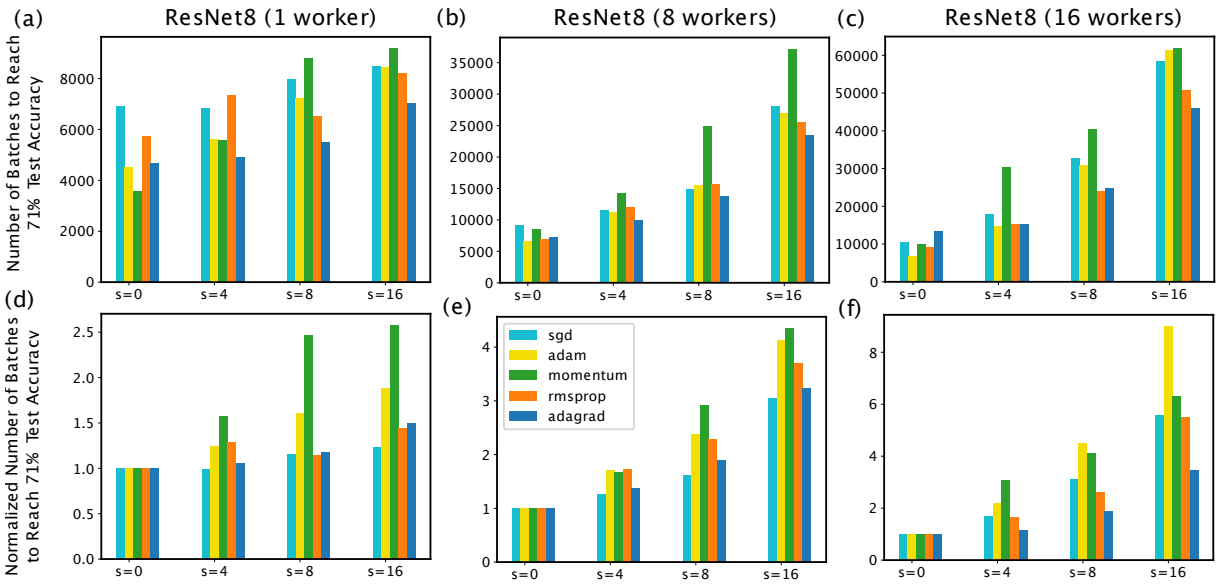

Figure 5.2: **(a)(b)(c)** The number of batches to reach 71% test accuracy on 1, 8, 16 workers with staleness $s = 0, ..., 16$ using ResNet8. We consider 5 variants of SGD: SGD, Adam, Momentum, RMSProp, and Adagrad. For each staleness level, algorithm, and the number of workers, we choose the learning rate with the fastest time to 71% accuracy from $\{0.001, 0.01, 0.1\}$. **(d)(e)(f)** show the same metric but each algorithm is normalized by the value under staleness 0 ($s = 0$), respectively, with possibly different learning rate.

Section 5.3.3)[1].

This is perhaps not surprising, given the fact that deeper models pose more optimization challenges even under the sequential settings [60, 80], though we point out that existing literature does not explicitly consider model complexity as a factor in distributed ML [68, 129]. Our results suggest that the staleness level acceptable in distributed training can depend strongly on the complexity of the model. For sufficiently complex models it may be more advantageous to eliminate staleness altogether and use synchronous training.

5.3.3 Algorithms' Sensitivity to Staleness

Staleness has uneven impacts on different SGD variants. Figure 5.2 shows the amount of work (measured in the number of batches) to reach the desired model quality for five SGD variants. Figure 5.2(d)(e)(f) reveal that while staleness generally increases the number of batches needed to reach the target test accuracy, the increase can be higher for certain algorithms, such as Mo-

[1]ResNet8 takes more batches to reach the same model quality than deeper networks in Figure 5.1(a) because, with SGD, ResNet8's final test accuracy is about 73% in our setting, while ResNet20's final test accuracy is close to 75%. Therefore, deeper ResNet can reach the same model accuracy in the earlier part of the optimization path, resulting in lower number of batches in Figure 5.1(a). However, when the convergence time is normalized by the non-stale (s=0) value in Figure 5.1(b), we observe the impact of staleness is higher on deeper models.

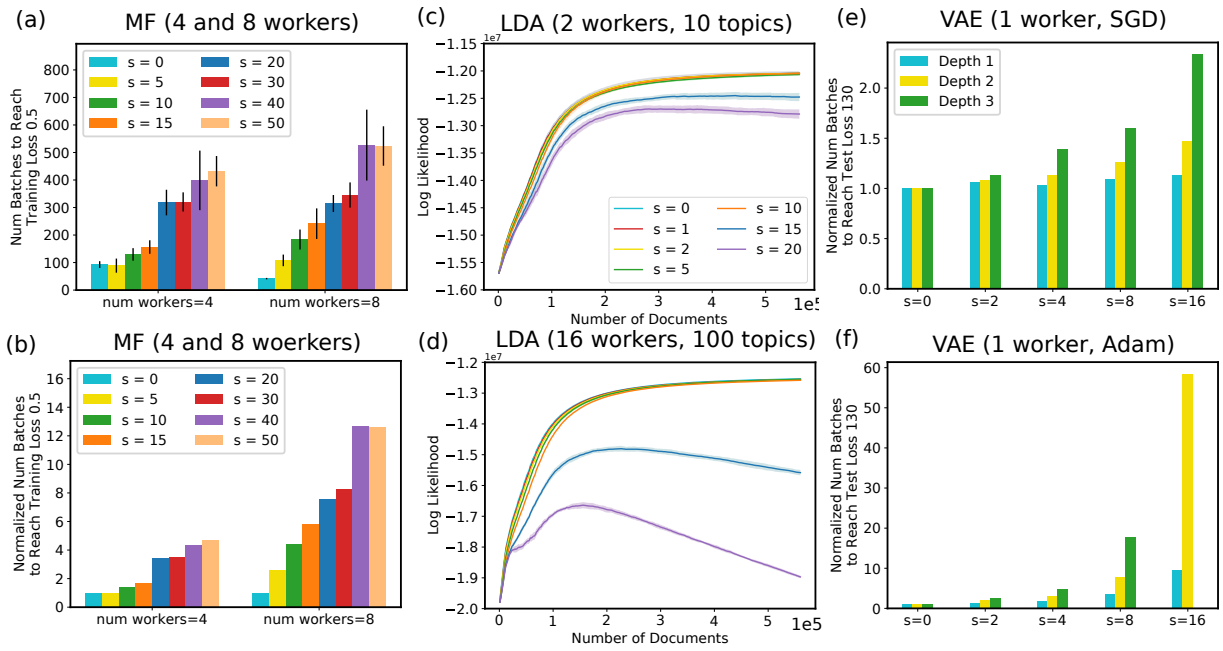

Figure 5.3: **(a)** The number of batches to reach training loss of 0.5 for Matrix Factorization (MF). **(b)** shows the same metric in (a) but normalized by the values of staleness 0 of each worker setting, respectively (4 and 8 workers). **(c)(d)** Convergence of LDA log likelihood using 10 and 100 topics under staleness levels $s = 0, ..., 20$, with 2 and 16 workers. The convergence is recorded against the number of documents processed by Gibbs sampling. The shaded regions are 1 standard deviation around the means (solid lines) based on 5 randomized runs. **(e)(f)** The number of batches to reach test loss 130 by Variational Autoencoders (VAEs) on 1 worker, under staleness $s = 0, ..., 16$. We consider VAEs with depth 1, 2, and 3 (the number of layers in the encoder and the decoder networks, separately). The numbers of batches are normalized by $s = 0$ for each VAE depth, respectively. Configurations that do not converge to the desired test loss are omitted in the graph, such as Adam optimization for VAE with depth 3 and $s = 16$.

mentum. On the other hand, Adagrad appear to be robust to staleness[2]. Our finding is consistent with the fact that, to our knowledge, all existing successful cases applying non-synchronous training to deep neural networks use SGD [32, 46]. In contrast, works reporting subpar performance from non-synchronous training often use momentum, such as RMSProp with momentum [24] and Momentum [40]. Our results suggest that these different outcomes may be partly driven by the choice of optimization algorithms, leading to the seemingly contradictory reports of whether non-synchronous execution is advantageous over synchronous ones.

Effects of More Workers

The impact of staleness is amplified by the number of workers. In the case of MF, Figure 5.3(b) shows that the convergence slowdown in terms of the number of batches (normalized by the

[2]Many synchronous systems uses batch size linear in the number of workers (e.g., [68]). We preserve the same batch size and more workers simply makes more updates in each iteration.

convergence for $s = 0$) on 8 workers is more than twice of the slowdown on 4 workers. For example, in Figure 5.3(b) the slowdown at $s = 15$ is $\sim$3.4, but the slowdown at the same staleness level on 8 workers is $\sim$8.2. Similar observations can be made for CNNs (Figure 5.3). This can be explained by the fact that additional workers amplifies the effect of staleness by (1) generating updates that will be subject to delays, and (2) missing updates from other workers that are subject to delays.

DNNs

We present additional results for DNNs. Figure 5.4 shows the number of batches, normalized by $s = 0$, to reach convergence using 1 hidden layer and 1 worker under varying staleness levels and batch sizes. Overall, the effect of batch size is relatively small except in high staleness regime ($s = 32$).

Figure 5.5 shows the number of batches to reach convergence, normalized by $s = 0$ case, for 5 variants of SGD using 1 worker. The results are in line with the analyses in the main text: staleness generally leads to larger slow down for deeper networks than shallower ones. SGD and Adagrad are more robust to staleness than Adam, RMSProp, and SGD with momentum. In particular, RMSProp exhibit high variance in batches to convergence (not shown in the normalized plot) and thus does not exhibit consistent trend.

Figure 5.6 shows the number of batches to convergence under Adam and SGD on 1, 8, 16 simulated workers, respectively normalized by staleness 0's values. The results are consistent with the observations and analyses in the main text, namely, that having more workers amplifies the effect of staleness. We can also observe that SGDS is more robust to staleness than Adam, and shallower networks are less impacted by staleness. In particular, note that staleness sometimes accelerates convergence, such as in Figure 5.6(d). This is due to the implicit momentum created by staleness [150].

MF

We show the convergence curves for MF under different numbers of workers and staleness levels in Figure 5.7. It is evident that higher staleness leads to a higher variance in convergence. Furthermore, the number of workers also affects variance, given the same staleness level. For example, MF with 4 workers incurs very low standard deviation up to staleness 20. In contrast, MF with 8 workers already exhibits a large variance at staleness 15. The amplification of staleness from increasing number of workers is consistent with the discussion in the main text. See the main text for experimental setup and analyses.

LDA

Figure 5.3(c)(d) show the convergence curves of LDA with different staleness levels for two settings varying on the number of workers and topics. Unlike the convergence curves for SGD-based algorithms (see Appendix), the convergence curves of Gibbs sampling are highly smooth, even under high staleness and a large number of workers. This can be attributed to the structure of log likelihood objective function [71]. Since in each sampling step we only update the count statistics based on a portion of the corpus, the objective value will generally change smoothly.

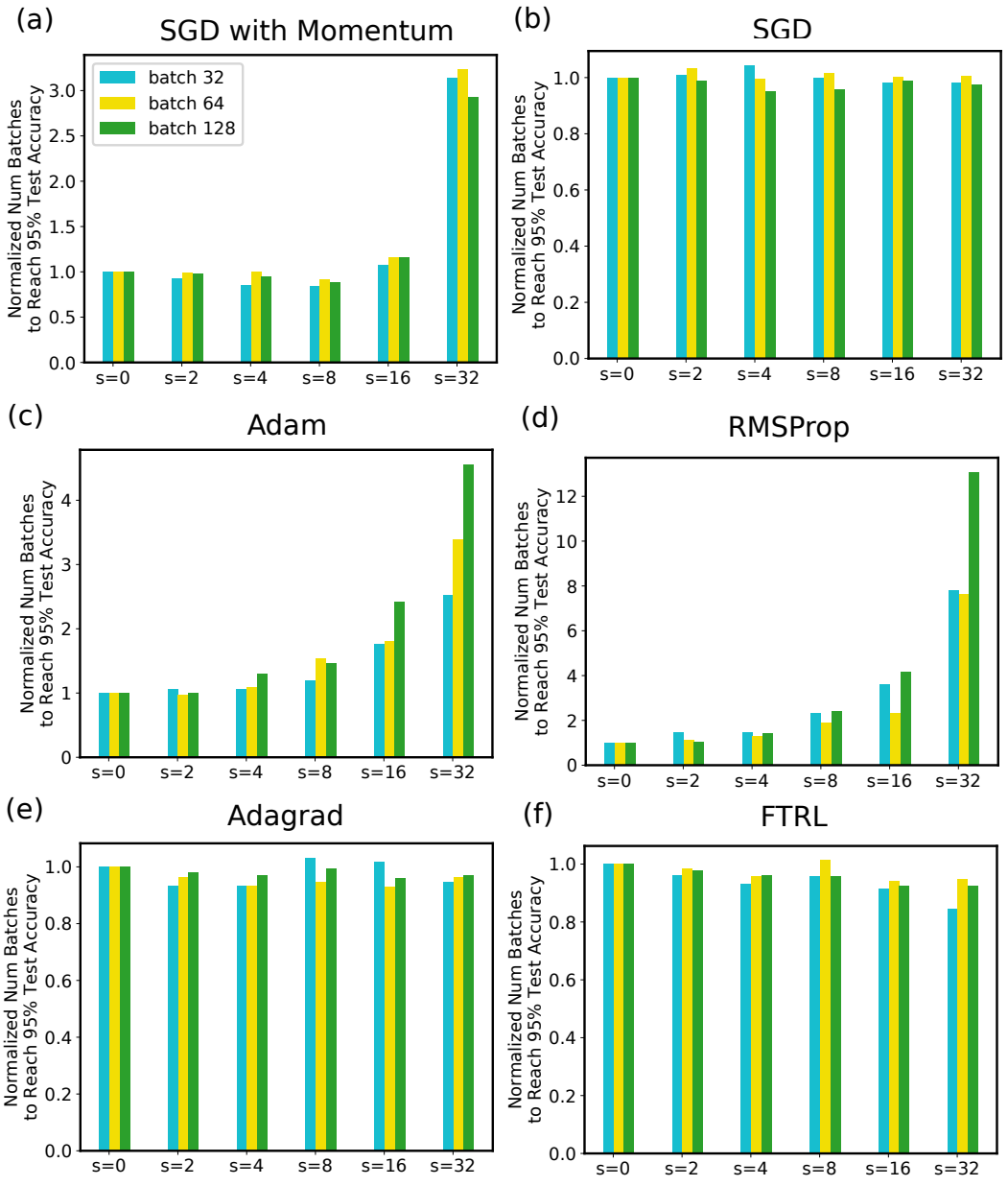

Figure 5.4: The number of batches to reach 95% test accuracy using 1 hidden layer and 1 worker, respectively normalized by $s = 0$.

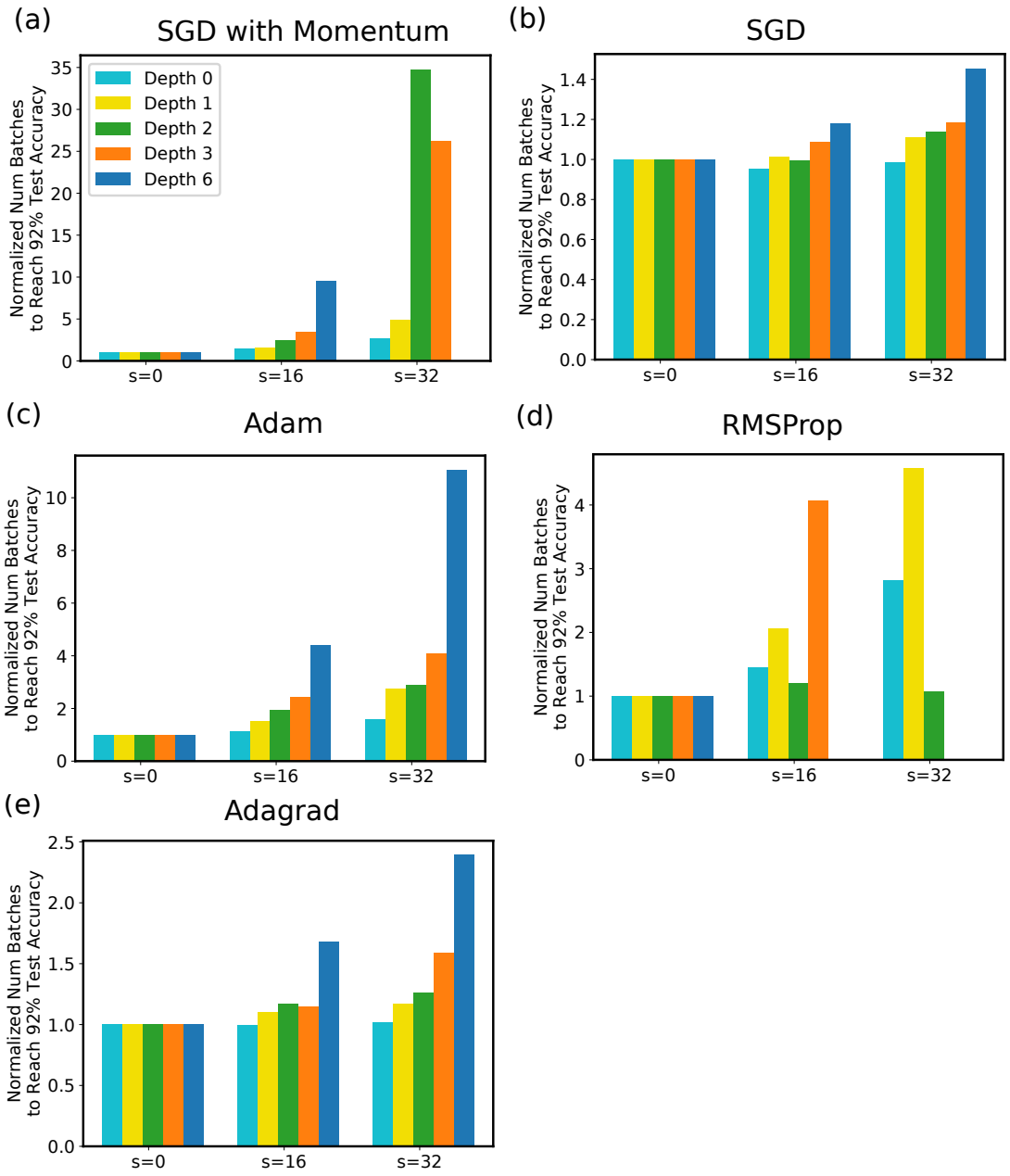

Figure 5.5: The number of batches to reach 92% test accuracy using DNNs with varying numbers of hidden layers under 1 worker. We consider several variants of SGD algorithms (a)-(e). Note that with depth 0 the model reduces to MLR, which is convex. The numbers are averaged over 5 randomized runs. We omit the result whenever convergence is not achieved within the experiment horizon (77824 batches), such as SGD with momentum at depth 6 and $s = 32$.

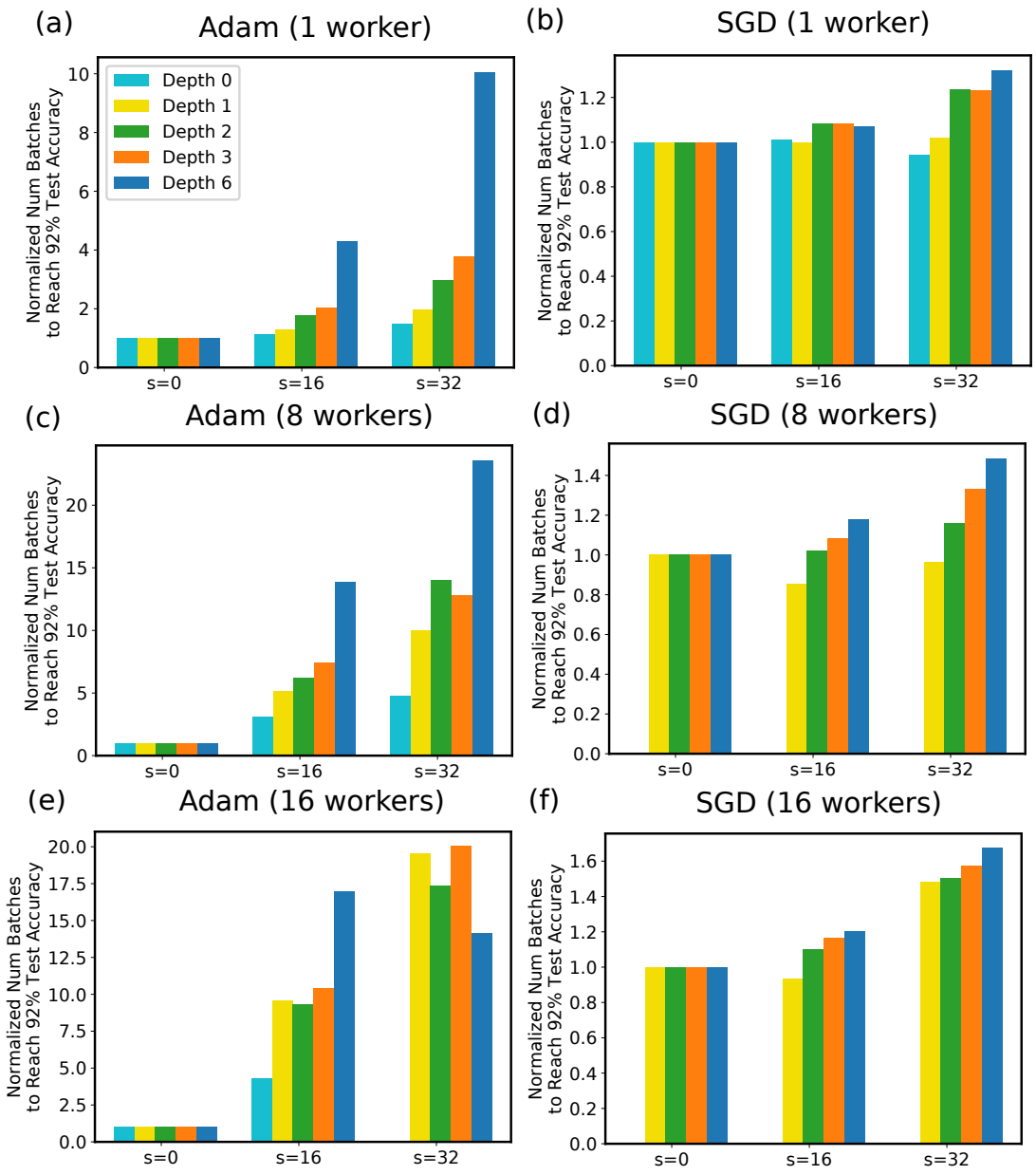

Figure 5.6: The number of batches to reach 92% test accuracy with Adam and SGD on 1, 8, 16 workers with varying staleness. Each model depth is normalized by the staleness 0's values, respectively. The numbers are average over 5 randomized runs. Depth 0 under SGD with 8 and 16 workers did not converge to target test accuracy within the experiment horizon (77824 batches) for all staleness values, and is thus not shown.

Staleness levels under a certain threshold ($s \leq 10$) lead to convergence, following indistinguishable log likelihood trajectories, regardless of the number of topics ($K = 10, 100$) or the number of workers (2–16 workers, see Appendix). Also, there is very minimal variance in those trajectories. However, for staleness beyond a certain level ($s \geq 15$), Gibbs sampling does not

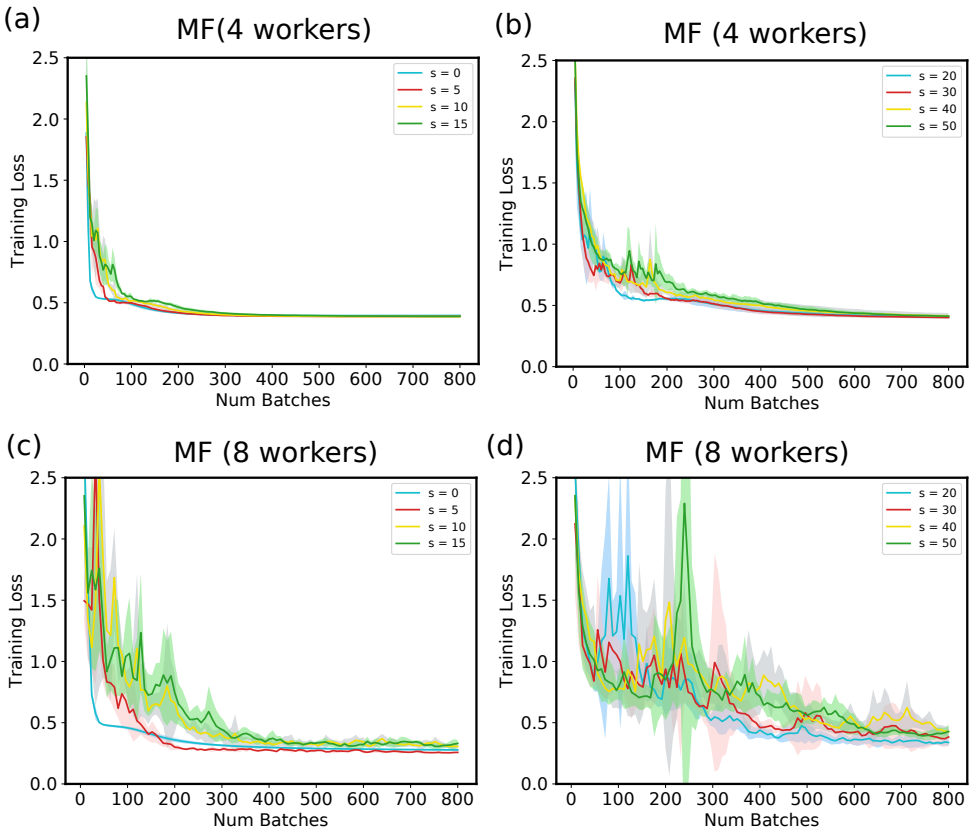

Figure 5.7: Convergence of Matrix Factorization (MF) using 4 and 8 workers, with staleness ranging from 0 to 50. The x-axis shows the number of batches processed across all workers. Shaded area represents 1 standard deviation around the means (solid curves) computed on 5 randomized runs.

converge to a fixed point. The convergence trajectories are distinct and are sensitive to the number of topics and the number of workers. There appears to be a "phase transition" at a certain staleness level that creates two distinct phases of convergence behaviors[3]. We believe this is the first report of a staleness-induced failure case for LDA Gibbs sampling.

We present additional results of LDA under different numbers of workers and topics in Figure 5.8 and Figure 5.9. These panels extends Figure 5.3(c)(d) in the main text.

VAE

In Figure 5.3(e)(f), VAEs exhibit a much higher sensitivity to staleness compared with DNNs (Figure 5.1(e)(f)). This is the case even considering that VAE with depth 3 has 6 weight layers, which has a comparable number of model parameters and network architecture to DNNs with 6 layers. We hypothesize that this is caused by the additional source of stochasticity from the sampling procedure, in addition to the data sampling process.

Figure 5.10 shows the number of batches to reach test loss 130 by Variational Autoencoders

[3]We leave the investigation into this distinct phenomenon as future work.

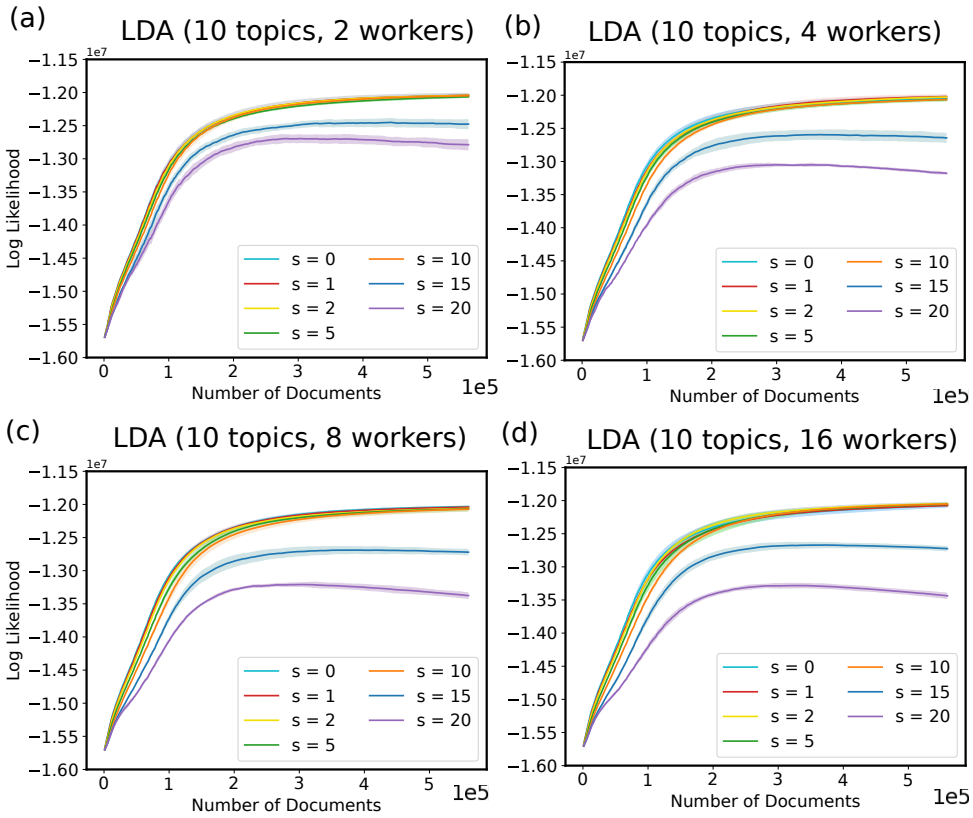

Figure 5.8: Convergence of LDA log likelihood using 10 topics with respect to the number of documents processed by collapsed Gibbs sampling, with varying staleness levels and number of workers. The shaded regions are 1 standard deviation around the means (solid lines) based on 5 randomized runs.

(VAEs) on 1 worker, under staleness 0 to 16 and 4 SGD variants. We consider VAEs with depth 1, 2, and 3 (the number of layers in the encoder and decoder networks). The number of batches are normalized by $s = 0$ for each VAE depth, respectively. See the main text for analyses.

Recurrent Neural Networks

We consider staleness $s = 0, 4, 8, 16$ on 8 workers. The model quality is measured in perplexity. Figure 5.11 shows the number of batches needed to reach the desired model quality for RNNs with varying network depths. We again observe that staleness impacts deeper network variants more than shallower counterparts, which is consistent with our observation in CNNs and DNNs.

Figure 5.12 shows the number of batches needed to reach the desired model quality for RNNs with on 4 SGD variants: SGD, Adam, Momentum, and RMSProp. Similar to the discussion in the main text, different algorithms respond to staleness differently, with SGD and Adam more robust to staleness than Momentum and RMSProp[4].

[4]We however, note that we have not tuned the learning rate in this experiment. RMSProp might benefit from a lower learning rate at high staleness.

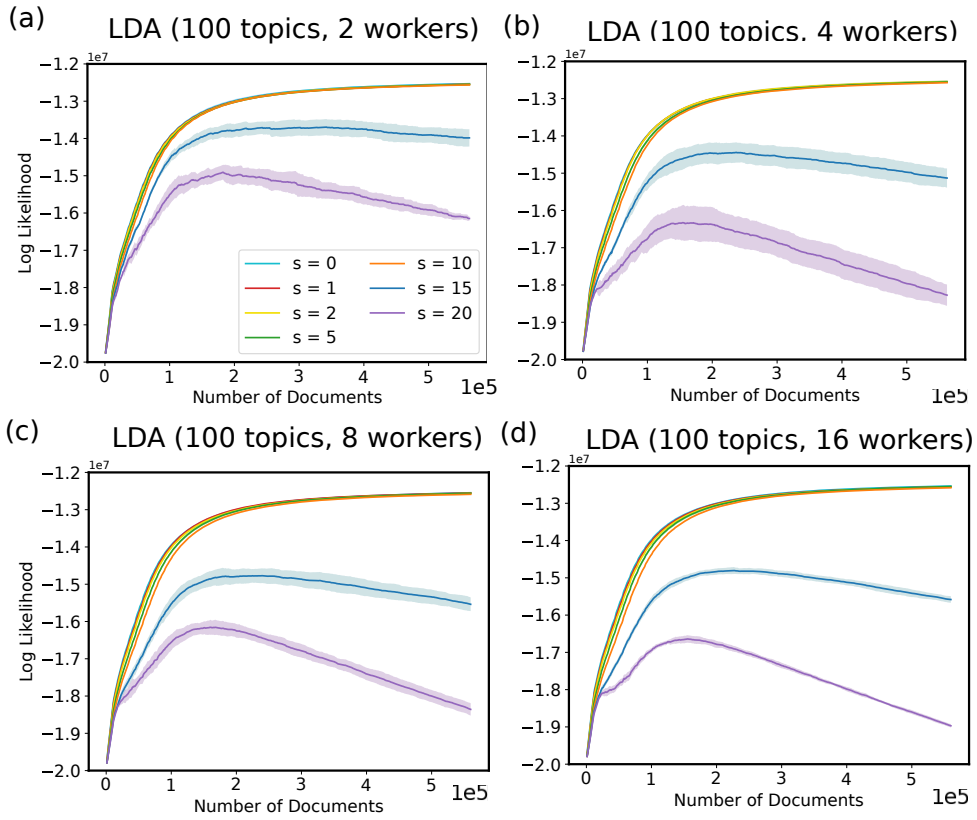

Figure 5.9: Convergence of LDA log likelihood using 100 topics with respect to the number of documents processed by collapsed Gibbs sampling, with varying staleness levels and the number of workers. The shaded regions are 1 standard deviation around the means (solid lines) based on 5 randomized runs.

5.4 Discussion

Through the empirical studies, we show that staleness appears to be a key governing parameter in learning. Overall staleness slows down the convergence, and under high staleness levels the convergence can progress very slowly or fail. The effects of staleness are highly problem dependent, influenced by model complexity, choice of the algorithms, the number of workers, and the model itself, among others.

The findings have clear implications for distributed ML. To achieve actual speed-up in absolute convergence, any distributed ML system needs to overcome the slowdown from staleness, and carefully trade off between system throughput gains and statistical penalties. Many ML methods indeed demonstrate certain robustness against low staleness, which should offer opportunities for system optimization.

