TOWARDS BUILDING EFFICIENT COMPUTATION AND COMMUNICATION MODELS FOR DEEP LEARNING SYSTEMS

Sahil Tyagi

Submitted to the faculty of the Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in the Department of Intelligent Systems Engineering,

Indiana University Bloomington

September 2024

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

Martin Swany, Ph.D.
(Chair)

Dingwen Tao, Ph.D.

Fan Chen, Ph.D.

David J. Crandall, Ph.D.

Date of Defense: 9/9/2024

In loving memory of Bibbi and Pitaji.

And for everyone in Basai.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to all the people who supported me throughout this journey. Your constant love, support and camaraderie through the ups and downs was instrumental in the completion of this work.

I am grateful to my advisor, Dr. Martin Swany for his unwavering support and guidance throughout my doctoral journey. Martin was crucial in forging my research trajectory, and the autonomy I received at work helped mold me into an independent researcher. A heartfelt thanks to my committee members: Dr. Fan Chen, Dr. Dingwen Tao and Dr. David Crandall for their invaluable feedback and suggestions in shaping this thesis. I extend my gratitude to Dr. Alex Shroyer and Jeremy Musser for debugging all cluster-related issues and datacenter visits, and Allan Streib for resolving all the raised tickets in a timely manner. I am grateful to the agencies that supported my research over the years: Indiana University Bloomington (IUB) and the National Science Foundation (NSF). Thank you Dr. Judy Fox and Dr. Prateek Sharma for the early collaboration opportunities at IUB.

Above all, I am grateful to my family; your support made it possible for me to come here and pursue my passion. I could not have done this without you.

Sahil Tyagi

TOWARDS BUILDING EFFICIENT COMPUTATION AND COMMUNICATION MODELS FOR DEEP LEARNING SYSTEMS

Deep Learning (DL) and Federated Learning (FL) continue to be a challenge, especially with the growing size and complexity of state-of-the-art (sota) Deep Neural Networks (DNNs). With big data and Internet-of-Things (IoT) resulting in an exponential growth of multimodal data over the last two decades, these combined factors further exacerbate the training performance of DNNs. Parallelization techniques with specialized accelerators and high-speed interconnects lower the training time and improve the overall scaling efficiency. However, such methods assume high-performance, homogeneous resources and high-network speeds, which may not always hold true in the cloud and shared clusters. Systems with low-compute, heterogeneous nodes and fluctuating network performance can significantly raise either training time or cost. Similarly, FL on the edge suffers poor performance due to transient, low-compute devices, high-latency, low-bandwidth networks and skewed data distribution. Thus, there is a large and complex trade-off space between parallel and statistical performance of distributed DL and FL systems.

This thesis addresses such challenges by developing novel algorithms and adaptive systems to aid distributed training across a wide array of compute resources, from high-performance datacenters to resource-constrained edge devices. We make contributions at the intersection of distributed learning and systems research, broadly classified under three scopes: optimizing computation cost in DL, attenuating communication cost in distributed DL and accelerating convergence in FL. This thesis improves the convergence time and quality in heterogeneous clusters under varying communication patterns, and explores time-cost trade-offs of training in the cloud. The proposed adaptive compression techniques significantly reduce the communication cost while preserving model quality, and identifies the optimal collective under networks with variable latency and bandwidth. For federated settings, we develop a framework for fast learning over unbalanced, streaming-data, and propose a novel, semi-synchronous training mechanism over bounded networks.

TABLE OF CONTENTS

Section 3: Accelerating Convergence in Federated Learning

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

AG . . . . . All-Gather

AR . . . . . All-Reduce

AI . . . . . Artificial Intelligence

ASP . . . . . Asynchronous Parallel

BLEU . . . . . Bilingual Evaluation Understudy

BSP . . . . . Bulk-Synchronous Parallel

CF . . . . . Compression Factor

CIFAR . . . . . Canadian Institute For Advanced Research

CNC . . . . . Compression to No-Compression ratio

CPU . . . . . Central Processing Unit

CUDA . . . . . Compute Unified Device Architecture

DGC . . . . . Deep Gradient Compression

DL . . . . . Deep Learning

DNNs . . . . . Deep Neural Networks

DefDP . . . . . Default Data-Partitioning

DDL . . . . . Distributed Deep Learning

EWMA . . . . . Exponentially Weighted Moving Average

FedAvg . . . . . Federated Averaging

FL . . . . . Federated Learning

FLOPS . . . . . Floating-point operations per-second

GPU . . . . . Graphic Processing Unit

Gbps . . . . . Gigabit per-second

HL . . . . . Heterogeneity-Level

HPC . . . . . High-Performance Computing

IID . . . . . Independent and Identically Distributed

IoT . . . . . Internet-of-Things

KDE . . . . . Kernel Density Estimates

LR . . . . . Learning-rate

LSTM . . . . . Long Short-Term Memory

LSSR . . . . . Local-to-Synchronous Step Ratio

LWTop-*k* . . . . . Layer-wise Top-*k* compression

MOO . . . . . Multi-Objective Optimization

MPI . . . . . Message Passing Interface

MSTop-*k* . . . . . Multiple Samplings Top-*k* compression

NCCL . . . . . NVIDIA Collective Communications Library

NIC . . . . . Network Interface Card

PS . . . . . Parameter Server

PTB . . . . . Penn Treebank

PID . . . . . Proportional-Integral-Derivative

PrC . . . . . Prior-Compression

PoC . . . . . Post-Compression

QoS . . . . . Quality-of-Service

ResNet . . . . . Residual Network

SelDP . . . . . `SelSync` Data-Partitioning

SGD . . . . . *Stochastic Gradient Descent*

sota . . . . . state-of-the-art

SSP . . . . . Stale Synchronous Parallel

`tc` . . . . . Traffic Control

TPU . . . . . Tensor Processing Unit

TTA . . . . . Time To Accuracy

ViT . . . . . Vision Transformer

VGG . . . . . Visual Geometry Group

VM . . . . . Virtual Machine

CHAPTER 1
INTRODUCTION

1.1 Overview

In the last decade, DNNs have achieved sota performance in diverse tasks like computer vision and recognition [1, 2], natural language processing, machine translation, anomaly or outlier detection [3], chat or prompt-based search engine, etc. [4, 5]. Algorithmic innovation drives the major milestones achieved in the field of Artificial Intelligence (AI) over the years. According to the scaling hypothesis [6], effective DNNs are built on a scalable architecture. In late 20th century, this was done by stacking multiple layers of fully connected layers to solve non-linear problems [7]. Later, DL models like AlexNet [1] and VGG [8] stacked many convolutional and dense layers to attain cutting-edge results. Skip-connections based residual networks [9] and attention-based transformers [4, 5] have varying depths based on the number of layers. Although scaling up such architectures produces generalizable and accurate models, this comes at the cost of high computational requirements for training or inference. However, as explained in Sutton's bitter lesson [10], greater AI systems are built by scaling computation and learning model parameters iteratively and repeatedly, rather than solely developing new ways to embed human knowledge into new compute agents. In 2018, OpenAI [11] observed that the computational requirements to train DNNs approximately doubles every 3.4 months [12], while updated surveys stretch this window to $\approx$ 5-6 months [13].

As per Von Neumann architecture [14], the overall computing speedup is governed by how fast and efficiently instructions are executed, along with data-transfer speeds from memory store to compute agent over the bus, as illustrated in Fig. (1.1). Thus, each of the three components, 'compute', 'memory' and 'interconnect' are crucial for optimal execution. In DL context, the output is some input data-dependent prediction and the task is to learn latent features or model parameters (i.e., weights and biases). 'Compute' efficiency in DL can be improved by performing more operations per-unit of time, which may correspond to Floating-point operations per-second (FLOPS) or MAC (multiply and accumulate) operations. As training is iterative and composed of many repetitive iterations or steps, compute in the context of DL may also imply the total computation cost of training a

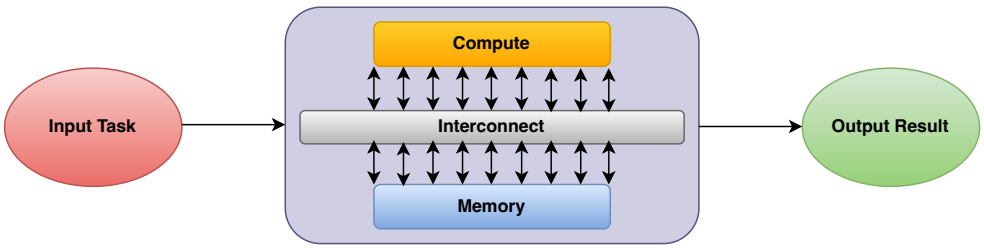


Figure 1.1: Conventional execution model in a Von Neumann computer. End-to-end execution cost depends on compute, memory and interconnect bandwidth.

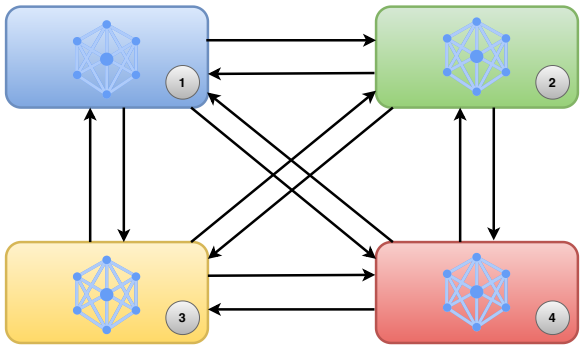


Figure 1.2: Multiple machines run collaboratively in a distributed manner and perform a task that is exceedingly large or slow for a single computer. The topology or node arrangement may vary across systems (a decentralized per-to-peer is shown here).

model to convergence. DL training also requires ample 'memory' to accommodate batches of training data, model parameters, gradients, activation maps and optimization-specific data (e.g., first or second-order gradient information). A slow 'interconnect' may also pose transmission delays between memory and compute that could lead to application stalling. The compute component such as Central Processing Unit (CPU), memory at different hierarchies and interconnects like buses represent different bottlenecks. Development of specialized accelerators like GPU, Tensor Processing Unit (TPU) [15] and FPGAs (field programmable gate arrays) [16] have significantly accelerated compute (especially for DL tasks) and rendered interconnects (i.e., data-movement) as the major execution bottleneck. Along with algorithmic innovations, significant compute gains in commodity hardware (with Moore's law [17]), development of specialized accelerators like GPUs and TPUs, wide availability low-cost resources and High-Performance Computing (HPC) have aided the development and training of DNNs.

In addition to the growing size and complexity of DNNs, explosion of big data and IoT has

resulted in an exponential growth of multimodal data, thus making it crucial to integrate DL with distributed computing. This involves running computations across a cluster of machines to achieve a common objective [18] Fig. (1.2). Instead of a single computer, a network of interconnected nodes communicate with each other to solve large, complex tasks collaboratively. The goal of a distributed system is to provide scalability, consistency, transparency, availability and efficiency. Systems today are available over a cluster, grid, cloud, peer-to-peer (P2P) networks, etc. Frameworks like Mapreduce-based Hadoop [19], Spark [20], Storm [21], and parallelization libraries like OpenMP [22], OpenSHMEM [23] and Message Passing Interface (MPI) [24] provide abstractions to run large scales of computations. Running DL models on distributed systems presents challenges akin to those observed in a conventional Von Neumann computer with respect to compute, interconnect speed and data-movement. *Distributed training strategies also incur significant cost due to computation, communication and I/O overheads*. It is crucial to optimize these components in order to improve the overall parallel efficiency of DL in a distributed cluster. Further, as Moore's law trickles down to the edge, it enables greater computational capability for *collaborative* or *federated* learning [25], and train models while conscious of network bandwidth while preserving data privacy and locality.