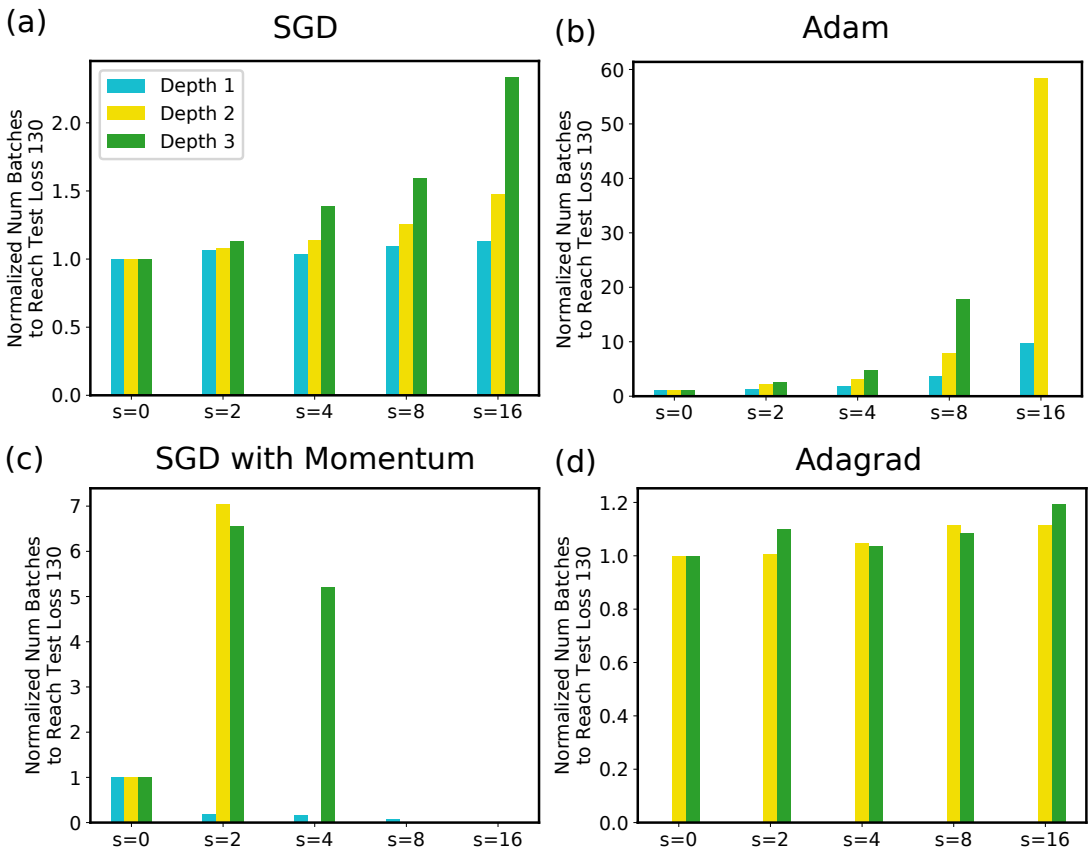

Figure 5.10: The number of batches to reach test loss 130 by Variational Autoencoders (VAEs) on 1 worker, under staleness 0 to 16. We consider VAEs with depth 1, 2, and 3 (the number of layers in the encoder and the decoder networks). The numbers of batches are normalized by $s = 0$ for each VAE depth, respectively. Configurations that do not converge to the desired test loss are omitted, such as Adam optimization for VAE with depth 3 and $s = 16$.

5.5 Additional Related Work

Staleness is reported to help absolute convergence for distributed deep learning in [32, 46, 227] and has minimal impact on convergence [75, 129, 228]. But Chen et al. and Cui et al. [24, 40] show significant negative effects of staleness. LDA training is generally insensitive to staleness [4, 83, 222, 237], and so is MF training [38, 138, 247]. However, none of their evaluations quantifies the level of staleness in the systems. By explicitly controlling the staleness, we decouple the distributed execution, which is hard to control, from ML convergence outcomes.

We focus on algorithms that are commonly used in large-scale optimization [24, 46, 68], instead of methods specifically designed to minimize synchronization [102, 157, 183]. Non-synchronous execution has theoretical underpinning [83, 128, 129, 173, 247]. Here we study algorithms that do not necessarily satisfy assumptions in their analyses.

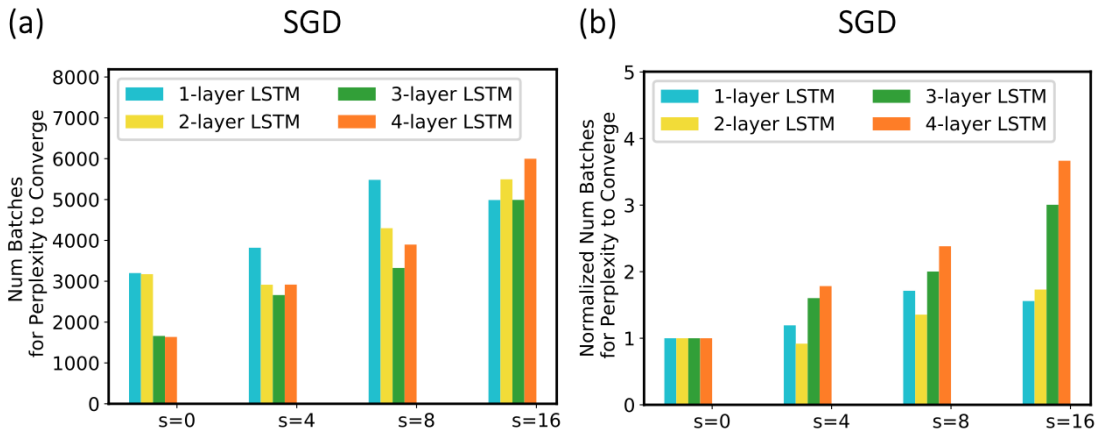

Figure 5.11: The number of batches to converge (measured using perplexity) for RNNs on 8 workers, under staleness 0 to 16. We consider LSTMs with depth $\{1, 2, 3, 4\}$.

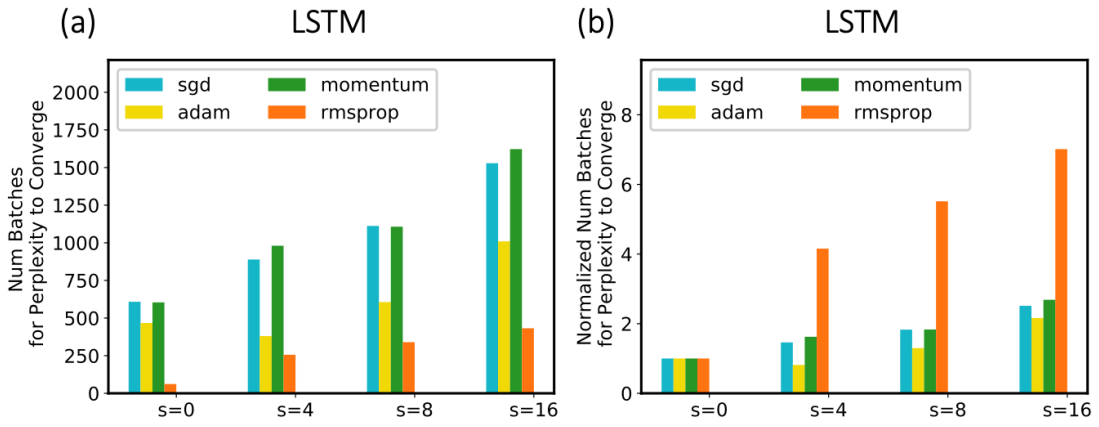

Figure 5.12: The number of batches to converge (measured using perplexity) for RNNs on 8 workers, under staleness 0 to 16. We consider 4 variants of SGD: vanilla SGD, Adam, Momentum, RMSProp.

Part II

Composability and Representations of ML Parallelisms

Overview

In the first part of the thesis, we have studied several key parallelization aspects including scheduling, communication, memory management, and consistency model, and developed aspect-specific optimizations based on the proposed methodology *adaptive parallelism*. This part of the thesis aims to generalize this approach – instead of developing system or algorithm optimizations at an aspect-by-aspect basis, we develop new representations to more effectively express the entirety of ML parallelization. We believe that a unified representation of parallelisms offers at least two outstanding benefits (as we will show later): improved programmability in writing parallel ML code, and opportunities of co-optimizing multiple parallelization aspects to further improve parallelization performance.

This part of thesis is organized into two chapters with respect to two distinct classes of parallel ML workloads: parallelization of dynamic DL models, which is commonly termed as *dynamic batching*, and distributed parallelization as done in previous chapters.

In particular, Chapter 6 develops new representations for expressing the *dynamic batching* parallelism, commonly found difficult but necessary in the training or inference for DL models with dynamic-varying structures. It presents two programming models developed progressively: dynamic declaration (Section 6.2.2), and vertex-centric representations (Section 6.4), and their corresponding system implementations, DyNet (Section 6.3) and Cavs (Section 6.5). DyNet and Cavs have advanced the performance of the parallelization of dynamic NNs by nearly on order of magnitude.

Chapter 7 gleans insight from our past developments on distributed parallelization of ML training, and offers a unified, principled representation (Section 7.3) to encapsulate various forms of (seemingly different) distributed parallelization strategies or optimizations. On top of the representation, it then presents a system, AutoDist (Section 7.4), that allows complex parallelization strategies to be composed from base atomic parallelization aspects. AutoDist simplifies the way of writing distributed parallel ML code by offering a unified interface, but with matched or even better performance compared to many specialized systems with diverse interfaces. It also lays the foundation for the automatic parallelization, which will be developed in the last part of this thesis (Chapter 8).

The results presented in this part of the thesis have appeared in the following publications:

- DyNet Contributors. DyNet: The Dynamic Neural Network Toolkit. In *arXiv:1701.03980 preprint*.
- Hao Zhang[†], Shizhen Xu[†], Graham Neubig, Qirong Ho, Guangwen Yang, and Eric P. Xing. Cavs: An Efficient Runtime System for Dynamic Neural Networks. In *2018 USENIX*

Annual Technical Conference (USENIX ATC 2018)).

- Hao Zhang, Christy Li, Zhijie Deng, Aurick Qiao, Qirong Ho, and Eric P. Xing. AutoDist: A Composable and Automated Synchronization System for Distributed Deep Learning. *Preprint 2020.*

Chapter 6

Dynamic Neural Networks Parallelization

Recent deep learning (DL) models have moved more and more from static network architectures to dynamic ones, handling data where the network structure changes every example, such as sequences of variable lengths, trees, and graphs. However, existing DL programming models are inefficient in handling dynamic network architectures because of:

- Substantial overheads caused by repeating dataflow graph construction and processing for every input data example;
- Difficulties in batched parallel execution of multiple samples;
- Inability to incorporate graph optimization techniques such as those used in static graphs.

In this chapter, we co-develop representations and systems for dynamic neural network parallelisms. We start by giving a former introduction of dynamic neural networks, and existing programming models for such models.

6.1 Dynamic Neural Networks

Successful NN models generally exhibit suitable architectures that capture the structures of the input data. For example, convolutional neural networks [117, 233], which apply fixed-structured operations to fixed-sized images, are highly effective precisely because they capture the spatial invariance common in computer vision domains [192, 201]. However, apart from images, many forms of data are structurally complex and can not be readily captured by fixed-structured NNs. Appropriately reflecting these structures in the NN design has shown effective in sentiment analysis [203], semantic similarity between sentence pairs [194], and image segmentation [130].

To see this, we will take the constituency parsing problem as an example. Sentences in natural languages are often represented by their constituency parse tree [203], whose structure varies depending on the content of the sentence itself (Figure 6.1(a)). Constituency parsing is an important problem in natural language processing that aims to determine the corresponding grammar type of all internal nodes given the parsing tree of a sentence. Figure 6.1(b) shows an example of a network that takes into account this syntactic structure, generating representations for the sentence by traversing the parse tree bottom-up and combining the representations for each sub-

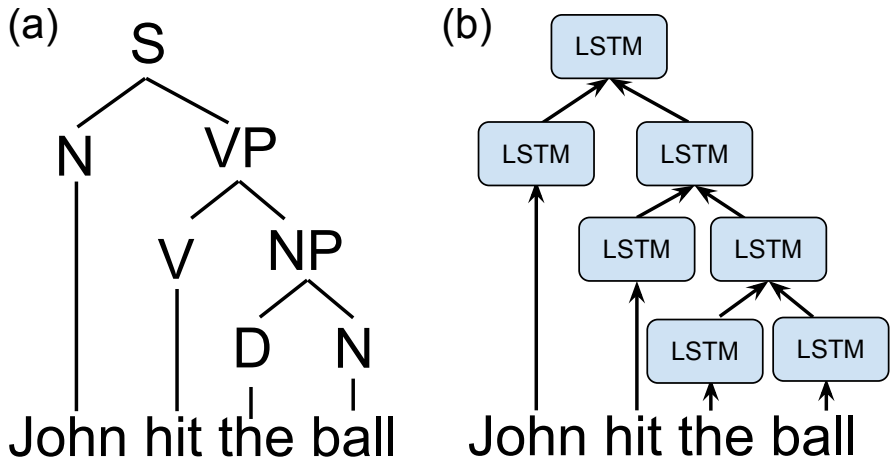

Figure 6.1: An example of a dynamic NN: (a) a constituency parsing tree, (b) the corresponding Tree-LSTM network. We use the following abbreviations in (a): S for sentence, N for noun, VP for verb phrase, NP for noun phrase, D for determiner, and V for verb.

tree using a dynamic NN called Tree Structured Long Short-term Memory (Tree-LSTM) [203]. In particular, each node of the tree maps to a LSTM function [84]. The internal computations and parameters of the LSTM function is defined in Figure 6.7. At each node, it takes a variable number of inputs and returns to the parent node a vector representing the parsing semantics up to that point, until the root LSTM node returns a vector representing the semantics of the entire sentence.

The important observation is that the NN structure varies with the underlying parsing tree over *each* input sample, but the same LSTM cell is constant in shape and repeated at each internal node. Similar examples can be found for graph input [130, 131] and sequences of variable lengths [7, 198]. We refer to these NNs that exhibit different structures for different input samples as *dynamic neural networks*, in contrast to the static networks that have fixed network architecture for all samples. We next introduce the most adopted dynamic NNs – recurrent and recursive neural networks (RNNs) [55, 84, 194].

6.1.1 Recurrent and Recursive Neural Networks (RNNs)

RNNs are a class of NNs generally applied on modeling structured inputs or outputs, e.g., sequences or graphs. At the core of an RNN is a cell function with trainable parameters. It will be dynamically applied at different places of the input structure, and optionally produce an output if needed. Figure 6.2(a) illustrates such a cell function: it takes an input element x, forwards it through a few mathematical transformations, and generates some intermediate state h and an output o. Depending on what transformations are applied, different variants of RNNs have been derived, such as long-short term memory units (LSTM) [84] and gated recurrent units (GRU) [33]. However, *the internals of the cells themselves are secondary; the dynamics of the net as a whole are mainly reflected by the structures that the NN works on*.

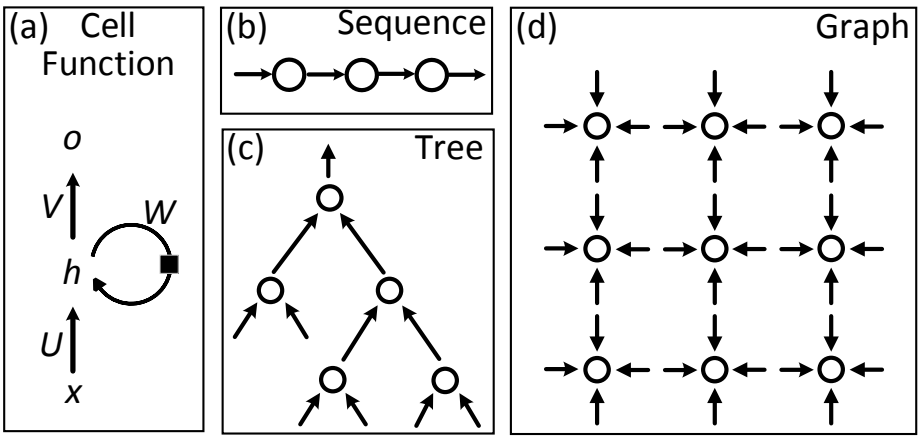

Figure 6.2: A cell function shown in (a) could be applied on different structures such as a (b) chain (c) tree, or (d) graph.

Sequence RNNs

When the input to the RNN are sequences (e.g., sentences) as in Figure 6.2(b), the cell function is applied across all elements of the sequence. At each step t, it takes the element x_t (e.g., a word) from the input sequence, and the state variable h_{t-1} maintained by the model at step $t-1$. It computes an output o_t, and a new state h_t that will be used by the next step $t+1$. Hence, This sequence RNN encodes not only the data values, but also the dependencies present in the sequence. If represented as a dataflow graph, the graph exhibits a chain structure. As the input or output sequences usually have variable length (e.g., translating an arbitrary-length English sentence into Chinese), the dataflow graph needs to be dynamically changed, i.e., the steps of the chain must be adapted to fit the length of the input or output.

Tree-structured RNNs

Further, RNNs can be enhanced to model data with more complex structures suited for downstream tasks. For example, tree-structured RNNs (Tree-RNNs, Figure 6.2(c)), have been used to classify the sentiment of sentences [164] given an associated binary tree representing the sentence structure [193, 203]. In this case, a leaf of the tree maps to a word of the sentence, an internal node corresponds to a multi-word phrase. To process this structure, the cell function scans the tree recursively, starting from leaf nodes until reaching the root. At the node t, it computes the state $h_t = f(h_{t_l}, h_{t_r}, x_t)$, where x_t is the input to the node, and h_{t_l}, h_{t_r} are the states of its left and right children, respectively. As the tree structure vary from example to example, the dataflow graph corresponded to a Tree-RNN is highly dynamic.

Graph-structured RNNs

Similarly, RNNs can be extended to compute over more general graphs, such as N-ary trees or graphs (Figure 6.2(d)), as long as their parameters are learnable. In fact, various NNs have been developed toward having more dynamic workflows [130, 203], and proven quite effective

because of their ability to encode structured information. While we take RNNs as examples for explanation, we note there are many other dynamic NNs in the literature or production with their dynamics reflected in various perspectives: variably sized inputs/outputs [7, 18, 54, 55, 84, 198], variably structured inputs/outputs [130, 193, 203], or with nontrivial inference algorithms [67, 70, 112, 249].

6.2 Programming and Parallelizing Dynamic NNs

6.2.1 Static Declaration

There is a natural connection between NNs and directed graphs: we can map the graph nodes to the computational operations or parameters in NNs, and let the edges indicate the direction of the data being passed between the nodes. In this case, we can represent the process of training NNs as batches of data flowing through computational graphs, i.e., *dataflow graphs* [1, 10, 158], as we have introduced in chapter 2.3.

Based on dataflow graphs, one dominant paradigm in the training of DL models, adopted by almost all toolkits (TensorFlow, MxNet), is *static declaration* [1, 154]. In static declaration, the declared graph represents the flow of data through computational functions, and are defined using symbolic programming [1, 10], once before beginning training or testing of the model. The training of these models is performed through auto-differentiation, in which users are only required to assemble their model architectures by connecting operators using high-level language interface (e.g., Python), after which the framework will automatically derive the correct algorithm for training [9]. With proper optimization, the execution of these static dataflow graphs can be highly efficient. Specifically, by separating model declaration and execution, it makes it possible for the graph to be further processed and optimized before runtime [1]. In addition, the evaluation of multiple data samples in a dataflow graph can be naturally *batched* (i.e., parallelized) to leverage the improved computational capability of modern hardware (e.g., GPUs), which is extremely advantageous for DL workloads [117].

```
/* static declaration */
// all samples must share one graph
declare a static dataflow graph 𝒟.
for p = 1 → P:
  read the pth data batch {x_k^p}_{k=1}^K.
  batched computation: 𝒟({x_k^p}_{k=1}^K).
```

Figure 6.3: The workflow of static declaration. Notations: $\mathcal{D}$ notates both the dataflow graph itself and the computational function implied by it; p is the index of a batch while k is the index of a sample in the batch.

Figure 6.3 summarizes its workflow, which assumes all data samples share a fixed NN structure declared symbolically in a dataflow graph $\mathcal{D}$. While these static dataflow graphs have major efficiency advantages, their applicability highly relies on a key assumption – the dataflow graph

(i.e., NN architecture) fixed throughout the runtime. Hence, *in its original form, static declaration cannot express dynamic NNs with structures changing with data samples*. Two variants of static declaration are derived to slightly mitigate this issue.

Static unrolling. Static unrolling [1] is a standard way to express sequence RNNs with fixed steps. To handle variable-length data, it declares an RNN that has number of steps equal with the length of the longest sequence in the dataset. It then appends zeros at the end of other sequences to have equal length, and feeds them in batches to the dataflow graph for computation. Static unrolling enables batched computation of multiple sequences, but obviously results in substantial unnecessary computation[1].

Dynamic unrolling. Dynamic unrolling implements basic control flow functionality within static graphs, allowing for the declaration of graph operators similar to `while` loops. At each iteration of the training, the cell function of the RNN will be executed a conditional number of times determined at runtime by the length of the longest sentence in the batch. It then pads the sequences in the batch and performs batched computation – apparently it also waste computational resources.

Notably, both of these methods essentially *cannot* support more complex structures than sequences. There are currently two remedies to this problem: expanding the graph programming language to allow it to explicitly include controls structure necessary to implement these applications, or forgo the efficiency gains afforded by static dataflow graphs and instead use a dynamic declaration framework that reconstructs the graph for every training example – the latter approach and the resultant open source framework, DyNet (to which this thesis has contributed to), are discussed in the next sections.

6.2.2 Dynamic Declaration

With a slight modification on static declaration, we can derive a more flexible programming model, *dynamic declaration*, illustrated at Figure 6.4.

```
/* dynamic declaration */
for p = 1 → P:
  read the pth data batch {x_k^p}_{k=1}^K.
  for k = 1 → K:
     declare a dataflow graph D_k^p for x_k^p.
     single-instance computation: D_k^p(x_k^p).
```

Figure 6.4: The workflow of dynamic declaration.

By creating a unique dataflow graph $\mathcal{D}_k^p$ for each sample x_k^p according to its associated structure, dynamic declaration is able to express sample-dependent dataflow graphs. It however causes extra overhead on graph construction and puts constraints on runtime optimization, which usually lead to inefficient execution. Particularly, since a dataflow graph $\mathcal{D}_k^p$ needs to be constructed per sample, the overhead is linearly increasing with the number of samples, and sometimes yields

[1]It is also possible to split sentences into several buckets of different lengths, which alleviates this problem somewhat but adds some degree of code complexity and is not a fundamental solution.

downgraded performance [136]. Moreover, we can hardly benefit from any well-established dataflow graph optimization. We will have to perform graph processing/optimization for each dataflow graph and every single sample; but incorporating this optimization itself has a non-negligible overhead. More importantly, as we are unable to batch the computation of different structured graphs, we note in Figure 6.4 single-instance computation $\mathcal{D}_k^p(x_k^p)$ would be very inefficient due to the absence of batched computation (hence underutilized GPUs).

6.3 DyNet: Dynamic Neural Network Toolkit

We next describe a specialized, open-sourced ML framework, DyNet, which implements dynamic declaration, and partially addresses some of the aforementioned problems (but not all). While the thesis author was an active contributor to DyNet design and code, he did not participate in the scientific studies involving benchmarking the performance of DyNet. Therefore, this section mainly aims to reveal the two key ingredients adopted in DyNet, by *quoting from a white paper provided by all DyNet contributors* [158], but skips a thorough evaluation of it.

6.3.1 DyNet Design Overview

In contrast to the two-step process of definition and execution used by the static declaration paradigm, DyNet is built on top of the dynamic declaration model, which takes a single-step approach, in which the user defines the computation graph programmatically as if they were calculating the outputs of their network on a particular training instance. For example, in the case of image processing, this would mean that for every training example, the user would "Load a 64x64 image into their computation graph, perform several convolutions, and calculate either the predictive probabilities (for test) or loss (for training)." Notably, there are no separate steps for definition and execution: the necessary computation graph is created, *on the fly*, as the loss calculation is executed, and a new graph is created for each training instance. This requires very lightweight graph construction.

This general design philosophy is advantageous because it allows the user to: (1) define a different computation architecture for each training example or batch, allowing for the handling of variably sized or structured inputs using flow-control facilities of the host language, and (2) interleave definition and execution of computation, allowing for the handling of cases where the structure of computation may change depending on the results of previous computation steps. Furthermore, it reduces the complexity of the computation graph implementation since it does not need to contain flow control operations (those in TensorFlow [236]) or support dynamically sized data – these are handled by the host language (C++ or Python).

To alleviate the linearly increasing graph construction overhead and lack of batching capability, DyNet deliberately incorporates two optimizations: *efficient graph construction*, and *on-the-fly dynamic batching*.

6.3.2 Efficient Graph Construction

DyNet's backend is designed with this rapid computation graph construction in mind. First and foremost, the backend is written in C++ with a specific focus on ensuring that graph building operations are as efficient as possible. Specifically, one of the major features of DyNet is that it performs its own memory management for all computation results[2]. When the DyNet library is initialized, it allocates three large blocks of memory, the one for storing results of forward calculation through the graph, one for storing backward calculations, and one for storing parameters and the corresponding gradients. When starting processing of a new graph, the pointers to the segments of memory responsible for forward and backward computation are reset to the top of the corresponding segments, and any previously used backward memory is zeroed out with a single operation. When more memory is required during forward execution or in advance of running the backward algorithm, the custom allocator simply returns the current pointer, then increments it according to the size of the memory segment requested. Thus, in this extremely simple memory allocation setup, deallocation and allocation of memory are simple integer arithmetic. This is also an advantage when performing operations on the GPU, as managing memory requires no interaction with the GPU itself, and can be handled entirely by managing, on the CPU, pointers into GPU memory.

Interacting with the DyNet library is also efficient. When writing programs in C++, it is possible to interact directly on the machine code level. The Python wrapper is also written with efficiency in mind, attempting to keep the wrapping code as minimal as possible, delegating the operations directly to the C++ core when possible, writing the glue code in Cython rather than Python, and trying to minimize Python objects allocation. At the same time, we also attempt to provide a "Pythonic" and natural high-level API. There is empirical evident that DyNet Python speed is indeed very close to the C++ one.

6.3.3 On-the-fly Dynamic Batching

The execution of dynamic NNs cannot be trivially parallelized over a batch of training data (contrast to static NNs in static declaration), due to the ever-changing computational patterns with data samples. DyNet incorporates on-the-fly dynamic batching [159] to slightly alleviate this issue, which identifies parallelizable operations in multiple different computational graphs and batches them with two steps.

Checking Computing Compatibility Groups

It first partitions operations in different graphs into different groups. Operations in the same group have the potential to be batched together. The partitioning is done by creating, for each operation, a string signature expressing the input, output, the operation type, and other computation-critical information, and then grouping operations that share similar signatures.

[2]The data structure used to represent the computation graph is managed by the standard C++ memory management, care is taken to avoid performing expensive allocations in the construction of this structure as well.

Determining Execution Order

With these groups, it selects an execution order to batch parallelizable operations in the same group. This execution order needs to satisfy at least two requirements: (1) an operation must be executed only when all its inputs are ready, i.e., after all its dependent operations in the graph haven been executed; (2) only nodes in the same group, without dependencies between each other, are deemed as parallelizable. Given multiple computational graphs and operations therein, finding execution order subject to these constraints is NP-hard. DyNet develops *depth-based batching* and *agenda-based batching* as heuristics to greedily generate sub-optimal orders.

Finally, this non-trivial batching procedure must be executed quickly so that overhead due to batch scheduling calculations does not cancel out the efficiency gains from operation batching. Empirically, this dynamic batching mechanism, implemented on top of DyNet, gives healthy improvements in computation time across a series of notable challenging-to-batch models, including BiLSTM, TreeLSTM, and Transition-Parsing: 3.6–9.2x on the CPU, and 2.7–8.6x on GPU.

6.4 Vertex-centric Representation

While dynamic declaration is convenient to developers as it removes the restriction that computation must be completely specified before training begins, it exhibits a few limitations. First, constructing a graph for every sample results in substantial overhead, which grows linearly with the number of input instances. In fact, we find graph construction takes longer time than the computation in some frameworks (see Section 6.6.3); even for frameworks with optimized graph construction implementations, such as DyNet, the overhead is still substantial. It also prevents the application of complex static graph optimization techniques (see Section 6.5.4). Moreover, since each sample owns a dataflow graph specifying its unique computational pattern, batching together similarly shaped computations across instances is non-trivial. Without parallelization, the computation is inefficient due to its lack of ability to exploit modern computational accelerator. While some progress has been made in recent research [136, 159], such as the on-the-fly dynamic batching introduced in Section 6.3.3, how to automatically batch the computational operations from different graphs remains a difficult problem.

To fundamentally address these challenges, we present a vertex-centric programming model, and an efficient runtime system Cavs (Section 6.5), for dynamic NNs. Instead of declaring a dataflow graph per sample, the representation decomposes a dynamic NN into two components: a static vertex function $\mathcal{F}$ that is only declared (by the user) and optimized once before execution, and an input-specific graph $\mathcal{G}$ obtained via I/O at runtime. Thus, it inherits the flexibility of symbolic programming [1, 57, 158] for DL; it requires users to define $\mathcal{F}$ by writing symbolic expressions in the same way as in static declaration. With $\mathcal{F}$ and $\mathcal{G}$, the workflow of training or testing a dynamic NN is cast as scheduling the execution of $\mathcal{F}$ following the structure of the input graph $\mathcal{G}$. Cavs will perform auto-differentiation, schedule the execution following dependencies in $\mathcal{G}$, and guarantee efficiency and correctness. We next describe the design of the vertex-centric representation and the system Cavs in detail.

6.4.1 Vertex-centric Programming Model

Our motivation comes from a key property of dynamic NNs: most dynamic NNs are designed to exhibit a recursive structure; Within the recursive structure, a static computational function is being applied following the topological order over instance-specific graphs. For instance, if we denote the constituency parsing tree in Figure 6.1 as a graph $\mathcal{G}$, where each node of the tree maps to a vertex in $\mathcal{G}$, we note the Tree-LSTM can be interpreted as follows: a computational cell function, specified in advance, is applied from leaves to the root, following the dependencies in $\mathcal{G}$. $\mathcal{G}$ might change with input samples, but the cell function itself is always static: It is parametrized by a fixed set of learnable parameters and interacts in the same way with its neighbors when applied at different vertices of $\mathcal{G}$.

These observations motivate us to decompose a dynamic NN into two parts: (1) a static computational *vertex function* $\mathcal{F}$ that needs to be declared by the programmer once before runtime; (2) a dynamic *input graph* $\mathcal{G}$ that changes with every input sample[3]. With this representation, the workflow of training a dynamic NN can be cast as scheduling the evaluation of the symbolic construct encoded by $\mathcal{F}$, following the graph dependencies of $\mathcal{G}$, as illustrated in Figure 6.5.

```
/* vertex-centric model */
declare a symbolic vertex function F.
for p = 1 → P:
  read the pth data batch {x_k^p}_{k=1}^K.
  read their associated graphs {G_k^p}_{k=1}^K.
  compute F over {G_k^p}_{k=1}^K with inputs {x_k^p}_{k=1}^K.
```

Figure 6.5: The workflows of the vertex-centric programming model.

By exploiting the properties of dynamic NNs, we argue that this representation addresses the aforementioned issues in the following ways.

Minimize Graph Construction Overheads

It only requires users to declare $\mathcal{F}$ using symbolic expressions, and construct it once before execution. This bypasses repeated construction of multiple dataflow graphs, avoiding overheads. While it is still necessary to create an I/O function to read input graphs $\mathcal{G}$ for each sample, this must be done by any method, and only once before training commences, and it can be shared across samples.

Batched Execution

With the proposed representation, Cavs transforms the problem of evaluating data $\{x_k^p\}_{k=1}^K$ (at the pth batch) on different dataflow graphs $\{\mathcal{D}_k^p\}_{k=1}$ [136, 159] into a simpler form – scheduling the execution of the vertex function $\mathcal{F}$ following the dependencies in input graphs $\{G_k^p\}_{k=1}$. For

[3]In the following text, we will distinguish the term *vertex* from *node*. We use *vertex* to denote a vertex in the input graph while *node* to denote an operator/variable in a dataflow graph. Hence, a vertex function can have many nodes when it represents a dataflow graph.

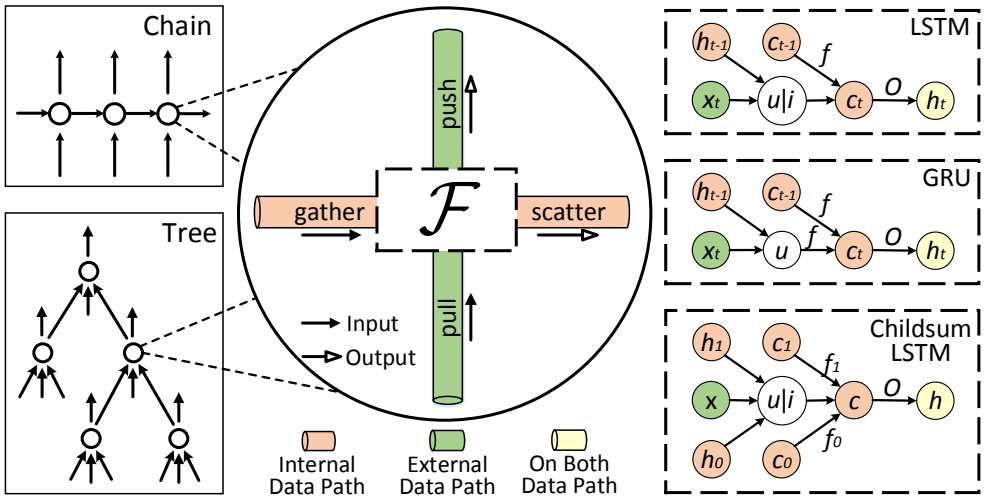

Figure 6.6: The vertex-centric representation expresses a dynamic model as a dynamic input graph $\mathcal{G}$ (left) and a static vertex function $\mathcal{F}$ (right).

the latter problem, we can easily batch the execution of $\mathcal{F}$ on multiple vertices at runtime (Section 6.5.1), leveraging the batched computational capability of modern accelerators and libraries.

Open to Graph Optimizations

Since the vertex function $\mathcal{F}$ encodes a dataflow graph which is static throughout runtime, it can benefit from various graph optimizations originally developed for static declaration, such as kernel fusion, streaming, and our proposed lazy batching, which are not effective in dynamic declaration (see Section 6.5.4).

Based on this motivation, we next describe the programming interfaces and the Cavs system. We highlight the following design challenges: (1) how to design minimal APIs in addition to the symbolic programming interface to minimize user code; (2) how to schedule the execution of $\mathcal{F}$ over multiple input graphs to enable batched computation; (3) how to manage memory to support the dynamic batching; (4) how to incorporate static graph optimization in Cavs's execution engine to exploit more parallelisms.

6.4.2 Programming Interface

Conventional dataflow graph-based programming models usually entangle the computational workflow in $\mathcal{F}$ with the structure in $\mathcal{G}$, and require users to express them as a whole in a single dataflow graph. Instead, Cavs separates the static vertex function $\mathcal{F}$ from the input graph $\mathcal{G}$ (see Figure 6.6). While users use the same set of symbolic operators [1, 56] to assemble the computational workflow in $\mathcal{F}$, Cavs proposes four additional APIs, `gather`, `scatter`, `pull`, `push`, to specify how the messages shall be passed between connected vertices in $\mathcal{G}$:

- `gather(child_idx)`: `gather` accepts an index of a child vertex, gets its output, and returns a list of symbols that represent the output of the child.

- `scatter(op)`: `scatter` reverses `gather`. It sets the output of the current vertex as `op`. If this vertex is `gather`ed, the content of `op` will be returned.

`gather` and `scatter` are motivated by the GAS model in graph computing [61] – both are vertex-centric APIs that help users express the overall computational patterns by thinking locally like a vertex: `gather` receives messages from dependent vertices, while `scatter` updates information to parent vertices (see discussion in Section 6.7).