The primary optimization algorithm for DNNs training, Gradient descent [7, 26] is inherently stochastic. A model's trajectory towards the minima over a non-convex loss landscape depends on multiple training variables or *hyperparameters*. There is a statistical aspect of DNNs training on account of its stochasticity as the work performed over the iterations does not fully compose. Thus, the features learned by a model cannot be equally attributed to each of its training iterations. In fact, there are critical or sensitive regions observed over the training phase in DL where some iterations perform more significant updates than others [27, 28].

To achieve high throughput and good quality DNNs, both parallel and statistical efficiency thus need be considered in the context of distributed systems. Unlike conventional parallel applications where all the computations performed are equally useful, DNNs have a statistical efficiency aspect as well. In distributed settings, the cumulative progress made (like improvement in overall accuracy or decrease in loss) is not equal to the progress made by each worker uniformly over the iterations. To develop systems and algorithms with high execution speedups and sota convergence quality, it is imperative to consider both.

1.2 Thesis Contributions

The objective of this thesis is to study, propose and validate methods that optimize compute and communication for distributed DL and FL systems while accounting for both parallel and statistical performance. The former relates to parallel scaling in distributed training with regard to computation, communication or I/O, while the latter pertains to convergence quality and generalizability. Classified under three scopes, we develop six novel methods to optimize computation and communication in distributed learning, while cognizant of a DNNs' statistical performance to ensure high convergence quality:

- **Optimizing computation cost in deep learning**
    1. Optimizing distributed training over heterogeneous systems
    2. Balancing <cost, time> trade-offs of DL in the cloud
- **Mitigating communication cost in distributed deep learning**
    3. Adaptive gradient compression in distributed learning
    4. Flexible compression-communication mechanism over unpredictable networks
- **Accelerating convergence in federated learning**
    5. Real-time learning over streaming data
    6. Federated learning with semi-synchronous communication

1.2.1 Optimizing Distributed Training over Heterogeneous Systems

Conventional distributed DL algorithms are ideal for datacenter-grade machines composed of homogeneous resources. However, resource heterogeneity is pervasive across computing infrastructures and applications are often deployed on servers with different size and configurations. In communication-bound synchronous training where updates are aggregated at the end of each iteration, training over heterogeneous nodes results in "stragglers" where fast workers must wait on slower nodes to cross the synchronization barrier. As a result, parallel scaling and overall training time is exacerbated in synchronous training due to heterogeneity. Although asynchronous training

does not have a synchronization barrier, its statistical performance, i.e., final test accuracy is worse due to "staleness" in model updates.

We develop `OmniLearn` [29], a framework that improves parallel efficiency in synchronous, and statistical efficiency in asynchronous training over heterogeneous clusters. Our key insight is that worker batch-sizes need *not* be identical across workers; instead mini-batches should be proportional to a workers' computational capability. This variable-batching approach assigns different batches to workers so as to equalize their respective computation times. To minimize noise from small-batch updates that may degrade model convergence in synchronous training, we propose weighted aggregation where higher value is assigned to worker updates computed over larger batches. For asynchronous training, we perform learning-rate scaling based on a worker's local batch-size. We define a new metric to measure the degree of heterogeneity in a cluster, referred to as its *Heterogeneity-level* (HL). To improve the optimal batch estimation even further, we develop a throughput-based proportional-control mechanism inspired by Proportional-Integral-Derivative (PID) controllers. This approach minimizes the effects of stragglers in synchronous and staleness in asynchronous training, thus improving parallel and statistical efficiency respectively. As heterogeneity in a cluster may change dynamically, `OmniLearn` adaptively adjusts batches by detecting changes across workers' computation times. For additional control stability, we enforce techniques like batch-size bounds and dead-banding to prevent noisy updates and accuracy degradation from large-batch training.

1.2.2 Balancing <Cost, Time> Trade-Offs of DL in the Cloud

Another challenging problem in distributed training is choosing the "right" resource configuration. This is critical when deploying jobs in the cloud where different cluster configurations may have varying time and cost for a given Distributed Deep Learning (DDL) job, thus presenting a large and complex trade-off space. To address this, we develop a system called `Scavenger` to determine the optimal cluster and batch-size configuration for training DNNs in the cloud.

Distributed training can be scaled either 'horizontally' by adding more workers and raising the cluster-size, or 'vertically' by raising the batch-size. However, increasing the cluster-size, batch-size, or both gives diminishing returns where dollar cost ($) of training can significantly rise for marginal reductions in training time. Thus, there is a pareto-relationship between cost and time,

which we capture in `Scavenger` by estimating and constructing the cost-time curves of DNNs through *parallel* and *statistical performance models*. We model parallel performance as a linear relationship of computation time with respect to batch-size, and model the synchronization time with respect to cluster-size. In a centralized topology, communication cost grows linearly with cluster-size, while decentralized topologies using ring or tree AR collectives have a lower bandwidth cost over larger clusters [30]. Our proposed parallel performance model can estimate iteration time with high accuracy irrespective of cluster topology. We then develop an analytical model for statistical efficiency to estimate the epochs required to train a model to convergence. Statistical efficiency is measured via 'gradient noise', a low-overhead scaling indicator that measures the divergence between approximated and true gradients. Using parallel and statistical performance models, we estimate the total training time and cost for any cluster and batch-size job configuration, along with different pricing policies in the cloud. We propose different policies that prioritize low-cost, low-time or both (i.e., knee-point) and returns the optimal job configuration for DL in the cloud.

1.2.3 Adaptive Gradient Compression in Distributed Learning

As the size of DNNs increases, so does the proportional size of updates to be communicated among workers in synchronous data-parallel training. Updates are exchanged at every iteration, resulting in high communication cost that inhibits linear scaling. Gradient compression is a lossy technique to ameliorate this problem in DDL. Here, the total volume of gradients to be exchanged among workers is reduced by a certain amount, as determined by its degree of compression or *compression factor* (CF). Although gradient compression reduces the communication volume and improves parallel scaling, it may degrade the statistical performance of a model depending on the chosen CF. The ideal CF that lowers the synchronization overhead while maintaining good accuracy varies for each DL model and training configuration, and thus may require careful tuning.

Here, we propose `GraVAC` [31], an adaptive compression algorithm that dynamically varies compression while balancing parallel efficiency (improved with high CFs) and statistical efficiency (improved with low CFs). We define a new metric called '*Compression gain*' that compares information loss in the gradients on account of compression, thus tracking statistical efficiency of compressed training. Low CFs *retain* most of the gradient signal and thus have a high gain value, while high CFs *lose* most of the information and hence, have lower compression gain. Combining

compression gain with system throughput, we measure '*Compression throughput*' to capture the pareto-relationship between the parallel aspect (i.e., system throughput) and statistical aspect (i.e., compression gain) of gradient compression. We also develop scaling policies for an adaptive compression regime, where we use low CFs in critical or sensitive regions, and gradually increase compression in later stages. Compared to static compression, `GraVAC` achieves up to 6.67× speedup while attaining similar test accuracy or loss.

1.2.4 Flexible Compression-Communication over Unpredictable Networks

Although intra-node workers enjoy fast synchronization over high-bandwidth interconnects, inter-node communication comparatively suffers from high synchronization cost due to limited network bandwidth. Additionally, the effective bandwidth available to a node may vary with time as well, exacerbating parallel scaling. Although gradient compression reduces the communication cost, it incurs an additional compression overhead and uses collectives like AG for communication, which may perform even worse than uncompressed training when using optimized collectives like ring or tree AR.

We propose an AR-friendly Top-*k* compressor called `AR-Top`k [32] that is faster than AG under certain conditions. It is implemented to either prioritize the most significant worker updates or minimize staleness among workers. We further analyze the cost of compression using the latency-bandwidth communication cost model, and develop heuristics to determine what compression and collective to use, given a cluster topology, DL model and network configuration. As network performance may vary over time, we model compression as a Multi-Objective Optimization (MOO) problem to find the optimal CF that best balances parallel and statistical efficiency in gradient compression. For a given compressor, parallel efficiency is improved by keeping the compression and communication cost low for a given CF, while statistical efficiency is improved by maximizing the compression gain metric (first proposed in `GraVAC` at §1.2.3). We evaluate `AR-Top`k by varying network latency and bandwidth throughout training of multiple DNNs and show how MOO-based technique dynamically changes CF and collective for optimal communication.

1.2.5 Real-time Learning over Streaming Data

Training DNNs over the edge presents a plethora of challenges like high network latency and limited bandwidth, insufficient storage and memory, as well as unbalanced and skewed data distribution across devices. This is further exacerbated when learning over streaming data in an online manner. Thus, edge systems suffer both systems and statistical heterogeneity; the former arises from variances in stream-rates, compute resources and available bandwidth across devices, while latter stems from training over skewed and non-IID data.

We develop a framework called `ScaFLES` [33] to accelerate training across devices with heterogeneous stream-rates under IID and non-IID settings. We mitigate streaming latency due to heterogeneous inflow-rates by adjusting batches proportional to each device's streaming-rate. Thus, workers with high-inflow volume train with larger batches while low stream-rates process smaller batches. To preserve statistical efficiency, we perform weighted aggregation, where each device's update is scaled by its streaming-rate. Additionally, we enforce a linear learning-rate scaling rule to improve generalization gap on account of large-batch training. As backpropagation [26] is compute intensive and edge devices have limited compute capability compared to datacenter-grade servers, data inflow cannot be processed at line-rate and the buffering queue can grow quickly. To avoid running out of storage or memory, we present multiple policies to manage buffering queues. Further, we introduce a stochastic *data-injection* policy to improve test accuracy over unbalanced and non-IID data while preserving privacy. Last, we optimize communication by choosing a high-frequency, low-volume synchronization approach where we exchange either compressed or uncompressed updates based on its significance. The importance of each update in this dynamic approach is measured by comparing prior and post-compression gradient updates. With the aforementioned optimizations, `ScaFLES` enables up to 3.29$\times$ faster training over heterogeneous data-streams with high convergence quality.

1.2.6 Federated Learning with Semi-Synchronous Communication

FL faces parallel scaling challenges like high communication overhead over constrained networks, and statistical performance degradation due to unbalanced and skewed data. Although algorithms like FedAvg and SSP lower the communication cost considerably, they tend to converge to lower

final accuracy compared to fully-synchronous training.

In this work, we present SelSync [34, 35], a federated algorithm to expedite training while converging to accuracy-levels at par with synchronous training. SelSync adaptively switches between local and synchronous updates based on their significance, referred to as "semi-synchronous" training. The key insight is that gradient sensitivity varies throughout training; gradients are large initially, but get smaller and eventually saturate as the model converges. We define 'relative gradient change' to track changes in the gradient trajectory and magnitudes, and show how it corroborates to a model's statistical performance (i.e., test loss or accuracy). We then apply a rule to synchronize updates among devices only if the relative gradient change exceeds a certain user-defined threshold. Different threshold values trade-off between parallel and statistical efficiency such that a low threshold favors synchronous updates, while a high threshold prioritizes parallel efficiency by applying local updates. SelSync also presents a new data partitioning scheme to improve convergence quality in semi-synchronous methods. To improve convergence over non-IID data, we extend data-injection from ScaFLES and compare training speedups as well as convergence quality with FedAvg, SSP and synchronous training across balanced and unbalanced versions of popular training datasets.

1.3 Thesis Structure

The contents of this thesis are outlined as follows: Chapter (2) provides the background and motivation of distributed DL and FL, and explains the premise of parallel and statistical efficiency in DDL systems.

Chapter (3) investigates how training suffers over heterogeneous clusters under synchronous and asynchronous communication models, and presents OmniLearn to mitigate stragglers or staleness with variable-batching and throughput-based proportional-control. Chapter (4) elucidates on how gradient noise works as a scaling indicator of statistical efficiency, followed by analytical performance models to predict the cost and time of a training configuration deployed in the cloud.