However, unlike graph computing, in dynamic NNs, the vertex function $\mathcal{F}$ usually takes input from not only the internal vertices of $\mathcal{G}$ (internal data path in Figure 6.6), but also the external environment, e.g., an RNN can take inputs from a CNN feature extractor or some external I/O (external data path in Figure 6.6). Cavs therefore provides another two APIs to express such semantics:

- `pull()`: `pull` grabs inputs from the external of the current dynamic structure, e.g., another NN, or I/O.
- `push(op)`: `push` reverses `pull`. It sets the output of the current vertex as `op`. If this vertex is `pull`ed by others, the content of `op` will be returned.

Once $\mathcal{F}$ declared, together with an input graph $\mathcal{G}$, they encode a recursive dataflow graph structure, which maps to a subgraph of the implicit full dataflow graph of the model that may needs to be explicitly declared in traditional programming models. Via `push` and `pull`, Cavs allows users to connect any external static dataflow graph to a dynamic structure encoded by $(\mathcal{F}, \mathcal{G})$, to express more complex model architectures, such as the LRCN [52] (i.e., connecting a CNN to an RNN), or an encoder-decoder LSTM network [198] (i.e., connecting two different recursive structures). With these four APIs, we present in Figure 6.7 an example user program how the N-ary child-sum Tree-LSTM [203] can be simply expressed by using them and other conventional mathematical operators.

Expressiveness

With these four APIs, this new programming model can be seen as a middle ground between static and dynamic declaration. In the best case that the NN is fully recursive (e.g., most recurrent or recursive NNs), it can be represented by a single vertex function and an input graph. While in the worst case, that every sample has a unique input graph while every vertex in the graph has a unique way to interact with its neighboring vertices (i.e., the NN is dynamic but non-recursive nor recurrent), Cavs reduces to dynamic declaration that one has to define a vertex function for each vertex of each input graph. Fortunately, in general, dynamic NNs in this scenario are usually avoided because of difficulties in designing, programming and training such architectures.

As a summary, we list the pros and cons of the three programming models – static declaration, dynamic declaration, and the vertex-centric programming model, in Table 6.1 for a side-by-side comparison.

```
def F():
  for k in range(N):
    S = gather(k) # gather states of child vertices
    c_k, h_k = split(S, 2) # get hidden states c and h
  x = pull() # pull the first external input x

  # specify the computation
  h = sum_{k=0}^{N-1} h_k
  i = sigmoid(W^(i)× x + U^(i)× h + b^(i))
  for k in range(N):
    f_k = sigmoid(W^(f)× x + U^(f)× h_k + b^(f))
  o = sigmoid(W^(o)× x + U^(o)× h + b^(o))
  u = tanh(W^(u)× x + U^(u)× h + b^(u))
  c = i ⊗ u + sum_{k=0}^{N-1} f_k ⊗ c_k
  h = o ⊗ tanh(c)

  scatter(concat([c, h], 1)) # scatter c, h to parents
  push(h)                    # push to external connectors
```

Figure 6.7: The vertex function of an N-ary child-sum TreeLSTM [203] in Cavs. Within $\mathcal{F}$, users declare a computational dataflow graph using symbolic operators. The defined $\mathcal{F}$ will be evaluated on each vertex of $\mathcal{G}$ following graph dependencies.

Programming Model	Frameworks	Expressiveness	Batching	Graph Construction Overhead	Graph Optimization
static declaration	Caffe, TensorFlow	×	√	low	beneficial
dynamic declaration (eager evaluation)	PyTorch, Chainer	√	×	N/A	unavailable
dynamic declaration (lazy evaluation)	DyNet	√	√	high	limited benefits
TensorFlow Fold	TensorFlow-Fold	√	√	high	unknown
vertex-centric	Cavs	√	√	low	beneficial

Table 6.1: A side-by-side comparison of existing programming models for dynamic NNs, and their advantages and disadvantages.

6.5 Cavs: An Efficient Runtime System for Dynamic Neural Networks

Derived from the vertex-centric representation and interfaces, we next explain the design of the Cavs runtime system, by going through scheduling, memory management, and related system optimizations.

6.5.1 Scheduling

Once $\mathcal{F}$ is defined and $\mathcal{G}$ is obtained from I/O, Cavs will perform computation by scheduling the evaluation of $\mathcal{F}$ over data samples $\{x_i\}_{i=1}^N$ and their input graphs $\{\mathcal{G}_i\}_{i=1}^N$.

Forward Pass

For a sample x_i with its input graph $\mathcal{G}_i$, the scheduler starts the forward pass from the input vertices of $\mathcal{G}_i$, and proceeds following the direction indicated by the edges in $\mathcal{G}_i$: at each sub-step, the scheduler figures out the next activated vertex in $\mathcal{G}_i$, and evaluates all expressions in $\mathcal{F}$ at this vertex. It then marks this vertex as *evaluated*, and proceeds with the next activated vertex until reaching a terminal vertex (e.g., the loss function). A vertex of $\mathcal{G}$ is activated if and only if all its dependent vertices have been evaluated.

Backward Pass

The backward pass is continued right after the forward. The scheduler first resets the status of all vertices as *not evaluated*, then scans the graph in a reverse direction, starting from the ending point of the forward pass. It evaluates $\partial\mathcal{F}$ at each vertex until all vertices have been evaluated in the backward pass.

To train a NN to convergence, the above process has to be iterated on all samples $\{x_i\}_{i=1}^N$ and their input graphs $\{\mathcal{G}_i\}_{i=1}^N$, for many epochs. We next describe our batched execution policy to speed the computation.

Batching Policy in Scheduling

Algorithm 6 illustrates the batched backpropagation process in Cavs. Particularly, given a data batch $\{x_k\}_{k=1}^K \subseteq \{x_i\}_{i=1}^N$ and associated graphs $\{\mathcal{G}_k\}_{k=1}^K$, this policy groups multiple vertices and performs batched evaluation of $\mathcal{F}$ in order to reduce kernel launches and exploit parallelism. Specifically, a forward pass over a batch $\{x_k\}_{k=1}^K$ are performed in multiple steps. At each step t, Cavs analyzes $\{\mathcal{G}_k\}_{k=1}^K$ at runtime and determines a set V_t that contains all activated vertices in graphs $\{\mathcal{G}_k\}_{k=1}^K$. It then evaluates $\mathcal{F}$ over these vertices by creating a *batched execution task*, with the task ID set to t[4]. The task is executed by the Cavs execution engine (Section 6.6.3). Meanwhile, the scheduler records this task by pushing V_t into a stack $\mathcal{S}$. To perform backward

[4]Whenever the context is clear, we use V_t to denote both the set of vertices to be batched together, and the batched execution task itself.

Algorithm 6: Backpropagation with the batching policy.

Function `Forward` $(\{x_k\}_{k=1}^K, \{\mathcal{G}_k\}_{k=1}^K, \mathcal{F})$**:**
  set task ID $t \leftarrow 0$, task stack $S \leftarrow \emptyset$
  while *NOT all vertices in* $\{\mathcal{G}_k\}_{k=1}^K$ *are* evaluated **do**
    figure out all activated vertices in $\{\mathcal{G}_k\}_{k=1}^K$ as a set V_t
    push V_t into $\mathcal{S}$
    evaluate $\mathcal{F}$ on V_t: `GraphExecute`$(V_t, \mathcal{F})$ (see Section 6.6.3)
    set the status of all vertices in V_t as *evaluated*
    set $t \leftarrow t + 1$
  return $\mathcal{S}$
Function `Backward` $(\mathcal{S}, \{\mathcal{G}_k\}_{k=1}^K, \partial\mathcal{F})$**:**
  set t as the size of $\mathcal{S}$
  while $\mathcal{S}$ *is not empty* **do**
    pop the top element of $\mathcal{S}$ as V_t
    evaluate $\partial\mathcal{F}$ on V_t: `GraphExecute`$(V_t, \partial\mathcal{F})$ (Section 6.5.4)
    set $t \leftarrow t - 1$

```
struct DynamicTensor {
  vector<int> shape;
  int bs;
  int offset;
  void* p; };
```

Figure 6.8: The definition of a dynamic tensor.

pass, the scheduler pops out an element V_t from $\mathcal{S}$ at each step – the execution engine will evaluate the derivative function $\partial\mathcal{F}$ over vertices in V_t, until all vertices of $\{\mathcal{G}_k\}_{k=1}^K$ are evaluated.

Compared to the on-the-fly dynamic batching (Section 6.3.3) in DyNet, Cavs determines how to batch fully dynamically during runtime using simple breadth-first search with negligible cost (instead of analyzing full dataflow graphs before every iteration of the execution). Since batched computation requires the inputs to an expression over multiple vertices to be placed on a continuous memory buffer, we develop a new memory management support for it.

6.5.2 Memory Management

In static declaration [1, 158], a symbol in the user program usually corresponds to a fixed-sized *tensor* object with a batch size dimension. While in Cavs, each batching task V_t is determined at runtime. For the batched computation to be efficient, Cavs must guarantee for each batching task, the inputs to each expression of $\mathcal{F}$ over a group of runtime-determined vertices coalescing in memory.

Cavs proposes a novel data structure *dynamic tensor* to address this challenge (Figure 6.8). A dynamic tensor is a wrapper of a multi-dimensional array [1, 211]. It contains four attributes:

`shape`, `bs`, a pointer `p` to a chunk of memory, and `offset`. `shape` is an array of integers representing the specific shape of the tensor excluding the batch dimension. It can be inferred from the user program and set before execution. The batch size `bs` is dynamically set by the scheduler at runtime at the beginning of a batching task. To access a dynamic tensor, one moves `p` forward with the value of `offset`, and reads/writes number of elements equal to $\texttt{bs} \cdot \prod_i \texttt{shape}[i]$. Therefore, `bs` together with `offset` provide a view of the tensor, and the state of the tensor will vary based on their values. Given a vertex function $\mathcal{F}$, Cavs creates dynamic tensors $\{\alpha_n\}_{n=1}^N$ for each non-parameter symbol $s_n (n = 1, \ldots, N)$ in $\mathcal{F}$, and also $\{\nabla\alpha_n\}_{n=1}^N$ as their gradients, while it creates static tensors for model parameters.

Figure 6.9 illustrates how the memory is assigned during the forward pass by manipulating dynamic tensors. In particular, in a training iteration, for a batching task V_t, the scheduler sets `bs` of all $\{\alpha_n\}_{n=1}^N$ to $M_t = |V_t|$ (the number of vertices in V_t). The execution engine then performs batched evaluation of each expression in $\mathcal{F}$. For an expression $s_l = \texttt{op}(s_r)$[5], Cavs first accesses α_r (the dynamic tensor of the RHS symbol s_r) – it offsets α_r.`p` by α_r.`offset`, and reads a block of $M_t \prod_i \alpha_r.\texttt{shape}[i]$ elements, and presents it as a tensor with batch size M_t and other dimensions as α_r.`shape`. It then applies batched computational kernels of the operator `op` over this memory block, and writes the results to α_l (the dynamic tensor of the LHS symbol s_l) on the continuous block in between $[\alpha_l.\texttt{p}+\alpha_l.\texttt{offset}, \alpha_l.\texttt{p}+\alpha_l.\texttt{offset}+M_t \prod_i \alpha_l.\texttt{shape}[i]]$. Upon the completion of V_t, the scheduler increases `offset` of all $\{\alpha_n\}_{n=1}^N$ by $M_t \prod_i \alpha_n.\texttt{shape}[i]$, respectively. It then starts the next task V_{t+1}. Hence, intermediate results generated in each batching task at forward pass are stored continuously in the dynamic tensors, and their offsets are recorded.

At the entrance of $\mathcal{F}$, the vertices $\{v_m\}_{m=1}^{M_t}$ in V_t need to interact with its dependent vertices in previous V_{t-1} to `gather` their outputs as inputs (L3 of Figure 6.7), or `pull` inputs from the external (L5 of Figure 6.7). Cavs maintains memory buffers to enable this (Figure 6.9). It records the offsets of the dynamic tensors for each $v_m \in V_t$, and therefore during the execution of `gather` operator, the memory slices of specific children can be indexed. As shown in Figure 6.9, `gather` and `scatter` share the same temporary buffer for memory re-organization, but `push` and `pull` operate on external memory buffers.

Algorithm 7 summarizes the memory management during forward and backward pass. The backward execution follows an exactly reverse order of the forward pass (Section 6.5.1), which we skip in the text. With this strategy, Cavs guarantees memory continuity for any batched computation of $\mathcal{F}$ and $\partial\mathcal{F}$. Compared to dynamic batching in DyNet, Cavs performs memory movement only at the entrance and exit of $\mathcal{F}$, instead of for each expression (operator). We empirically find this significantly reduces overhead of memory-related operations (Section 6.6.4).

[5]Note that the user-defined expressions can be arbitrary, e.g., with more than one argument or return values

Algorithm 7: Memory management at forward and backward pass.

Function `Forward` $(\{V_t\}_{t=1}^{T}, \{\alpha_n\}_{n=1}^{N}, \mathcal{F})$**:**
  for $t = 1 \to T$ **do**
    for $n = 1 \to N$ **do**
      $\alpha_n.\texttt{bs} \leftarrow M_t$
    for *each expression like* $s_l = op(s_r)$ *in* $\mathcal{F}$ **do**
      if $op \in \{\texttt{gather}, \texttt{pull}\}$ **then**
        $C \leftarrow \prod_i \alpha_l.\texttt{shape}[i], q \leftarrow \alpha_l.\texttt{p} + \alpha_l.\texttt{offset}$
        for $v_m \in V_t (m = 1 \to M_t)$ **do**
          $\texttt{src} \leftarrow \texttt{IndexBuffer}(\texttt{op}, m), \texttt{dest} \leftarrow q + (m-1)C$
          $\texttt{memcpy}(\texttt{dest}, \texttt{src}, C)$
      else if $op \in \{\texttt{scatter}, \texttt{push}\}$ **then**
        $C \leftarrow \prod_i \alpha_r.\texttt{shape}[i], q \leftarrow \alpha_r.\texttt{p} + \alpha_r.\texttt{offset}$
        for $v_m \in V_t (m = 1 \to M_t)$ **do**
          $\texttt{dest} \leftarrow \texttt{IndexBuffer}(\texttt{op}, m), \texttt{src} \leftarrow q + (m-1)C$
          $\texttt{memcpy}(\texttt{dest}, \texttt{src}, C)$
      else
        perform batched computation: $\alpha_l = \texttt{op\_kernel}(\alpha_r)$.
    for $n = 1 \to N$ **do**
      $\alpha_n.\texttt{offset} += M_t \prod_i \alpha_n.\texttt{shape}[i]$

Function `Backward` $(\{V_t\}_{t=1}^{T}, \{\alpha_n\}_{n=1}^{N}, \{\nabla\alpha_n\}_{n=1}^{N}, \partial\mathcal{F})$**:**
  for $n = 1 \to N$ **do**
    $\nabla\alpha_n.\texttt{offset} \leftarrow \alpha_n.\texttt{offset}$
  for $t = T \to 1$ **do**
    for $n = 1 \to N$ **do**
      $\alpha_n.\texttt{bs} \leftarrow M_t, \nabla\alpha_n.\texttt{bs} \leftarrow M_t$
    for *an expression* $\nabla s_r = grad\_op(\nabla s_l, s_l, s_r)$ *in* $\partial\mathcal{F}$ **do**
      for $\alpha \in \{\alpha_l, \alpha_r, \nabla\alpha_l, \nabla\alpha_l\}$ **do**
        $\alpha.\texttt{offset} \leftarrow \alpha.\texttt{offset} - M_t \prod_i \alpha.\texttt{shape}[i]$
      if $grad\_op \in \{\texttt{gather}, \texttt{pull}\}$ **then**
        $C \leftarrow \prod_i \nabla\alpha_r.\texttt{shape}[i], q \leftarrow \nabla\alpha_r.\texttt{p} + \nabla\alpha_r.\texttt{offset}$
        for $v_{t_m} \in V_t (m = 1 \to M_t)$ **do**
          $\texttt{src} \leftarrow \texttt{IndexBuffer}(\texttt{op}, t_m), \texttt{dest} \leftarrow q + (m-1)C$
          $\texttt{memcpy}(\texttt{dest}, \texttt{src}, C)$
      else if $grad\_op \in \{\texttt{scatter}, \texttt{push}\}$ **then**
        $C \leftarrow \prod_i \nabla\alpha_l.\texttt{shape}[i], q \leftarrow \nabla\alpha_l.\texttt{p} + \nabla\alpha_l.\texttt{offset}$
        for $v_{t_m} \in V_t (m = 1 \to M_t)$ **do**
          $\texttt{dest} \leftarrow \texttt{IndexBuffer}(\texttt{op}, t_m), \texttt{src} \leftarrow q + (m-1)C$
          $\texttt{memcpy}(\texttt{dest}, \texttt{src}, C)$
      else
        Read $\nabla\alpha_l, \alpha_l, \alpha_r$ if necessary, perform batched computation following
        $\nabla\alpha_r = \texttt{grad\_op}(\nabla\alpha_l, \alpha_l, \alpha_r)$, write the (batch) results continuously to $\nabla\alpha_r$

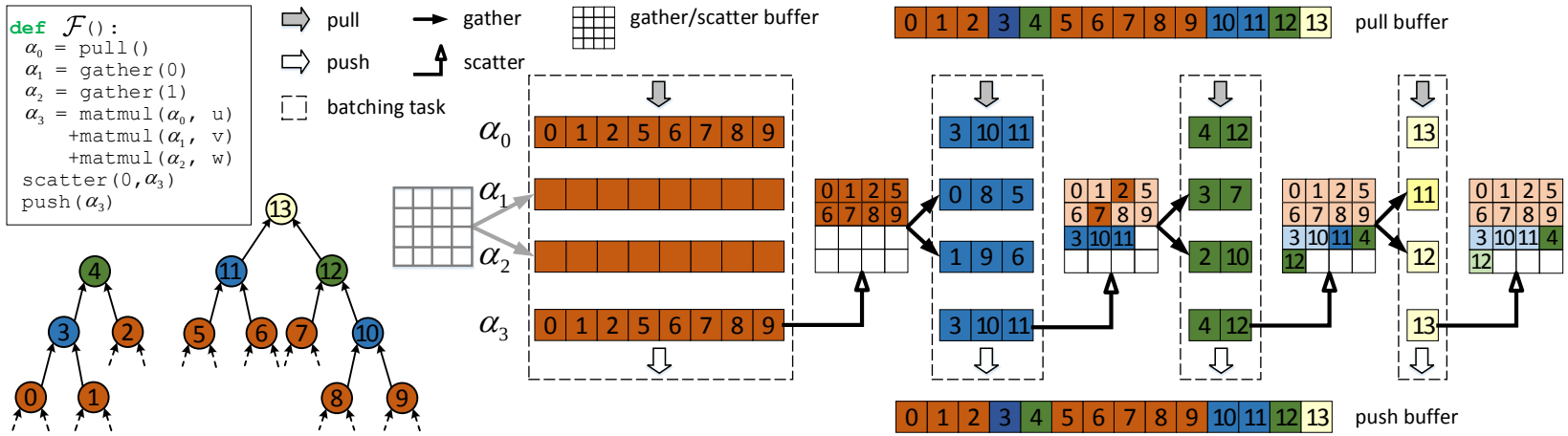

Figure 6.9: The memory management at the forward pass of $\mathcal{F}$ (top-left) over two input trees (bottom-left). Cavs first analyzes $\mathcal{F}$ and inputs – it creates four dynamic tensors $\{\alpha_n\}_{n=0}^{3}$, and figures out there will be four batch tasks (dash-lined boxes). Starting from the first task (orange vertices $\{0, 1, 2, 5, 6, 7, 8, 9\}$), Cavs performs batched evaluation of each expression in $\mathcal{F}$. For example, for the pull expression $\alpha_0 = \texttt{pull}()$, it indexes the content of α_0 on all vertices from the *pull buffer* using their IDs, and copies them to α_0 continuously; for scatter and push expressions, it scatters a copy of the output (α_3) to the *gather buffer*, and pushes them to the push buffer, respectively. Cavs then proceeds to the next batching task (blue vertices). At this task, Cavs evaluates each expression of $\mathcal{F}$ once again for vertices $\{3, 10, 11\}$. (e.g., for a pull expression $\alpha_0 = \texttt{pull}()$, it pulls the content of α_0 from pull buffer again; for a gather expression $\alpha_2 = \texttt{gather}(1)$ at vertex 3, it gathers the output of the second child of 3, which is 1); it writes results continuously at the end of each dynamic tensor. It proceeds until all batching tasks are finished.

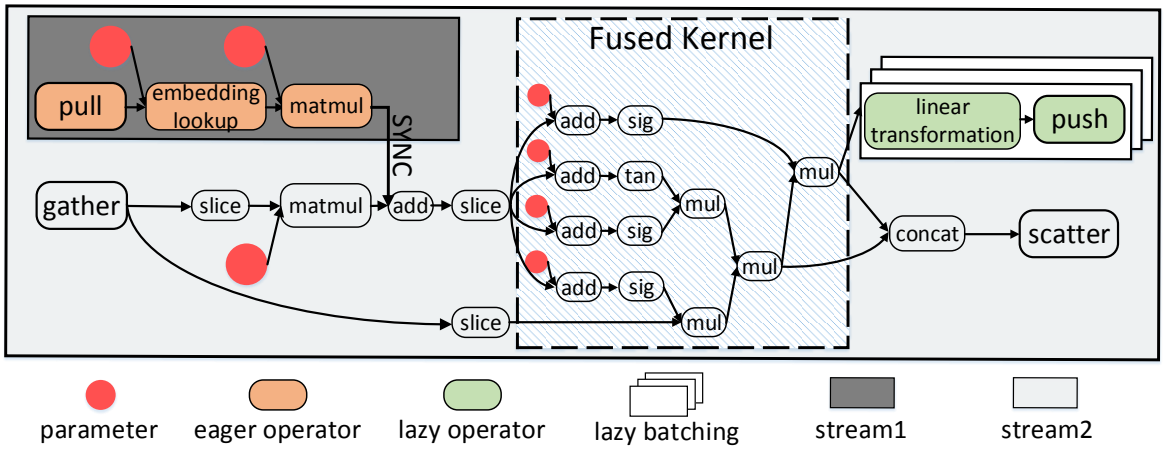

Figure 6.10: The dataflow graph encoded by $\mathcal{F}$ of Tree-LSTM.

6.5.3 Auto-differentiation

Cavs by nature supports auto-differentiation. Given a vertex function $\mathcal{F}$ it derives $\partial\mathcal{F}$ following the auto-differentiation rules: for each math expression such as $s_l = \texttt{op}(s_r)$ in $\mathcal{F}$, Cavs generates a corresponded backward expression in $\partial\mathcal{F}$ as $\nabla s_r = \texttt{grad\_op}(\nabla s_l, s_l, s_r)$. For the four proposed operators, with the memory management strategy described above, we note `scatter` is the backward operator of `gather` in the sense that if `gather` collects inputs from `gatherBuffer` previously written by `scatter` at the forward pass, a `scatter` needs to be performed to write the gradients to the `gatherBuffer` for its dependent vertices to `gather` at the backward pass. Hence, for an expression like $s_l = \texttt{gather}(\texttt{child\_idx})$ in $\mathcal{F}$, Cavs will generate a backward expression $\texttt{scatter}(\nabla s_l)$ in $\partial\mathcal{F}$. Similarly, the gradient operator of `scatter` is `gather`. The same auto-differentiation rule applies for `push` and `pull` as well.

6.5.4 Optimizing Execution Engine

Since Cavs separates out a static dataflow graph encoded by $\mathcal{F}$, we can replace the original $\mathcal{F}$ with an optimized one that runs more efficiently, as long as maintaining correctness. We next described our optimization strategies.

Lazy Batching and Streaming

In addition to batched execution of $\mathcal{F}$, the lazy batching and streaming explore potential parallelisms for a certain group of finer-grained operators in $\mathcal{F}$ or $\partial\mathcal{F}$ called lazy and eager operators. Note that streaming is a borrowed terminology from CUDA which means executing different commands concurrently with respect to each other on different GPU streams. As Cavs' optimizations are agnostic to the low-level hardware, we use streaming interchangeably with multi-threading if the underlying computing hardware is CPU.

Definition 6.5.1 *An operator in $\mathcal{F}$ ($\partial\mathcal{F}$) is a* lazy operator *if at the forward (backward) pass, for $\forall v \in \mathcal{G}, \forall \mathcal{G} \in \{\mathcal{G}_k\}_{k=1}^{K}$, the evaluation of $\mathcal{F}$ ($\partial\mathcal{F}$) at any parent (dependent) vertex of v does not*

rely on the evaluation of $\mathcal{F}$ at v. It is an eager operator *if the evaluation at v does not rely on the evaluation of $\mathcal{F}$ ($\partial\mathcal{F}$) at any dependents (parents) of v.*

Proposition 6.5.2 *Denote $\mathcal{D}_{\mathcal{F}}$ ($\mathcal{D}_{\partial\mathcal{F}}$) as the dataflow graph encoded by $\mathcal{F}$ ($\partial\mathcal{F}$), and $g, s \in \mathcal{D}_{\mathcal{F}}$ ($\mathcal{D}_{\partial\mathcal{F}}$) as nodes of* `gather` *and* `scatter` *operator, respectively. A node that has g as its dependent and is not on any path from g to s is a lazy operator. A node that has s as its ancestor and is not on any path from g to s is an eager operator.*

Figure 6.10 illustrates a forward dataflow graph of the vertex function of Tree-LSTM, with eager and lazy operators colored. A property of them is that their evaluation is not fully subject to the dependency reflected by the input graph $\mathcal{G}$. For instance, the `pull` operator in Figure 6.10 is eager and can be executed in prior – even before $\mathcal{F}$ has been evaluated at the vertices that `gather` tries to interact with; the `push` operator is lazy, so we can defer its execution without impacting the evaluation of $\mathcal{F}$ at parent vertices. Similarly, in $\partial\mathcal{F}$, the gradient derivation for model parameters are mostly lazy – their execution can be deferred as long as the gradients of hidden states are derived and propagated in time. Cavs leverages this property and proposes the lazy batching strategy. It defers the execution of all lazy operators in $\mathcal{F}$ and $\partial\mathcal{F}$ until all batching tasks $\{V_t\}_{t=1}^{T}$ has finished. It then performs a batched execution of these lazy operators over all vertices of $\{\mathcal{G}_k\}_{k=1}^{K}$. These operators includes, but is not limited to, the `push` operator that is doing memory copy, and operators for computing gradients of model parameters. Lazy batching helps exploit more parallelism and significantly reduces kernel launches. Empirically lazy batching brings 20% overall improvement (Section 6.6.4).

To leverage the exhibited parallelization opportunity between eager operators and the operators on the path from `gather` to `scatter` (Figure 6.10), Cavs proposes a streaming strategy that pipelines the execution of these two groups of operators. It allocates two streams, and puts the eager operators on one stream, and the rest (excluding lazy operators) on the other. Hence, independent operators in two streams run in parallel, while for those operators that depend on an eager operator, this dependency is respected by synchronization barriers.

Automatic Kernel Fusion

Given $\mathcal{F}$, before execution, Cavs will run a *fusion detector* [74] to scan its corresponded dataflow graph and report all *fuse-able* subgraphs therein, i.e., all nodes in a fuse-able subgraph can be fused as a single operator that behaves equivalently but takes less execution time (e.g., with fewer kernel launches and I/O, or faster computation). Currently, we only detect groups of directly linked elementwise operators, such as $+$, `sigmoid`, as shown in Figure 6.10, and we use a simple union-find algorithm to detect the largest possible fuse-able subgraphs. Given a fuse-able subgraph, Cavs adopts de facto automatic code generation techniques [44, 170, 172] to generate lower-level kernel implementations. Replacing the original fuse-able subgraphs with fused operators during execution is beneficial in many aspects: (1) it reduces the number of kernel launches; (2) on some devices such as GPUs, kernel fusion transform device memory access into faster device registers access. We empirically report another 20% improvement with automatic kernel fusion (Section 6.6.4).

6.5.5 Implementation

We implement Cavs as a pluggable C++ library that can be integrated with existing DL frameworks to provide or enhance their support for dynamic NNs. We next briefly discuss implementation details. For clarity, we assume the host framework is composed of three major layers (which is the case for most popular frameworks [1, 10, 158]): (1) a frontend that provides device-agnostic symbolic programming interface; (2) an intermediate layer that implements the core execution logic; (3) a backend with device-specific kernels for all provided operators.

Frontend

Cavs provides a base class `GraphSupport` in addition to conventional operators and the four proposed APIs. Users are required to instantiate it by providing a symbolic vertex function $\mathcal{F}$ – therefore an instantiation of `GraphSupport` represents a single dynamic structure. To construct more complex structures (e.g., encoder-decoder LSTM [198], LRCN [52])), users employ `push` and `pull` to connect this dynamic structure to external structures.

Intermediate Layer

At the intermediate layer, Cavs will create a unique scope [1], and generates a small dataflow graph for each instantiation of `GraphSupport`, connecting them appropriately with other parts of the model according to user programs. Cavs implements its core runtime logic at this layer, i.e., the scheduler, the memory management, and the graph execution engine, etc. During execution, the execution engine first analyzes the received dataflow graphs and incorporates described optimization in Section 6.5.4. The scheduler then instructs the system to read training samples and their associates graphs (e.g., adjacency matrices). It starts training by submitting batching tasks to the execution engine and assigning memory accordingly.

Backend

Following common practice [1, 57, 158], Cavs puts device-specific kernel implementations for each supported operator at this layer. Each operator implementation is a function that takes as inputs static tensors and produces static tensors as outputs – therefore the higher-layer logic, i.e., how the computation is scheduled or how the memory is assigned are invincible to this layer. Cavs will reuse the native operator implementations from the host framework, while it provides optimized implementations for the four proposed primitives (`gather`, `scatter`, `pull`, `push`). Specifically, `gather` and `pull` index different slices of a tensor and puts them together continuously on memory; `scatter` and `push` by contrast splits a tensor along its batch dimension, and copy different slices to different places. Cavs implements a customized memcpy kernel for there four operators so that copying multiple slices from (or to) different places is performed within one kernel, to further reduce kernel launches.

Distributed Execution

While Cavs's implementations are focused on improving the efficiency on a single node, they are compatible with most data-parallel distributed systems for deep learning [1, 40, 241], and can also benefit distributed execution on multiple nodes.

6.6 Evaluation

In this section, we evaluate Cavs on multiple NNs and datasets, obtaining the following major findings:

- Cavs has little overhead: on static NNs, Cavs demonstrates equal performance on training and inference with other systems; On several NNs with notably difficult-to-batch structures, Cavs outperforms all existing frameworks by a large margin.
- We confirm the graph construction overhead is substantial in both Fold [136] and dynamic declaration [158], while Cavs bypasses it by loading input graphs through I/O.
- We verify the effectiveness of our proposed design and optimization via ablation studies, and discuss Cavs' advantages over other DL systems for dynamic dataflow graphs.

6.6.1 Experiment Setup

Environment

We perform all experiments in this paper on a single machine with an NVIDIA Titan X (GM200) GPU, a 16-core CPU, and CUDA v8.0 and cuDNN v6 installed. As modern DL models are mostly trained using GPUs, we focus our evaluation on GPUs, but note Cavs' design and implementation do not rely on a specific type of device. We mainly compare Cavs to TensorFlow v1.2 [1] with XLA [66] and its variant Fold [136], PyTorch v0.3.0 [56], and DyNet v2.0 [158] with autobatching [159], as they have reported better performance than other frameworks [27, 205] on dynamic NNs. We focus on metrics for system performance, e.g., time to scan one epoch of data. Cavs produces exactly the same numerical results with other frameworks, hence the same per-epoch convergence.

Models and Dataset

We experiment on the following models with increasing difficulty to batch:

- `Fixed-LSTM` language model (LM): a static sequence LSTM with fixed steps for language modeling [197, 198, 240]. We train it using the PTB dataset[6] that contains over 10K different words. We set the number of steps as 64, i.e., at each iteration of training, the model takes a 64-word sentence from the training corpus, and predicts the next word of each word therein. Obviously, the computation can be by nature batched easily, as each sentence has exactly the same size.

[6] http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz.

- `Var-LSTM` LM: that accepts variable-length inputs. At each iteration the model takes a batch of natural sentences with different length from PTB, and predicts the next words;
- `Tree-FC`: the benchmarking model used in [136] with a single fully-connected layer as its cell function. Following the same setting in [136], we train it over synthetic samples generated by their code[7] – each sample is associated with a complete binary tree with 256 leaves (therefore 511 vertices per graph);
- `Tree-LSTM`: a family of dynamic NNs widely adopted for text analysis [130, 210]. We implement the binary child-sum Tree-LSTM in [203], and train it as a sentiment classifier using Stanford sentiment treebank (SST) dataset [194]. The dataset contains 8544 training sentences, each associated with a human annotated grammar tree, and the longest one has 54 words.

6.6.2 Overall Performance

Fixed-LSTM

We first verify the viability of our design on the easiest-to-batch case: `Fixed-LSTM` language model. We compare Cavs to three strong baselines:

- `CuDNN` [31]: a CuDNN-based fixed-step sequence LSTM, which is highly optimized by NVIDIA using handcrafted kernels and stands as the best performed implementation on NVIDIA GPUs;
- `TF`: the official implementation of `Fixed-LSTM` LM in TensorFlow repository[8] based on static declaration;
- `DyNet`: we implement a 64-step LSTM in DyNet based on dynamic declaration – we declare a dataflow graph per sample, and train with the autobatching [159] enabled;
- `Cavs` with batching policy, and all input samples have a same input graph – a 64-node chain.

We train the model to converge, and report the average time per epoch in Figure 6.11(a)(e), where in (a) we fix the hidden size h of the LSTM unit as 512 and vary the batch size bs, and in (e) we fix $bs = 64$ and vary h. Empirically, CuDNN performs best in all cases, but note it is highly inflexible. Cavs performs slightly better than `TF` in various settings, verifying that our system has little overhead handling fully static graphs, though it is specialized for dynamic ones. We also conclude from Figure 6.11 that batching is essential for GPU-based DL: $bs = 128$ is nearly one order of magnitude faster than $bs = 1$ regardless of used frameworks. For Cavs, the batching policy is 1.7x, 3.8x, 7.0x, 12x, 15x, 25x, 36x faster than non-batched at $bs = 2, 4, 8, 16, 32, 64, 128$, respectively.

[7] https://github.com/tensorflow/fold/tree/master/tensorflow_fold/loom/benchmarks.

[8] https://github.com/tensorflow/models/blob/master/tutorials/rnn/ptb/ptb_word_lm.py.

Var-LSTM

Next, we experiment with `Var-LSTM`, the most commonly used RNN for variable-length sequences. We compare the following three implementations (CuDNN-based LSTM cannot handle variable-length inputs):

- `TF`: an official TensorFlow implementation based on the dynamic unroll approach described in Section 6.7;
- `DyNet`: an official implementation from DyNet benchmark repository based on dynamic declaration[9];
- `Cavs`: where each input sentence is associated with a chain graph that has number of vertices equal to the number of words.

We vary h and bs, and report the results in Figure 6.11(b)(f), respectively. Although all three systems perform batched computation in different ways, `Cavs` is consistently 2-3 times faster than `TF`, and outperforms `DyNet` by a large margin. Compared to `TF`, `Cavs` saves computational resources. `TF` dynamically unrolls the LSTM unit according to the longest sentence in the current batch, but it cannot prevent unnecessary computation for those sentences that are shorter than the longest one.

Tree-FC

We then turn to `Tree-FC`, a dynamic model for benchmarking. Since vanilla TensorFlow is unable to batch its computation, we compare `Cavs` to (1) `DyNet` and (2) `Fold`, a specialized library built upon TensorFlow for dynamic NNs, with a depth-based dynamic batching strategy. To enable the batching, it however needs to preprocess the input graphs, translate them into intermediate representations and pass them to lower-level TensorFlow control flow engine for execution. We report the results in Figure 6.11(c)(g) with varying bs and h, respectively. For all systems, we allocate a single CPU thread for graph preprocessing or construction. `Cavs` shows at least an order of magnitude speedups than `Fold` and `DyNet` at $h \leq 512$. Because the size of the synthetic trees is large, one major advantage of `Cavs` over them is the alleviation of graph preprocessing/construction overhead. With a single CPU thread, `Fold` takes even more time on graph preprocessing than computation (Section 6.6.4).

Tree-LSTM

Finally, we compare three frameworks on `Tree-LSTM` in Figure 6.11(d)(h): `Cavs` is 8-10x faster than `Fold`, and consistently outperforms `DyNet`. One difference in this experiment is that we allocate as many CPU threads as possible (32 on our machine) to accelerate graph preprocessing for `Fold`, otherwise it will take much longer time. Further, we note `DyNet` performs much better here than on `Tree-FC`, as the size of the input graphs in SST (maximally 54 leaves) is much smaller than the synthetic ones (256 leaves each) in `Tree-FC` experiments. We observe `DyNet` needs more time on graph construction for large input graphs, and `DyNet`'s dynamic

[9] https://github.com/neulab/dynet-benchmark.

batching is less effective on larger input graphs, as it has to perform frequent memory checks to support its dynamic batching, which we will discuss in Section 6.6.4. We also compare Cavs with PyTorch – its per-epoch time on Tree-LSTM is 542s, 290x slower than Cavs when the batch size is 256. Compared to other systems, PyTorch cannot batch the execution of dynamic NNs.

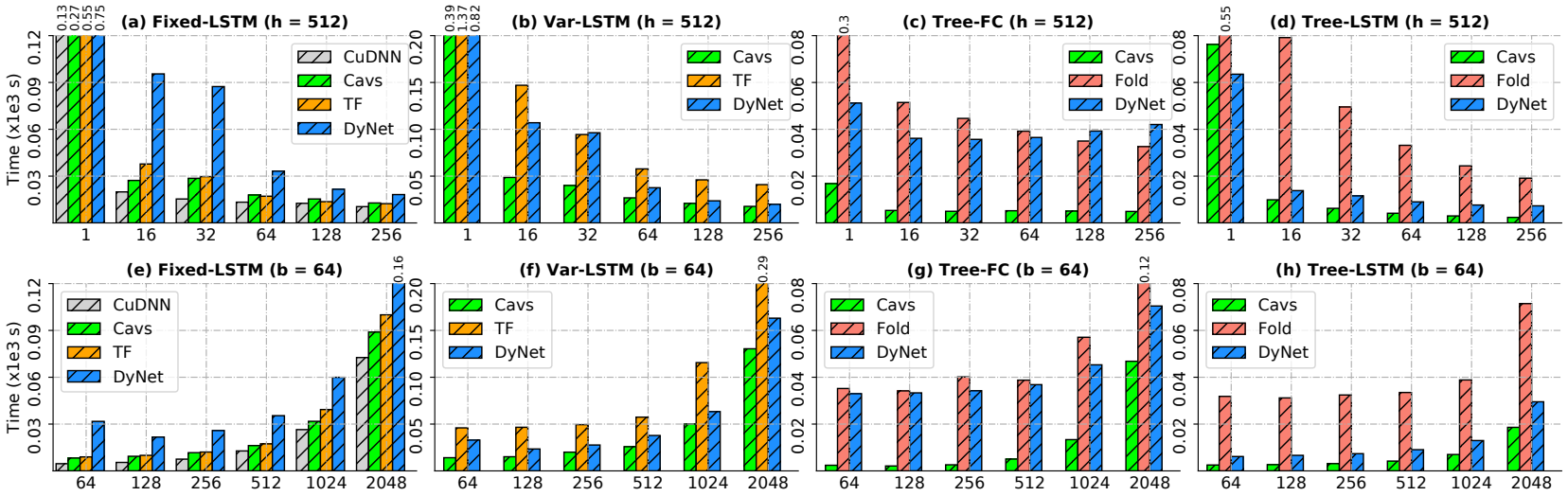

Figure 6.11: Comparing five systems on the averaged time to finish one epoch of training on four models: `Fixed-LSTM`, `Var-LSTM`, `Tree-FC` and `Tree-LSTM`. In (a)-(d) we fix the hidden size h and vary the batch size bs, while in (e)-(h) we fix bs and vary h.

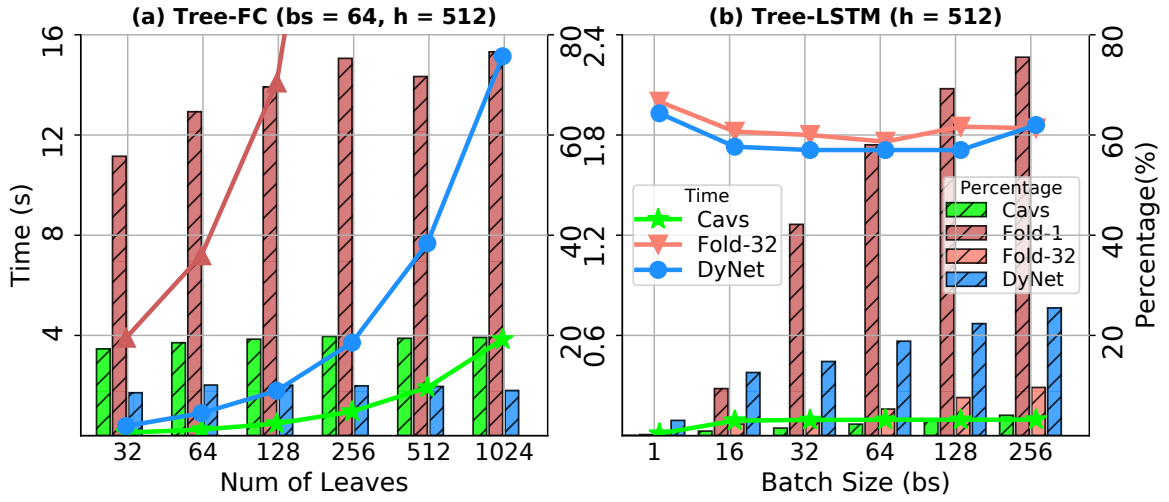

Figure 6.12: The averaged graph construction overhead per epoch when training (a) `Tree-FC` with different size of input graphs (b) `Tree-LSTM` with different batch size. The curves show absolute time in second (left y-axis), and the bar graphs show its percentage of the overall time (right y-axis).

6.6.3 Graph Construction

In this section, we investigate the graph construction overhead in Fold and DyNet. To batch the computation of different graphs, Fold analyzes the input graphs to recognize batch-able dynamic operations, then translates them into intermediate instructions, with which, TensorFlow generates appropriate control flow graphs for evaluation – we will treat the overhead caused in both steps as Fold's graph construction overhead. DyNet, as a typical dynamic declaration framework, has to construct as many dataflow graphs as the number of samples. Though DyNet has optimized its graph construction to make it lightweight, the overhead still grows with the training set and the size of input graphs. By contrast, Cavs takes constant time to construct a small dataflow graph encoded by $\mathcal{F}$, then reads input graphs through I/O. To quantify the overhead, we separate the graph construction from computation, and visualize in Figure 6.12(a) how it changes with the average number of leaves (graph size) of input graphs on training `Tree-FC`, with fixed $bs = 64, h = 512$. We compare (1) `Cavs` (2) `Fold-1` which is Fold with one graph processing thread and (3) `DyNet`. We plot for one epoch, both the (averaged) absolute time for graph construction and it percentage of the overall time. Clearly we find that all three systems take increasingly more time when the size of the input graphs grows, but Cavs, which loads graphs through I/O, causes the least overhead at all settings. In terms of the relative time, `Fold` unfortunately wastes 50% at 32 leaves, and 80% when the tree has 1024 leaves, while `DyNet` and `Cavs` take only 10% and 20%, respectively.

We also wonder how the overhead is related with batch size when there is fixed computational workload. We report in Figure 6.12(b) the same metrics when training `Tree-LSTM` with varying bs. We add another baseline `Fold-32` with 32 threads for Fold's graph preprocessing. As `Fold-1` takes much longer time than others, we report its time at $bs = 1, 16, 32, 64, 128, 256$ here (instead of showing in Figure 6.12): 1.1, 7.14, 31.35, 40.1, 46.13, 48.77. Except $bs =$

# leaves	time (s)	Speedup	bs	time (s)	Speedup
32	**0.6** / 3.1 / 4.1	**5.4** / 7.1	1	76 / 550 / **62**	**7.2** / 0.8
64	**1.1** / 3.9 / 8.0	3.7 / 7.5	16	**9.8** / 69 / 12	7.0 / 1.2
128	**2** / 6.2 / 16	3.0 / 7.9	32	**6.2** / 43 / 9.9	7.0 / 1.6
256	**4** / 10.6 / 33.7	2.7 / 8.7	64	**4.1** / 29 / 7.4	**7.2** / 1.8
512	**8** / 18.5 / 70.6	2.3 / 8.9	128	**2.9** / 20.5 / 5.9	7.1 / 2.0
1024	**16** / 32 / 153	2.1 / **9.7**	256	**2.3** / 15.8 / 5.4	7.0 / **2.4**

Table 6.2: The averaged computation time (`Cavs`/`Fold`/`DyNet`) and the speedup (`Cavs` vs `Fold`/`DyNet`) for training one epoch on `Tree-FC` with varying size of the input trees (left part), and on `Tree-LSTM` with varying batch size (right part).

1, all three systems (except `Fold-1`) take almost constant time for graph construction in one epoch, regardless of bs, while `Fold-32` and `DyNet` take similar time, but `Cavs` takes 20x less. Nevertheless, at the percentage scale, increasing bs makes this overhead more prominent, because larger batch size yields improved computational efficiency, therefore less time to finish one epoch. This, from one perspective, reflects that the graph construction is a main obstacle that grows with the number of training samples and prevents the efficient training of dynamic NNs in existing frameworks, while Cavs successfully overcomes this barrier.

Apart from the graph construction we report in Table 6.2 the computation-only time. `Cavs` shows maximally 5.4x/9.7x and 7.2x/2.4x speedups over `Fold`/`DyNet` on `Tree-FC` and `Tree-LSTM`, respectively. The advantages stem from two main sources: an optimized graph execution engine, and a better-suited memory management strategy, which we investigate next.

6.6.4 Graph Optimizations

Graph Execution Engine

To reveal how much each optimization in Section 6.5.4 contributes to the final performance, we disable lazy batching, fusion and streaming in Cavs and set this configuration as a baseline (speedup = 1). We then turn on one optimization at a time and record how much speedup it brings. We train `Fixed-LSTM` and `Tree-LSTM`, and report the averaged speedups one computation-only time in one epoch over the baseline configuration in Figure 6.13, with $bs = 64$ but varying h. Lazy batching and fusion consistently deliver nontrivial improvement – lazy batching is more beneficial with a larger h while fusion is more effective at smaller h, which are expected: lazy batching mainly parallelizes matrix-wise operations (e.g., `matmul`) commonly with $O(h^2)$ or higher complexity, while fusion mostly works on elementwise operations with $O(h)$ complexity [73].

Streaming, compared to the other strategies, is less effective on `Tree-LSTM` than on `Fixed-LSTM`, as we have found the depth of the input trees in SST exhibit high variance, i.e., some trees are much deeper than others. In this case, many batching tasks only have one vertex to be evaluated. The computation is highly fragmented and the efficiency is bounded by kernel launching latency. Lazy batching and fusion still help as they both reduce kernel launches (Section 6.5.4). Streaming, which tries to pipeline multiple kernels, can hardly yield obvious improvement.

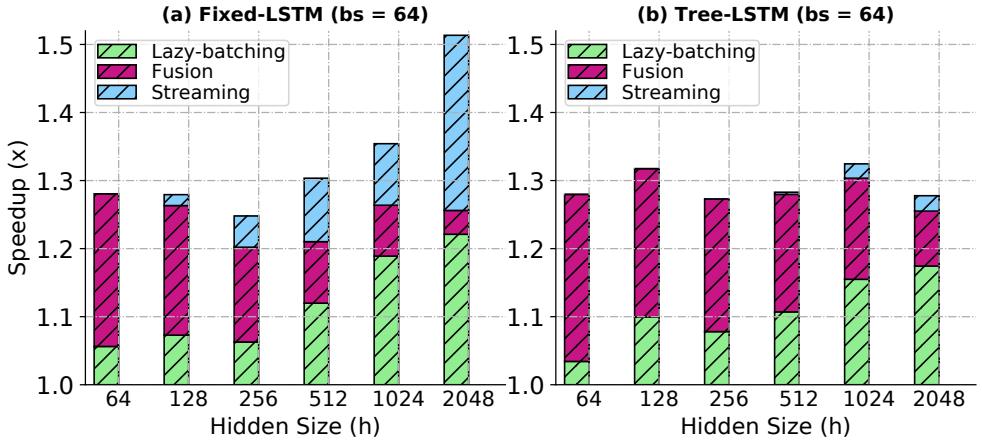

Figure 6.13: Improvement of each optimization strategy on execution engine over a baseline configuration (speedup = 1).

bs	Memory operations (s) (*Cavs / DyNet*)		Computation (s) (*Cavs / DyNet*)	
	Train	Inference	Train	Inference
16	**1.14** / 1.33	**0.6** / 1.33	**9.8** / 12	**2.9** / 8.53
32	**0.67** / 0.87	**0.35** / 0.87	**6.1** / 9.8	**1.9** / 5.35
64	**0.39** / 0.6	**0.21** / 0.6	**4.0** / 7.4	**1.3** / 3.48
128	**0.25** / 0.44	**0.13** / 0.44	**2.9** / 5.9	**0.97** / 2.52
256	**0.17** / 0.44	**0.09** / 0.44	**2.3** / 5.4	**0.77** / 2.58

Table 6.3: Breakdowns of average time per epoch on memory-related operations and computation, comparing Cavs to DyNet on training and inference of `Tree-LSTM` with varying bs.

Memory Management

Cavs' performance advantage also credits to its memory management that reduces memory movements while guarantees continuity. Quantitatively, it is difficult to compare Cavs to Fold, as Fold relies on TensorFlow where memory management is highly coupled with other system aspects. Qualitatively, we find Cavs requires less memory movement (e.g., memcpy) during dynamic batching. Built upon the `tf_while` operator, whenever Fold performs depth-based batching at depth d, it has to move all the contents of nodes in the dataflow graphs at depth $d-1$ to a desired location, as the control flow does not support cross-depth memory indexing. This results in redundant memcpy, especially when the graphs are highly skewed. By contrast, Cavs only copies contents that are necessary to the batching task. DyNet has a specialized memory management strategy for dynamic NNs. Compared to Cavs, it however suffers substantial overhead caused by repeated checks of the memory continuity – whenever DyNet wants to batch operators with same signatures, it checks whether their inputs are continuous on memory [159]. The checking overhead increases with bs and is more prominent on GPUs. Thanks to the simplicity of both systems, we are able to profile the memory-related overhead during both training and inference, and separate it from computation. We compare them on `TreeLSTM`, and report the breakdown time per epoch in Table 6.3 under different bs. We observe the improvement

is significant (2x - 3x) at larger bs, especially during inference where DyNet has its continuity checks concentrated.

6.7 Additional Related Work

Graph Execution Optimization

Optimizing the execution of DL dataflow graphs comes in mainly two ways: better operator implementations or optimizing the execution of (sub-)graphs. As Cavs is implemented as a plug-in to enhance existing frameworks, it benefits from any improved implementations of specific operators (e.g., cuDNN) [31, 69, 72, 90]. In addition, Cavs has optimized implementations for its proposed four primitives (`gather`/`scatter`/`pull`/`push`). At the graph level, a variety of well-developed techniques from other areas, such as kernel fusion, common subexpression elimination, and constant folding, have been adapted and applied on speeding the computation of DL dataflow graphs [1, 27, 57, 66]. They are usually incorporated after the graph declaration, but before the execution, so that the actual computation is conducted on an optimized graph other than the original one. However, these graph optimizations are less beneficial in dynamic declaration, in which the graph changes with the sample, and needs to be re-processed and re-optimized every iteration, and may cause substantial overhead. On the contrary, Cavs separates the static vertex function from the dynamic-varying input graph, so it benefits from most of the aforementioned optimizations, as we have shown in Section 6.6.4. We draw insights from these strategies and reflect them in Cavs' execution engine. We further propose lazy batching and streaming to exploit more parallelism exposed by our programming model.

Vertex-centric Models

The vertex-centric programming model has been extensively developed in the area of graph computing [25, 61, 139, 196]. Cavs draws insights from the GAS model [61], but faces totally different challenges in system and interface design, such as expressiveness, scheduling for batched execution of different graphs, guaranteeing the memory continuity, etc., as we have discussed in Section 6.4.2.

Dynamic Neural Network Frameworks

Tensorflow Fold [136] also performs dynamic batching for dynamic dataflow graphs. Fold turns dynamic dataflow graphs into a static control flow graph to enable batched execution, but introduces a complicated functional programming-like languages and a large graph preprocessing overhead. There are also some "imperative" frameworks, such as PyTorch [56] and Chainer [205] that allow users to construct dynamic NNs but performs instant evaluation of each user expression. As model construction and execution are coupled, they are usually difficult to perform dynamic batching or graph optimization. Overall, they are still far from efficient when handling dynamic NNs.

Chapter 7

Distributed Parallelization

A substantial number of software systems, including those in this thesis, have been created to parallelize training algorithms over multiple CPUs or GPUs distributed across nodes. Ideally, such parallel ML systems should democratize training of large models on large datasets, by reducing training times to allow quicker ML development. In practice, the majority of papers at ML conferences and journals (such as NeurIPS, ICML, KDD, AAAI, JMLR, etc.) or models trained in production report experiments taken on single workstations. What are the opportunities to increase the adoption of parallel ML systems? We believe these opportunities are:

1. **ML systems either require high expertise, or are narrowly-specialized**: they either require low-level programming to specify how to partition distributed operations [1, 153, 165] (Section 2.4.2), or are effective only on a limited set of models [26, 186];

2. **Multiple parallel ML strategies can be composed to achieve near-ideal performance, but we need new interfaces and systems for composition:** the research in Part I of this thesis and by others [107, 168] has shown that different parallel ML strategies can be applied to different parts of an ML model, in order to achieve near-linear (i.e., ideal) increases in throughput [245]. Composing multiple strategies is technically demanding, and new systems and programming interfaces are needed to increase accessibility to ML practitioners;

3. **ML systems should be composable, to facilitate rapid adaptation to unseen ML models:** ML systems are usually monolithic software projects, and it is practically impossible to apply multiple parallel ML strategies to the same ML model. Decomposing these ML systems into composable units will allow fast experimentation on newly developed ML models, which will likely require new parallel ML strategies.

4. **Distributed-parallel ML strategies are hard to choose**: different distributed strategies exhibit sharp differences in performance when applied to different ML model building blocks, and users must select the right strategy via domain knowledge or trial-and-error. For some ML models, none of the available strategies may be a good fit at all;

5. **Parallel performance tuning requires deep expertise**: a training system's optimal tuning parameters (e.g., number of partitions of a variable, number of collective reduction groups,

staleness) will change when going from one model to the other, requiring trial-and-error to discover.

The chapter aims to address the aforementioned challenges 1-3, and defers a solution of Challenge 4-5 to the last part (Chapter 8) of this thesis.

Generalized from the thesis methodology adaptive parallelism presented in Part I, in this chapter, we introduce a concept, *composibility* of ML parallelisms. In the context of data-parallel training, we demonstrate an end-to-end compiler system, AutoDist for distributing training – AutoDist allows mechanically composing model- and resource-adaptive distributed synchronization strategies from a few base, atomic synchronization aspects.

7.1 Introduction

Existing parallel DL systems, including those presented in previous chapters, have explored various aspects to improve the performance of synchronization, e.g., communication architectures [107, 127, 186], message computation and augmentation [76, 132], bridging models [38, 83], etc. These systems have demonstrated improved training speed or scalability on specifically-chosen DL "model architectures", such as convolutional [81], fully-connected, but do not always generalize well to other architectures, or even models that incorporate multiple architectures [51]. However, from a practical usability standpoint, users are expected to choose and use the best system for their model of interest, which often requires both ML *and* systems savvy; or they may find that no system provides adequate parallel training performance for novel and complex models with multiple architectures, or substantial additional efforts are needed in tuning or augmenting existing systems to extrapolate to a new model.

To overcome the lack of flexibility of existing monolithic DL systems with built-in distributed strategies, we posit that such systems need to adopt a compiler-like architecture, and the principle of *composability* – different aspects of performance-improving strategies, such as different choices of communication topologies (e.g., master-slave, peer-to-peer), update aggregation structures (ring, tree, hierarchy), and message encoding (e.g., "regular" stochastic gradients, sufficient factors [226]) can be mechanically selected and recombined to adapt to different building blocks of DL models [107, 245]. We refer to this approach as *strategy composition* for DL systems. The composition opens space to improve the performance of synchronization.

Exploiting the structures of DL models for optimal strategy composition. Because elements of a composite distributed-system strategy may apply deferentially to different building-blocks such as layers of a DL model, e.g., a parameter server [32, 83, 127] for sparse gradients, and collective allreduce [68, 186] for convolutional layers, we can exploit the structures of given algorithms or design novel algorithms that co-optimize multiple aspects and assign to every model building block, so to provide the greatest parallel throughput.

Improving DL training systems with task-adaptive composability. Existing distributed DL training systems are usually built around a single or only a few parallelization aspects, e.g., collective allreduce for Horovod [186]. Our experiments show that this approach is over-specialized, in the sense that no parallelization aspect we studied performs well across *all* DL models. The architecture of DL training systems is often tightly coupled with one aspect; thus, improving

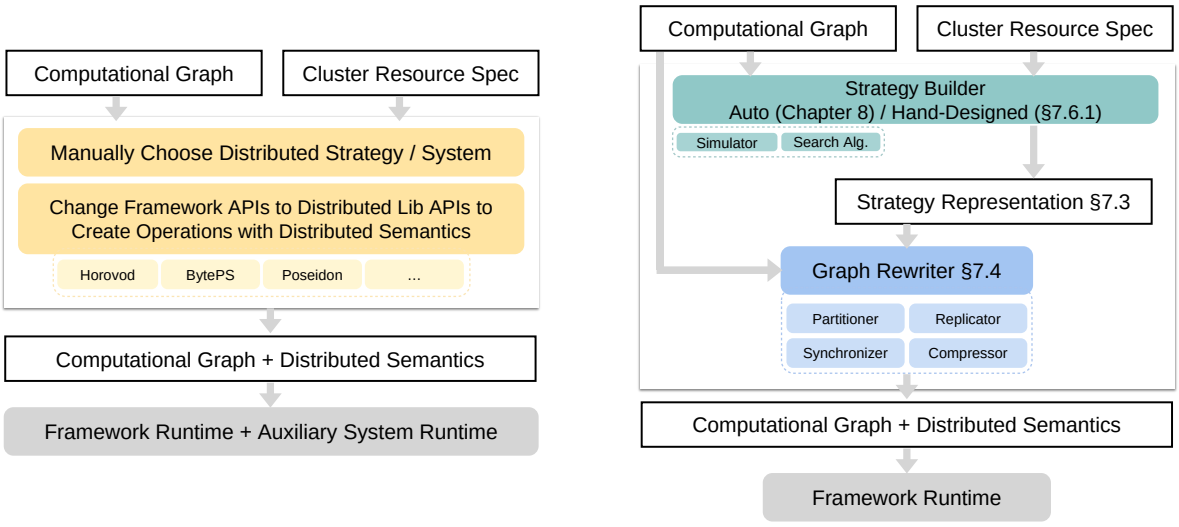

Figure 7.1: How **(Left)** prior DL training systems, and **(Right)** a composable compiler-like system differ in parallelizing an ML model.

these systems' performance on new models requires low-level modifications to their implementations. We see a need to break the design of DL training systems into composable units based on different aspects. We believe a modular design, based on these units, can help DL training systems adapt quickly to newly-emerging DL models.

Developing systems that achieve high performance on DL models "out-of-the-box". State-of-the-art DL models, such as BERT [51], can incorporate multiple layer types of varying sizes connected in complex ways. In Section 7.6.3, we show that applying DL parallel training systems to complex models "out-of-the-box" (i.e., with default recommended settings) often results in lower-than-expected performance, when compared to reported results on typical benchmark models. While it could be argued that a skilled user can tune these systems and restore most of the "missing" performance, we think a better alternative is to develop automatic systems, which take advantage of strategy tuning algorithms to achieve high performance out-of-the-box. This would make parallel DL training more attractive and easier to use for a wider audience.

To explore these opportunities, we propose to design a composable and automated system for distributed parallel DL training. Most prior systems are built upon one (or a few) parallelization aspects, and apply those uniformly across all parts of a DL model (Figure 7.1(Left)). In contrast, we aim to pursue a compiler system that automatically generates a distributed strategy for a given model and resource specification (Figure 7.1(Right)), by selecting from a pool of atomic parallelization aspects that can come from prior systems literature, and be extended as new research emerges. Such a system would exhibit the following obvious advantages:

Unified representation. It possesses a *unified representation for synchronization* that jointly encodes multiple contributing factors to distribution performance (Section 7.3). With the representation, it goes through a compilation process: it generates a suitable strategy conditioned on the user model and resource specification, taking into account the statistical properties of model building blocks, as the thesis methodology *adaptive parallelism* suggests. The strategy is

explicitly represented, executable by low-level systems, and transferable across users.

Composable system. The system allows strategy composition. Each aspect of parallelization maps to a primitive distribution function implementation (a.k.a. *kernel*) in the backend system. A complete distribution strategy is hence materialized by composing multiple primitive kernels, based on the strategy representation. This approach isolates strategy prototyping from low-level distributed systems, allows composition, assessment and execution of complex distribution strategies via a unified interface, and is extensible to emerging parallelization techniques.

Automatic parallelization. The introduced representation spans a combinatorial space enclosing all possible strategies (i.e., assignments of specialized parallelization aspects to model building blocks). It allows to build a end-to-end strategy auto-optimization pipeline (presented in Chapter 8) to efficiently optimize strategies against model and resources, as presented in Equation 1.1, which not only improves parallel performance, but is also an added convenience for users.

Following this discussion, we next ground our discussion and development in the context of data-parallel training, and derive the representations for *synchronization strategies*, and corresponded compiler system – AutoDist.

7.2 Overview

7.2.1 Data-parallel Distributed Training and Synchronization Strategy

Throughout this chapter, we will adopt the dataflow graph as the principled representation for ML models. We refer to the dataflow graph of DL models and its computational function using $\{(\mathcal{O}, \mathcal{V}_\Theta), \mathcal{E}\}$, where the node is either a computational operation $o \in \mathcal{O}$ or a stateful variable $v \in \mathcal{V}_\Theta$, and edges ($\mathcal{E}$) are tensors. In this section, we focus on *data-parallel* training, which parallelizes $\mathcal{G}$ over multiple devices. Simply, we instantiate Equation 2.2 using dataflow graph notations as

$$\mathcal{V}_\Theta^{(t+1)} \leftarrow \mathcal{V}_\Theta^{(t)} + \epsilon \sum_{p=1}^{P} \nabla_{\mathcal{G}_p}(\mathcal{V}_\Theta^{(t)}, \mathcal{X}_p), \tag{7.1}$$

where the original single-device graph $\mathcal{G}$ is replicated into P replicas $\{\mathcal{G}_p\}_{p=1}^{P}$, and each replica $\mathcal{G}_p$ performs computation on the pth device over its independent split of training data $\mathcal{X}_p$. All replicas use a consistent set of trainable parameters $\mathcal{V}_\Theta$. In addition to the forward-backward computation, the core step making this possible is *synchronization* (mapping to $\sum$ in the Equation 7.1), which collects parameter updates from multiple devices and maintains consistency on $\mathcal{V}_\Theta$.

The synchronization step is expensive when training large models over many nodes [32, 83, 128]. Accordingly, many systems are developed to improve the performance of this synchronization step, each with its variety of *synchronization aspects*. Examples include communication architectures, topologies and bridging models [107, 186, 221, 245], partitioning and placement strategies [99, 148, 216], message encoding schemes [132], among others. However, multiple aspects interact with each other, together with complex structures in $\mathcal{G}$ and $\mathcal{V}_\Theta$, dictate their performance against different models and cluster specifications, thereby leading to an intricate

space of synchronization strategy considerations. Existing strategies and systems usually treat $\mathcal{G}$ and $\mathcal{V}_\Theta$ as an entirety, and make specific *instance choices* for each aspect and ignore such intricacy. This presents opportunities that, though a unified strategy representation and a composable backend, the synchronization performance could by improved by co-optimizing multiples synchronization aspects against each building block of the model.

To formalize a generic representation for synchronization (Section 7.3), as a first step, we break existing instance-based synchronization methods into atomic units and re-catalog them into the following aspects:

- *Replication* of the model graph $\mathcal{G}$: how and which device to replicate the model graph for data parallelism.
- *Partitioning and placement* of variables $\mathcal{V}_\Theta$: partitioning, sharing and placement mechanisms of trainable variables.
- *Synchronization architecture and aggregation structures*: How to set up a network topology for message synchronization (e.g., bipartite parameter servers [83], P2P broadcast [226], or fully-connected all-reduce [186]). How and where to aggregate updates $\nabla_{\mathcal{G}_p}$, e.g., full summation at a central device [83], partial summation across a tree [106].
- *Message encoding*: How to encode and decode messages $\nabla_{\mathcal{G}_p}$ before communication, e.g., various forms of gradient updates, compression, sufficient factors [226], etc.
- *Bridging models*: How to bridge computation (of $\sum_{p=1}^{P} \nabla_{\mathcal{G}_p}$) and communication, e.g., with various synchronous, asynchronous, or bounded-asynchronous methods [42, 83, 221].

Upon above, we denote $\mathcal{S}$ as an expression (developed in Section 7.3) that instructs how $\mathcal{G}$ should be synchronized when distributed on the cluster. We call it a *synchronization strategy*. It spans multiple synchronization aspects cataloged above, and is model-dependent and resource-dependent. Hence, a valid and complete $\mathcal{S}$ would instantiate proper values for each aspect. We use $\mathcal{D}$ to describe the specifications of the target cluster. It includes information such as the number of nodes, their addresses, number of CPUs and GPUs per node, and their interconnection information (e.g., Ethernet bandwidth).

7.2.2 AutoDist Workflow

The workflow of AutoDist resembles that of a conventional compiler. AutoDist takes the dataflow graph $\mathcal{G}$ and the cluster specification $\mathcal{D}$ as inputs. It makes an appropriate choice for each aspect in its corresponded representation space (Section 7.3), w.r.t. $\mathcal{G}$ and $\mathcal{D}$, considering the computational or statistical properties present in $\mathcal{G}$ and the resource condition in $\mathcal{D}$. It then composes them into a complete strategy expression $\mathcal{S}$. Figure 7.2 illustrates such an example. AutoDist decouples the strategy generation from specific system implementation. The generation of $\mathcal{S}$, depending on user choices, is done either via *hand-designed strategy builders*, or an *AutoStrategyBuilder* that derives the optimal $\mathcal{S}$ via automatic strategy optimization (Chapter 8). For instance, we can have a strategy builder for Horovod [186] that takes $\mathcal{G}$ and $\mathcal{D}$ to build a strategy with collective allreduce as the communication architecture, ring as the reduction structure.

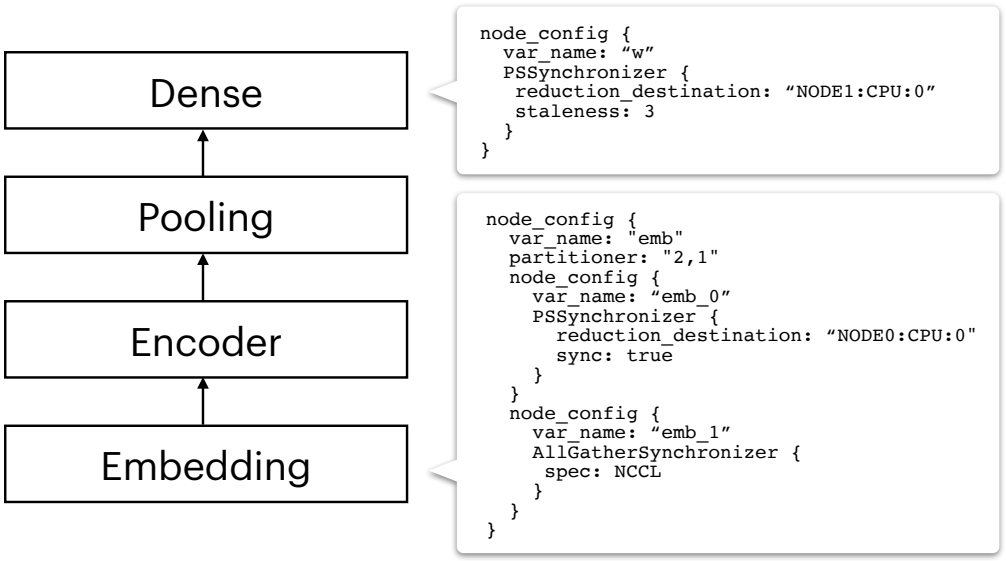

Figure 7.2: An example of how the synchronization strategy is composed based on the DL model $\mathcal{G}$ and resource information in $\mathcal{D}$.

The expression is then broadcasted across all corresponded workers within $\mathcal{S}$. An AutoDist process then on each worker is invoked locally to transform the graph $\mathcal{G}$ based on $\mathcal{S}$, into a distributed graph. AutoDist backend implements a library of primitive and generic graph transformation functions, which we will call *graph transformation kernels*. By mapping each aspect in $\mathcal{S}$ with its graph transformation kernel, the transformation happens in a composable way, by applying a series of kernels instructed in $\mathcal{S}$. Upon the completion of graph transformation, AutoDist launches the specific ML frameworks with the desired distributed dataflow graph to start distributed training.

We next develop each component of $\mathcal{S}$ in detail.

7.3 Synchronization Strategy Representation

The synchronization expression $\mathcal{S}$ defines the space for high-level strategy generation and serves as a composition instruction for low-level strategy-based graph rewriting (Section 7.4). We define the synchronization representation with two levels of semantics: *graph level* that expresses how $\mathcal{G}$ will be transformed globally to fit a parallel environment, and *node level* that specifies per-variable synchronization configurations. Figure 7.3(a) illustrates an example of strategy expression.

7.3.1 Graph-level Representation

At the graph level, we introduce the *graph config* to include a single semantic, *replication*, to capture the aspect of how $\mathcal{G}$ should be replicated across multiple devices in data-parallel envi-

ronment. We use a tuple $(\mathcal{G}, \{d_i\}_{i=1}^m)$ to notate that $\mathcal{G}$ will be replicated on a set of destination devices $\{d_i\}_{i=1}^m \subset \mathcal{D}$.

7.3.2 Node-level Representation

At the node level, for each trainable variable $v_i \in \mathcal{V}_\Theta$, we introduce a *node config* to express its variable-specific synchronization setup, specifying the following aspects.