Next, we examine GraVAC in Chapter (5) to establish the notion of parallel and statistical efficiency with regard to gradient compression, and compare it with other static as well as dynamic compression techniques. Chapter (6) proposes MOO-based AR-Top$k$ compression and studies the

cost of different compression-communication strategies in terms of the latency-bandwidth model, followed by measuring end-to-end performance of different DNNs when network latency and bandwidth is varied at fixed intervals.

Chapter (7) addresses the challenges of distributed training over streaming data across heterogeneous devices, and explains how `ScaFLES` overcomes them to achieve fast and good convergence quality. Chapter (8) studies how the relative gradient change metric is a good approximation to second-order Hessian information and correlates to model convergence, followed by extensive empirical evaluation and comparison with techniques like FedAvg, SSP and synchronous training.

Finally, Chapter (9) summarizes the contributions of this thesis and presents it over a broader scope with current implications and future research directions.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, we present background details regarding the 'parallel' and 'statistical' aspects that are crucial for large-scale DL/FL systems; the former explains various parallelization techniques, impact of job placement and topology on parallel scaling, computational optimizations in DNNs, etc. while the latter discusses the effects of different hyperparameters that affect the statistical performance of DNNs and need to be considered when scaling applications over distributed systems.

2.1 Gradient Descent

The workhorse of DL, Gradient descent [26] is an iterative learning algorithm that minimizes a loss function by comparing output predictions with ground truth. For DL tasks, loss $\mathcal{L}(\cdot)$ is generally a non-convex, differentiable Lipschitz-smooth (L-smooth) function such that $||\nabla\mathcal{L}(x) - \nabla\mathcal{L}(y)|| \leq L||x - y|| \; \forall \; x, y$ where $L > 0$.

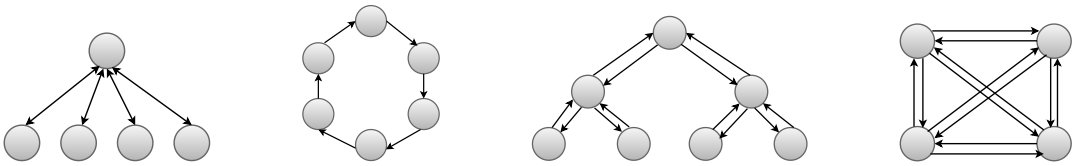
$$w_{i+1} = w_i \; - \; \eta \; \frac{\partial \mathcal{L}}{\partial w_i} \tag{2.1}$$

By Eqn. (2.1), loss function is minimized by updating model parameters 'w' at iteration 'i' in a direction opposite to that of the gradients of $\mathcal{L}(\cdot)$ (with respect to w), scaled by the learning-rate (or step-size) 'η'. The updated parameters are then used at iteration '$(i + 1)$'. DL is an iterative-convergent process where models converge after multiple traversals over the entire training data or *epochs*, and each epoch is comprised of many steps or iterations. Convergence quality is greatly influenced by training-specific variables or *hyperparameters*, such as duration of training (i.e., total epochs or iterations), learning-rate, choice of activation function (e.g., Sigmoid, ReLU, Tanh, etc. [36]), optimization (e.g., Nesterov's momentum, AdaGrad, Adam, RMS-Prop [37]), loss function (like MSE or mean squared error [38]), weight initialization (Xavier, He) [39], etc. The gradients, or partial derivatives of $\mathcal{L}(\cdot)$ with respect to 'w' are computed via backpropagation [7]. In DL, data is fed to the input layer while results are generated at the output layer. By chain rule, gradients are first computed on the outer layers and propagate back to the inner layers. In *Full gradient descent*,

updates are computed over the entire training data on each iteration. Despite its noise-free gradients [40], this approach has a high per-iteration computation cost. On the other hand, *Stochastic gradient descent* or SGD selects a random sample from the training set to compute gradients and update model parameters. However, SGD does not leverage multi-sample vectorization capabilities available across modern multi-core systems and GPU accelerators, leaving considerable performance gains on the table. Additionally, gradients computed over a single sample in SGD tend to be noisy compared to those calculated over entire training dataset [40]. For a convex problem running for 'I' iterations, SGD has a convergence rate of $\mathcal{O}(\frac{1}{I})$, while non-convex problems converge in $\mathcal{O}(\frac{1}{\sqrt{I}})$ [41, 42]. *Mini-batch gradient descent* is the middleground between the two where multiple samples are aggregated into a batch to compute model updates [43]. A collection of training samples to process updates at each iteration ensures the gradients have a high signal-to-noise ratio and can be efficiently vectorized on parallel processors.

2.2 Exploiting Parallelism in Deep Learning

Backpropagation in DL is inherently sequential as inner-layer gradients can only be computed once updates from the outer-layers are available (from chain rule). However, DL can be parallelized in a coarse-grained manner to reduce training time and improve parallel scaling when training data is colossal, or when DNNs are too large to be hosted on a single node. DL training can be distributed under *data-parallelism*, *model-parallelism* or *pipeline-parallelism*. In data-parallelism, multiple nodes collaboratively train a model where each device holds an independent copy of the current model state and individually processes its training samples [44, 45]. Updates may be aggregated in different ways, depending on the specific synchronization constraints in training (such as bulk-synchronous parallel [46], asynchronous-parallel [47], etc.). DNNs are trained with model-parallelism when the memory requirements to host and store a model (i.e., hold model parameters, gradients, intermediate activation maps, training batches, etc.) is larger than that available on a single node [45]. Here, a model is split across multiple nodes such that partial states residing on each are combined to form the global model state, and each worker runs computations only over its local model subset. It can be implemented by splitting a model vertically or horizontally, where the former partitions across different layers while the latter splits multiple layers into one



(a) (b) (c) (d)

Figure 2.1: Common cluster topologies in distributed computing: (a) Star (b) Decentralized Ring (c) Decentralized Tree (d) Decentralized Peer-to-Peer (P2P).

subset. Model-parallelism is communication-intensive as intermediate activation maps need to be exchanged across layers in forward pass, and error tensors in backward pass. Pipeline-parallelism further improves worker utilization by splitting a mini-batch into micro-batches and keeps the execution pipeline busy, thus improving cluster utilization and overall application throughput [48].

2.3 Cluster Topology in Distributed Deep Learning

For any parallelization strategy, the physical arrangement of nodes or their topology influences its communication cost incurred from model synchronization. Cluster topology determines the degree of distribution and latency between the nodes, thus influencing the overall application throughput. Broadly, a topological arrangement can either be 'centralized' or 'decentralized'.

Centralized Systems:

Bidirectional communication is enabled between a central node and workers across the cluster in a centralized topology. In synchronous data-parallel training, a centralized Parameter Server (PS) [49, 50] collects and aggregates gradients from all workers, then applies and distributes model updates back to the workers. Fig. (2.1a) illustrates a PS strategy in a star topology configuration. For better fault tolerance and bandwidth utilization, PS may even be sharded with multiple model copies across nodes. Another centralized training approach called ensemble learning [51] combines local models to produce an averaged model with better generalizability. Centralized synchronous training is shown to have a convergence-rate of $\mathcal{O}(\frac{1}{\sqrt{NI}})$ for 'N' workers running for 'I' iterations [52, 53], and a communication complexity $\mathcal{O}(N)$ since incoming and outgoing traffic flows between 'N' workers and PS.

Decentralized Systems:

Decentralized systems have no central node for aggregation. Instead, workers independently compute their updates, and may communicate with a subset of other nodes in the cluster. Communication may be optimized in certain decentralized systems, such as ring topology in Fig. (2.1b) where a worker communicates with only its left and right neighbors, while tree shown in Fig. (2.1c) allows direct communication only between a node's parents and children. Peer-to-peer (P2P) network shown in Fig. (2.1d) allows a node to directly access every other node. P2P networks are highly scalable as workers can directly communicate with each other (thus avoiding additional hops) and the network does not saturate due to heavy traffic to or from a single node (as commonly seen in centralized systems). Decentralized topologies have a communication complexity of $\mathcal{O}(\text{degree}(N))$. Gossip-based methods also fall under decentralized mechanisms where each worker communicates with only a subset of other workers in each iteration [54]. However, arriving at a consistent model state across all workers is challenging in such settings. Decentralized learning is facilitated by communication collectives such as allreduce, reduce, scatter, broadcast and allgather, implemented and optimized in libraries like OpenMPI [55], NVIDIA Collective Communications Library (NCCL) [56], Gloo [57], etc. Additionally, development of high-speed interconnects like PCIe Gen4 (with 16GT/s), PCIe Gen5 (with 32GT/s) and NVLinks have made intra-node multi-GPU DL even faster [58]. Open-source frameworks like Horovod [59] further reduce synchronization cost via optimized ring-based reduction. [60, 61] used different flavors of AR at inter and intra-node node levels to further reduce communication overhead.

Communication Overhead across Centralized and Decentralized Systems:

Latency is a measure of the communication delay in passing a message between endpoints (e.g. CPU and GPU) or over the network, while bandwidth measures the data-transfer rate of a connection. Akin to access costs in memory hierarchy where data movement to main memory is slower than that to registers or caches, latency and bandwidth cost get worse as we move from intra-node to inter-node scaling. In the context of distributed training, message-size implies the byte-size of gradients/parameters to be exchanged. For a given message-size in a cluster, different topologies have varying costs or overheads based on their respective synchronization mechanisms. These commu-

nication costs can be analyzed via α-β communication cost model [62, 30], where α is the latency in message transfer while $1/\beta$ corresponds to the available bandwidth. From the α-β model, centralized PS based on star-topology has a communication cost where its latency is fixed irrespective of cluster-size, while its bandwidth cost increases linearly with the cluster and message-size [30]. On the other hand, decentralized systems based on ring topology have a latency cost that rises with cluster-size, and bandwidth cost that is nearly independent of cluster-size and only increases with the message-size [63]. For tree-based AR, latency and bandwidth cost have logarithmic optimality with respect to the cluster-size. Communication cost is also influenced by the specific implementation of the collective used; AR over Compute Unified Device Architecture (CUDA) capable GPUs is faster in NCCL and Gloo, while Gloo is faster than MPI in CPU-based training [64]. Thus, overall throughput in a distributed cluster depends on the point-to-point or collective communication used, as well the specific implementation of the collective over a connection's latency and bandwidth.

2.4 Resource Heterogeneity in Batch and Stream Processing Systems

The aforementioned parallelization techniques and communication cost in different topological arrangements described in §2.2 and 2.3 assume homogeneous resources on every node. However, resource heterogeneity is prevalent across cloud, HPC and edge. Heterogeneity may be present intrinsically in a cluster of diverse servers, so a batch-processing job like DL training may be slow due to inherent performance characteristics of the individual servers. However, it may also arise due to workload characteristics, for example, in real-time stream processing systems where ingress data volume may vary for each worker. In distributed systems, resource heterogeneity can be classified either based on the computational or networking aspect of an application.

Compute Heterogeneity:

The compute nodes available in the cloud and data-centers may not always be homogeneous. In fact, there is a high variance in performance and resource configurations available across these machines, either in terms of CPU cores, memory, type and number of available GPUs or TPUs [15], interconnect speeds, etc [65]. In synchronous DL, heterogeneity can give rise to stragglers that cause slowdowns from slower workers being the bottleneck [66], or cause staleness issues in asynchronous

training [67]. Even if compute resources are identical among workers, shared clusters with varying job distributions across nodes can change the effective resources available per-job, per-worker (i.e., due to interference). To alleviate compute heterogeneity, techniques like Anytime Mini-batch [68] impose constraints like a fixed time budget so heterogeneous workers may process variable-batches in that duration. Although staleness considerably deteriorates test performance, [67] proposes an asynchronous SGD variant where learning-rate is adjusted with respect to gradient staleness. In streaming real-time systems, training over streaming data can also introduce heterogeneity due to skewed volume of ingress data across devices, resulting in more compute on high-volume nodes compared to low-volume devices [33].