Variable Partitioning

Operation partitioning in dataflow graphs is a widely explored topic in parallel DL systems [99, 188, 216]. As a complement to existing works that propose a general partitioning mechanism for arbitrary operations, AutoDist only concerns the partitioning of nodes that participate in data-parallel synchronization, i.e., nodes in $\mathcal{V}_\Theta$. Since a variable node could be partitioned directly along any axis of its tensor buffers (unlike an arbitrary computational operation), we represent the partitioning of a variable v_i simply as a vector $\boldsymbol{p}_i = [p_i^j]_{j=1}^{k_i}$, with k_i equals the number of dimensions of v_i, and p_i^j an integer representing how many partitions are generated along the jth axis of v_i. We thus obtain $\prod_{j=1}^{k^i} p_i^j$ *partitioned variables* of v_i after the partitioning. The representation allows to express a rich set of optimizations in existing synchronization systems, such as various types of sharding strategies for load balancing in PS [127, 168], but is much more flexible as it is variable-specific.

Variable Sharing and Placement

A variable, partitioned or unpartitioned, can be shared (e.g., PS), or replicated across multiple devices (e.g., collective communication) for synchronization. We bring in *placement* that decides where the variable locates at. When it is instantiated as a set of devices $\{d_i\}_{i=1}^m \subset \mathcal{D}$, the variable is replicated across all $\{d_i\}_{i=1}^m$. A single device value $v_i : d_i$ then indicates v_i is placed on d_i and the computation or synchronization happening on different devices have shared access to v_i. While device placement can be extended onto all nodes in $\mathcal{G}$ [148], AutoDist only models the placements of $\mathcal{V}_\Theta$, as the rest of nodes in $\mathcal{G}$ are replicated across $\mathcal{D}$ in data-parallel training, captured by the introduced replication semantic in graph config.

Synchronization

Data-parallel synchronization requires collecting and applying variable updates $\nabla\mathcal{G}_p$ from distributed replicas, and synchronizing all replicas with new states of v_i. This distinguishes synchronization form single-directional communication between devices, as normally addressed in model-parallel systems [99, 216]. Specific to this characteristic, for each variable v_i, we introduce a *synchronizer config* to capture the process, which has its leading dimension as *synchronizer type*, indicating the communication primitives used for synchronization, notated as combinations of communication primitives. For example, v_i with synchronizer type as (`Reduce`, `Broadcast`) and placement as d_i follows a bipartite PS architecture to *reduce* and apply the

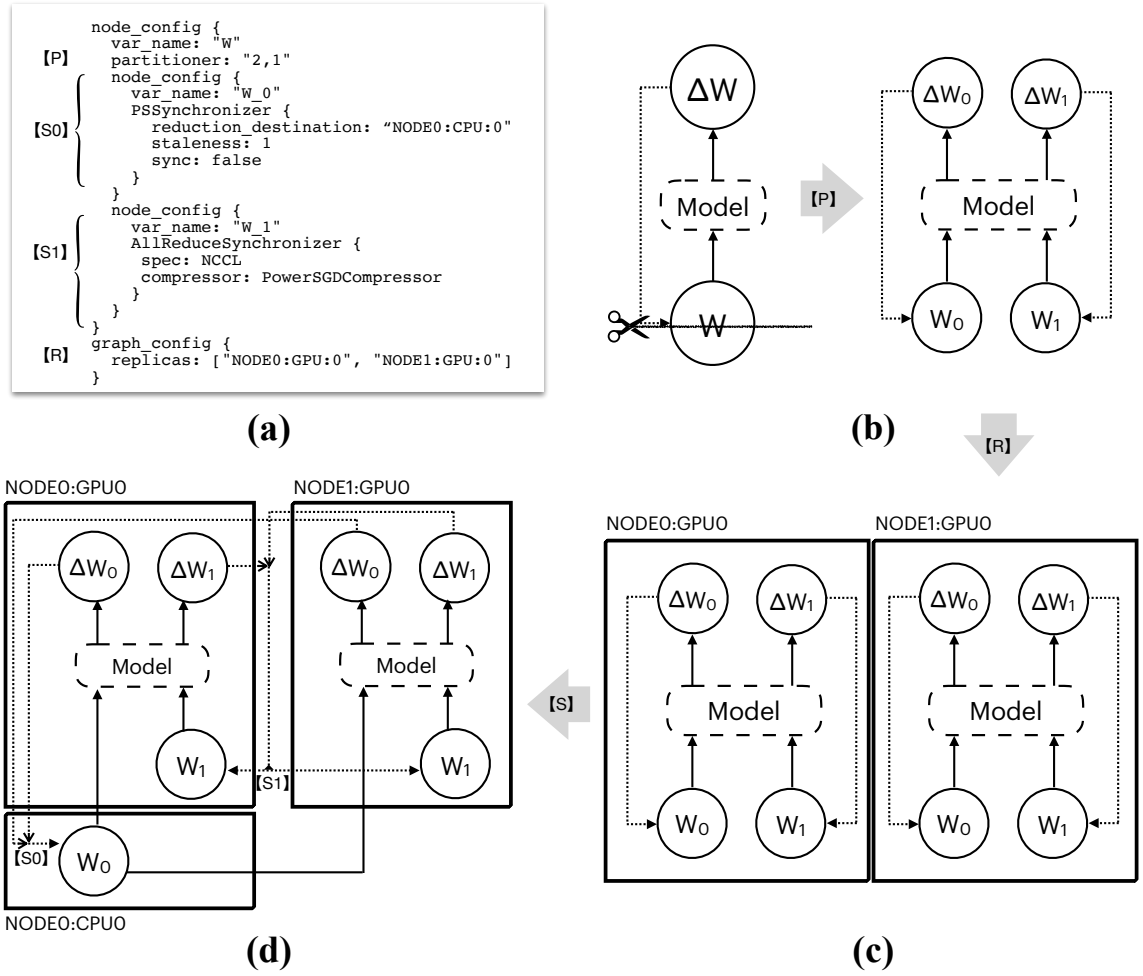

Figure 7.3: A composed strategy (Section 7.3) and its mapped graph-rewriting operations. A solid arrow is from a tensor producer to a stateless consumer; dotted refers to a stateful variable update. (a) The strategy expression. (b) It shards variable W into W_0, W_1 along axis 0. (c) It replicates onto two devices. (d) W_0 is synchronized by sharing the variable at one device and applying (`Reduce`, `Broadcast`), W_1 is synchronized with `AllReduce`.

updates $\{\nabla \mathcal{G}_p\}_{p=1}^{P}$ onto the destination d_i, and *broadcast* the updated states back to all participating replicas[1]. Similarly, type (`AllReduce`) straightforwardly indicates using AllReduce for joint aggregation and synchronization. This forms a unified representation of diverse synchronization architectures.

Based on the synchronizer type, we bring in more type-specific semantics such as: (1) *transfer destination*: $d_t(d_t \in \mathcal{S})$, to express a two-level hierarchy to reduce updates or broadcast parameters (of v_i) first to the transfer station d_t, then to the final destination d_i. (2) *reduction sequence*: list of devices $(d_1, \ldots, d_n)$ indicating the order of devices when performing collective communication. (3) *group*: schemes to group multiple communication primitives with the same group, together into one single operation to reduce constant min-cost (see Section 7.6.3 for a thorough study). The offers flexibility to specify, for each v_i, various forms of communication topologies, aggregation structure, grouping mechanisms (e.g., ps, hierarchical ps, ring or tree reduce), and related optimization conditioned on the context of v_i in $\mathcal{G}$ and the resource $\mathcal{D}$.

Message Encoding and Decoding

Many synchronization systems exploit certain structures (e.g., low-rank) present in the original message $\nabla_{\mathcal{G}_p}$, and encode them into alternative forms for faster network transfer. We introduce *encoder* and *decoder* for each v_i to model this aspect. Optimizations like sufficient factor broadcasting [226], or compression [132] hence can be covered by specifying each variable with a corresponding encoder and decoder methods in its node config.

Bridging Models

This aspect specifies how far the parallel replicas can allow computations and messages to occur out-of-order. Based on previous studies [43, 83] and definitions, we use an integer *staleness* to express the degree of consistency for synchronizing the variable v_i – when staleness is set to K, the fastest worker replica cannot be more than K steps ahead of the slowest worker in Equation 7.1.

7.3.3 Expressiveness

Table 7.1 enumerates several notable synchronization systems and shows how their specialized optimizations can be represented by the proposed synchronization representation. Since AutoDist focuses on synchronization in data-parallel training, the proposed representation deliberately skips the support for the partitioning of computational operations. Due to our assumptions on the synchronization aspect be represented using dataflow graph rewriting, the proposed representation also excludes optimizations commonly done in lower execution layer of the system, such as scheduling [78, 168, 245] and memory management [28, 40]. Based on this representation, we next present AutoDist system design to enable such strategy composition.

[1] A sparse version of it has the synchronizer type (`Gather`, `Scatter`).

System	Synchronization Architecture	Variable Sharding	Variable Placement	Collective Fusion	Encoding	Reduction Structure
Poseidon [245]	(`Reduce`, `Broadcast`) for normal gradients, (`Broadcast`) for rank-1 gradients	Fixed-size partitioning for all $V_{\mathcal{G},\theta}$	Place on PS	N/A	Sufficient factor	Local CPU as transfer device
Parallax [107]	(`Gather`, `Scatter`) for sparse gradients, (`AllReduce`) for dense gradients	Fixed-size partitioning for all sparse $v \in V_{\mathcal{G},\theta}$	Placed on PS for sparse v, replicated and placed on all worker nodes for dense v	N/A	N/A	Local CPU as transfer device for shared var, and ring for `AllReduce`.
Horovod [186]	(`AllReduce`) for dense gradients, (`AllGather`) for sparse gradients	N/A	Replicated and placed on all worker nodes	fixed-size fusion	FP16 compressor	Specifying device order (tree, ring, etc.)
BytePS [168]	(`Reduce`, `Broadcast`) for all gradients	Fixed-size partitioning for all $V_{\mathcal{G},\theta}$	Placed on PS	N/A	FP16 compressor	Local CPU as transfer device

Table 7.1: The proposed strategy space in AutoSync covers many advanced strategies in existing systems. Moreover, it offers a superset of semantics beyond them.

7.4 AutoDist: Composable System via Graph Rewriting

AutoDist backend is based on *dataflow graph rewriting*. Given user-defined $\mathcal{G}$, AutoDist incorporates distribution semantics by adding, deleting or modifying nodes and edges and their attributes in $\mathcal{G}$, using designated graph transformation kernels. Afterward, the transformed graph with distributed semantics is submitted to the runtime for execution.

7.4.1 Strategy Verification

Before a strategy can be applied to $\mathcal{G}$ for rewriting, it goes through a light compilation process for verification. The process looks for incompatibilities, such as reduction destinations out of $\mathcal{D}$ or inconsistent with the variable placement, or combinations of communication primitives that do not guarantee synchronization of variables, etc.

7.4.2 Graph Rewriting Kernels

AutoDist backend offers a library of primitive graph rewriting kernels. Each sub-expression in $\mathcal{S}$ is mapped to a kernel type with specific configurations. Each type of kernel defines how to rewrite the graph from its current state to the next state, which manifests the corresponding semantics in the sub-expression. By designing each local kernel at the right level of abstraction, they can be flexibly composed to alter the graph based on $\mathcal{S}$. We describe a few representative kernels illustrated with illustrations in Figure 7.3.

Partitioner (Figure 7.3(b)) reads node-level partitioning configurations from $\mathcal{S}$ for each $v_i \in \mathcal{V}_\Theta$. It splits the variable across given axes into multiple smaller variables, as well as its gradients and subgraphs corresponded to its state-updating operations. However, it does not split the operations that consume the original variable – which will instead consume the value re-concatenated from all partitions. Without allowing recursive partition, each of the new smaller variables has its own node config (generated at strategy generation time), will be added into $\mathcal{V}_\Theta$ as an independent variable identity in the following transformation steps. *Replicator* (Figure 7.3(c)) reads graph-level configuration from $\mathcal{S}$. It replicates the original graph onto target devices. Unless overridden by other graph-transformation kernels or by developers, the replicated operations or variables have their placement same with the target replication destination in $\mathcal{S}$. *Synchronizer* (Figure 7.3(d)) reads node-level configurations for each of the original and partitioned variable in $\mathcal{V}_\Theta$, where *Compressor* as its component rather than graph-transformation kernel is responsible for gradient encoding and decoding therein. Depending on the synchronizer type, it rewrites the graph: (i) either to share a variable on a destination device across replicas ((`Reduce`, `Broadcast`) synchronizer) with specified staleness in $\mathcal{S}$, (ii) or to synchronize states of replicated variables via collective communication (`AllReduce` synchronizer) following specified device structures in $\mathcal{S}$. Moreover, the kernel implementations are dispatched to handle either dense or sparse cases.

Besides existing kernels, the system design allows convenient extensions to emerging synchronization optimizations, by allocating new dimensions in the representation and introducing corresponded graph-rewriting kernels in the backend.

7.4.3 Parallel Graph Rewriting

With a valid $\mathcal{S}$ and graph-transformation kernels, parallelizing the ML program is essentially applying a series of kernels to rewrite the graph $\mathcal{G}$ into a distributed graph $\mathcal{G}'$. In contrast to other graph optimization systems [29, 100, 119], direct generation of $\mathcal{G}'$ leads of the size of $\mathcal{G}'$ proportionally growing with both the size of $\mathcal{G}$ and $\mathcal{D}$, quickly becoming unmanageable in terms of memory or rewriting efficiency if any of $\mathcal{G}$ and $\mathcal{D}$ is large. AutoDist uses local and deferred graph rewriting to mitigate the issue. Instead of writing all distributed semantics in one graph, AutoDist first broadcasts the strategy expression (whose size only corresponds to $\mathcal{G}$) across $\mathcal{D}$, and invokes graph transformation on each node afterward. Each node then transforms the single-node graph in parallel, but only rewrites a local subgraph of $\mathcal{G}'$ that corresponds to graph execution on that node. They maintain a consistent picture of $\mathcal{G}'$ by checking against $\mathcal{D}$ and $\mathcal{S}$, and uses absolute device notations when generating shared or distributed nodes. This allows to scale to extremely large graphs or clusters. After transformation, the executions of the dataflow graph is delegated to specific framework runtime on each node, so AutoDist keeps agnostic to frameworks.

7.4.4 Implementations

AutoDist is built as a Python library (12K LoC), and relies on specific ML frameworks to provide distributed runtime for dataflow graph execution. The current AutoDist offers APIs compatible with TensorFlow [1], but the design and concepts are compatible with other frameworks.

Coordination

In addition to described components, AutoDist implements a cluster coordinator. It sets up connections from the launching node, denoted as chief, to each worker specified in $\mathcal{D}$. After chief generates a strategy, it broadcasts the serialized strategy across nodes, then coordinates each worker to apply the graph rewriting based on the strategy to $\mathcal{G}$, and starts the framework process to execute the transformed graph.

Usages

AutoDist interface is designed to make sure ML practitioners can write distributed ML code as if they were writing single-node code. Using AutoDist involves construct a desired strategy builder with a resource specification, and then properly scopes (or decorates) the model declaration code using the provided interface (See Section 2.4.2 for an introduction of the scope and decorator interfaces), so that AutoDist can capture the corresponded dataflow graph representation for upcoming strategy generation. Then one can use the AutoDist session instead of the framework session to execute the graph. Figure 7.4 illustrates a code example.

Developing a synchronization strategy in AutoDist is equivalent to implementing a custom strategy builder. In AutoDist, this can be done using pure high-level languages (e.g., Python), without modifying low-level distributed system implementations. For emerging and new synchronization technologies, adding their support in AutoDist requires implementing their corre-

```
import autodist
# define the cluster specification
resource_spec = ...
ad = autodist.AutoDist(resource_spec,
        strategy_builder="AUTO")
with ad.scope():
  # Build your model here
  fetches = ...
  feeddict = ...
  ad.Session().run(fetches, feeddict)
```

Figure 7.4: An code snippet illustrating how to use AutoDist to distribute single-node ML code.

sponded graph transformation kernels, and registering them as new aspects in strategy representation so as to make them available in strategy generation.

7.5 Automatic Strategy Optimization

A unified and explicit representation of the synchronization strategy and a ready-to-evaluate system enables to auto-optimize $\mathcal{S}$ conditioned on $\mathcal{G}$ and $\mathcal{D}$, formulated as

$$\max_{\mathcal{S}} \mathcal{U}(\mathcal{S}, t(\mathcal{G}, \mathcal{D}, \mathcal{S}), \mathcal{D}), \tag{7.2}$$

where $t(*)$ denotes the graph transformation and compilation performed by AutoDist, and $\mathcal{U}$ characterizes a utility function that we want to maximize. It is possible to optimize different unities, e.g., convergence, system throughput, clould instance cost, or even combinations of them. Solving Equation 7.2 automatically alleviates the knowledge barriers on selecting appropriate distributed strategies. We defer the development of a solution to Equation 7.2 to Chapter 8. We focus on the evaluation of the system composability in AutoDist in the next section.

7.6 Evaluation

In this section, we evaluate the AutoDist system, aiming to answer the following questions:

- Is a composable design of data-parallel synchronization system *feasible*? How does the system perform when representing existing strategies?
- Is a composable design of synchronization system *necessary*? How do multiple aspects of synchronization influence the data-parallel training performance?

$\mathcal{D}$	Cluster Setup	GPU Distribution	Bandwidth Spec
A1	16x Cluster A nodes	[1] x 16	40GbE
A2	11x Cluster A nodes	[1] x 11	40GbE
B1	2x g3.16	[4] x 2	25 GbE
B2	3x g3.16	[4] x 3	25 GbE
B3	4x g3.16	[4] x 4	25 GbE
B4	1x g4dn.12	[4] x 1	50 GbE
B5	2x g4dn.12	[4] x 2	50 GbE
B6	3x g4dn.12	[4] x 3	50 GbE
B7	4x g4dn.12	[4] x 4	50 GbE
B8	8x g4dn.12	[4] x 8	50 GbE
B9	1x g3.4, 1x g3.16	[1, 4]	10/25 GbE
B10	1x g3.16, 1x g4dn.12	[4] x 2	25/50 GbE
B11	2x g3.16, 2x g4dn.12	[4] x 4	25/50 GbE
B12	1x g4dn.2, 1x g4dn.12	[1, 4]	25/50 GbE

Table 7.2: Cluster specifications we have experimented with, listed with their reference name ($\mathcal{D}$), setup information, GPU distributions, and bandwidth specifications. For the detailed AWS instance specification, refer to Table 2.1.

7.6.1 Experiment Setup

Clusters

We focus on GPU clusters since it is the main setup for distributed DL training. We conduct experiments on two cluster setups:

- **Cluster A** includes maximally 16 nodes for the ORCA cluster 2.7, each equipped with a GeForce TITAN X GPU, an Intel 16-core CPU and 64GB RAM, interconnected via a 40-Gigabit Ethernet switch;
- **Cluster B**, based on AWS, consists of up to 8 nodes, each node is one of the g3.4xlarge, g3.16xlarge, g4dn.2xlarge, g4dn.12xlarge instance types. Due to AWS constraints, they all have 10GbE single-flow bandwidth. On top of these two clusters, we list all the resource specifications we have experimented in Table 7.2 for later reference.

We rely on TF 2.0 for both dataflow graph evaluation and distributed execution, and *do not alter* any runtime or communication libraries in the native TF. As a note, TF 2.0 is complied with CUDA10.0, CUDNN 7.1, and NCCL 2.4.7, and uses gRPC for network communication. We choose baselines that depend on the same runtime in later sections.

Benchmark Models and Metrics

Following MLPerf [141], we choose a diverse set of ML models listed in Table 7.3 for evaluation. We focus on the *system throughput*, and report *per-iteration training time*, instead of statistical convergence to evaluate the system. We conduct fully synchronous training, and exclude syn-

Model	Task	Training data	Batchsize	#Params
ResNet101 [80]	IC	ImageNet	32	45M
InceptionV3 [202]	IC	ImageNet	32	24M
VGG16 [191]	IC	ImageNet	32	138M
Densenet121 [87]	IC	ImageNet	32	8M
Transformer [209]	MT	WMT'14 ende	5K	62M
NCF-dense [82]	CF	MovieLens-20Mx16x32 [141]	256	122M
BERT-3L [51]	LM	Wiki & BookCorpus	32	11M
BERT-6L [51]	LM	Wiki & BookCorpus	32	36M
BERT-12L [51]	LM	Wiki & BookCorpus	32	110M
BERT-large [51]	LM	Wiki & BookCorpus	8	340M

Table 7.3: ML models we have experimented with. All the model implementations are from the `tensorflow/models` repository. IC: image classification, MT: machine translation, LM: language modeling. For neural collaborative filter (NCF), we follow MLPerf [141] and use a enlarged version (x16x32), and we use dense instead of sparse gradients for its embedding variables to test systems' capability.

chronization aspects that would change the nature or results of the ML training as in the original single-node program. We manage to train all benchmark models to the reported accuracy [141] – hence we skip this comparison in later sections.

7.6.2 Hand-composed Strategy Performance

In this section, we implement a few notable synchronization strategies as fixed strategy builders in AutoDist, and compare them to those specialized systems in terms of system throughput. We conduct all the experiments on Cluster B (B4-B8). We consider the following baselines:

- `MirroredStrategy` where each "strategy" from the *tf.distribute* library can be seen as a fixed strategy builder based on specialized dataflow graph writing. The mirrored strategy is based on ring allreduce[2];
- `TF-PS-LB` where the native TF-PS is reported to have major drawbacks in load balancing [168, 245]. We improve it with a greedy load balancing strategy – we maintain a load recorder, and place each variable on the next PS which has the least loads. With this enhancement, we have observed much stronger performance under TF2.0 distributed runtime – which we will use as a PS baseline;
- `Horovod`: We deploy Horovod 0.19.0 with NCCL 2.4.7. It is a strong collective communication baseline specialized in using `AllReduce`, `AllGather` for parameter synchronization;
- `AllReduce` builder: We implement an `AllReduce` builder in AutoDist which exactly

[2]Among all strategies we only managed to setup the `MirroredStrategy` running on up to 4 GPUs on a single node during the writing of this thesis.

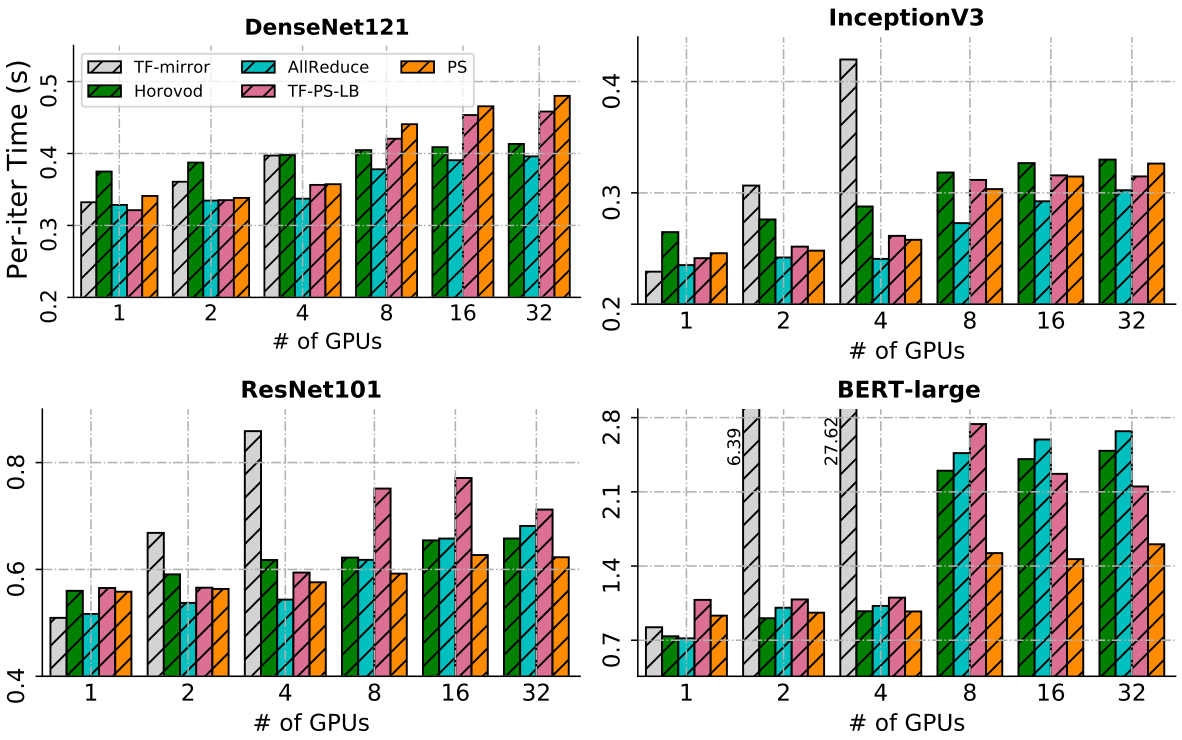

Figure 7.5: Based on TF distributed runtime, we comparing `MirrorStrategy`, `TF-PS-LB`, `Horovod` with manually implemented `AllReduce` and `PS` builders on AutoDist. See Section 7.6.2 for a detailed analysis.

reproduces the parameter synchronization in Horovod. Related hyperparameters for each model (e.g., merge scheme, see Section 7.6.3) is hand-tuned and the best performance is reported.

- `PS` builder: Manually implemented parameter server builder with many known optimizations incorporated: it performs hierarchical `Reduce` and `Broadcast` [127] on nodes with >1 GPUs to reduce network traffic; it partitions large variables (upon a hand-tuned size threshold based on model) into number of partitions equal to number of nodes in $\mathcal{S}$, and evenly distributes shards on nodes to achieve load balance [107, 168]. We use it as a strong PS baseline based on TF distributed runtime.

We compare these methods on 4 benchmark models: BERT-large, DenseNet121, ResNet101, InceptionV3, in Figure 7.5. We observe that collective-based strategies outperform the rest by a large margin, on all single node settings (1, 2, 4 GPUs), in which settings NCCL is advantageous. The improved version of TF parameter server, `TF-PS-LB` shows constant and acceptable performance other than reported [107, 168]. It scales well on CNNs (DenseNet121, InceptionV3, and ResNet101), outperforms `Horovod` occasionally in several settings (unlike reported), which verifies TF distributed runtime is a valid testbed. All baselines seem to scale poorly or sublinearly on BERT-large, among which `PSBuilder` performs best.

The reconstructed `AllReduceBuilder` shows almost similar performance to `Horovod`

on all CNNs, and slightly worse performance on BERT-large though, probably because Horovod has a smarter mechanism[3] for collective communication fusion. The hand-optimized `PSBuilder` frequently outperforms all other methods in distributed settings (>4 GPUs), particularly evident on BERT-large. We will show different ranks between collective communication and PS in Section 7.6.3 later.

To summarize, using distributed TF as runtime, we show that the composed hand-optimized strategies in AutoDist exhibits at least matched performance with baseline systems, and offers the best all-round performance on all models, thanks to its flexibility to represent strategies as builders.

7.6.3 Study of Synchronization Aspects

In this section, we re-examine several synchronization aspects commonly adopted in the latest development.

Synchronization Architecture

When choosing synchronization architectures, seemingly contradictory conclusions have been drawn in recent literature, claiming the supremacy of one architecture over another, e.g., Parallax [107], based on NCCL, shows a constant improvement of using collective communication to synchronize dense gradients over PS. ByteScheduler [168], using a different distributed runtime, demonstrates PS with credit-based scheduling is better.

Based on the native TF distributed runtime and NCCL, we compare the two architectures on a series of *dense* models across different resources shown in Figure 7.6, using the three builders `TF-PS-LB`, `AR` and `PS` (Section 7.6.2). On BERT-3L, AllReduce demonstrates outstanding advantage (more than 2x faster) over both unpartitioned (PS1) and partitioned PS (PS2), but when we increase the model size to BERT-large, partitioned PS outperforms AR by a small margin. On another model NCF-dense, the ranks between PS2 and AR change depending on the cluster. Both BERT and NCF have embedding variables that are significantly larger than other variables, preventing a normal PS without variable partitioning to achieve load balancing. This disadvantage no longer holds on ResNet101-B3 and Tranformer-B12, where PS1 shows small advantage over PS2. ResNet101 has most of its parameters as small convolutional filters, hence balancing them across multiple nodes is possible without partitioning, skipping overheads introduced by splitting and concatenation. Comparing Transformer-A1 and Transformer-B2, we observe AllReduce betters PS significantly on 16 nodes (A1), but underperforms PS substantially if switching to B2 (0.3s increase on per-iter time). We also perform experiments on two models with sparse gradients: NCF-sparse and LM1B [22] and the results echo the advantages of PS over `AllGather` on synchronizing gradient with sparsity above a certain level – this is modeled and supported by the proposed synchronization presentation.

In summary, we see that even for synchronizing dense gradients, different architectures seem to demonstrate uncertain pros and cons, depending on models and resources.

[3] https://github.com/horovod/horovod/blob/master/docs/autotune.rst.

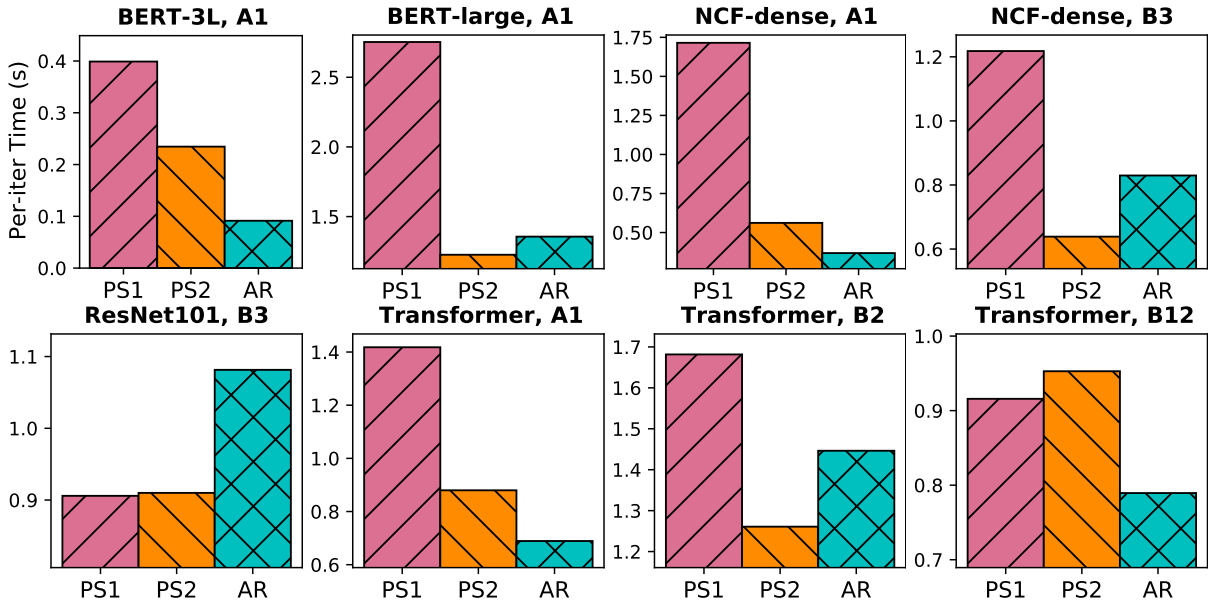

Figure 7.6: Comparing synchronization architectures across a diverse set of models and cluster specifications. PS1: load balanced PS builder without partitioning, PS2: load balanced PS builder with partitioning. AR: collective AllReduce based builder. Per-iter training time is reported.

Collective Communication Merge Scheme

A common optimization in collective-based synchronization is to merge multiple collective operations into a fused one, to reduce the linear min-cost by the invocation of each collective. While this optimization has been adopted broadly [119, 186], its effects and how to properly tune it against different models remain largely unknown. To study it, we implement a `AllReduce(chunk: int)` builder with a *chunk*-based merging scheme, similar to tf.distribute. Under this scheme, starting from the first variable, the synchronization of every $n =$ `chunk` adjacent variable will be merged into one collective. Based on this builder, we experiment with different values of a `chunk`, on diverse model-resource combinations shown in Figure 7.7.

We observe the chunk effects differently on different models – among the values we examined, the performance improvement due to collective merging can go up to more than 2x (ResNet101), or with almost no impact (VGG16). On NCF-dense, we observe merging undermines the performance. We hypothesize this is because NCF-dense uniquely has two adjacent large embedding variables; communicating them in one collective might be bandwidth-bounded – in this situation, partitioning them might be a better choice. Constantly, the performance drops when the chunk size is larger than the number of variables in the model. Merging all variables into one collective operation prevents opportunities of pipelining communication with computation during BP, which is a key optimization nowadays distributed runtime has adopted. In choosing the optimal value of `chunk`, we cannot draw a constant conclusion: while on BERT-6L, BERT-12L, Transformer, 3 transformer-based architectures, this value seems to be near 64, it shifts toward 128 on BERT-large. On Densenet121, we see a sudden drop of per-iter time when

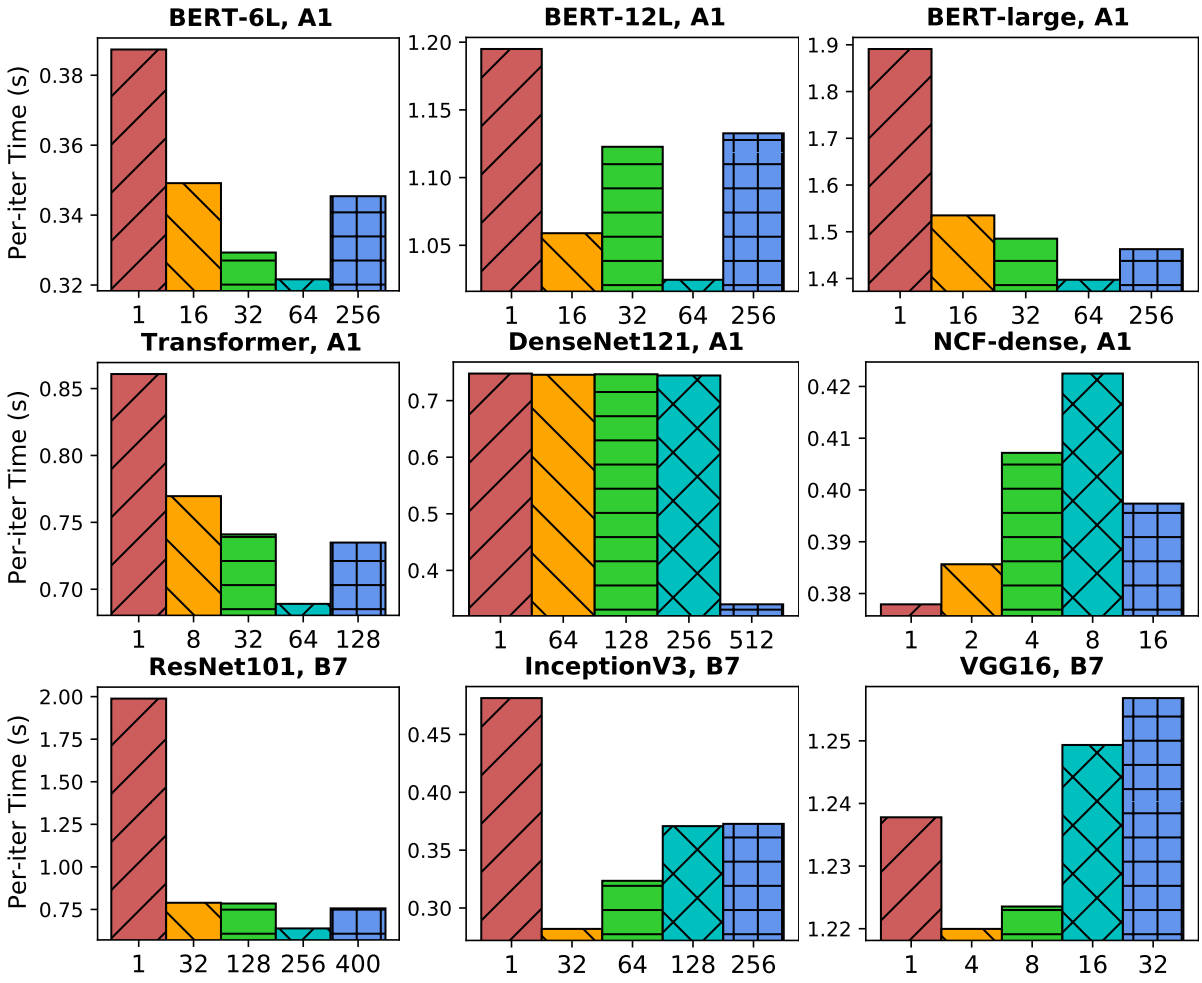

Figure 7.7: Using a `AllReduce(chunk: int)` builder, we study how the performance is impacted by the collective merge scheme on different models and resource specifications. X-axis lists the value of `chunk`, the per-iter training time (lower is better) is reported.

the chunk goes up to 512, but the performance of VGG16 is not influenced by it. Chunk-based merging is obviously suboptimal since it is inflexible and not fully aware of the model structure. How to create the optimal merge scheme that copes with the model structure is an NP problem, explored in Chapter 8 later.

To conclude, the optimal choice of each aspect changes with models, resources, and the choice of other aspects, letting alone the fact that there exists variability caused by system artifacts. Understanding and interpreting such intrinsic system-ML subtleties are nontrivial. We do not think a typical ML practitioner should be reasonably expected to have this knowledge, and we argue this demonstrates the need for model- and resource-dependent modeling of synchronization strategies, and corresponding auto-optimization methods.

7.7 Additional Related Work

Specialized Synchronization Systems

Poseidon [245] and Parallax [107] use hybrid communication architectures exploiting specific properties present in the model. Horovod [186] brings in collective communication to reduce gradients for distributed DL, and a line of collective communication-based systems [68, 95, 232] have reduced the training time of specific NNs (e.g., ResNet) from days to minutes on specialized hardware. Most of these systems are specialized based on one or two synchronization aspects or hardwares, and exhibit varying performance on different models or resources.

Synchronization performance can also be optimized at lower levels of systems – such as better scheduling to pipeline computation and communication [78, 79, 168, 245]. While AutoDist does not include these optimizations as it assumes each aspect to map with a dataflow graph rewriting function, its representation and auto-optimization techniques can be integrated with and benefit from these system runtimes.

DL Graph Rewriting and Optimization

An emerging line of work, such as TVM [29, 30], TASO [100], XLA [66], perform automatic operator optimization and graph rewriting on a single device, mostly for inference graphs. AutoDist draws insights from these work but faces different challenges: the training graph needs to handle states, and the synchronization in data parallel training has unique semantics and problems.

Representation of Parallelisms

Some existing works [99, 148, 188, 216] have studied the representation of parallelisms in different context, and propose automatic parallelization methods. Device placement [148] automatically places nodes of a graph using a trained NN. Mesh-TensorFlow [188] defines "meshes" to describe how operations are partitioned and dispatched across devices. The follow-up work GShard [124] by Google implement the parttiioning in XLA, and support SPMD other than MPMD. FlexFlow [98, 99] proposes the "SOAP" representation, and uses a stochastic MCMC-based algorithm to search for the optimal partitioning strategies for coarse-grained NN layers. Tofu [216] poses a similar problem on finer-grained operators, and derives possible partitioning configurations using symbolic interval analysis, and automatically optimize the partitioning. These techniques focus on partitioning or placement, and partially intersect with AutoDist as variables in the dataflow graphs are also represented as nodes (hence can be partitioned or placed using their techniques). However, AutoDist focuses on how to characterize the problem of synchronization in data-parallel training and modeling various synchronization-related aspects, and aims to co-optimize them to maximize the training performance.

Part III

Automatic ML Parallelization

Introduction

Echoing the fourth and fifth challenge mentioned in the opening of Chapter 7 – ML scale-up is frequently underestimated. What does it really take to train an ML model, whose training code was originally written for a single CPU/GPU, on multiple machines? The need to choose the appropriate distributed strategies, and heavily tune the distributed code to the satisfying system or statistical performance, which is yet another iterative process in addition to model development.

Built on top of the composable representation and system deveoped in previous chapters, this part of the thesis presents automation mechanisms for ML parallelization, to lower the barrier of distributed ML. In Chapter 8, we build an end-to-end pipeline, *AutoSync*, to automatically optimize synchronization strategies with ML-based simulators given model structures and resource specifications, lowering the bar for data-parallel distributed ML. By learning from low-shot data collected in only 200 trial runs, AutoSync can discover synchronization strategies up to 1.6x better than manually optimized ones. We develop transfer-learning mechanisms to further reduce the auto-optimization cost – the simulators can transfer among similar model architectures, among similar cluster configurations, or both. We also present a dataset that contains over 10000 synchronization strategies and run-time pairs on a diverse set of models and cluster specifications.

The results presented in this part of the thesis have also appeared in the following paper:

- Hao Zhang, Christy Li, Zhijie Deng, Lawrence Carin, and Eric P. Xing. AutoSync: Learning to Synchronize for Data-Parallel Distributed Deep Learning. *The Thirty-fourth Conference on Neural Information Processing Systems (NeurIPS 2020)*.

Chapter 8

Automatic Synchronization Strategy Optimization

8.1 Introduction

Evidenced in Chapter 7, existing systems struggle to provide excellent *all-round* performance on diverse models due to their oversimplified assumptions about the synchronization, and rigid application of fix-formed synchronization strategies (e.g., parameter server (PS) [127, 221] for BytePS [168], Allreduce for Horovod [186]), ignoring the characteristics of models or clusters. More importantly, different strategies often exhibit sharp performance differences when applied to different ML [107, 245]; the burden of selecting the *right* strategy for the model of interest is placed on ML practitioners, who may not have domain expertise on the trade-offs among these systems. Given the combinatorial number of choices for various synchronization factors, (e.g., architecture, variable partitioning, and placement configuration), it is prohibitively costly to manually search for the optimal strategy, and the search has to be redone every time a new model is developed.

To address these challenges, this chapter aims to answer: Can one automate the selection of the optimal synchronization strategy, given a model and cluster specification? With multiple synchronization-affecting factors identified in Chapter 7 for data-parallel distributed DL, We construct a valid and large strategy space spanned by multiple factors, by factorizing the strategy with respect to each trainable building block of a DL model. To efficiently navigate the space and locate the optimal strategy, we build an end-to-end pipeline, *AutoSync*. AutoSync leverages domain knowledge about synchronization systems to reduce the search space, and is equipped with a *domain adaptive simulator*, which combines principled communication modeling and data-driven ML models, to estimate the runtime of strategy proposals without launching real distributed execution. To further reduce practical development cost, we study the transferability of trained simulators across different models and resource specifications, which shows promising adaptability to unseen models or cluster configurations.

We evaluate AutoSync on a broad set of models and clusters, and show that there exist ample strategies in the proposed space that outperform hand-optimized systems by a significant margin. AutoSync can effectively find strategies that reduce the training time by 1.2x - 1.6x than hand-

optimized ones on multiple model architectures (e.g., NCF [82], BERT [51] and VGG16 [191]), within an acceptable budget. Leveraging transfer learning, AutoSync simulators can be trained on cheaper trial data collected on smaller models or clusters, and used to derive strategies without additional training for larger models or costlier clusters. As an additional contribution, we collect a dataset with over 10000 data points containing (model, resource, strategy) tuples and their corresponding runtime on real clusters. We share the dataset with the community to encourage extended studies.

8.2 Problem Formulation

8.2.1 Notations

We first expand the notations in Chapter 7 a bit. We represent a DL model using its dataflow graph $\mathcal{G} = \{(V_{\mathcal{G},\theta}, V_{\mathcal{G},o}), E_{\mathcal{G}}\}$ where $V_{\mathcal{G}}$ are nodes in $\mathcal{G}$ including trainable variables $V_{\mathcal{G},\theta} = \{v_i\}_{i=1}^{|V_{\mathcal{G},\theta}|}$ or computational operations $V_{\mathcal{G},o}$, and $E_{\mathcal{G}}$ are tensors (edges) transferred between nodes. For simplicity, we use V equivalently with $V_{\mathcal{G},\theta}$ to notate the set of variables. In addition to $\mathcal{G}$, we define a cluster as a device graph $\mathcal{D} = \{V_{\mathcal{D}}, E_{\mathcal{D}}\}$, where $V_{\mathcal{D}} = \{d_p\}_{p=1}^{|V_{\mathcal{D}}|}$ represents devices (e.g., CPUs or GPUs), and $E_{\mathcal{D}} = \{b_{i,j}\}$ is a symmetric matrix with the entry $b_{i,j}$ representing the connectivity (e.g., bandwidth) between d_i and d_j. In data-parallel training, we replicate $\mathcal{G}$ on all devices, and update each trainable variable v_i using the aggregation of the stochastic gradients $\nabla_{v_i}(\mathcal{G}, X_p)$ computed by each worker device d_p on its data partition X_p, following $v_i^{(t+1)} \leftarrow v_i^{(t)} + \epsilon \sum_{p=1}^{P} \nabla_{v_i^{(t)}}(\mathcal{G}, X_p)$, for $v_i \in V_{\mathcal{G},\theta}$. Since devices are distributed across the cluster, obtaining the aggregation requires *synchronization* support, which collects updates $\nabla_{v_i^{(t)}}$ and provides all devices the shared access to a consistent version of $v_i^{(t+1)}$.

Existing systems aim to optimize some individual factor to expedite synchronization, ignoring that the optimal of each factor significantly changes with $\mathcal{G}$ and $\mathcal{D}$. For ML practitioners, it is challenging to select appropriate synchronization strategies for their models of interest without domain expertise.

8.2.2 Formulation

Alternatively, following Equation 1.1 and the motivation in Section 7.5, we pose the strategy selection as an optimization problem, in which the *per-iteration runtime* (e.g., time taken to process a batch on all nodes of the entire cluster, equivalent with system throughput), denoted as f, is minimized given $\mathcal{G}$ and $\mathcal{D}$ by solving

$$\min_{\mathcal{S}} f(\mathcal{G}, \mathcal{D}, \mathcal{S}), \text{ s.t. } \mathcal{C}, \tag{8.1}$$

where we use $\mathcal{S}$ to denote a model and resource dependent representation of the synchronization strategy, and $\mathcal{C}$ as a set of constraints (developed in Section 8.3.2). Approaching this problem analytically needs continuous characterizations of $\mathcal{S}$ and f, which are unavailable. In light of the recent advance in AutoML [30, 229, 251], we define a domain-specific space considering

multiple synchronization-affecting factors following the representation defined in Section 7.3, and resort to search-based methods to find a near-optimal $\mathcal{S}^*$.

8.2.3 Search Space

When constructing the search space, we have the following considerations. First, instead of optimizing a single factor in a piecemeal fashion as commonly done in existing systems, we seek a unified space covering multiple synchronization-affecting factors, to capture the subtleties between them and their dynamics with different $\mathcal{G}$ and $\mathcal{D}$ via co-optimization. On the other hand, we want to establish direct correspondences between $\mathcal{S}$ and each participating variable of $\mathcal{G}$ in synchronization, so that the strategy can adapt with specific variable-wise mathematical properties.

Following the synchronization strategy representation proposed in Section 7.3, we decompose existing fixed-formed systems orstrategies into following orthogonal factors:

- *Variable partitioning*, represented as $\boldsymbol{p}_i = [p_i^j]_{j=1}^{k_i}$ for the variable v_i, where k_i is the number of tensor dimensions of v_i, and p_i^j represents the *partition degree* on the jth axis.
- *Variable placement*, defined as $\{d_i\}_{i=1}^{m} \subset V_{\mathcal{D}}$ which is the set of devices the variable resides. The placement being a single device means v_i is shared across all devices.
- *Synchronization architecture*: we define two types of architecture primitives, namely parameter server (PS) and collective communication (CC), and their architecture-specific semantics. In PS, we use *reduction hierarchy* to indicate whether parameters are transmitted hierarchically (e.g., from a central CPU to multiple GPUs co-located on the same machine). In CC, we define *merge group*, where communication primitives assigned with the same group are merged and communicated via a single message, and *device order*, specifying the message passing order across devices (e.g., tree, ring).
- *Message encoding and decoding*, notated as c_i for v_i, introduces compression or decompression schemes to represent how messages are processed before (after) communication, enabling optimizations [132, 226, 245] that exploit structures (e.g., low-rank) exhibited in the messages to reduce message size and fasten network transfer.

Since we focus on synchronous training and optimizing system throughput, we exclude optimizations beyond data-parallel training (e.g., operation partitioning in model-parallel training), or introduce deviation of parameter updates (e.g., staleness which was included in Section 7.3).

Based on these factors, we express the strategy as $\mathcal{S} = \{s_i\}_{i=1}^{|V|}$ where s_i, as a sub-strategy, includes the discrete choices of above factors for each $v_i \in \mathcal{G}$. Note that we decide whether to partition variables or not before any other aspects, so a sub-strategy needs to be generated for each variable partition. The multiple factors span a combinatorial space whose size grows with the size of $\mathcal{G}$ and $\mathcal{D}$.

We next develop the *learning to synchronize* framework to approximate the optimal strategy $\mathcal{S}^*$.

8.3 AutoSync: Learning to Synchronize

Despite the large space, solving Equation 8.1 poses an additional challenge: searching for $\mathcal{S}^*$ requires evaluating f for each strategy proposal, which involves distributed execution on clusters and is prohibitively expensive. To make the search tractable, we present the learning to synchronize framework, illustrated in Figure 8.1 and Algorithm 8, with two novel components: runtime simulation of arbitrary $\mathcal{S}, \mathcal{G}, \mathcal{D}$ (Section 8.3.1), and knowledge-guided strategy search (Section 8.3.2).

In detail, to reduce real execution, we develop a *domain adaptive simulator* to estimate f. The estimation is made possible (without training data) by first designing features agnostic to $\mathcal{G}, \mathcal{D}$. The features describe critical impacting factors on the runtime using predefined modeling, and can be generalized to any unseen $\mathcal{G}$ and $\mathcal{D}$. Then, we enhance them using ML models and various raw features extracted from specific $\mathcal{G}, \mathcal{D}$. In order to direct the search to the subspace where good strategies may locate, we instantiate constraints $\mathcal{C}$ in Equation 8.1 using prior knowledge on synchronization systems.

The final framework, named as *AutoSync*, approaches $\mathcal{S}^*$ alternately. In the initial phase, the simulator uses the training-free features to propose the initial batch of strategy candidates, which are executed on real clusters to obtain ground truth runtime. The low-shot data are feedbacked to train the simulator to adapt to $\mathcal{G}, \mathcal{D}$, achieving improved capability in differentiating high-performing strategies. The process alternates until the optimization budget for real distributed cluster evaluation is exhausted. We next describe the design of the simulator and the guided search.

Algorithm 8: The AutoSync strategy auto-optimization process

Input: model $\mathcal{G}$, device $\mathcal{D}$, budget B
Output: near-optimal strategies $\mathcal{S}^*$

Initialize $b = 0$
while $b < B$ **do**
  Sample proposals $\{\mathcal{S}_i\}_{i=1}^{N}$ subject to the constraints c_{lb} and c_{am} (Section 8.3.2)
  Filter $\{\mathcal{S}_i\}_{i=1}^{N}$ by rejecting $\mathcal{S}_i$ with high $\hat{f}(\mathcal{S}_i)$, then obtain $\{\mathcal{S}_i\}_{i=1}^{M}$
  Select final candidate set $\{\mathcal{S}_k\}_{k=1}^{K} \subset \{\mathcal{S}_i\}_{i=1}^{M}$ by minimizing a weighted sum of $\sum_{k=1}^{K} \hat{f}(\mathcal{S}_k)$ and the pairwise similarity of $\{\mathcal{S}_k\}_{k=1}^{K}$ (Section 8.3.1)
  Launch trial execution for $\{\mathcal{S}_k\}_{k=1}^{K}$ and get ground truth $\{f_k\}_{k=1}^{K}$ where $f_k = f(\mathcal{S}_k)$
  Update S^* with the best-performed $\mathcal{S}_k$
  Improve ML simulators by training with additional data $\{f_k\}_{k=1}^{K}$
  $b = b + K$
return $\mathcal{S}^*$

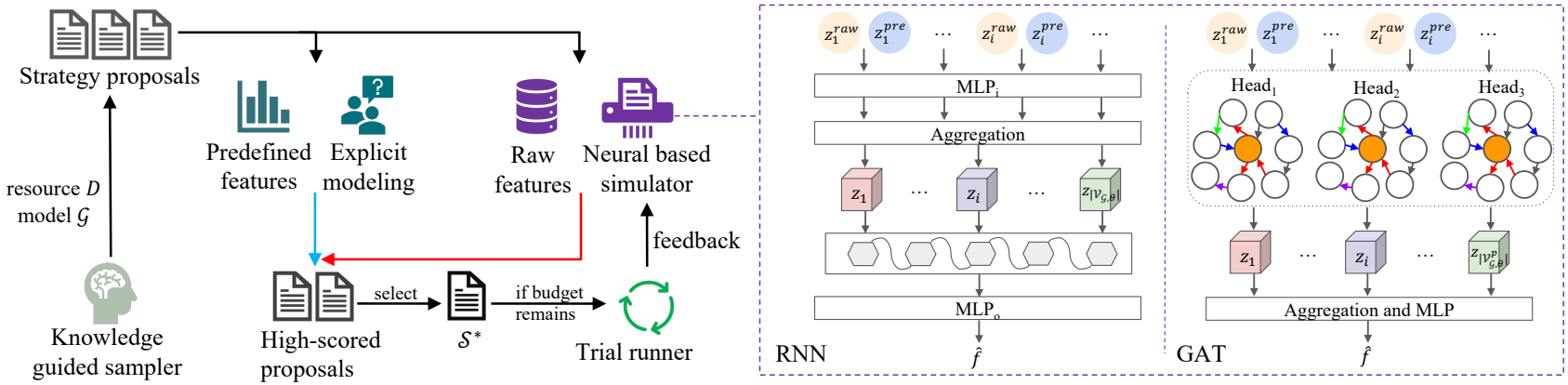

Figure 8.1: **Left**: Learning to synchronize framework. Initially, the simulator utilizes domain-agnostic features (Section 8.3.1) to explicitly estimate the runtime so to select promising strategies for evaluation (the blue line). After trials, the real runtime data are feedbacked to train the ML-based simulator to adapt to specific $\mathcal{G}, \mathcal{D}$, enhancing its capability in differentiating high-quality strategies. Gradually, the ML-based simulator takes over and directs the search (the red line). **Right**: Illustrations of the RNN and GAT simulators.

8.3.1 Domain Adaptive Simulator

The simulator takes $(\mathcal{G}, \mathcal{D}, \mathcal{S})$ as input, and estimates its per-iteration runtime (equivalent with throughput). Because variable partitioning will alter $V_{\mathcal{G},\theta}$, we first infer the new set of variables $V'_{\mathcal{G},\theta}$ based on $\mathcal{G}, \mathcal{S}$, and let the simulator work with each variable $v_i \in V'_{\mathcal{G},\theta}$, which contains variable shards after partitioning original variables. We define a particular $(\mathcal{G}, \mathcal{D})$ target as a *domain*. When equipped with domain-agnostic features, the simulator is capable of estimating the runtime of any $(\mathcal{G}, \mathcal{D})$ without training. This is realized by systematic modeling of the runtime in distributed execution.

Predefined Modeling

We model the per-iteration runtime T of parallelizing $\mathcal{G}$ on $\mathcal{D}$ using two contributing components: computation time T_{comp}, and parameter synchronization time T_{sync}. Since: (1) many runtime systems (e.g., TensorFlow [1] or PyTorch [56]) introduce scheduling or parallelization between communication and computation, in practice, there are significant overlaps between the two components; (2) in data-parallel training, it is commonly observed that one component usually dominates the other [78], we simply obtain T via

$$T = \max(T_{\text{comp}}, T_{\text{sync}}).$$

We factorize T_{comp} and T_{sync} w.r.t. variables, similar to $\mathcal{S}$, and estimate $T_{\text{comp}}(v_i), T_{\text{sync}}(v_i)$ for each v_i based on its s_i in $\mathcal{S}$. $T_{\text{comp}}(v_i)$ can be approximated by profiling its corresponding operation on a single-device. To calculate $T_{\text{sync}}(v_i)$, we split $V'_{\mathcal{G},\theta}$ into variables using PS as V^{PS} and using collective communication as V^{CC}, and derive two analytic forms of $T_{sync}(v_i)$.

Modeling $v_i \in V^{PS}$. Synchronizing v_i via PS ($v_i \in V^{PS}$) involves: (1) workers send gradients to servers, (2) servers update parameters, (3) servers send the updated parameters to workers, where (2) is negotiable and (1)(3) are symmetric processes and cost the same amount of time. We denote the original size (e.g., byte size) of the gradient of the variable v_i as m_i (note m_i takes into consideration the sparsity of the gradients when applicable), and assume it will apply the encoding/decoding scheme $c_i \in s_i$, so the actual size of the message (related to v_i) to be transferred across devices is $c_i(m_i)$, where we also use c_i to denote the compression function that reduces the original size m_i to $c_i(m_i)$.

Let w_i denote the number of workers involved in synchronizing v_i. The parameter transfer process involves transferring data between the GPU device memory and the host memory (RAM) within the same machine, and between the host memory across machines. The first process introduces GPU kernel latency and device-host communication. The second process introduces network overhead (*e.g.* latency) and network communication. Hence, the communication time T^{PS}_{server} on the server hosting v_i, indexed as j, is

$$T^{PS}_{\text{server}}(v_i) = \underbrace{\sum_{k=1}^{w_i} \mathbb{I}_d(j,k) \cdot \frac{c_i(m_i)}{b_{j,k}} \cdot r_{i,k}^{\mathbb{I}_p}}_{\text{network transfer}} + \underbrace{\sum_{k=1}^{w_i} \mathbb{I}_d(j,k) \cdot r_{i,k}^{\mathbb{I}_p} \cdot \phi}_{\text{network overhead}} + \underbrace{\delta}_{\text{GPU kernel latency}}, \tag{8.2}$$

where $r_{i,k}$ is the number of replicas of $\mathcal{G}$ on worker k if the worker k has multiple GPUs that host multiple replicas of $\mathcal{G}$ respectively; $\mathbb{I}_d(j,k)$ and $\mathbb{I}_p$ are true when server j and worker k locate on different machines and when hierarchical reduction is used, respectively. To interpret Equation 8.2, the first term corresponds to sending messages from each worker k to the server j (and vise versa). The second term captures network overheads that scale linearly with the number of workers, or with the number of replicas when $\mathbb{I}_p$ is false. The third term captures the constant GPU memcpy latency.

In addition to the formula, we can construct domain-agnostic features of v_i as:

$$\boldsymbol{z}_i^{PS} = [\text{network transfer}, \text{coefficient of } \phi, \text{coefficient of } \delta].$$

As synchronizing any v_i between any pair of nodes can happen simultaneously and is upper bounded by the multi-flow bandwidth, the communication bottleneck may be caused by the slowest transmission, or the total time of transmissions. Thus, we define global features for estimating T_{sync}^{PS} as:

$$\boldsymbol{z}^{PS} = \max\{\boldsymbol{z}_i^{PS} \text{ for } v_i \in V^{PS}\} \oplus \sum\{\boldsymbol{z}_i^{PS} \text{ for } v_i \in V^{PS}\}/|V^{PS}|,$$

where $\oplus$ is vector concatenation, and $\max, \sum$ are elementwise.

Modeling $v_i \in V^{CC}$. For $v_i \in V^{CC}$, we model 5 mostly used collective primitives: `AllReduce`, `ReduceScatter`, `AllGather`, `Broadcast` and `Reduce` [156]. Take an example when there are w workers and the device order in the substrategy s_i is a ring [186]. Each primitive sends and receives $\frac{2(w-1)}{w}$, $\frac{w-1}{w}$, $\frac{w-1}{w}$, $1, 1$ times, respectively, in its applicable scenario for parameter synchronization (e.g., `AllReduce` for dense gradients or `AllGather` for sparse gradients [186]). Therefore, we can obtain $T_{sync}^{CC}(v_i)$ using the following formula:

$$T_{sync}^{CC}(v_i) = \underbrace{\mathbb{I}_1 \frac{2(w_i-1)c_i(m_i)}{w_i b_m} + \mathbb{I}_2 \frac{(w_i-1)c_i(m_i)}{w_i b_m} + \mathbb{I}_3 \frac{c_i(m_i)}{b_m}}_{\text{network transfer}} + w_i \cdot \phi + \delta, \tag{8.3}$$

where $b_m = \min_{(k_1,k_2)\in\text{ring}} b_{k_1,k_2}$ denotes the lowest bandwidth between devices in the ring, since the throughput of a ring is restricted by the lowest bandwidth in the network [212]. $\mathbb{I}_1, \mathbb{I}_2, \mathbb{I}_3$ are true when `AllReduce`, `ReduceScatter` and `AllGather`, `Broadcast` and `Reduce` are activated, respectively. The formula is derived based on counting how many times each message (i.e. gradients) needs to be passes across the ring, taking into considerations both network transfer overhead as well as the device-host memory swap latency. The total synchronization time for variables assigned with collective communication is $T_{sync}^{CC}(V^{CC}) = \sum_{i=1}^{|V^{CC}|} T_{sync}^{CC}(v_i)$.

In a similar way with $v_i \in V^{PS}$, the domain-agnostic features for $v_i \in V^{CC}$ are:

$$\boldsymbol{z}_i^{CC} = [\text{network transfer}, \text{coefficient of } \phi, \text{coefficient of } \delta],$$

which is a concatenation of terms from Equation 8.3, and the global features of T_{sync}^{CC} are:

$$\boldsymbol{z}^{CC} = \max\{\boldsymbol{z}_i^{CC} \text{ for } v_i \in V^{CC}\} \oplus \sum\{\boldsymbol{z}_i^{CC} \text{ for } v_i \in V^{CC}\}/|V^{CC}|.$$

Concatenating $\boldsymbol{z}^{PS}$ and $\boldsymbol{z}^{CC}$ obtains the set of domain-agnostic features $\boldsymbol{z}^{pre}$ for $(\mathcal{G}, \mathcal{D}, \mathcal{S})$. We can either use the estimated T to rank different $\mathcal{S}$, or use the constructed features as inputs to ML models, which we elaborate next.

Domain-specific Modeling

Once trial data are acquired, we augment $\boldsymbol{z}^{pre}$ using raw features $\boldsymbol{z}^{raw}$ extracted from $\mathcal{G}, \mathcal{D}, \mathcal{S}$, and train ML models so to capture their domain-specific characteristics. For each v_i, $\boldsymbol{z}_i^{raw}$ vectorizes attributes including variable placement, synchronization architecture, encoding/decoding type, and merge group from $\mathcal{S}$, bandwidth and the number of replica devices of each node from $\mathcal{D}$, and variable size, dimensions, the sparsity of gradients, data types, and information of partitioned shards from $\mathcal{G}$. Combining $\boldsymbol{z}^{pre}$ and variable-specific raw features $\boldsymbol{z}_i^{raw}$, we adopt three different ML models: (1) a linear model, (2) a recurrent neural network (RNN), and (3) a graph attention networks (GAT) [246], to learn from trial data and make more accurate predictions of f.

Linear model. The linear model, simply written as $\hat{f} = \boldsymbol{z}^{pre} \cdot \boldsymbol{\theta}$, introduces trainable weights $\boldsymbol{\theta}$ and predicts the runtime $\hat{f}$ using only global features $\boldsymbol{z}^{pre}$.

RNN. We use a RNN to model different $\mathcal{G}$ with varying number of variables, so as to inject $\boldsymbol{z}_i^{raw}$. The RNN first concatenates $\boldsymbol{z}_i^{pre}$ with $\boldsymbol{z}_i^{raw}$, and then transforms the combined features via an MLP. The results are aggregated into $|V_{\mathcal{G},\theta}|$ features corresponding to the variables in the $V_{\mathcal{G},\theta}$. A bidirectional LSTM scans them following the forward-backward propagation order preserved in $\mathcal{G}$. At last, the prediction $\hat{f}$ is obtained via an extra MLP (Figure 8.1). Detailed formulas are below:

$$
\begin{aligned}
\boldsymbol{z}_i &= \text{MLP}_i\left(\boldsymbol{z}_i^{raw} \oplus \boldsymbol{z}_i^{pre}\right), \\
\boldsymbol{h}_{i+1} &= \text{LSTM}\left(\boldsymbol{h}_i, \boldsymbol{z}_i\right), \\
\boldsymbol{h}'_{i+1} &= \text{LSTM}\left(\boldsymbol{h}'_i, \boldsymbol{z}_{|V'_{\mathcal{G},\theta}|+1-i}\right), \\
\hat{f} &= \text{MLP}_o([\boldsymbol{h}_{|V'_{\mathcal{G},\theta}|}, \boldsymbol{h}'_{|V'_{\mathcal{G},\theta}|}]).
\end{aligned}
$$

GAT. We bring in GAT to model the raw graph structure of $\mathcal{G}$. To do so, we prune $\mathcal{G}$ into a graph $\mathcal{G}^p = (V^p_{\mathcal{G},\theta}, E^p_{\mathcal{G}})$ with only variable (including partitioned) nodes and edges connecting them. For each node, similar to RNN, its raw and predefined features are concatenated, and features of all nodes and corresponding edge information are fed to a GAT for encoding graph structure as: $\{\boldsymbol{z}_i\} = \text{GAT}(\{\boldsymbol{z}_i^{raw} \oplus \boldsymbol{z}_i^{pre} \text{ for } v_i \in V^p_{\mathcal{G},\theta}\}, E^p_{\mathcal{G}})$, shown in Figure 8.1. The node features are then aggregated into graph-level ones, followed by an MLP to get the runtime estimation $\hat{f}$, following formulas:

$$
\begin{aligned}
\{\boldsymbol{z}_i\}_{i=1}^{|V'_{\mathcal{G},\theta}|} &= \text{GAT}(\{\boldsymbol{z}_i^{raw} \oplus \boldsymbol{z}_i^{pre}, \text{ for } v_i \in V'_{\mathcal{G},\theta}\}, E'_{\mathcal{G}}), \\
\boldsymbol{z} &= \frac{1}{|V'_{\mathcal{G},\theta}|} \sum (\{\boldsymbol{z}_i\}_{i=1}^{V'_{\mathcal{G},\theta}}), \\
\hat{f} &= q(\boldsymbol{z}),
\end{aligned}
$$

Training Objective

Accurately predicting runtime can be challenging due to uncertain factors on a distributed cluster, we instead train the simulators with the pair-wise logistic ranking loss [19, 30]:

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\mathbb{I}(f_i + \sigma\mathbb{I}_1(P_i, P_j) > f_j)\log(1+\exp(\hat{f}_j - \hat{f}_i)),$$

where f_i, f_j are ground truth runtime, $\mathbb{I}$ is an indicator function, and n denotes the number of training examples. Note that we augment the original ranking loss with a penalty term $\sigma\mathbb{I}_1(P_i, P_j)$, where σ is a non-negative threshold, P_i is the total number of partitions in $\mathcal{S}_i$, and $\mathbb{I}_1(x, y)$ outputs 1 when $x > y$, -1 when $x < y$, and 0 otherwise. This term additionally considers inherent partitioning overheads, and alleviates the bias toward heavily partitioning variables.

8.3.2 Knowledge-guided Search

The search uses the simulator to evaluate a strategy, and proposes candidates with low predicted runtime $\hat{f}$. It then selects a small number of qualified candidates from a large set of proposals for trial execution, as shown in Figure 8.1. We implement two search algorithm variants: random search and genetic algorithm (GA) [47]. The detailed search algorithm is illustrated in Algorithm 8.

As the strategy space is exponentially large, it is inefficient to grid search the entire space. We instantiate the constraints $\mathcal{C}$ in Equation 8.1 using two system design principles to restrict the exploration within promising regions.

Algorithm 9: Implementation of the load balance constraint c_{lb}

Inputs: $\mathcal{S}$, $\mathcal{D}$, $V'_{\mathcal{G},\theta}$

Function `load_balance_constraint` ($\mathcal{S}$, $\mathcal{D}$, $V'_{\mathcal{G},\theta}$) **:**

- Set loads of each device in $\mathcal{D}$ as $l_{cur} = [0, ..., 0]$
- Sum up the bandwidth of devices in $\mathcal{D}$: $b_{all} = \sum_{i=1}^{|\mathcal{D}|} b_{i,i}$
- Sort $V'_{\mathcal{G},\theta}$ by byte size in descending order and get $V^{s'}_{\mathcal{G},\theta}$
- **for** v *in* $V^{s'}_{\mathcal{G},\theta}$ **do**
  - $l_{all} = \sum_{i=1}^{|\mathcal{D}|} l_{cur}[i]$
  - $logits = l_{all} * [b_{1,1}/b_{all}, \ldots, b_{|V_{\mathcal{D}}|,|V_{\mathcal{D}}|}/b_{all}] - l_{cur}$
  - $j \sim$ Categorical(softmax($logits$))
  - Set the placement of v in $\mathcal{S}$ as $d_j \in \mathcal{D}$
  - $l_{cur}[j] = l_{cur}[j] +$ byte_size(v)

Load Balancing Constraint

The load balancing constraint c_{lb} is vital to alleviate the communication bottleneck. When deciding the placement for each $v_i \in V^{PS}$, c_{lb} enforces to sample the placement from a multinomial

distribution over all participating nodes in $\mathcal{D}$, where each node's probability of being chosen correlates to its current communication load and maximum bandwidth, so that nodes with higher available bandwidth are more likely to be sampled. This allows generating randomized solutions while approximately maintaining a balanced status across all nodes. The implementation of c_{lb} is presented in Algorithm 9.

Adjacent Merging Constraint

The fusion of collection operations should correspond to the model forward-backward propagation order in $\mathcal{G}$, as merging two operations in the head and tail of the model would prevent low-level scheduling overlapping communication and computation. The adjacent merging constraint c_{am} is introduced to ensure variables adjacent to each other in $\mathcal{G}$ are more likely to be grouped together. The implementation of c_{am} is presented in Algorithm 10. We show the c_{lb} and c_{am} empirically improve search efficiency and quality in Section 8.4.1.

Algorithm 10: Implementation of the adjacent merge constraint c_{am}

Inputs: $\mathcal{D}$, $V'_{\mathcal{G},\theta}$, $\mathcal{G}$, the number of total collective merging groups N

Function `adjacent_merge_constraint` $(\mathcal{D}, V'_{\mathcal{G},\theta}, \mathcal{S}, N)$ **:**

- Sort $V'_{\mathcal{G},\theta}$ based on their location in the backpropagation order indicated by $\mathcal{G}$, from input to loss function, and get $V^{s'}_{\mathcal{G},\theta}$
- Set loads $l_{cur} = [0, ..., 0]_{i=1}^{N}$ corresponding to N merge groups
- $l_{avg} = \frac{1}{N} \sum_{v \in V^{s'}_{\mathcal{G},\theta}}$ byte_size(v)
- $g = 0$
- **for** v *in* $V^{s'}_{\mathcal{G},\theta}$ **do**
  - **if** $l_{cur}[g] \geq l_{avg}$ **then**
    - $g = g + 1$
  - **if** $l_{cur}[g] < l_{avg}$ **then**
    - Set the group of v in $\mathcal{S}$ as g
    - $l_{cur}[g] = l_{cur}[g] +$ byte_size(v)

Select Evaluation Candidates

Similar to AutoTVM [30], we select the final set of candidates, that have minimized weighted sum of $\hat{f}$ and internal similarity, for distributed execution.