Network Heterogeneity:

Variable and unpredictable networks that are prevalent at the edge and cloud constitute another source of heterogeneity in distributed computing. In the cloud, different VM instances come with different classes of networking tiers [65] in terms of ingress-egress bandwidth [69, 70]. FL on mobile and edge devices tend to have a slower network configuration than high-end servers in datacenters. Network heterogeneity can also arise from the cluster topology (e.g., star, mesh, ring, etc. discussed in §2.3), type of communication link (like ethernet or infiniband) or physical medium (fiber cable, copper, etc.). Latency and bandwidth can fluctuate even on the same connection over time due to various reasons. Because of congestion, effective bandwidth may be reduced under high network traffic, thus increasing the data-transfer time on saturated links. With Quality-of-Service (QoS) prioritization, concurrent jobs on shared clusters can have different QoS levels such that high-priority jobs are allocated a larger chunk of the total bandwidth. As a consequence, low-priority jobs may suffer communication slowdown due to lesser bandwidth available to them. Even without explicit prioritization, concurrent jobs in shared clusters split finite networking resources, resulting in lower bandwidth that is available to each job. Network switches and routers may also use different scheduling algorithms to allocate bandwidth for different traffic flows that determines how resources are shared among applications.

The scale of network latency can vary as well, whether we train on the edge or high-networking datacenters. Local area networks (LANs) have very low latency that ranges from just microseconds to a few milliseconds, inter-rack latency is on the scale of tens of milliseconds, while inter-node

latency is even worse [71]. Heavy network load and congestion also leads to latency surges under high traffic flows due to queuing delays as multiple jobs try to perform data transfers simultaneously. Network routing that decides the path of a packet also determines its latency. Although routing protocols determine the optimal transmission path, different paths can result in varying latencies due to network congestion and physical distance. These factors affect latency and bandwidth, and consequently the parallel scaling of communication-intensive tasks like DDL and FL.

2.5 Communication Mechanisms in Data-Parallel Training

We now describe various distributed data-parallel techniques along with their respective communication patterns.

Bulk-Synchronous Parallel (BSP) Training:

$$w_{i+1} = w_i - \eta \frac{1}{N} \sum_{n=1}^{N} \frac{\partial}{\partial w_i} (\frac{1}{|b|} \sum_{x_{(n,i)} \in \mathcal{B}_n} \mathcal{L}(x_{(n,i)}, w_t)) \tag{2.2a}$$

$$t_{step} = t_{compute} + t_{sync} + t_{IO} \tag{2.2b}$$

In BSP, or simply synchronous training [46], model updates are aggregated among workers at the end of every iteration. This is done by averaging updates on a central PS [49] or via AR in decentralized systems [59]. BSP scales weakly to effectively increase the amount of work performed per-iteration. Assuming each worker 'n' processes a mini-batch 'b' from distribution $\mathcal{B}_n$, we process 'Nb' samples per-iteration with 'N' workers, as shown in Eqn. (2.2a). Each training step is identical from a computational perspective; Eqn. (2.2b) breaks down an iteration into the time it takes to evaluate loss and compute gradients (i.e., $t_{compute}$), communication or synchronization cost to aggregate updates (i.e., t_{sync}), along with data movement and I/O overhead (i.e., t_{IO}). Each step is blocking due to the synchronization barrier in BSP, i.e., all workers halt training until updates are complete across all workers. The convergence-rate of BSP for non-convex optimization is $\mathcal{O}(\frac{1}{\sqrt{NI}})$ over 'N' workers running for 'I' steps [41].

Asynchronous-Parallel (ASP) Training:

$$w_{i+1} = w_i - \eta \frac{1}{|b|} \sum_{x_{(n,i)} \in \mathcal{B}_n} \frac{\partial}{\partial w_i} \mathcal{L}(x_{(n,i)}, w_{n,(i-\tau_{(n,i)})}) \ \ \forall \ \ n \in [\, 1, 2, 3, ...N \,] \tag{2.3a}$$

$$t^{(n)}_{step} = t^{(n)}_{compute} + t^{(n)}_{pull\text{-}PS} + t^{(n)}_{IO} \tag{2.3b}$$

In ASP, workers train local model replicas independently in a non-blocking manner [47, 72]. Hogwild! implemented ASP in a lock-free manner on shared memory systems [73]. Without an explicit synchronization barrier, workers can push/pull updates to/from a central PS asynchronously. The iteration breakdown in Eqn. (2.3b) shows that worker 'n' pulls model parameters from the PS in $t^{(n)}_{pull\text{-}PS}$ time and incurs compute and I/O overheads $t^{(n)}_{compute}$ and $t^{(n)}_{IO}$ respectively

Across 'N' workers in Eqn. (2.3a), each worker 'n' computes its gradients over a mini-batch sample $x_{(n,i)}$ with stale model parameters $w_{n,(i-\tau_{(n,i)})}$ such that parameters on 'n' at iteration 'i' are delayed by $\tau_{(n,i)}$. If τ is large, updates are inexact since parameters used to calculate them are old and stale, thus degrading model accuracy. Staleness (τ) is further exacerbated in heterogeneous clusters due to computationally slower nodes. When all workers are homogeneous and staleness is not an issue, centralized ASP systems converge at $\mathcal{O}(\frac{1}{\sqrt{NI}})$. Decentralized ASP [74, 75] entails symmetric peer-to-peer communication so that each active worker communicates its updates to a passive worker and vice versa, converging in $\mathcal{O}(\frac{1}{\sqrt{I}})$. Although communication cost of ASP is lower than BSP, the former comes at a cost of inconsistent model state across workers. Comparatively, asynchronous training performs lesser work per-iteration by applying updates computed over mini-batch 'b', while BSP processes 'Nb' in a single step. If staleness is excessively high, ASP may have a lower convergence quality than BSP, or require many more iterations to converge (thus increasing the total training time).

Stale-Synchronous Parallel (SSP) Training:

$$w_{n,i+1} = w_{0,0} - \eta \sum_{j=1}^{i-s} \sum_{k=1}^{N} \frac{\partial}{\partial w_{k,j}} (\frac{1}{|b|} \sum_{x_{(k,j)} \in \mathcal{B}_n} \mathcal{L}(x_{(k,j)}, w_{k,j}))$$
$$-\eta \sum_{j=i-s}^{i} \frac{\partial}{\partial w_{n,j}} (\frac{1}{|b|} \sum_{x_{(n,j)} \in \mathcal{B}_n} \mathcal{L}(x_{(n,j)}, w_{n,j})) - \eta \sum_{(k,j) \in \mathcal{S}_{n,i+1}} \frac{\partial}{\partial w_{k,j}} (\frac{1}{|b|} \sum_{x_{(k,j)} \in \mathcal{B}_n} \mathcal{L}(x_{(k,j)}, w_{k,j})) \tag{2.4}$$

SSP training runs workers asynchronously, albeit in a bounded manner [76]. Here, fast workers make more progress and update model weights at the PS, but halt training after exceeding the slowest worker by a certain user-specified 'staleness-threshold'. This metric bounds the iteration difference between the fast and slow workers, and consequently decides the maximum staleness in updates that are sent to the PS.

The gradient descent update for SSP is shown in Eqn. (2.4) for bounded staleness 's' on worker 'n'. When workers are $\leq s$ steps apart, each sees model updates in the range $[1, i-s]$ at iteration 'i'. Additionally, there are adaptive updates between iterations $[i-s, i]$, shown in the second summation term. The last term corresponds to the iteration phase $[i-s]$ where $\mathcal{S}_{n,i+1}$ is a subset of updates over other workers. SSP has a convergence-rate of $\mathcal{O}(\sqrt{\frac{2(s+1)N}{I}})$ [76], so its equivalent to BSP if $s = 0$, or local execution as $s \to \infty$ [77]. Other variants of SSP like Dynamic SSP [78] calculates staleness 's' dynamically at runtime based on per-worker batch processing time to reduce waiting times. Free-SSP [79] optimizes under resource contention from concurrent jobs in a cluster by adding a penalty term on slow workers that halt training and are ultimately discarded from the cluster.

2.6 Federated Learning

Collaborative or *Federated learning* (FL) performs on-device training when client devices are resource-constrained [80], communication bandwidth is limited [81] and data privacy and locality is critical [25, 82]. This training paradigm stems from the boom in big data and IoT, as personal edge or mobile devices collect and generate new data to train foundation models or device-local context-aware DNNs. For e.g., traffic camera feeds, smart sensors in private networks, fog servers, smartphones, etc. Due to privacy constraints, data cannot be migrated to a centralized location or

collected to be partitioned in an IID manner as it may pose a security risk depending on its sensitivity (like health or financial data). Consequently, non-IID and skewed data distributions affect the convergence quality of the final model [25, 83]. FL systems thus face challenges pertaining to both parallel and statistical efficiency, often benchmarked with frameworks like LEAF [84] and FedML [85].

$$w_{n,i+1} = \begin{cases} w_{n,i} - \eta \cdot \frac{\partial}{\partial w_i}\left(\frac{1}{|b_n|}\sum_{x_{(n,i)}\in\mathcal{B}_n} \mathcal{L}(x_{(n,i)}, w_{n,i})\right) & \text{if } i \,\%\, T_s > 0 \\ w_{n,i} - \eta \cdot \frac{1}{N}\sum_{n=1}^{\mathcal{S}_i\in N} \frac{|b_n|}{|B|}\frac{\partial}{\partial w_i}\left(\frac{1}{|b_n|}\sum_{x_{(n,i)}\in\mathcal{B}_n} \mathcal{L}(x_{(n,i)}, w_i)\right) & \text{if } i \,\%\, T_s = 0 \end{cases} \tag{2.5}$$

Federated Averaging or FedAvg [25] learns a model across decentralized clients with local data by periodically aggregating updates to produce a globally shared model. Such a periodic synchronization approach is akin to Local-SGD [86, 87] where updates are periodically averaged among workers. However, FedAvg additionally accounts for data imbalance and compute heterogeneity between clients by performing weighted averaging during synchronization phase. Updates are communicated after every T_s steps in Eqn. (2.5), scaled as $|b_n|/|B|$ from worker 'n', where the numerator is the local mini-batch and denominator is the global batch-size. FedAvg with $T_s = 1$ is equivalent to synchronous training, while a high synchronization interval $T_s \to \infty$ is closer to One-shot SGD [88], a form of ensemble learning where local models are aggregated at the end of the training phase.

$$\min_{w_{(n,i)}} \frac{1}{|b|}\sum_{x_{(n,i)}\in\mathcal{B}_n} \mathbb{E}[\ \mathcal{L}(x_{(n,i)}, w_{n,i})\ ] + \frac{\rho}{2}||w_{n,i} - w_{*,i}||^2 \tag{2.6a}$$

$$w_{n,i} = w_{n,i-1} - \eta\ \{\frac{1}{|b|}\frac{\partial}{\partial w_{n,i-1}}\mathcal{L}(x_{(n,i-1)}, w_{n,i-1}) + \rho \cdot (w_{n,i-1} - w_{*,i-1})\ \} \tag{2.6b}$$

$$w_{*,i+1} = \frac{1}{N}\sum_{n=1}^{N} w_{n,i} \quad \text{if} \quad i \,\%\, T_s = 0 \tag{2.6c}$$

A modification to FedAvg called FedProx [89], inspired by Elastic-Averaging or EA-SGD [90, 91] adds a penalty or proximal term to the loss function (Eqn. (2.6a)) and local SGD update (Eqn.

(2.6b)). Adding the ρ-term bounds the impact of variable local updates and minimizes the divergence between local and global model states. Akin to FedAvg, model parameters are periodically aggregated during synchronization phase, determined by T_s in Eqn. (2.6c). Other works like [83] explain accuracy degradation in non-IID FL by weight divergence or earth mover's distance (EMD) between the distribution of classes over a single worker compared to the overall data distribution. This divergence is measured as $\left|\frac{w_{FedAvg}-w_{SGD}}{w_{SGD}}\right|$. EMD is small over IID data, and larger over non-IID data. Local models are aggregated in the synchronization phase, weighted by the number of samples processed by each worker. Final accuracy can further be improved by explicitly creating a small subset of training data shared across all clients, thus improving overall data distribution.