Formally, we solve the following optimization problem:

$$\min_{\{\mathcal{S}_i\}_{i=1}^{K} \subset \{\mathcal{S}_i\}_{i=1}^{M}} \sum_{i=1}^{K} \hat{f}(\mathcal{S}_i) + \alpha \sum_{i=1}^{K} \sum_{j=1}^{K} \text{sim}(\mathcal{S}_i, \mathcal{S}_j), \tag{8.4}$$

where $\{\mathcal{S}_i\}_{i=1}^{M}$ are the M qualified strategies filtered first using the simulator score, α denotes a trade-off coefficient, and sim is a pairwise similarity function between strategies. Solving Equa-

tion 8.4 helps deliver a set of candidates that trade off between low predicted runtime (exploitation) and low similarity (exploration). The above problem is a typical problem of submodular minimization, and we resort to the greedy algorithm for an approximate solution [114].

Estimating the Similarity between Strategies

Since we do not have a continuous representation of strategies, we have developed two approaches to estimate the similarities between strategies. The first approach estimates the similarity of the two strategies $\mathcal{S}_1, \mathcal{S}_2$ by comparing each sub strategy $s_{i1} \in \mathcal{S}_1$ and $s_{i2} \in \mathcal{S}_1$ corresponding to the variable v_i, and counting how many choices of each synchronization-affecting factor are the same, and use the final count as a measure of their similarity (higher is more similar). The second approach takes the cosine similarity between the hidden outputs of the simulator, whose inputs are $\mathcal{S}_1$ and $\mathcal{S}_2$, as a proxy of the strategy similarity. Empirically, we find the two approaches perform similarly, but the first similarity function does not depend on the simulator thus can be used when no trained simulator is available (e.g., for the baseline `AutoSync(-s)`).

8.3.3 Low-cost Optimization using Transfer Learning

Performing end-to-end strategy optimization from scratch for a target domain $\mathcal{G}_t, \mathcal{D}_t$ on large clusters or large models is costly, especially if $\mathcal{G}_t$ will be only trained once (one-shot training). For unseen $\mathcal{G}_t, \mathcal{D}_t$, we consider transferring simulators trained on the data from a source domain $\mathcal{G}_s, \mathcal{D}_s$. This would be advantageous if the source domain data already exist, or any of $\mathcal{G}_s$ and $\mathcal{D}_s$ is small so their trial run data is cheap to collect. The transferability is made possible by our feature design – the predefined features are universal due to its domain-agnostic nature. The per-variable raw features are extracted from $\mathcal{G}$, $\mathcal{S}$, and $\mathcal{D}$, where $\mathcal{G}$ is the generic low-level dataflow graph representation, capable of expressing all NN models; $\mathcal{S}$, based on $\mathcal{D}$, is generated from a strategy space invariant to models as well. Hence, the feature components of different $(\mathcal{G}, \mathcal{D}, \mathcal{S})_i$ are invariant, leaving the only variability as the the length of features, caused by the different number of variables in different $\mathcal{G}$, which, however, is absorbed by models like GAT and RNN that operate on inputs with variable lengths. Section 8.4.2 validates the effectiveness of the transferable feature representation.

8.4 Evaluation

We evaluate AutoSync's components and end-to-end performance in Section 8.4.1. We investigate transferability of trained simulators across domains in Section 8.4.2.

Models

We generate strategies on top of AutoDist, and resort to distributed TensorFlow for distributed execution [65]. We treat f as *system performance*, and conduct synchronous training[1] on 10

[1] In this chapter, we do not consider synchronization-affecting factors that would alter the algorithm or result as in the original single-node code, such as lossy compression [132], staleness [83], etc.

models with standard settings suggested by MLPerf [141], including an enlarged dense (x16x32) version of the neural collaborative filtering (NCF) [82], Transformers [209] and BERT variants [51], and various CNNs [80, 87, 191], listed in Table 7.3. We managed to train all models to the suggested accuracy, hence skip the comparisons on convergence.

Clusters

We conduct experiments on two clusters ($\mathcal{D}$):

1. *Cluster A* is an in-house cluster with 11 nodes, each equipped with a TITAN X GPU and 40GbE Ethernet switch;
2. *Cluster B* is based on AWS, consists of 4x g4dn.12xlarge nodes, each with 4 NVIDIA T4 GPUs and 50GbE full bandwidth.

They correspond to the cluster A2 and B7 in Table 7.2.

Hand-optimized Baselines

We introduce two strong hand-optimized synchronization systems as external baselines.

- `Horovod` [186] is one of the most adopted open source synchronization systems for data-parallel distributed ML. It uses *AllReduce* (*AllGather*) to synchronize dense (sparse) gradients of all model variables, and utilizes BO to autotune the merge scheme for multiple collective operation based on collected trial data in warm-up runs[2]. Per our experiments on our cluster setup, it reports up to 2x speedup than a Google-provided parameter server implementation based on distributed TensorFlow as runtime[3]. `Horovod` uses NCCL 2.4.7 for collective communication, which is the same with AutoSync.
- `PS` is a highly tuned parameter server implementation with multiple optimizations from recent PS literature incorporated. We elaborate a few notable optimizations: (1) it maintains load balance by partitioning large variables and evenly placing the shards across servers depending on their available bandwidth (i.e. correlates to its maximum bandwidth and the current load as a parameter server); (2) it uses the BO algorithm to decide the partitioning size following ByteScheduler [168]; (3) it communicates sparse gradients (*IndexSlices* in TensorFlow) using *Gather* and *Scatter* primitives (instead of *Reduce* and Broadcast) [107] to reduce communication overhead; (4) it performs hierarchical Reduce (or Gather) and Broadcast (or Scatter) on nodes with more than 1 GPUs to reduce network traffic. The optimizations are implemented on top of TensorFlow 2.0 as well, so the backend also relies on distributed TensorFlow 2.0 for distributed execution, fair with AutoSync.

We use them as collective- and PS-based baselines, respectively. Note that both methods use the same TF and NCCL version for distributed execution and communication with AutoSync, preventing variations caused by systems.

[2]Details of the optimization can be found at https://github.com/horovod/horovod/blob/master/docs/autotune.rst.

[3]Details at https://github.com/tensorflow/examples/blob/master/community/en/docs/deploy/distributed.md

Setting	Linear	RNN	GAT
NCF-dense, A	0.771	**0.894**	0.810
NCF-dense, B	0.826	**0.913**	0.830
VGG-16, A	**0.868**	0.796	0.753
VGG-16, B	0.833	**0.839**	0.692
BERT-base, A	0.758	0.746	**0.775**
BERT-base, B	0.807	**0.867**	0.760
BERT-large, A	0.780	**0.847**	0.771
BERT-large, B	**0.796**	0.755	0.784

Table 8.1: Comparisons of different model instantiations of the simulator.

Public Datasets

Throughout our research, we have collected a dataset containing nearly 10K data points of $\{(\mathcal{G}, \mathcal{D}, \mathcal{S}), f\}$, where $\mathcal{G}$ is one of the DL models in Table 7.3, $\mathcal{D}$ is one of the cluster setups from Table 7.2, $\mathcal{S}$ is randomly sampled strategy from the proposed strategy space, and f is the groundtruth runtime collected via real distributed execution.

The dataset contains strategies sampled for all 11 models in Table 7.3, ranging from fixed-formed strategies such as those used in specialized systems, and randomly explored strategies by AutoSync during the strategy auto-optimization. The data are organized into multiple folders where each folder corresponds to a domain of $(\mathcal{G}, \mathcal{D})$. Hence the dataset used in the main paper is subset containing several domains as parts of the whole collected dataset. For quick access, we have provided scripts that read $\mathcal{G}$ as dataflow graphs in standard TensorFlow 2.0 format, and read the strategies and runtime into json formats. The way to access the dataset can be found in Appendix.

8.4.1 End-to-end Results and Ablation Studies

Comparing Model Instantiations

To compare the linear model, RNN and GAT, we construct datasets using trial data collected on 6 different settings, and train them as simulators, respectively. We report their ranking accuracy on held-out test sets in Table 8.1. RNN, by leveraging raw features, outperforms linear model mostly. GAT, though additionally modeling the graph structure of $\mathcal{G}$, does not demonstrate substantial advantages. We hypothesize that GAT might need more data for training, which are unavailable in our budgeted search. We hence use RNN by default in the rest of the chapter.

Search Algorithm Comparisons

We implement both random search and genetic algorithm (GA) [47] for searching. Table 8.2 right compares them regarding optimizing the strategy for VGG16. While GA finds strategies with better average quality, it constantly gets stuck at the local minima if working with an untrustworthy simulator $\hat{f}$ (instead of using real execution data f). Hence in our experiments, we

stats	RS	GA
mean (s)	1.07	0.67
std (s)	0.63	0.03
min (s)	**0.60**	0.64
max (s)	2.34	0.79

Table 8.2: The per-iteration time statistics of 200 strategies found by RS and GA on (VGG16, A).

by default use random search.

Auto-optimization Results

We use AutoSync to optimize the strategies of NCF-dense (122M), VGG16 (138M), and BERT-large (340M). They cover 3 different NN families but are all considered "difficult-to-parallelize" workloads because of having >100M parameters. We compare two AutoSync variants with external baselines:

1. `AutoSync(-s)` where the simulator is disabled for searching. It randomly explores 30K strategies and selects 200 candidates that have minimized similarity (Section 8.3.2).
2. `AutoSync`: the full AutoSync with the budget of real evaluation on clusters as 200.

To obtain the runtime f, we run 10 warm-up iterations, then another 40 iterations of training, whose runtime is averaged as the groundtruth.

Figure 8.2 compares the *best* found strategy in 200 trials by two variants with the two manually optimized baselines. In 4 out of 6 settings, `AutoSync(-s)`, *without a simulator*, discovers strategies up to 1.4x faster than the best one in `PS` and `Horovod`. With the simulator, `AutoSync` finds strategies 1.2x to 1.6x faster than baselines. To interpret the speedup, practically BERT-large needs 2M steps [5] of training to its reported accuracy with batch size 128 (batch size 8 on 16 GPUs). A 1.2x speedup reduces the training time by 7 days, and saves approximately $2200 AWS credits *per training job* on Cluster B. Moreover, in practice a model needs to be re-trained when being applied to unseen data, but a trained simulator can be repeatedly used across jobs. Comparing `AutoSync` to `Autoysnc(-s)`: besides higher quality, the simulator guides the search to solutions sooner, e.g., on (BERT-large, A), `AutoSync` locates strategies 1.25x faster than `PS` in about 50 trials, and 1.2x using only 30 trials on Cluster B. Similar patterns are observed in other settings as well.

Search Space Evaluation

We define the metric *hit rate* – the percentage of strategies found better than hand-optimized baselines during explorations, illustrated in the right axis of Figure 8.2. Except on (NCF-dense, B), `AutoSync(-s)` reports positive hit rates and the rate is considerably large on (VGG16, B) (42.0%) and (BERT-large, B) (28.5%). When augmented with a simulator, `AutoSync` frequently hits better strategies, especially on more complex models VGG16 and BERT-large

Setting	full	raw only	predefined only
NCF-dense, A	**0.894**	**0.894**	0.883
NCF-dense, B	**0.913**	0.907	0.873
VGG16, A	**0.796**	0.785	0.734
VGG16, B	**0.839**	0.837	0.816
BERT-large, A	0.847	0.848	**0.850**
BERT-large, B	**0.755**	0.746	0.735

Table 8.3: Studies on feature importance (pairwise ranking accuracy is reported).

($> 70\%$). This verifies that factorizing the strategy w.r.t. each variable and co-optimizing multiple factors form a promising space with a profound set of strategies better than manually optimized.

Knowledge Constraints

Figure 8.2 contrasts another variant, `AutoSync(-s,-k)`, where the knowledge constraints in Section 8.3.2 are removed from `AutoSync(-s)`. `AutoSync(-s,-k)` mostly visits strategies below hand-optimized baselines, especially on complex models with a larger search space – on BERT-large all strategies by `AutoSync(-s,-k)` are far below baselines hence are skipped in the plots. Incorporating system design principles as knowledge is key to make the search manageable.

Feature Importance

We ablate $\boldsymbol{z}^{raw}$ and $\boldsymbol{z}^{pre}$ and reveal their individual effect in Table 8.3. Specifically, we train RNN simulators with: (1) only predefined features $\boldsymbol{z}^{pre}$, (2) only raw features $\boldsymbol{z}^{raw}$, (3) the full features, under 6 $(\mathcal{G}, \mathcal{D})$ settings, and compare their ranking accuracy on test sets. Using full features achieves the best accuracy, demonstrating that the predefined and raw features may contain complementary information. On the other hand, the predefined features can be beneficial at the initial phase of search as it is possible to directly rank strategies using them without training.

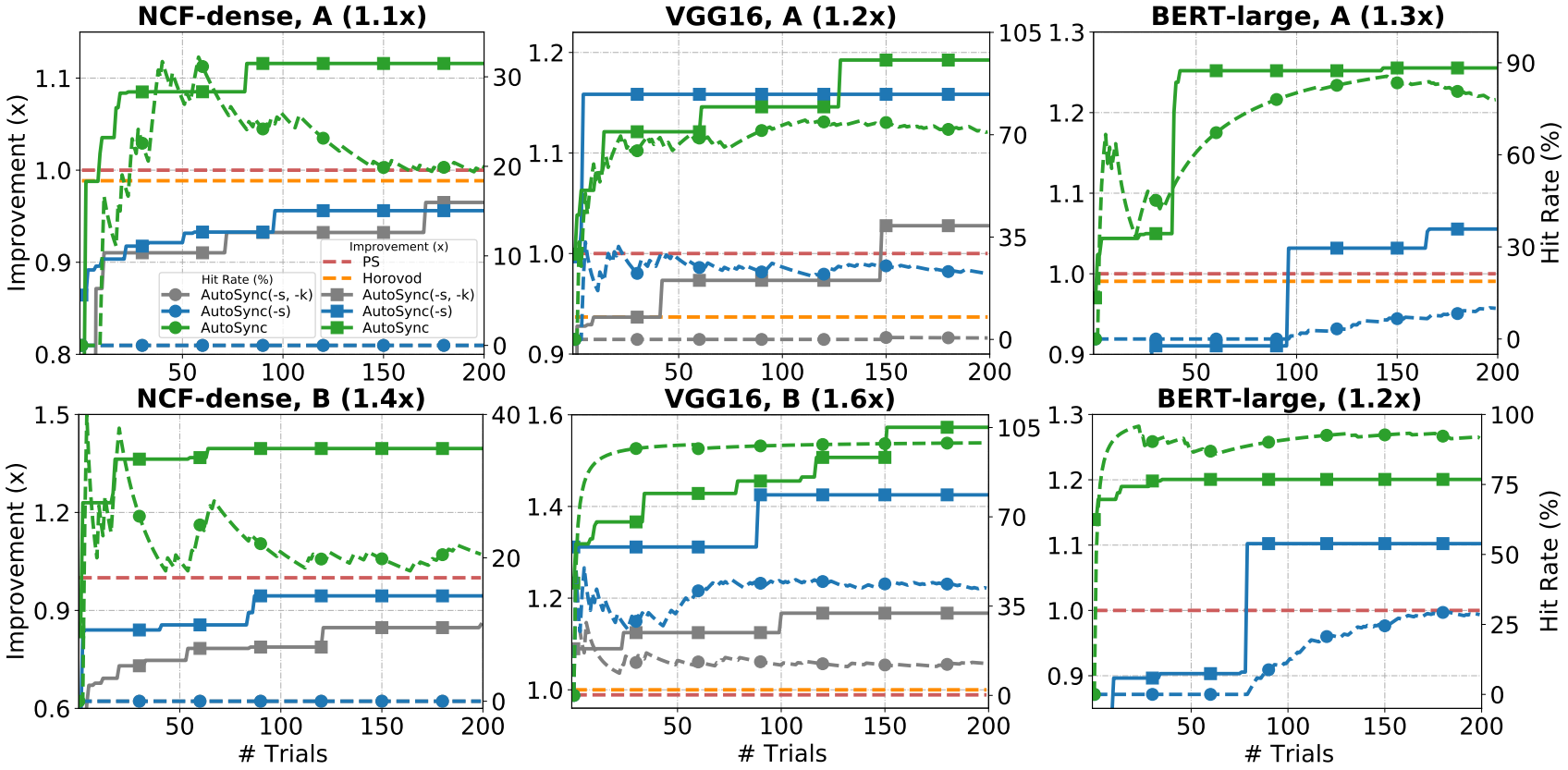

Figure 8.2: Comparing `AutoSync`, `AutoSync(-s)`, `AutoSync(-s,-k)` on (left axis) the improvement (higher is better) of the best found strategy over baseline, (right axis) the percentage of strategies better than baseline (higher is better), w.r.t. the number of trials conducted in 200 trials. The baseline (1x) is the better one of `PS` and `Horovod`). The average over 3 runs is reported. A curve is skipped from the plot if it is too below the baseline.

8.4.2 Transferring Trained Simulators

Transferability Studies

Different from Section 8.4.1, we train RNN simulators using trial data from a source domain $\mathcal{G}_s, \mathcal{D}_s$ to rank the strategies in unseen target domains $\mathcal{G}_t, \mathcal{D}_t$, and report the ranking accuracy in Table 8.4 (more in supplementary). In general, we note that: (1) Models with similar architectures exhibit higher transferability. With fixed $\mathcal{D}$, transformer [209] based models transfer between each other pretty well, with the lowest accuracy at 0.76; the transferability is slightly compromised when $\mathcal{D}$ is changed, due to increased domain distance. This hints we can use a simulator trained for smaller BERT models to optimize the strategy of a larger BERT model. (2) When $\mathcal{G}$ is fixed, transferability is observed across $\mathcal{D}$. In practice, we might pretrain a simulator using in-house cheap clusters (e.g., cluster A), and deploy the simulator for training jobs on more expensive and larger-scale clusters.

Source → Target	Accuracy
(BERT-3L, A) → (BERT-3L, B)	0.8672
(Transformer, A) → (BERT-base, A)	0.7674
(BERT-3L, A) → (BERT-base, A)	0.7591
(BERT-3L, B) → (BERT-base, B)	0.8100
(NCF-dense, A) → (BERT-base, B)	0.7904
(VGG16, A) → (BERT-base, B)	0.7298
(BERT-3L, A) → (BERT-base, B)	0.6992
(BERT-base, A) → (BERT-base, B)	0.6852
(VGG16, B) → (BERT-base, B)	0.6774
(Transformer, A) → (BERT-base, B)	0.6672
(BERT-3L, A) → (Transformer, A)	0.8866
(Transformer, B) → (Transformer, A)	0.8305
(Transformer, A) → (Transformer, B)	0.8171
(BERT-3L, A) → (Transformer, B)	0.808
(NCF-dense, A) → (NCF-dense, B)	0.7694
(VGG16, A) → (VGG16, B)	0.7966
(BERT-3L, A) → (VGG16, B)	0.5583
(ResNet50, A) → (ResNet101, A)	0.6057
(VGG16, A) → (ResNet101, A)	0.5600
(VGG16, A) → (ResNet50, A)	0.7156
(VGG16, A) → (DenseNet121, A)	0.7596
(ResNet101, A) → (ResNet101, B)	0.7187
(DenseNet121, A) → (ResNet50, B)	0.5266
(VGG16, A) → (InceptionV3, A)	0.7857

Table 8.4: The target domain test accuracy under different transfer learning settings.

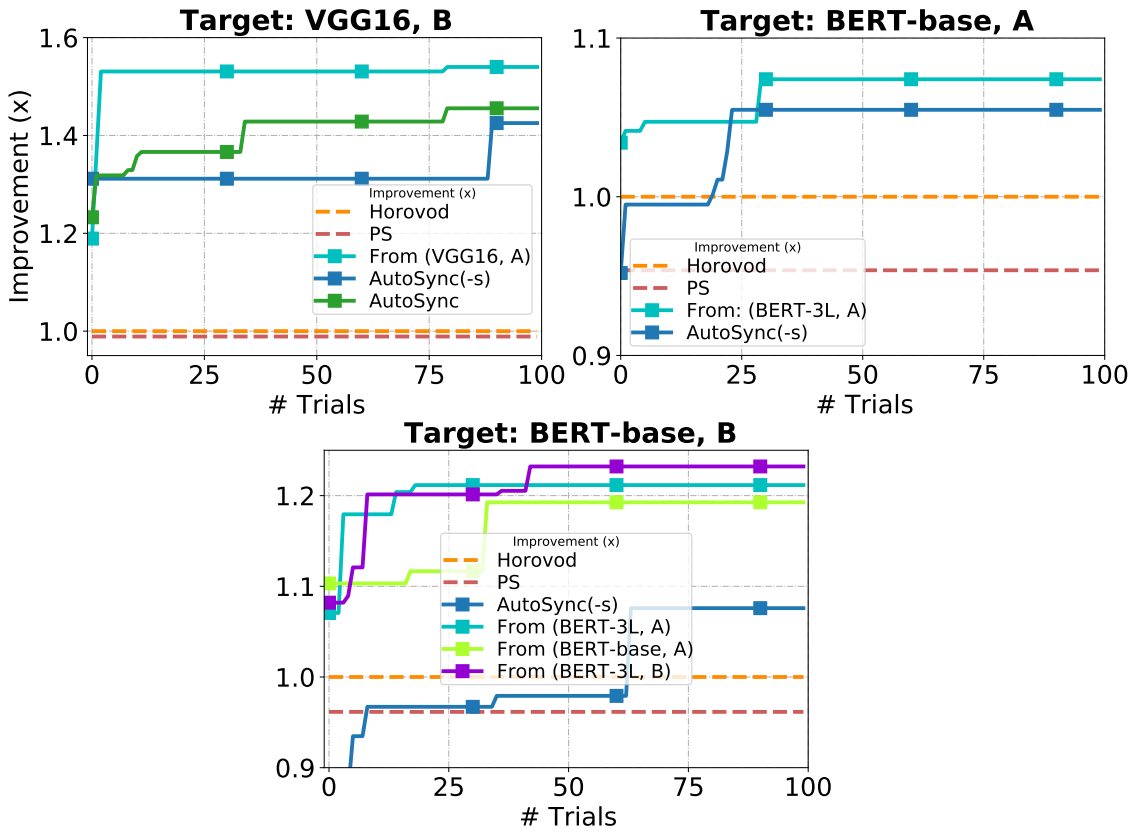

Figure 8.3: Transferring trained simulators from different source domains to 3 target domains, compared to untransferred `AutoSync-` and `AutoSync` with a budget of 100 trials. The average of three runs is reported.

End-to-end Results on Transfer Learning

We now transfer trained simulators to guide end-to-end optimizations. We deliberately target difficult-to-parallelize models and expensive clusters: (BERT-base, A), (BERT-base, B), and (VGG16, B). We set a smaller budget of 100 trials, and *do not use* any data from target domains for model selection or finetuning. Figure 8.3 illustrates the optimization progress. On (VGG16, B), surprisingly, a well-trained simulator from (VGG16, A) can find strategies 1.5x faster as soon as in 5 trials. On (BERT-base, B), we experiment with 3 source domains and notice that the domain distance does impact the end-to-end results: source (BERT-3L, B) achieves better quality and efficiency than (BERT-base, A), both better than (BERT-3L, A); (BERT-3L, A) has the largest domain distance from the target. Overall, we manage to transfer simulator trained on "cheap" domains to find good strategies in only a few trials in "expensive" domains. Besides the reduction on the number of trials, transferring a simulator effectively decreases the wall-clock time taken in auto-optimization as it bypasses simulator training, which is advantageous for scenarios where single-shot model training happens often.

8.5 Additional Related Work

ML for Systems

There is a surge of interest in applying ML to solve system problems. Mirhoseini *et al*. [148, 149] develops reinforcement learning (RL) frameworks to decide the placement of nodes in dataflow graphs. Paliwal *et al*. [163] combines RL and genetic algorithms (GA) to minimize the execution cost of NN graphs for compilers. AutoTVM [29] builds an ML-based pipeline to generate operator implementations better than hand-designed. AutoSync belongs to this line of work: it uses ML to optimize data-parallel synchronization and addresses the unique challenges therein.

Synchronization System Autotuning

Many data-parallel ML systems demonstrate certain levels of autotuning capability. Horovod and ByteScheduler introduce adjustable *knobs* and *credit size*, respectively, and use Bayesian optimization (BO) [58] to autotune their values. Parallax [107] develops a 3-parameter linear model, learned via trial data, to find the best partitioning for sparse variables in PS. These work focuses on autotuning one or two hyperparameters of a specific strategy; AutoSync contrasts them by co-optimizing a generic and holistic representation of synchronization. Among them, the closest to ours is AutoTVM [30]. We draw insights from AutoTVM, but address a fundamentally different problem – data-parallel auto-distribution of ML training on clusters – which requires the problem-specific design of strategy representations, features, runtime simulations, etc.

Chapter 9

Conclusion and Future Work

We conclude this thesis in this chapter by summarizing our contributions (Section 9.1), then discussing limitations (Section 9.2) and suggesting potential future research directions (Section 9.2).

9.1 Conclusions

Machine learning has become an essential component in industry production and in our daily lives. The availability of big datasets and sophisticated learning algorithms enables ML techniques to be applied at increasingly larger scales and more complex applications, but also poses unique challenges on computation, that simply prolonging Moore's law is no longer possible to solve, and calls for new software systems and parallelization techniques. This thesis identifies these problems in performance and usability for ML scale-up, and proposes solutions to bridge the gap between ML parallelization performance and parallel system usability.

In Part I of this thesis, we develop a principled methodology, *adaptive parallelism*, that factors the ML parallelization strategy with respect to the model characteristics and cluster specifications, and derives performance optimizations that improve ML parallelization in various apsects:

- **Scheduling and communication** (Chapter 3). We introduce an adaptive scheduling algorithm, wait-free backpropagation (WFBP), and a hybrid communication architecture. Based on the two key techniques, we build the Poseidon system, one of the state-of-the-art system that scales out CNNs to GPU clusters with limited communication bandwidth.
- **Memory** (Chapter 4). We present a memory management mechanism, memory swapping, to address the memory inadequacy of modern ML accelerators. GeePS, as an implementation of memory swapping for distributed shared-memory, allows training large models that is only bounded by the largest layer rather than the overall model size.

  Based on these set of techniques, we present a system-ML co-designed model, HD-CNN. By increasing model complexity and training scale, it achieved state-of-the-art image classification results on ImageNet.
- **Consistency** (Chapter 5). We study how staleness impacts the training of various models

and deep learning building blocks using empirical simulations, and provide insights on how the consistency model should also flexibly adapt to model specifications to trade-off between system throughput and statistical efficiency.

Generalizing the instantiations of adaptive parallelism in each specific aspect , Part II formulate ML parallelization as an optimization problem (Equation 1.1), and seek to solve it end-to-end, automatically, so to lower the barrier of distributed ML and bring it to the masses. More precisely, we concern the two broad parallelization paradigms:

- **Dynamic batching** (Chapter 6). We create a new open source DL framework, DyNet, specialized for dynamic neural networks using a new programming model – dynamic declaration. As an earlier framework for dynamic NNs, DyNet gains substantial adoptions in the NLP community.

  One step further, we develop a new vertex-centric representation for dynamic NNs. The representation motivates a vertex-centric programming model – which enables static dataflow graph optimization techniques to be utilized in dynamic neural network training, and more importantly, automates dynamic batching. Based on these techniques, we implement the system Cavs for dynamic neural networks. Cavs achieves state-of-the-art training performance on a variety of neural networks with dynamic structures.

- **Distributed parallelization** (Chapter 7). We formalizes a unified representation to express a rich set of distributed ML parallelisms that are originated from different aspects (synchronization architecture, node/layer partitioning and placement, consistency control, gradient compression, etc.). Based on the representation, parallelization strategies for new models can be composed using building blocks and exercised by assembling modular graph rewriting kernels. Using it, we develop an end-to-end compiler system, AutoDist, for ML auto-parallelization on distributed clusters. AutoDist simplifies parallel ML programming, isolates the sophistication of distributed systems from the ML prototyping, offers all-round performance on unseen models and heterogeneous clusters.

Based on the groundwork in previous chapters, in Part III of this thesis, we focus on:

- **Automatic Parallelization** (Chatper 8). we develop end-to-end solutions to automatically generate data-parallel distributed strategies given an ML model and a cluster configuration. The proposed AutoSync framework significantly lowers the technical barrier of distributed ML and helps make it accessible to a larger community of users.

The set of techniques developed in this thesis lead to the *first prototype implementation of an end-to-end compiler system* for ML training on distributed environments. The ideas and systems developed in this thesis have already made an impact in both academia and industry. Since the publications of the related papers [40, 230, 233, 245] of this thesis, we have observed growing interest in optimizing parallel ML system scheduling [168], memory [86, 93], communication [107], following the suggested methodology. Several techniques developed in this thesis, such as the scheduling technique in Chapter 3 and memory management technique in Chapter 4, have been incorporated into nowadays popular ML frameworks – TensorFlow [1]. The programming models and optimizations developed in Chapter 6 have inspired the design

of newer frameworks for dynamic models, such as PyTorch [56]. Several system implementations developed in this thesis are under commercialization in Petuum Inc., a Pittsburgh-based ML startup.

9.2 Limitations and Future Directions

This thesis is a first step toward a larger research agenda in building automatic compiler systems for distributed ML training. In this section, we discuss several limitations and try to answer the question: *What are the limitations of the proposed formulation in Equation 1.1*? In particular, are there any parallelization goals that cannot be optimized through this formulation? are there any ML programs that cannot go through this stack for auto-parallelization? are there any parallelization optimizations that cannot be expressed within the proposed representations? We informally answer these questions, and suggest potential future research directions to address them.

9.2.1 Performance Characterization for Different Parallelization Aspects

While the research results presented in this thesis mostly focus on system throughput as the optimization objectve, the ideal goal of ML parallelization is to reduce the *wallclock time to convergence*. To fit the convergence as an optimization objective in Equation 1.1, we lack the theoretical characterizations between the representations of some aspects and the ML statistical convergence. Take consistency model as an example, we need to answer the question: how is the statistical convergence precisely affected by the value of staleness, preferably in an analytic form? The similar rationale applies when we want to auto-optimize other utility functions (f in Equation 1.1) against parallelization strategies.

To address these limitations, it is necessary to develop a set of metrics for how each aspect affects ML model training convergence, computational cost and efficiency, and messaging cost and efficiency, and how these measurements change in response to different computing devices and network environments. As motivating examples, we can consider the following ML-oriented metrics: statistical variance of updates $\Delta_{\mathcal{L}}$ (which affects the stability and convergence speed of the training algorithm), convergence rate (objective function change per unit time). We shall also consider the following system-oriented metrics: bottleneck analysis (computing time spent on useful operations versus waiting for network), network latency and network bandwidth usage, message or packet loss rate, and metrics to capture stragglers by measuring the time between pairs of messages to/from a computing device. Future work shall develop rigorous definitions for each metric, their relations with specific instance choices of parallelization aspects, as well as a standard protocol for how each metric should be measured and how they can measured together.

9.2.2 Joint Optimization with Cluster Resource Scheduling

This thesis focuses on optimizing the distribution strategy of a model with *a fixed set of cluster resources*. In practice, the amount of resources allocated for a training job is usually non-transparent, hard to be decided by an average practitioner, and dynamically changing – normally

known as the problem of *ML cluster resource scheduling*.

To see this, organizations may have an inventory of on-premises machines or a limited budget for cloud VMs. They want to make sure that their existing resources are utilized efficiently by their teams, and facilitate resource sharing between many data scientists and training jobs with minimal manual intervention and performance overhead. Individual data scientists want the training jobs they submit to be completed as soon as possible so they can iterate faster to find the best models and hyper-parameters. However, the optimal amount of resources to dedicate to each training job is a highly non-transparent decision for a data scientist to make, dynamically changing, and often depends on the specific structure, hyper-parameters, and distribution strategies of the model. In order to decide the right amount of a resource, the user must know the marginal performance benefit of allocating each unit of that resource to their job. However, distributed machine learning jobs experience quickly diminishing returns. This point of diminishing returns depends on subtle properties of the model being trained, such as the ratio between the size of gradients generated and the computational complexity of each iteration, and the ability of the ML framework to optimize and overlap communication with computation. Without expert knowledge of the implementation details of the ML framework or prior experimentation, it is extremely challenging for a user to know the resource needs of their job.

While this thesis focuses on the *strategy-level* decisions for distributed DL training, i.e., looking for the optimal parallel strategies, the resource-level decisions – *how resources should be managed to maximize the overall training outcome* – are non-trivial and highly correlated with the factors considered in this thesis. We suggest future research to investigate how to answer the two questions *how many resources to allocate* and *which distribution strategy to use* in an integrated way. Solving these two problems jointly may lead to a fully automated end-to-end distributed training system for DL models, but needs to further expand the scope of Equation 1.1 to allow dynamic cluster resources $\mathcal{D}$, and more powerful representations and solutions that can capture the complex interplay between training strategies and training resources.

9.2.3 Better Intermediate Representations for ML Programs

Many approaches developed in this thesis, except in Chapter 6, adapts a dataflow graph as the representation for downstream compilation, optimization, or graph writing. Existing dataflow graph representations, such as `tf.GraphDef` in TensorFlow [1], are *static* and do not capture the *dynamic structures* in complex ML models, as elaborated in Chapter 6.

A even more severe problem arises when ML developers move more and more to imperative style programming, such as TensorFlow Eager, PyTorch, and JAX. In these programs, dynamics are expressed through typical Python programming constructs like for-loops and if-then-else. The computation structures and communicated messages are non-explicit. Obtaining a static description of the model from these programs are very difficult, if not impossible.

To solve this challenge, a necessary step is to join the line of ongoing effort in creating a more principled, unifed intermediate representation to expresses ML programs from all origins and in all kinds. Ongoing effort such as MLIR [120], Relay [176] are mostly in a preliminary stage, and focus on building IRs to facilitate the optimization of operator imlementations on different accelerate devices, and operator-level parallelism. We suggest future research to join the effort and build IRs to facilitate the graph-level optimization and graph-level parallelism. This would

require future research to define composable parallelization semantics and rewriting kernels on the IR in a similar way with what this thesis has done on dataflow graphs, and better distributed runtime to executing the IRs efficiently on distributed clusters.

Appendix A

Software and Dataset Developed in This Thesis

- Poseidon v1 (Chapter 3): https://github.com/sailing-pmls/pmls-caffe
- Poseidon v2 (Chapter 3): https://poseidon-release.readthedocs.io/en/latest/
- GeePS (Chapter 4): https://github.com/cuihenggang/geeps
- DyNet (Chapter 6): http://dynet.io/
- Cavs (Chapter 6): https://github.com/zhisbug/Cavs
- AutoDist (Chapter 7): https://github.com/petuum/autodist
- AutoSync (Chapter 8): https://github.com/petuum/autodist/tree/master/autodist/autosync
- AutoSync datasets containing approximately 10K tuples of strategy, runtime, resource specifications (Chapter 8): https://github.com/petuum/autodist

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016. [Cited on pages 1, 1, 2, 5, 23, 40, 53, 53, 53, 54, 63, 64, 113, 113, 113, 113, 114, 117, 119, 124, 124, 130, 130, 130, 131, 131, 139, 140, 151, 167, 182, and 184.]