2.7 Communication Scheduling

Although distributed data-parallel training improves parallel scaling, the mechanism of forward and backward pass in DL is inherently sequential with a bidirectional execution flow over a DNNs' computation graph. For e.g., in the forward pass of a fully-connected network, computation flows from the input to output layer in forward phase, and from output to input layer in the backward pass. A DNNs' execution graph (i.e., within a worker) can thus be fused with the communication step to aggregate updates (i.e., across the workers) and improve the parallel scaling in DDL.

Wait-free backpropagation (WFBP) [92] hides communication cost by overlapping with computation based on the structure and density of a DNNs computation graph. For e.g., most of the computation cost is incurred in the initial and middle convolutional layers [2]. The outer fully-connected layers in models like VGG [8] and AlexNet [1] account for more than 90% of the total parameters, while the remaining 10% convolutional layers are responsible for about 90% of the total computation cost. WFBP exploits this property by skipping completion of the whole backpropagation routine and transmitting gradients from the large, fully-connected layers while still computing gradients on the inner convolutional layers. This approach of interleaving computation and communication partially hides the synchronization cost and improves parallel scaling. This can further be optimized by combining layers with fewer parameters together and transmitting them as one, i.e., 'tensor fusion' [93, 59]. Merging multiple short communication tasks into one reduces the overall transmission cost, so smaller convolutional and recurrent layers [94] are batched together instead

of communicating them on a per-layer basis. ByteScheduler [95] also partitions and reorders its tensors into slices or buckets, using an auto-tuning scheduler based on Bayesian optimization. [96] combines domain knowledge of DL execution engines (like Tensorflow [97] or PyTorch [98]) and tensor-sizes for data transmission, fusing tensors based on the directed acyclic graph (DAG) generated by the underlying execution engine. Scheduling communication intelligently hides overall synchronization overhead or avoids otherwise redundant, expensive communication calls.

2.8 Compression in Deep Learning

Compression is crucial to improve the overall parallel scaling and resource utilization of DL applications. At intra-worker level, techniques like pruning reduce memory footprint either by lowering the model-size [99, 100] or the size of training data [101], thus improving training and inference performance. At inter-worker level, compressing gradients improves parallel scaling by lowering the exchange volume in communication phase [102, 103].

Pruning:

Pruning techniques [99, 104] identify and remove redundant or insignificant model parameters either during the training or inference stage. Dropout [105], where some weights are randomly dropped during training effectively acts as a pruning mechanism to prevent overfitting. Although the actual size of DNNs is not reduced, some connections are skipped which lowers the amount of computations performed in an iteration. Methods like Deep Compression [106] combine pruning with quantization and Huffman coding [107] to reduce model-size.

Gradient Compression:

Gradient compression reduces the volume of gradients to communicate in distributed training. The factor or degree to which compression is performed is measure by its *Compression factor (CF)*. For e.g., a compressor that reduces the tensor volume to 10% of the original tensor has a CF 10×, 1% volume gives 100× CF and 1000× for 0.1% reduction. The objective of gradient compression is to reduce training time or its iteration cost, as shown in Eqn. (2.7). Here, $t_{compute}$ is the gradient computation time, t_{IO} is the data-movement and I/O overhead, $t^{(c)}_{comp-decomp}$ is the over-

head to compress and decompress the gradients while $t_{sync}^{(c)}$ is the communication cost of gradients compressed to CF 'c'.

$$t_{step}^{(c)} = t_{compute} + t_{sync}^{(c)} + t_{IO} + t_{comp-decomp}^{(c)} \tag{2.7}$$

Since lossy compressors apply compressed or partial updates, this results in a loss of statistical performance (i.e., lower final accuracy or loss). The generalization loss may increase with larger CF 'c' as more updates are approximated in the update step. To compensate for this loss, compression techniques employ *error-feedback* [108, 109] that performs residual gradient accumulation, as shown in Eqn. (2.8).

$$g_{ef}^{(i)} = g_{o}^{(i)} \; + \; \texttt{residual}^{(i-1)} \;\; \text{s.t.} \;\; g_{c}^{(i)} = \mathcal{C}(g_{ef}^{(i)}) \;\; \& \;\; \texttt{residual}^{(i)} = g_{ef}^{(i)} - g_{c}^{(i)} \tag{2.8}$$

In error-feedback, gradients that are ineligible for communication at step $(i-1)$ are not discarded, but appended to the backpropagated gradients $g_o^{(i)}$ at iteration 'i'. Compression mechanism $\mathcal{C}(\cdot)$ converts the error-fed gradients $g_{ef}^{(i)}$ to compressed form $g_c^{(i)}$ and updates the residuals. Residual gradients can thus be viewed as increasing batch-sizes that grow corresponding to the update interval once a gradient becomes eligible for communication.

Gradient compression techniques can broadly be classified as *sparsification*, *quantization*, *low-rank approximations*, or any combination of the above. In sparsification, we select a subset of gradient values from the original tensors and set the remaining to 0 [110]. Top-*k* sparsification selects the largest $k\%$ gradients by magnitude and sets the remainder to 0 [111, 112, 113]. Gaussian-*k* [114] approximates the gradients as a gaussian distribution and estimates threshold corresponding to a target CF via quantile or percent point function (ppf). As the gradients may not strictly be gaussian, the estimated threshold can be skewed. DGC [115] sparsifies momentum by accumulating velocity instead of gradients, and avoids exploding gradients [116] from error-feedback by applying gradient clipping. Random-*k* [111] randomly chooses $k\%$ of the gradients, while sparsity inducing distribution-based compression (SIDCo) [117] models gradients as a sparse distribution like double exponential or double gamma, and estimates the threshold for a target CF accordingly. To mitigate

estimation bias, SIDCo performs multi-stage threshold estimation such that to compress to CF 'c_1', gradients are first compressed to 'c_2' such that $c_2 < c_1$, which is then further compressed to (c_1/c_2) to get the final compression factor.

Quantization either lowers the bit-size of single precision gradients or via codebooks so each gradient is associated with a predefined state, thus bounding the range of values held by the gradients [118]. 1-bit SGD [119] achieves $32\times$ compression by quantizing 32-bit gradients to 1-bit. Quantized or QSGD [118] performs stochastic quantization via randomized rounding while keeping statistical properties of the original gradients intact. Automatic mixed precision (AMP) reduces single precision floats to half-precision gradients (i.e., 16-bit) and lowers the communication volume by half [120]. Terngrad [121] quantizes gradients over three levels: $[-1, 0, +1]$ in a layerwise manner across clipped residual gradients, while signSGD [122] only uses the sign of gradient for SGD update. SIGNUM [122, 123] applies momentum on top of signSGD, while EF-signSGD [108] adds error-feedback and scaled gradient magnitude to signSGD to improve convergence quality. EF-blockSGD [109] further improves EF-signSGD and fixes the gradient vanishing problem due to $\mathbb{L}1$-gradient norm scaling by quantizing and bucketing gradients into blocks and scaling the magnitude of each block separately.

Low-rank approximation techniques presume that gradients are over-parameterized and tend to have a low-rank structure. These methods decompose gradient $G \in \mathbb{R}^{(p\times q)}$ into lower rank matrices (U, V) such that $U \in \mathbb{R}^{(p\times r)}$ and $V \in \mathbb{R}^{(r\times q)}$ [124]. By exchanging (U, V) instead of G, we achieve a CF of about $\left(\frac{pq}{pr+rq}\right)$. Compressors like Atomo [125] decompose gradient tensors based on a sparsity budget, while GradZip [126] compresses gradients to lower rank matrices (U, V) that minimizes the deviation between the original and reconstructed gradients with an added regularization term.

Additionally, hybrid compression techniques combine different mechanisms together to reduce communication intensity with good convergence quality. For e.g., RedSync [127] combines sparsification with quantization, while Sparse Ternary Compression (STC) [128] combines upstream Top-*k* compression, downstream compression, ternarization and optimal Golomb encoding [129] of weight updates. On the other hand, adaptive compression techniques like AdaComp [130] keeps track of local gradients and tunes its CF based on the significance of collected updates, while Accordion [131] adjusts its compression between low and high CFs based on critical regime identification

[27].

2.9 Statistical Efficiency in Deep Learning

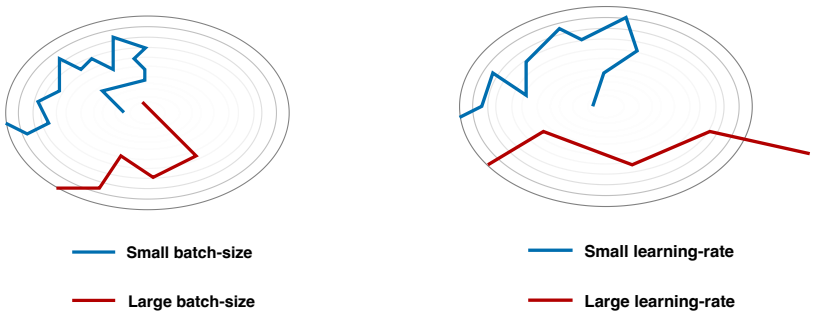
Training DNNs entails learning some some underlying optimization landscape corresponding to the loss function over a given dataset. The hyperparameters set prior training or adjusted over the course impact statistical efficiency, which in turn determines the final convergence quality (such as generalization loss, perplexity, accuracy, BLEU score, etc.). Parallelizing DNNs across multiple nodes affects its statistical efficiency and needs to be considered for optimal convergence and generalization.

2.9.1 Hyperparameter Tuning

Choosing the right hyperparameters is critical for convergence, whether for highest accuracy or lowest training time. As mentioned in §2.1, the statistical performance in DL is influenced by the duration of training (epochs or steps/iterations), learning-rate schedule, weight decay, dropout, $\mathbb{L}1$ or $\mathbb{L}2$ regularization [132], activation function to introduce non-linearity (such as sigmoid, ReLU, Tanh) [39], mini-batch size, optimization (e.g., vanilla SGD, Nesterov's momentum, adaptive moment, adaptive gradient, RMS-Prop, etc.) [37], loss function (mean squared error, mean absolute error, cross-entropy loss, cosine similarity, Kullback-Leibler divergence loss, etc.) [38, 133] or weight initialization (such as Xavier or He) [39]. We describe some of them as follows:

***Length of Training*:** In well-conditioned optimization problems, model parameters adjust towards the direction of low-curvature where loss decreases in a predictable way. Repeating this over the iterations or epochs attains better model convergence. However, in poor and suboptimal conditions, DNNs may not necessarily converge in an obvious or expected manner as the model may saturate to a saddle point. Hence, training for longer duration may be ineffective in such scenarios.

***Model Optimization*:** Apart from vanilla SGD update in Eqn. (2.1), other optimization techniques may update model parameters faster or towards a flatter minima over the non-convex loss surface. For e.g., gradient descent with Nesterov's momentum [134] adds a momentum term to the gradients, effectively applying EWMA smoothing and reducing noise. Adagrad or Adaptive gradient descent [135] dynamically adjusts the learning-rate individually for every model parameter by



(a) (b)

Figure 2.2: Optimization trajectory with (a) Small and large batch-sizes over the loss landscape. Large batches tend to have less noise and attain optimal training loss in bigger steps and fewer iterations. (b) Low and high learning-rates. A large step-size may overshoot and diverge from the minima.

considering the summed outer product of gradients from previous iterations. RMS-Prop [43, 37] extends Adagrad by adjusting the step-size of each parameter using EWMA instead of a cumulative sum. Adaptive moment (Adam) and Adamax [136] combines multiple optimization schemes by keeping EWMA of prior gradients (i.e., momentum) as well as the decaying average of gradient variance for previous iterations to account for bias-correction from diminishing or sparse gradients.

***Mini-batch size*:** As mentioned in §2.1, training is accelerated as we scale from full to mini-batch gradient descent. However, the quality of gradients estimated with a full-batch is different from those computed over smaller batches. With full-batch training, there is an implicit true convergence landscape corresponding to the loss calculated over the entire dataset. Gradients are only approximated to this true loss with mini-batch gradient descent, and smaller batches suffer from high variance and noisy gradients [40]. Although accumulating updates over the iterations increases the effective batch-size and averages out gradient variance which directs the model towards the minima [60, 137], the individual updates by themselves are not equally effective or important. On the other hand, gradients computed over a large batch-size are highly representative of the true gradients from full-batch. This is the main statistical benefit of weakly-scaled synchronous data-parallel training where batches are parallelized to effectively produce larger batches. As shown in Fig. (2.2a), larger batches achieve low 'training loss' (note: not test loss) by taking larger steps over the optimization landscape [138, 139]. Comparatively, smaller batches are noisy and need to take many more steps so as to arrive at the minima. Despite the parallel efficiency and low training loss on larger batches, we observe poor generalization or performance on test loss/accuracy [140, 141, 142]. This is because

large-batches either overfit, or saturate to saddle points, are not as exploratory as small-batches, or simply don't train for longer (i.e., within a fixed epoch budget) [143].

***Learning-rate schedule*:** Vanilla SGD works as per Eqn. (2.1), where gradients are scaled by step-size η. Learning-rate is another crucial training hyperparameter that determines model convergence or divergence. From Fig. (2.2b), a small learning-rate takes smaller steps and gradually moves in the direction of the minima [40]. With a large step-size, training DNNs has a tendency to overshoot while taking bigger steps and may diverge the model. This is why typical training routines either use adaptive learning-rates [144, 145, 146] or a dynamic learning-rate schedule where the step-size is periodically decayed by some factor [147]. Adaptive learning-rate techniques adjust model parameters at different granularities (either entire model, layerwise or individual parameters) [43, 144, 148]. The intuition behind dynamic step-size routines is that as a converging model fine-tunes itself and gradients simultaneously begin to saturate, the factor by which SGD update is applied needs to be adjusted as well. Techniques like learning-rate warmup [61] have also been mapped to gradient compression [115, 131] for successful convergence, where a small CF is used in the beginning and uniformly increased over the course of training. In warmup, step-size 'η' for 'I' iterations raises per-step learning-rate as $(\frac{\eta}{I})$ until 'I' steps, and set to η afterwards. The small values used (either in step-size or CF) initially account for the early sensitive phase of training [149, 27].

2.9.2 Statistical Efficiency in Distributed Deep Learning

The parallel and statistical gain of synchronous data-parallel training comes from distributing the workload across multiple workers via weak-scaling [150], i.e., increasing the global batch-size proportionally to the total number of workers. By doing so, we effectively increase the amount of work performed per-iteration by processing more training samples in a step. If a single node processes 'b' samples in a step, BSP training across 'N' workers implies training on 'Nb' samples in one iteration. From a parallel scaling perspective, training is limited by systems constraints like heterogeneity and frequent, high-volume communication. From a statistical standpoint, distributed DL is affected by factors like skewed data distributions, large batch-sizes, noisy gradients, poor conditioning, greedy optimization [40], learning-rate routines, etc. which may affect model generalization.

***Learning-rate and Large-batch Training*:** Tuning the learning-rate for large-batches helps

DNNs achieve high convergence targets as there is a strong correlation between the two. [151] uses a strict learning-rate schedule and weight decay to improve convergence in large-batch training as part of its collapsed ensemble technique. [152] presents large-batch training on convolutional networks by tuning step-size with a factor of $\sqrt{N}$ when scaling to 'N' workers in order to keep the expected gradient variance fixed. Later, [61] proposed a linear scaling rule to increase learning-rate by 'N' when the global batch-size increased proportionally. [153] successfully trained ResNet50 [9] on ImageNet [154] with a batch-size of 32000 by using RMS-prop with warmup, batch-normalization [155] and a slow start learning-rate schedule. [156] trades between adjusting the learning-rate for using larger batches. Additionally, it reduces the total iterations by increasing the step-size and scales batch-size proportionally, or increase batches inversely to the momentum coefficient (when using SGD with momentum). Layerwise adaptive rate scaling, or LARS [141, 157] is an adaptive learning-rate technique for large-batch training where each layer's step-size is determined by the $\mathbb{L}2$-norm of its weights and gradients, which mitigates the vanishing and exploding gradients issue. [60] combines large-batch training with mixed-precision, LARS, batch normalization and optimized collective communication. AdaBatch [158] adaptively increases the batch-size analogous to learning-rate schedule, while [159] identifies the optimal batch-size for a fixed learning-rate that maximizes test performance, which is proportional to the learning-rate and size of training data.

***Training Sensitivity in DNNs*:** The adaptive learning-rate and warmup strategies described in the last section attenuate the optimization challenges of the first few epochs. This is done because a randomly initialized model aggressively adjusts its parameters towards the loss minima in the early training phases [149, 160, 161]. The gradients computed in this period tend to be larger and change drastically. Under good conditioning and hyperparameters, DNNs typically converge after many iterations. As it approaches a saddle point and the model saturates, gradients in the later stages become smaller and stop changing in a significant manner. Although the length of early phases varies for each model, dataset and training hyperparameters, it can be detected by changes in the eigen-values of the Hessian [142, 149, 162]. However, computing second-order gradient information is computationally expensive and slow. But recent works have shown that it can be well approximated with first-order gradient information which is much cheaper to compute and can easily be integrated as part of the backpropagation routine [131, 40].

Aside from early training phase, there are additional critical or sensitive regions in training

influenced by DL hyperparameters. Previous works [27, 28] have observed that information gain in model parameters does not increase monotonically during training, but is comprised of a rapid increase in information phase, followed by information reduction phase. Critical periods are often centered around the former, and any major hyperparameter changes in this phase (such as learning-rate or CF in compression) affect convergence quality that may not necessarily be compensated simply by training for longer. Similar to early phases, such critical regions have prior been detected via Hessian conditioning, Fisher information matrix (FIM) [27, 163] or gradient variance. These metrics serve as a measure of the curvature of loss landscape for a given model and its convergence quality.

***Adaptive Deep Learning Systems*:** The rate of feature-learning in DNNs is not monotonically increasing, thus the techniques used for training them should be adaptive as well. Adaptive methods [135] for gradient optimization or tuning hyperparameters (from §2.9.1) account for the statistical efficiency in DL, but fail to account for scaling efficiency when parallelizing training across many nodes. Considering both parallel and statistical efficiency is vital to develop distributed learning systems that train in minimal time while achieving high convergence-rates. However, this presents various challenges due to poorly understood first-principles of distributed SGD over non-convex optimization landscapes and complex trade-offs between parallel and statistical efficiency. Schedulers like Optimus [164] performs resource allocation and job scheduling by making overly optimistic assumptions like a linear-rate of model learning throughout training and develops an empirical performance model based on that. Cynthia [165] also develops an analytical performance model for cloud resource provisioning specifically for DL jobs. HyperSched [166] performs grid search over the hyperparameter space and allocates additional resources to the best-performing job configuration. This way, HyperSched accounts for statistical efficiency by prioritizing model configuration with lowest loss or highest accuracy.

Recent studies have explored using first and second-order gradient information to develop adaptive learning systems. [40] proposes large-batch training and describes an adaptive scheme to adjust batch-size based on the amount of noise in the gradients, i.e., variance between local and aggregated gradients in synchronous data-parallel training. AdaScale [167] performs learning-rate adjustment for large-batch training such that step-size is scaled proportionally to gradient noise. KungFu [168] designs adaptation policies to adjust parameters like cluster-size and global batch-size using gradi-

ent signal-to-noise ratio. Pollux [169] performs job scheduling for DL applications by combining parallel and statistical efficiency into a single metric. [159] scales training by computing gradient noise-scale as a function of learning-rate, batch-size and size of training dataset. [170, 171] use second-order information to adapt model hyperparameters for large-scale distributed training.

CHAPTER 3

OPTIMIZING DISTRIBUTED TRAINING OVER HETEROGENEOUS SYSTEMS

3.1 Introduction

Systems for distributed training typically assume homogeneous workers, i.e., nodes with similar computational and networking capabilities. However, heterogeneity is prevalent in computing infrastructure, be it public cloud or private datacenters. Synchronous training (i.e., BSP) on heterogeneous servers can deteriorate parallel scaling as training may suffer straggler slowdown on account of slow or low-resource workers. On the other hand, asynchronous training (i.e., ASP) over heterogeneous clusters can degrade statistical efficiency due to staleness issues arising from late updates. In this chapter, we present `OmniLearn` [66, 29], a framework to mitigate the effects of heterogeneity across asynchronous and synchronous communication patterns. This work is inspired by proportional controllers [172] to balance computations across heterogeneous resources, and works under fluctuating resource availability as well by dynamically adjusting worker batches in an online manner.

3.2 Motivation

DL is an iterative-convergent process; with optimal conditioning and hyperparameters, a model converges after certain epochs or iterations. By Eqn. (2.2), BSP performs weak-scaling to produce more accurate gradients by processing more samples per-iteration [40], with high communication cost from frequent synchronization. On the other hand, ASP in Eqn. (2.3) prioritizes parallel scaling over statistical performance with workers training asynchronously over smaller batches (and thus producing noisier gradients).

However, resource heterogeneity is ubiquitous across data-centers, cloud and edge. Distributed applications are often deployed on servers with different size and resource allocations. It is crucial to train DNNs efficiently on heterogeneous clusters, either for FL on the edge or training on low-end servers in datacenters [173]. Additionally, modern applications must contend with dynamic heterogeneity due to fluctuating resource availability from concurrent jobs running on shared clus-

ters. Burstable cloud VMs [174, 175] effectively act as heterogeneous resources as well since they can burst at different times. Hence, distributed training must be omnivorous, opportunistic, able to leverage different types of compute resources and adapt to dynamic resource allocations over time.

Based on the communication pattern, data-parallel training suffers when resource allocation across the workers is not uniform. BSP training suffers from parallel inefficiency, while ASP suffers from statistical inefficiency. BSP has a synchronization barrier at the end of every iteration where model updates are aggregated across all workers. In heterogeneous settings, parallel scaling is degraded due to straggler slowdown; workers with lower or fewer resources take longer to process a mini-batch, thus causing high-end workers to sit idle and wait until all workers have calculated their local updates. This increases the average iteration time across the entire cluster and consequently, the end-to-end training time. The high iteration cost thus degrades parallel scaling in synchronous training.

Although stragglers do not affect parallel scaling of ASP, the convergence quality of a model suffers from stale updates [74] under resource heterogeneity. This occurs when progress made by faster workers to update the global model at a centralized server is overridden by slower workers that are still computing updates from a local model replica several iterations behind the global state. Although ASP typically requires more iterations to converge than BSP since it processes fewer samples per-iteration, asynchronous training may not necessarily reach the same convergence targets as BSP due to staleness issues.

We measure '*static*' heterogeneity in a cluster based on the effective compute resources available to each worker. To exclusively study the impact of heterogeneity and not that of parallel scaling, we keep the cumulative resources fixed across various heterogeneous cluster configurations. For a cluster of CPUs, we define heterogeneity by its *Heterogeneity-level (HL)*. It compares the maximum CPU-cores available on the largest worker to the total cores on the smallest worker, as shown in Eqn. (3.1). For e.g., a 4-node cluster withf CPU-cores: (6, 6, 4, 24) has HL 24/4 = 6.0, while a homogeneous cluster with (10, 10, 10, 10) cores has *HL1*. Although both clusters have the same 40 cores cumulatively, the distribution of resources varies across workers in the two cases. For DL accelerators like GPUs or TPUs, static heterogeneity can also be measured by comparing FLOPS, compute cores available (like CUDA or tensor cores [176]) or total memory available across workers (in memory-constrained devices).