[2] Charu C Aggarwal and ChengXiang Zhai. *Mining text data*. Springer Science & Business Media, 2012. [Cited on page 12.]

[3] Akshay Agrawal, Akshay Naresh Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, et al. Tensorflow eager: A multi-stage, python-embedded dsl for machine learning. *arXiv preprint arXiv:1903.01855*, 2019. [Cited on page 25.]

[4] Amr Ahmed, Moahmed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander J Smola. Scalable inference in latent variable models. In *Proceedings of the fifth ACM international conference on Web search and data mining*, pages 123–132, 2012. [Cited on pages 27, 67, 93, and 105.]

[5] NVIDIA AI. Bert meets gpus. https://openai.com/blog/ai-and-compute/, 2019. [Cited on pages xiii, 15, 16, and 175.]

[6] Andrew Gibiansky. Bringing hpc techniques to deep learning. https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/, 2017. [Cited on page 27.]

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014. [Cited on pages 111 and 113.]

[8] Paul Barham and Michael Isard. Machine learning systems are stuck in a rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 177–183, 2019. [Cited on page 22.]

[9] Michael Bartholomew-Biggs, Steven Brown, Bruce Christianson, and Laurence Dixon. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics*, 124(1):171–190, 2000. [Cited on page 113.]

[10] James Bergstra, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, David Warde-Farley, Ian J. Goodfellow, Arnaud

Bergeron, and Yoshua Bengio. Theano: Deep Learning on GPUs with Python. In *NIPSW*, 2011. [Cited on pages 113, 113, and 130.]

[11] Simone Bianco, Remi Cadene, Luigi Celona, and Paolo Napoletano. Benchmark analysis of representative deep neural network architectures. *IEEE Access*, 6:64270–64277, 2018. [Cited on pages xiii, 15, and 16.]

[12] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003. [Cited on page 12.]

[13] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. Variational inference: A review for statisticians. *Journal of the American statistical Association*, 112(518):859–877, 2017. [Cited on page 12.]

[14] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016. [Cited on pages 1 and 12.]

[15] Antoine Bordes, Y-Lan Boureau, and Jason Weston. Learning end-to-end goal-oriented dialog. *arXiv preprint arXiv:1605.07683*, 2016. [Cited on page 12.]

[16] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs. http://github.com/google/jax, 2018. [Cited on pages 25 and 34.]

[17] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020. [Cited on pages xiv, 1, 3, 3, 3, 18, and 19.]

[18] Jacob Buckman, Miguel Ballesteros, and Chris Dyer. Transition-based dependency parsing with heuristic backtracking. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing; 2016 Nov 1-5; Austin, Texas, USA. Stroudsburg (USA): Association for Computational Linguistics (ACL); 2016. p. 2313-18.* ACL (Association for Computational Linguistics), 2016. [Cited on page 113.]

[19] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning*, pages 129–136, 2007. [Cited on page 170.]

[20] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. *arXiv preprint arXiv:2005.12872*, 2020. [Cited on pages xiii, 17, and 18.]

[21] Chih-Chung Chang and Chih-Jen Lin. Libsvm: A library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):1–27, 2011. [Cited on page 22.]

[22] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical

language modeling. *arXiv preprint arXiv:1312.3005*, 2013. [Cited on page 156.]

[23] Hongshen Chen, Xiaorui Liu, Dawei Yin, and Jiliang Tang. A survey on dialogue systems: Recent advances and new frontiers. *Acm Sigkdd Explorations Newsletter*, 19(2):25–35, 2017. [Cited on page 12.]

[24] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016. [Cited on pages 50, 51, 54, 59, 90, 91, 91, 93, 97, 105, and 105.]

[25] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, page 1. ACM, 2015. [Cited on page 139.]

[26] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016. [Cited on pages 22 and 140.]

[27] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015. [Cited on pages 23, 40, 54, 54, 62, 63, 131, and 139.]

[28] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016. [Cited on pages 29, 30, and 148.]

[29] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018. [Cited on pages 25, 151, 159, and 180.]

[30] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, pages 3389–3400, 2018. [Cited on pages 159, 163, 170, 171, and 180.]

[31] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014. [Cited on pages 132 and 139.]

[32] Trishul Chilimbi, Yutaka Suzue Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *OSDI*, 2014. [Cited on pages 2, 42, 46, 47, 47, 53, 61, 61, 61, 61, 62, 62, 63, 63, 66, 67, 68, 80, 80, 80, 83, 84, 90, 91, 97, 105, 141, and 143.]

[33] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014. [Cited on page 111.]

[34] Adam Coates, Brody Huval, Tao Wang, David J. Wu, Andrew Y. Ng, and Bryan Catan-

zaro. Deep Learning with COTS HPC Systems. In *ICML*, 2013. [Cited on pages 63 and 66.]

[35] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3): 273–297, 1995. [Cited on page 17.]

[36] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014. [Cited on page 31.]

[37] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016. [Cited on page 31.]

[38] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R Ganger, Phillip B Gibbons, et al. Exploiting bounded staleness to speed up big data analytics. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 37–48, 2014. [Cited on pages 5, 27, 29, 67, 68, 71, 75, 75, 82, 91, 93, 105, and 141.]

[39] Henggang Cui, Alexey Tumanov, Jinliang Wei, Lianghong Xu, Wei Dai, Jesse Haber-Kucharsky, Qirong Ho, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, et al. Exploiting iterative-ness for parallel ml computations. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14, 2014. [Cited on page 27.]

[40] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 4. ACM, 2016. [Cited on pages 1, 1, 29, 30, 40, 46, 46, 49, 50, 51, 54, 54, 59, 61, 63, 64, 66, 90, 91, 93, 97, 105, 131, 148, and 182.]

[41] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth Gibson, and Eric P. Xing. Analysis of high-performance distributed ml at scale through parameter server consistency models. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence*, 2015. [Cited on pages 28, 29, and 50.]

[42] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth Gibson, and Eric P Xing. High-performance distributed ml at scale through parameter server consistency models. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. [Cited on page 144.]

[43] Wei Dai, Yi Zhou, Nanqing Dong, Hao Zhang, and Eric P Xing. Toward understanding the impact of staleness in distributed machine learning. *arXiv preprint arXiv:1810.03264*, 2018. [Cited on page 148.]

[44] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42 (12), 2009. [Cited on page 129.]

[45] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI*, page 1, 2004. [Cited on page 22.]

[46] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le,

Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large Scale Distributed Deep Networks. In *NIPS*, 2012. [Cited on pages 22, 27, 62, 63, 67, 84, 90, 90, 93, 97, 105, and 105.]

[47] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002. [Cited on pages 170 and 174.]

[48] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977. [Cited on page 12.]

[49] 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. [Cited on pages xiii, 11, 15, 16, 80, 80, and 89.]

[50] Jia Deng, Nan Ding, Yangqing Jia, Andrea Frome, Kevin Murphy, Samy Bengio, Yuan Li, Hartmut Neven, and Hartwig Adam. Large-scale object classification using label relation graphs. In *European conference on computer vision*, pages 48–64. Springer, 2014. [Cited on page 88.]

[51] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. [Cited on pages 1, 1, 3, 3, 40, 141, 142, 154, 154, 154, 154, 163, and 173.]

[52] Jeffrey Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2625–2634, 2015. [Cited on pages 81, 81, 120, and 130.]

[53] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul): 2121–2159, 2011. [Cited on pages xxii, 91, and 92.]

[54] Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A Smith. Transition-based dependency parsing with stack long short-term memory. *arXiv preprint arXiv:1505.08075*, 2015. [Cited on page 113.]

[55] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990. [Cited on pages 111 and 113.]

[56] Facebook. http://pytorch.org/, 2018. [Cited on pages 8, 40, 119, 131, 139, 167, and 183.]

[57] Facebook Open Source. Caffe2 is a lightweight, modular, and scalable deep learning framework. https://github.com/caffe2/caffe2, 2017. [Cited on pages 117, 130, and 139.]

[58] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*,

2018. [Cited on page 180.]

[59] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015. [Cited on page 11.]

[60] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010. [Cited on page 96.]

[61] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 17–30, 2012. [Cited on pages 22, 120, 139, and 139.]

[62] Ian Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. In *International conference on machine learning*, pages 1319–1327. PMLR, 2013. [Cited on page 31.]

[63] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014. [Cited on page 12 and 12.]

[64] Google. Protobuf. https://developers.google.com/protocol-buffers, 2008. [Cited on page 23.]

[65] Google. Distributed tensorflow. *https://github.com/tensorflow/examples/blob/master/community/en/docs/deploy/distributed.md*, 2016. [Cited on pages 33 and 172.]

[66] Google. Tensorflow xla. https://www.tensorflow.org/performance/xla/, 2017. [Cited on pages 25, 26, 131, 139, and 159.]

[67] Matthew R Gormley, Mark Dredze, and Jason Eisner. Approximation-aware dependency parsing by belief propagation. *arXiv preprint arXiv:1508.02375*, 2015. [Cited on page 113.]

[68] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017. [Cited on pages 2, 28, 64, 90, 93, 96, 97, 105, 141, and 159.]

[69] Édouard Grave, Armand Joulin, Moustapha Cissé, David Grangier, and Hervé Jégou. Efficient softmax approximation for gpus. *arXiv preprint arXiv:1609.04309*, 2016. [Cited on page 139.]

[70] Alex Graves, Santiago Fernàndez, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the International Conference on Machine Learning, ICML 2006*, 2006. [Cited on page 113.]

[71] Thomas L Griffiths and Mark Steyvers. Finding scientific topics. *Proceedings of the National academy of Sciences*, 101(suppl 1):5228–5235, 2004. [Cited on pages xxii, 12,

92, and 98.]

[72] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010. [Cited on page 139.]

[73] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988. [Cited on page 137.]

[74] Tobias Gysi, Carlos Osuna, Oliver Fuhrer, Mauro Bianco, and Thomas C Schulthess. Stella: A domain-specific tool for structured grid methods in weather and climate models. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pages 1–12. IEEE, 2015. [Cited on page 129.]

[75] Stefan Hadjis, Ce Zhang, Ioannis Mitliagkas, Dan Iter, and Christopher Ré. Omnivore: An optimizer for multi-device deep learning on cpus and gpus. *arXiv preprint arXiv:1606.04487*, 2016. [Cited on pages 91, 93, and 105.]

[76] 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. [Cited on page 141.]

[77] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014. [Cited on pages 1 and 12.]

[78] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018. [Cited on pages 32, 32, 64, 148, 159, and 167.]

[79] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288*, 2018. [Cited on page 159.]

[80] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015. [Cited on pages xiii, 1, 11, 16, 41, 54, 59, 93, 93, 96, 154, and 173.]

[81] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. [Cited on page 141.]

[82] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*, pages 173–182, 2017. [Cited on pages 11, 154, 163, and 173.]

[83] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip. B. Gibbons, Garth A. Gibson, Greg R. Ganger, and Eric P. Xing. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *NIPS*, 2013. [Cited on pages 2, 27, 29, 50, 51, 67, 74, 90, 91, 93, 105, 105, 141, 141, 143, 144, 144, 144, 148, and 172.]

[84] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*,

9(8):1735–1780, 1997. [Cited on pages 94, 111, 111, 111, and 113.]

[85] Zhiting Hu, Zichao Yang, Xiaodan Liang, R. Salakhutdinov, and E. Xing. Toward controlled generation of text. In *ICML*, 2017. [Cited on page 12.]

[86] 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. [Cited on pages 29, 30, 36, 70, and 182.]

[87] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017. [Cited on pages 11, 154, and 173.]

[88] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in neural information processing systems*, pages 103–112, 2019. [Cited on page 32 and 32.]

[89] Forrest N Iandola, Matthew W Moskewicz, Khalid Ashraf, and Kurt Keutzer. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2592–2600, 2016. [Cited on page 64.]

[90] Intel Open Source Technology Center. Intel(r) math kernel library for deep neural networks (intel(r) mkl-dnn). `https://github.com/01org/mkl-dnn`, 2017. [Cited on page 139.]

[91] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015. [Cited on page 63.]

[92] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 776–789. IEEE, 2018. [Cited on page 31.]

[93] 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. *arXiv preprint arXiv:1910.02653*, 2019. [Cited on pages 29, 30, and 182.]

[94] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed dnn training. *arXiv preprint arXiv:1905.03960*, 2019. [Cited on page 30.]

[95] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018. [Cited on pages 28, 64, and 159.]

[96] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Gir-

shick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *MM*, 2014. [Cited on pages 40 and 53.]

[97] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014. [Cited on pages 23, 29, 30, 53, and 78.]

[98] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. Exploring hidden dimensions in parallelizing convolutional neural networks. *arXiv preprint arXiv:1802.04924*, 2018. [Cited on pages 32, 33, and 159.]

[99] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018. [Cited on pages 32, 36, 143, 146, 146, 159, and 159.]

[100] 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*, pages 47–62, 2019. [Cited on pages 25, 151, and 159.]

[101] Wengong Jin, Regina Barzilay, and Tommi Jaakkola. Junction tree variational autoencoder for molecular graph generation. *arXiv preprint arXiv:1802.04364*, 2018. [Cited on page 12.]

[102] Michael I Jordan et al. On statistics, computation and scalability. *Bernoulli*, 19(4):1378–1390, 2013. [Cited on page 105.]

[103] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017. [Cited on page 22.]

[104] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014. [Cited on page 1.]

[105] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A Gibson, and Eric P Xing. Strads: a distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016. [Cited on page 93.]

[106] Seyoung Kim and Eric P Xing. Tree-guided group lasso for multi-response regression with structured sparsity, with an application to eqtl mapping. *The Annals of Applied Statistics*, 6(3):1095–1117, 2012. [Cited on pages 36 and 144.]

[107] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Automatic data-parallel training of deep neural networks. *arXiv preprint arXiv:1808.02621*, 2018. [Cited on pages 2, 5, 28, 48, 64, 64, 140, 141, 141, 143, 149, 155, 155, 156, 159, 162, 173, 180,

and 182.]

[108] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. [Cited on pages xxii and 92.]

[109] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013. [Cited on pages 12, 12, and 94.]

[110] Durk P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improved variational inference with inverse autoregressive flow. In *Advances in neural information processing systems*, pages 4743–4751, 2016. [Cited on page 12.]

[111] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016. [Cited on page 31.]

[112] Lingpeng Kong, Chris Dyer, and Noah A Smith. Segmental recurrent neural networks. *arXiv preprint arXiv:1511.06018*, 2015. [Cited on page 113.]

[113] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009. [Cited on page 11.]

[114] Andreas Krause and Daniel Golovin. Submodular function maximization., 2014. [Cited on page 172.]

[115] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. Master's thesis, University of Toronto, 2009. [Cited on page 53.]

[116] Alex Krizhevsky. One Weird Trick for Parallelizing Convolutional Neural Networks. In *arXiv:1404.5997*, 2014. [Cited on pages 1, 32, 33, 66, and 89.]

[117] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*, 2012. [Cited on pages xxi, xxii, 1, 1, 11, 22, 31, 41, 44, 46, 46, 46, 59, 92, 110, and 113.]

[118] Abhimanu Kumar, Alex Beutel, Qirong Ho, and Eric Xing. Fugue: Slow-worker-agnostic distributed learning for big models on big data. In *Artificial Intelligence and Statistics*, pages 531–539, 2014. [Cited on page 93.]

[119] Rasmus Munk Larsen and Tatiana Shpeisman. Tensorflow graph optimizations, 2019. [Cited on pages 25, 70, 151, and 157.]

[120] Chris Lattner, Jacques Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *arXiv preprint arXiv:2002.11054*, 2020. [Cited on pages 26 and 184.]

[121] Quoc V Le, Rajat Monga, Matthieu Devin, Kai Chen, Greg S Corrado, Jeff Dean, and Andrew Y Ng. Building High-level Features Using Large Scale Unsupervised Learning. In *ICML*, 2012. [Cited on pages 62, 62, 63, and 63.]

[122] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. [Cited on page 11.]

[123] Daniel D Lee and H Sebastian Seung. Algorithms for non-negative matrix factorization. In *Advances in neural information processing systems*, pages 556–562, 2001. [Cited on page 11.]

[124] 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. *arXiv preprint arXiv:2006.16668*, 2020. [Cited on page 159.]

[125] Hao Li, Asim Kadav, Erik Kruus, and Cristian Ungureanu. Malt: distributed data-parallelism for existing ml applications. In *Proceedings of the Tenth European Conference on Computer Systems*, page 3. ACM, 2015. [Cited on page 64.]

[126] Mu Li. *Scaling distributed machine learning with system and algorithm co-design*. PhD thesis, Carnegie Mellon University, 2017. [Cited on page 20.]

[127] 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 *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 583–598, 2014. [Cited on pages 1, 2, 2, 27, 40, 63, 90, 141, 141, 146, 155, and 162.]

[128] Mu Li, David G. Andersen, Alexander Smola, and Kai Yu. Communication Efficient Distributed Machine Learning with the Parameter Server. In *NIPS*, 2014. [Cited on pages 27, 105, and 143.]

[129] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Advances in Neural Information Processing Systems*, pages 2737–2745, 2015. [Cited on pages 93, 96, 105, and 105.]

[130] Xiaodan Liang, Xiaohui Shen, Jiashi Feng, Liang Lin, and Shuicheng Yan. Semantic object parsing with graph lstm. In *European Conference on Computer Vision*, pages 125–143. Springer, 2016. [Cited on pages xiii, xiii, 1, 17, 17, 110, 111, 112, 113, and 132.]

[131] Xiaodan Liang, Zhiting Hu, Hao Zhang, Chuang Gan, and Eric P Xing. Recurrent topic-transition gan for visual paragraph generation. *arXiv preprint arXiv:1703.07022*, 2017. [Cited on pages 1 and 111.]

[132] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017. [Cited on pages 2, 141, 143, 148, 164, and 172.]

[133] Lin Liu, Lin Tang, Wen Dong, Shaowen Yao, and Wei Zhou. An overview of topic modeling and its current applications in bioinformatics. *SpringerPlus*, 5(1):1608, 2016. [Cited on page 12.]

[134] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *NSDI*, 2013. [Cited on page 93.]

[135] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015. [Cited on pages xiii, 11, and 17.]

[136] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017. [Cited on pages 115, 117, 118, 131, 131, 132, 132, and 139.]

[137] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *arXiv preprint arXiv:1204.6078*, 2012. [Cited on page 22.]

[138] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014. [Cited on pages 93, 93, and 105.]

[139] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010. [Cited on page 139.]

[140] M. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Comput. Linguistics*, 19:313–330, 1993. [Cited on page 94.]

[141] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. Mlperf training benchmark. *arXiv preprint arXiv:1910.01500*, 2019. [Cited on pages xxii, 153, 154, 154, 154, and 173.]

[142] H. Brendan McMahan and Matthew Streeter. Delay-Tolerant Algorithms for Asynchronous Distributed Online Learning. In *NIPS*, 2014. [Cited on page 91.]

[143] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. Training deeper models by gpu memory optimization on tensorflow. In *LearningSys Workshop*, 2018. [Cited on pages 29 and 30.]

[144] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016. [Cited on page 22.]

[145] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017. [Cited on page 31.]

[146] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. In *ICLRW*, 2013. [Cited on page 3.]

[147] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013. [Cited on page 3.]

[148] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International*

Conference on Machine Learning-Volume 70, pages 2430–2439. JMLR. org, 2017. [Cited on pages 32, 36, 143, 146, 159, 159, and 180.]

[149] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. A hierarchical model for device placement. In *International Conference on Learning Representations*, 2018. [Cited on pages 32 and 180.]

[150] Ioannis Mitliagkas, Ce Zhang, Stefan Hadjis, and Christopher Ré. Asynchrony begets momentum, with an application to deep learning. In *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 997–1004. IEEE, 2016. [Cited on pages 91 and 98.]

[151] Dan Moldovan, James M Decker, Fei Wang, Andrew A Johnson, Brian K Lee, Zachary Nado, D Sculley, Tiark Rompf, and Alexander B Wiltschko. Autograph: Imperative-style coding with graph-based performance. *arXiv preprint arXiv:1810.08061*, 2018. [Cited on page 25.]

[152] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I Jordan. Sparknet: Training deep networks in spark. *arXiv preprint arXiv:1511.06051*, 2015. [Cited on page 64.]

[153] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, and Michael I Jordan. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 561–577, 2018. [Cited on page 140.]

[154] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013. [Cited on pages 5, 23, and 113.]

[155] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*, 2010. [Cited on page 93.]

[156] NVIDIA nccl tests. Performance reported by nccl tests, 2018. URL https://github.com/NVIDIA/nccl-tests/blob/master/doc/PERFORMANCE.md. [Cited on pages 28 and 168.]

[157] Willie Neiswanger, Chong Wang, and Eric P Xing. Asymptotically exact, embarrassingly parallel mcmc. In *Proceedings of the Thirtieth Conference on Uncertainty in Artificial Intelligence*, pages 623–632, 2014. [Cited on page 105.]

[158] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, et al. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017. [Cited on pages xiii, 17, 24, 113, 115, 117, 124, 130, 130, 131, and 131.]

[159] Graham Neubig, Yoav Goldberg, and Chris Dyer. On-the-fly operation batching in dynamic computation graphs. *arXiv preprint arXiv:1705.07860*, 2017. [Cited on pages 116, 117, 118, 131, 132, and 138.]

[160] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex

Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016. [Cited on pages 1 and 12.]

[161] OpenAI. Ai and compute. https://openai.com/blog/ai-and-compute/, 2018. [Cited on pages xiii, 14, and 15.]

[162] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. Reinforced genetic algorithm learning for optimizing computation graphs. *arXiv preprint arXiv:1905.02494*, 2019. [Cited on page 36 and 36.]

[163] Aditya Paliwal, Felix Gimeno, Vinod Gopal Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. Reinforced genetic algorithm learning for optimizing computation graphs. In *ICLR*, 2020. [Cited on page 180.]

[164] Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. Thumbs up?: sentiment classification using machine learning techniques. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pages 79–86. Association for Computational Linguistics, 2002. [Cited on page 112.]

[165] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, and Luca Antiga. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019. [Cited on pages 1, 2, 23, and 140.]

[166] Pitch Patarasuk and Xin Yuan. Bandwidth efficient all-reduce operation on tree topologies. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8. IEEE, 2007. [Cited on page 27.]

[167] 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*, page 3. ACM, 2018. [Cited on page 34.]

[168] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29, 2019. [Cited on pages 30, 36, 36, 64, 140, 146, 148, 149, 154, 155, 155, 156, 159, 162, 173, and 182.]

[169] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, 2010. [Cited on page 67.]

[170] Dan Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 1999. [Cited on page 129.]

[171] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019. [Cited on page 1.]

[172] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism,

locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48 (6):519–530, 2013. [Cited on page 129.]

[173] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011. [Cited on pages 29, 90, and 105.]

[174] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015. [Cited on page 11.]

[175] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016. [Cited on page 29.]

[176] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: A new ir for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 58–68, 2018. [Cited on pages 26 and 184.]

[177] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015. [Cited on page 11.]

[178] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018. [Cited on page 25.]

[179] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016. [Cited on pages xxii and 92.]

[180] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *IJCV*, pages 1–42, 2015. [Cited on page 54 and 54.]

[181] Ruslan Salakhutdinov. Learning deep generative models. *Annual Review of Statistics and Its Application*, 2:361–385, 2015. [Cited on page 12.]

[182] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, pages 285–295, 2001. [Cited on page 14.]

[183] Steven L Scott, Alexander W Blocker, Fernando V Bonassi, Hugh A Chipman, Edward I George, and Robert E McCulloch. Bayes and big data: The consensus monte carlo algorithm. *International Journal of Management Science and Engineering Management*, 11 (2):78–88, 2016. [Cited on page 105.]

[184] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient

descent and its application to data-parallel distributed training of speech dnns. In *INTERSPEECH*, pages 1058–1062, 2014. [Cited on pages 31 and 64.]

[185] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. On parallelizability of stochastic gradient descent for speech dnns. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 235–239. IEEE, 2014. [Cited on page 64.]

[186] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018. [Cited on pages 1, 1, 2, 2, 2, 3, 27, 27, 28, 64, 140, 141, 141, 141, 143, 144, 144, 149, 157, 159, 162, 168, 168, and 173.]

[187] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017. [Cited on pages 1, 32, 32, and 88.]

[188] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, and Cliff Young. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*, pages 10414–10423, 2018. [Cited on pages 32, 146, 159, and 159.]

[189] Julian Shun. *Shared-memory parallelism can be simple, fast, and scalable*. PUB7255 Association for Computing Machinery and Morgan & Claypool, 2017. [Cited on page 18.]

[190] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016. [Cited on pages 1 and 12.]

[191] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. [Cited on pages 1, 11, 40, 46, 46, 54, 89, 89, 154, 163, and 173.]

[192] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*, 2015. [Cited on pages xxi, 46, 46, 46, 54, 59, and 110.]

[193] Richard Socher, Cliff C Lin, Chris Manning, and Andrew Y Ng. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 129–136, 2011. [Cited on pages 112 and 113.]

[194] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013. [Cited on pages 110, 111, and 132.]

[195] Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. Ucf101: A dataset of 101 human actions classes from videos in the wild. *arXiv preprint arXiv:1212.0402*, 2012.

[Cited on page 81.]

[196] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 8(11):1214–1225, 2015. [Cited on page 139.]

[197] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *Thirteenth Annual Conference of the International Speech Communication Association*, 2012. [Cited on page 131.]

[198] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014. [Cited on pages 3, 3, 111, 113, 120, 130, and 131.]

[199] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018. [Cited on page 12.]

[200] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *CVPR*, 2015. [Cited on pages 31, 54, 59, and 89.]

[201] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015. [Cited on page 110.]

[202] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *arXiv preprint arXiv:1512.00567*, 2015. [Cited on pages 54, 59, and 154.]

[203] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015. [Cited on pages xvii, 110, 110, 111, 112, 112, 113, 120, 121, and 132.]

[204] The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 2016. [Cited on page 23.]

[205] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015. URL http://learningsys.org/papers/LearningSys_2015_paper_33.pdf. [Cited on pages 131 and 139.]

[206] Aaron Van den Oord, Nal Kalchbrenner, Lasse Espeholt, Oriol Vinyals, Alex Graves, et al. Conditional image generation with pixelcnn decoders. In *Advances in neural information processing systems*, pages 4790–4798, 2016. [Cited on page 12.]

[207] Vladimir Vapnik. *The nature of statistical learning theory*. Springer science & business

media, 2013. [Cited on page 17.]

[208] 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. *arXiv preprint arXiv:1802.04730*, 2018. [Cited on page 25.]

[209] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017. [Cited on pages 3, 18, 154, 173, and 178.]

[210] Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. In *Advances in Neural Information Processing Systems*, pages 2773–2781, 2015. [Cited on page 132.]

[211] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011. [Cited on page 124.]

[212] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil Devanur, and Ion Stoica. Blink: Fast and generic collectives for distributed ml. *arXiv preprint arXiv:1910.04940*, 2019. [Cited on page 168.]

[213] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. Collaborative deep learning for recommender systems. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1235–1244, 2015. [Cited on page 11.]

[214] Hongyi Wang, Scott Sievert, Shengchao Liu, Zachary Charles, Dimitris Papailiopoulos, and Stephen Wright. Atomo: Communication-efficient learning via atomic sparsification. In *Advances in Neural Information Processing Systems*, pages 9850–9861, 2018. [Cited on page 31.]

[215] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 41–53, 2018. [Cited on page 29.]

[216] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 26. ACM, 2019. [Cited on pages 32, 143, 146, 146, 159, and 159.]

[217] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Guilin Liu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. Video-to-video synthesis. In *Advances in Neural Information Processing Systems*, pages 1144–1156, 2018. [Cited on page 1.]

[218] Wei Wang, Gang Chen, Tien Tuan Anh Dinh, Jinyang Gao, Beng Chin Ooi, Kian-Lee Tan, and Sheng Wang. SINGA: Putting Deep Learning in the Hands of Multimedia Users. In *MM*, 2015. [Cited on page 63.]

[219] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. In *Proceedings*

of the Seventh ACM Symposium on Cloud Computing, pages 84–97. ACM, 2016. [Cited on page 64.]

[220] Jinliang Wei. *Scheduling for Efficient Large-Scale Machine Learning Training*. PhD thesis, Carnegie Mellon University, 2019. [Cited on pages xiv, 20, and 27.]

[221] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Managed Communication and Consistency for Fast Data-parallel Iterative Analytics. In *SoCC*, 2015. [Cited on pages 40, 42, 48, 49, 63, 75, 143, 144, and 162.]

[222] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *SoCC*, 2015. [Cited on pages 1, 2, 2, 5, 27, 29, and 105.]

[223] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*, pages 1509–1519, 2017. [Cited on page 31.]

[224] Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. Deep image: Scaling up image recognition. *arXiv*, pages arXiv–1501, 2015. [Cited on pages 68 and 89.]

[225] Pengtao Xie, Jin Kyu Kim, Yi Zhou, Qirong Ho, Abhimanu Kumar, Yaoliang Yu, and Eric Xing. Distributed Machine Learning via Sufficient Factor Broadcasting. In *arXiv*, 2015. [Cited on pages 42, 43, 48, and 64.]

[226] Pengtao Xie, Jin Kyu Kim, Qirong Ho, Yaoliang Yu, and Eric Xing. Orpheus: Efficient distributed machine learning via system and algorithm co-design. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13, 2018. [Cited on pages 27, 31, 33, 141, 144, 144, 148, and 164.]

[227] 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 *KDD*, 2015. [Cited on page 105.]

[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. *IEEE Transactions on Big Data*, 1(2):49–67, 2015. [Cited on page 105.]

[229] Haowen Xu, Hao Zhang, Zhiting Hu, Xiaodan Liang, Ruslan Salakhutdinov, and Eric Xing. Autoloss: Learning discrete schedules for alternate optimization. *arXiv preprint arXiv:1810.02442*, 2018. [Cited on page 163.]

[230] Shizhen Xu, Hao Zhang, Graham Neubig, Wei Dai, Jin Kyu Kim, Zhijie Deng, Qirong Ho, Guangwen Yang, and Eric P Xing. Cavs: An efficient runtime system for dynamic neural networks. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 937–950, 2018. [Cited on page 182.]

[231] Yahoo. Caffe on spark. http://yahoohadoop.tumblr.com/post/

129872361846/large-scale-distributed-deep-learning-on-hadoop, 2016. [Cited on page 64.]

[232] Masafumi Yamazaki, Akihiko Kasagi, Akihiro Tabuchi, Takumi Honda, Masahiro Miwa, Naoto Fukumoto, Tsuguchika Tabaru, Atsushi Ike, and Kohta Nakashima. Yet another accelerated sgd: Resnet-50 training on imagenet in 74.7 seconds. *arXiv preprint arXiv:1903.12650*, 2019. [Cited on pages 28 and 159.]

[233] Zhicheng Yan, Hao Zhang, Vignesh Jagadeesh, Dennis DeCoste, Wei Di, and Robinson Piramuthu. Hd-cnn: Hierarchical deep convolutional neural network for image classification. *ICCV*, 2015. [Cited on pages 1, 88, 88, 89, 110, and 182.]

[234] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. Imagenet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*, pages 1–10, 2018. [Cited on page 28.]

[235] Dong Yu, Kaisheng Yao, and Yu Zhang. The computational network toolkit [best of the web]. *IEEE Signal Processing Magazine*, 32(6):123–126, 2015. [Cited on pages 53, 61, 61, 61, and 64.]

[236] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, et al. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference*, page 18. ACM, 2018. [Cited on pages 26 and 115.]

[237] Jinhui Yuan, Fei Gao, Qirong Ho, Wei Dai, Jinliang Wei, Xun Zheng, Eric Po Xing, Tie-Yan Liu, and Wei-Ying Ma. Lightlda: Big topic models on modest computer clusters. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1351–1361. International World Wide Web Conferences Steering Committee, 2015. [Cited on pages 1, 93, and 105.]

[238] Joe Yue-Hei Ng, Matthew Hausknecht, Sudheendra Vijayanarasimhan, Oriol Vinyals, Rajat Monga, and George Toderici. Beyond short snippets: Deep networks for video classification. In *CVPR*, 2015. [Cited on page 87.]

[239] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59 (11):56–65, 2016. [Cited on page 22.]

[240] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014. [Cited on page 131.]

[241] Hao Zhang, Zhiting Hu, Jinliang Wei, Pengtao Xie, Gunhee Kim, Qirong Ho, and Eric Xing. Poseidon: A system architecture for efficient gpu-based deep learning on multiple machines. *arXiv preprint arXiv:1512.06216*, 2015. [Cited on pages 2, 27, 46, 80, and 131.]

[242] Hao Zhang, Gunhee Kim, and Eric P Xing. Dynamic topic modeling for monitoring market competition from online text and image data. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1425–1434, 2015. [Cited on page 12.]

[243] Hao Zhang, Zhiting Hu, Yuntian Deng, Mrinmaya Sachan, Zhicheng Yan, and Eric Xing. Learning concept taxonomies from multi-modal data. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1791–1801, 2016. [Cited on page 1.]

[244] Hao Zhang, Shizhen Xu, Graham Neubig, Wei Dai, Qirong Ho, Guangwen Yang, and Eric P Xing. Cavs: A vertex-centric programming interface for dynamic neural networks. *arXiv preprint arXiv:1712.04048*, 2017. [Cited on pages 1 and 24.]

[245] 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. [Cited on pages 1, 1, 2, 2, 28, 93, 140, 141, 143, 148, 149, 154, 159, 159, 162, 164, and 182.]

[246] Jiani Zhang, Xingjian Shi, Junyuan Xie, Hao Ma, Irwin King, and Dit-Yan Yeung. Gaan: Gated attention networks for learning on large and spatiotemporal graphs. *arXiv preprint arXiv:1803.07294*, 2018. [Cited on page 169.]

[247] Ruiliang Zhang and James Kwok. Asynchronous distributed admm for consensus optimization. In *International Conference on Machine Learning*, pages 1701–1709, 2014. [Cited on pages 90, 105, and 105.]

[248] Huasha Zhao and John Canny. Butterfly mixing: Accelerating incremental-update algorithms on clusters. In *Proceedings of the 2013 SIAM International Conference on Data Mining*, pages 785–793. SIAM, 2013. [Cited on page 27.]

[249] Shuai Zheng, Sadeep Jayasumana, Bernardino Romera-Paredes, Vibhav Vineet, Zhizhong Su, Dalong Du, Chang Huang, and Philip HS Torr. Conditional random fields as recurrent neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1529–1537, 2015. [Cited on page 113.]

[250] Martin Zinkevich, John Langford, and Alex J Smola. Slow learners are fast. In *Advances in neural information processing systems*, pages 2331–2339, 2009. [Cited on page 90.]

[251] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016. [Cited on page 163.]

[252] Yongqiang Zou, Xing Jin, Yi Li, Zhimao Guo, Eryu Wang, and Bin Xiao. Mariana: Tencent Deep Learning Platform and its Applications. In *VLDB Endowment*, 2014. [Cited on page 63.]