HL	Communication model	node #1	node #2	node #3	node #4
HL1	BSP	12	12	12	12
	ASP	10	10	10	10
HL2	BSP	12	12	8	16
	ASP	8	8	8	16
HL4	BSP	9	9	6	24
	ASP	10	10	4	16
HL8	BSP	6	6	4	32
	ASP	4	4	4	28

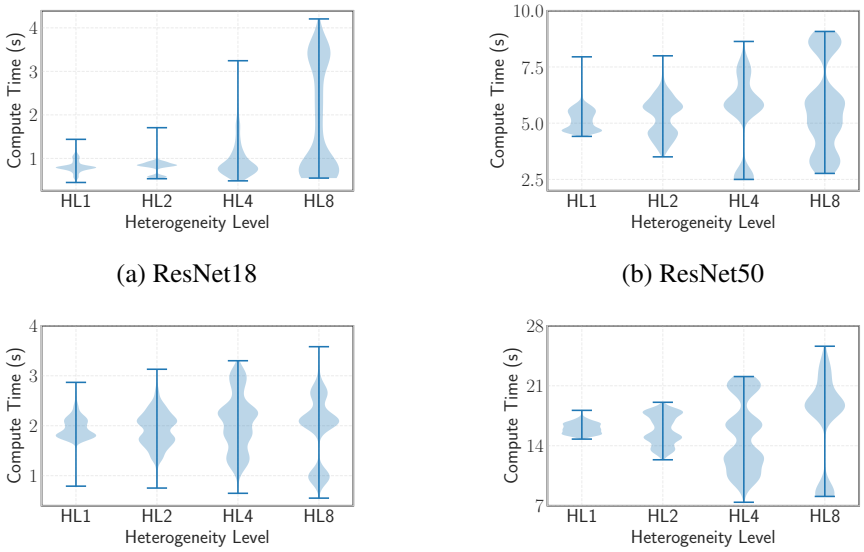
Table 3.1: CPU-cores allocation across 4 workers to simulate different HLs

$$\text{Heterogeneity-level (HL)} = \frac{\text{worker with max. \# cores/FLOPs}}{\text{worker with min. \# cores/FLOPs}} \tag{3.1}$$

Eqn. (3.1) measures the computational variance between workers in an offline manner, i.e., without actually running a DL model to measure performance differences. We emulate varying degrees of heterogeneity over a cluster of 4 docker containers [177], described in further detail in Appendix (A.2). In BSP *HL1*, each worker container is composed of 12 vCPUs. In our experiments, heterogeneity is increased by widening the divergence between workers #3 and #4. For e.g., in BSP the four containers at *HL8* are composed of 6, 6, 4 and 32 vCPUs respectively. With this notion of static heterogeneity, we look at its impact in BSP and ASP training in increasingly heterogeneous cluster configurations. Table (3.1) shows the CPU cores allocated to each worker in the 4-node cluster as we scale from a homogeneous cluster (i.e., *HL1*) to highly heterogeneous settings (=*HL8*). The cumulative core count is the same across all *HL*s, whether for BSP or ASP. However, total core count for the four workers in BSP is 48, while that of ASP is 40. This is because we want to keep cumulative cores *across* the entire cluster same across BSP and ASP. In addition to the workers, ASP training needs an additional PS node [49] allocated with 8 vCPUs to hold the central model state. We do not require a PS in BSP since we aggregate updates via decentralized AR in `OmniLearn`.

3.2.1 Impact of Heterogeneity in BSP Training

In BSP, we keep the local batch-size fixed and perform weak-scaling to increase the degree of parallelism. We refer to this identical, local mini-batch setting as '*uniform batching*'. Using different



(a) ResNet18 (b) ResNet50

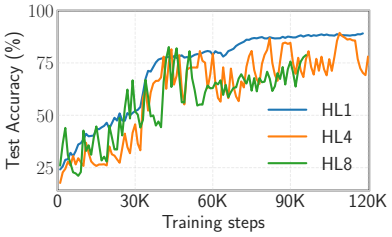
(c) AlexNet (d) VGG11

Figure 3.1: Distribution of worker computation times across different heterogeneity-levels (*HL*) in BSP. The variance in compute times increases with *HL*, resulting in stragglers causing slowdowns in synchronous training.

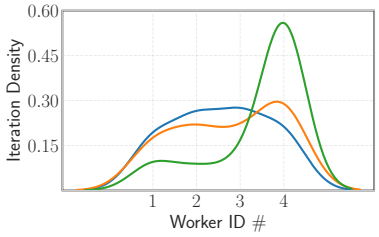
HL configurations outlined in Table (3.1), we evaluate the computation times of various DNNs. For *HL*1, 2, 4 and 8, we plot the density of worker computation times of ResNet18, ResNet50 [9], AlexNet [1] and VGG11 [8] with BSP, shown in Fig. (3.1). From the density distributions, workers with fewer resources take longer to calculate gradients for the same local mini-batch size. As heterogeneity gets worse with higher *HL*s, the distribution is more spread apart (first and third quartiles) and the median (second quartile) is higher as well. As a result, divergence in worker compute time rises, leading to stragglers and longer iteration times that degrades parallel scaling.

3.2.2 Impact of Heterogeneity in ASP Training

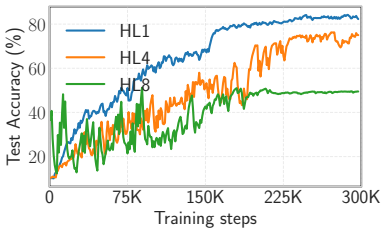
Uniform batching at high *HL*s degrades model convergence in ASP. This is because staleness gets worse at high *HL*s from computing updates on same-sized batches. Additionally, faster workers may perform multiple model updates in the same time it takes for a slower worker to compute even a single one. Thus, the amount of SGD updates performed in ASP over heterogeneous clusters is also skewed; faster workers execute a larger fraction of the total updates on central PS than slower ones. Statistical inefficiency arises due to late updates from slower workers overriding the multi-iteration progress of faster workers. We plot the convergence curves and iteration densities of the



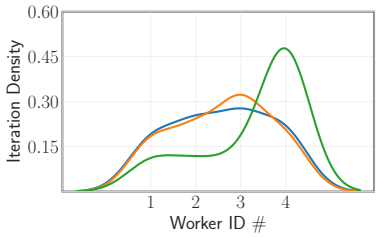
(a) ResNet18 Accuracy



(b) ResNet18 Iteration Density

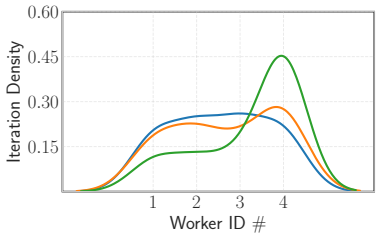


(c) ResNet50 Accuracy

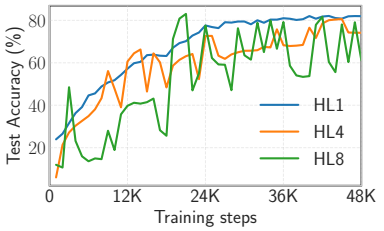


(d) ResNet50 Iteration Density

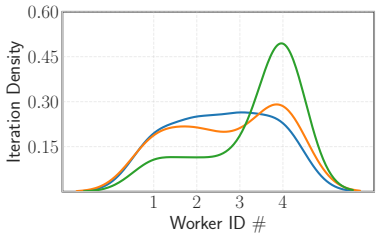
(e) AlexNet Accuracy



(f) AlexNet Iteration Density



(g) VGG11 Accuracy



(h) VGG11 Iteration Density

Figure 3.2: ASP model convergence and iteration KDE at different *HL*s. As heterogeneity rises, accuracy decreases due to staleness from uneven iteration densities at high *HL*s.

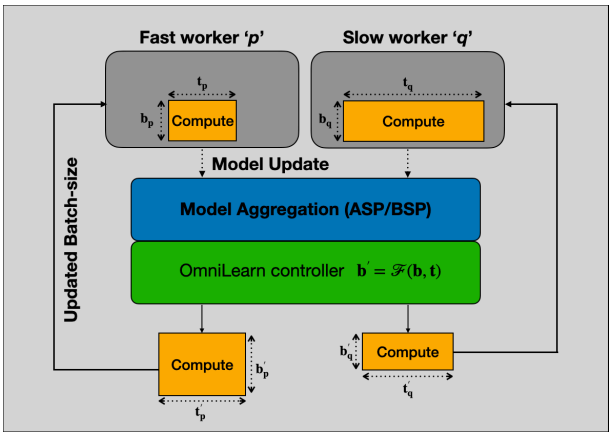


Figure 3.3: `OmniLearn` overview: Initially, fast and slow workers p, q train on the same mini-batch $b_p = b_q$ (shown as the breadth of 'Compute' block) that leads to stragglers (in BSP) or staleness (in ASP) as compute times $t_p < t_q$ (length of the 'Compute' block). Controller adjusts mini-batches to equalize batch computation times, i.e., $t_p^{'} \sim t_q^{'} \;:\; b_p^{'} > b_q^{'}$ since $p > q$ from a computational standpoint.

four models in Fig. (3.2). With high *HL*s 4 and 8, accuracy achieved by the models in noticeably lower than homogeneous configuration *HL1*. The lower convergence at these *HL*s can be explained from KDE plots of the total updates performed by each of the four workers (in Fig. (3.2b), (3.2d), (3.2f) and (3.2h)). Across all DNNs, *HL1* has a uniform distribution as all workers possess similar resources and the PS receives similar volumes of updates from all workers. As heterogeneity rises, iteration densities become more skewed, leaning more towards workers with more resources as they make more updates on the global model state than smaller workers. Here, worker #4 has more resources allocated to it (in accordance to Table (3.1)), and thus has a higher iteration density than other workers.

3.3 `OmniLearn` Design

3.3.1 Variable-Batching to address Static Heterogeneity

Conventional data-parallel training performs 'uniform batching' where each worker's local batch-size is fixed irrespective of the computational resources available on it. This results in parallel inefficiency over BSP, and statistical inefficiency in ASP. The key insight of `OmniLearn` is that since computational resources across workers may not necessarily be identical, neither should be the proportional work performed by them. This means that worker batch-sizes need not be identical

across workers; instead, mini-batch size allocated to a worker should be proportional to a worker's resource availability or computational capability. We thus assign varying amounts of work to each node in the cluster in order to minimize the divergence in worker compute times. Fig. (3.3) illustrates this batch-adjustment approach where initially, faster worker 'p' and slower worker 'q' train with identical batches $|b_p| = |b_q|$, but compute time of 'q' is considerably larger than 'p'. The mini-batch controller adjusts mini-batches of the two workers in order to equalize their respective computation times in the next update, i.e.. $t_p^{'} \sim t_q^{'}$. This works by assigning the faster worker 'p' with a larger batch-size than 'q'. The exact methodology of our static variable-batching and proportional-control based dynamic-batching techniques are described in subsequent sections.

***BSP Variable-Batching**:*

In variable-batching, workers are adjusted proportional to their computational resources (i.e., each worker's local *HL*). Although stragglers can be mitigated this way, the global batch-size should stay fixed as its a critical hyperparameter. Assuming per-worker batch-size $|b|$ in uniform batching, the global batch-size in BSP processed in a single iteration is $N|b|$. Keeping the global batch fixed, we reduce the batch-size on slower workers while proportionally raising it on faster workers.

$$|b_n| = \frac{c_n}{\sum_{n=1}^{N} c_n} \cdot N|b| \tag{3.2a}$$

$$g_{(n,i)} = \lambda_{\mathbf{n}} \odot \frac{\partial}{\partial w_i}\left(\frac{1}{|b_n|} \sum_{b_{(n,i)} \in \mathcal{B}_n} \mathcal{L}(b_{(n,i)}, w_i)\right) \; : \; \lambda_{\mathbf{n}} = \frac{|b_n|}{\sum_{i=1}^{N} |b_i|} \tag{3.2b}$$

$$w_{i+1} = w_i - \eta \cdot \sum_{n=1}^{N} g_{(n,i)} \tag{3.2c}$$

By measuring static heterogeneity with *HL*, we scale batches on worker 'n' based on its compute resource c_n (number of cores, FLOPS, etc.) relative to the whole cluster, shown in Eqn. (3.2a). Gradients computed over larger batches are more accurate and representative of true gradients compared to smaller batches [40, 142]. Thus, instead of simply averaging the updates, we perform a weighted mean in `OmniLearn` where we assign a higher score to gradients from workers with larger batches. As shown in Eqn. (3.2b), gradients from worker 'n' at iteration 'i' are scaled by

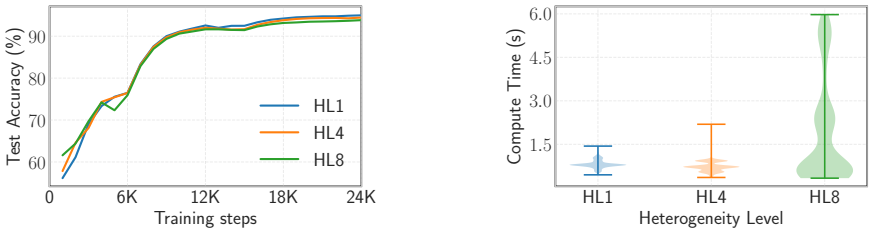
'λ_n', proportional to its batch-size. This preserves the global batch-size and convergence properties of SGD (Eqn. (3.2c)) [68].

Fig. (3.4) describes the parallel and statistical performance of BSP with variable-batching with compute time distribution across workers and DNNs' convergence curves. The *HL* configurations are simulated as per Table (3.1). It is to be noted that *HL1* configuration is akin to uniform-batching where every workers has the same batch-size. Despite weighted aggregation, we observed that model accuracy marginally decreases for most models as heterogeneity rises. For e.g., ResNet18 attained 95.15%, 94.54% and 94.03% accuracy at *HL1*, 4 and 8 respectively, while ResNet50 converged to 88.04% and 84.65% accuracy at *HL1* and *HL8*.

As for the compute-time distributions of ResNet18, ResNet50 and VGG11 in Fig. (3.4b), (3.4d) and (3.4h), the second quartile of HL8 is lower than *HL1*. This is because variable-batching tries to best balance the average computation time across the cluster by adjusting worker batches corresponding to their respective resource availability, However, we also noticed that the min-max compute times over the distribution is spread more apart at higher *HLs*. *This is because batches allocated strictly based on CPU core-counts are not an accurate estimate of training throughput*. Batch execution is also not strictly parallel over multi-core devices. This means that allocating a larger batch to a worker with more cores does not necessarily mean its corresponding computation time will be proportionally smaller.

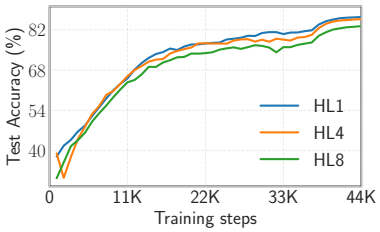
***ASP Variable-Batching**:*

Compared to BSP, ASP training performs lesser work per iteration by processing local batches of only a single worker to update global parameters on the PS. Like Eqn. (3.2a), we adjust worker batch-size based on its allocated resources so that their respective compute times are similar and staleness is attenuated. However, no gradient scaling (i.e., weighted aggregation) is performed in ASP since model updates are not aggregated among workers. This changes the effective batch-size for making model updates which may affect model generalization [140, 143]. To keep this hyperparameter consistent between BSP and ASP, we split the cumulative batch size $\sum_{n=1}^{N} b_n$ among 'N' workers as per Eqn. (3.2a). To account for ASP's small-batch training with noisy gradients, we apply a linear learning-rate rule in Eqn. (3.3b) where step-size on worker 'n' is scaled to η_n [61, 152] when updates coming from worker 'n' are performed over the smaller batch 'b_n'.

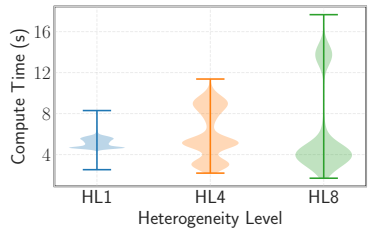


(a) ResNet18 Accuracy

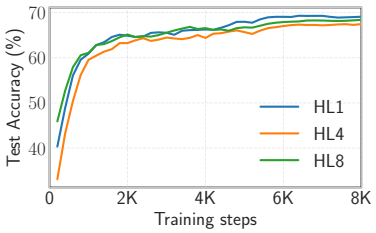
(b) ResNet18 Compute times



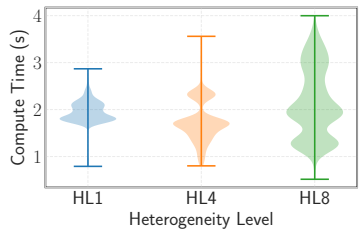
(c) ResNet50 Accuracy



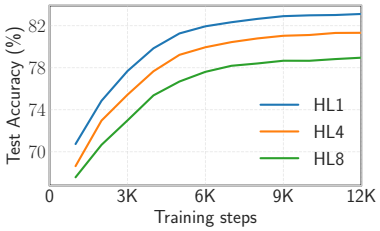
(d) ResNet50 Compute times



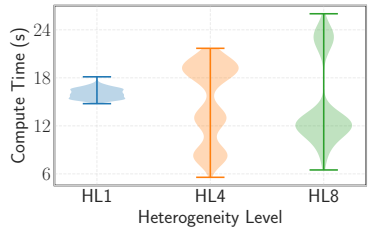
(e) AlexNet Accuracy



(f) AlexNet Compute times



(g) VGG11 Accuracy



(h) VGG11 Compute times

Figure 3.4: BSP variable-batching: Weighted aggregation reaches high convergence quality at larger *HLs* in ResNet18, ResNet50 and AlexNet, while VGG11 loses considerably more accuracy. At high *HLs*, although median of compute times is lower, the min-max are more spread apart as allocating batches based on core-count is not an accurate estimate of training throughput.

This is in harmony with prior work that explores the interplay between learning-rate and batch-size [156].

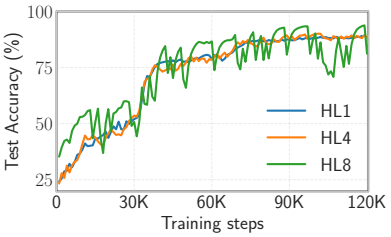
$$g_{(n,i)} = \frac{\partial}{\partial w_i}\left(\frac{1}{|b_n|}\sum_{b_{(n,i)}\in\mathcal{B}_n}\mathcal{L}(b_{(i,n)}, w_i)\right) \tag{3.3a}$$

$$w_{i+1}^{(n)} = w_i^{(n)} - \eta_n \cdot g_{(n,i)} \quad \text{where} \quad \eta_n = \eta \cdot \frac{b_n}{\sum_{i=1}^{N} b_i} \tag{3.3b}$$

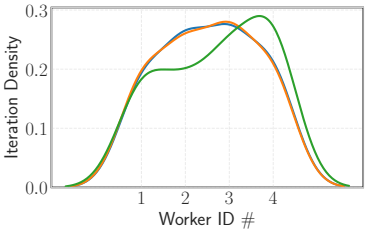
We plot the results of ASP variable-batching for different DNNs in Fig. (3.5). Although significantly better than staleness from uniform-batching at various *HLs* (from Fig. (3.2)), staleness is still not entirely eliminated in variable-batching due to throughput estimation errors. As a result, accuracy considerably oscillates over the epochs at higher *HL8* across all DNNs. At *HL4* and *HL8*, ResNet18 and AlexNet reach similar accuracy levels as uniform-batching at *HL1*. This is because staleness is significantly reduced with variable batching, as seen from the similar iteration densities in Fig. (3.5b) and (3.5f); *HL1* and *HL4* have near-identical densities while worker #4 with additional resources at *HL8* has a slightly higher iteration density than other workers. Similar density among workers implies that workers have comparable compute times and thus, contribute similar volume of parameter updates on the PS. On the other hand, ResNet50 and VGG11 saw a noticeable accuracy drop at *HL8*, corroborated by their unequal iteration densities in Fig. (3.5d) and (3.5h). At *HL8*, worker #4 performs the most updates in asynchronous training since it has the most CPU-cores allocated to it. For the same models, *HL4* has a similar density to *HL1* which is also reflected in their convergence curves (from Fig. (3.5c) and (3.5g)).

3.3.2 Proportional-Control Mechanism for Dynamic Heterogeneity

To mitigate stragglers or staleness, workers must adjust their respective batches adaptively. In the previously described 'static' variable-batching, mini-batches were allocated either based on a node's compute-core or FLOPS count. However, this open-loop estimation is not an accurate metric to estimate the actual worker throughput of DNNs and execution hardware. This is because throughput in distributed training depends on parallel scaling characteristics governed by Amdahl's law [150].

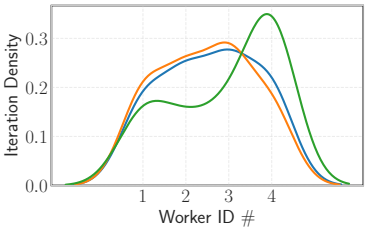


(a) ResNet18 Accuracy



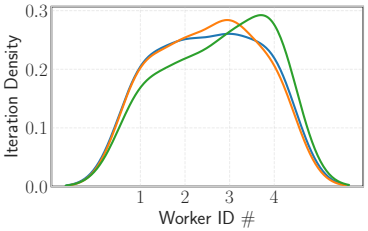
(b) ResNet18 Iteration Density

(c) ResNet50 Accuracy

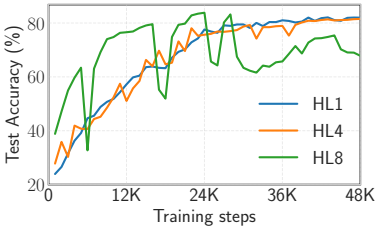


(d) ResNet50 Iteration Density

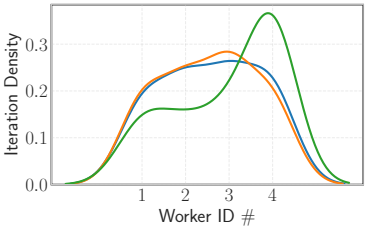
(e) AlexNet Accuracy



(f) AlexNet Iteration Density



(g) VGG11 Accuracy



(h) VGG11 Iteration Density

Figure 3.5: ASP variable-batching: Convergence quality improves at high *HLs* compared to uniform-batching in heterogeneous settings. From the iteration density estimates, accuracy improvement is attributed to staleness reduction in variable-batching.

Additionally, many scenarios in cloud and shared clusters yield fluctuating resource availability for which the static approach is ill-suited.

$$T_n^{(e)} = \frac{b_n^{(e)}}{t_n^{(e)}} \text{ and } \tau_n^{(e)} = t_n^{(e)} - \bar{t}^{(e)} : \bar{t}^{(e)} = \frac{1}{N}\sum_{j=1}^{N} t_j^{(e)} \tag{3.4a}$$

$$b_n^{(e+1)} = b_n^{(e)} + \Delta(b_n^{(e)}) \quad : \quad \Delta(b_n^{(e)}) = -\, T_n^{(e)}\tau_n^{(e)} \tag{3.4b}$$

`OmniLearn`'s dynamic mini-batching technique is designed to overcome throughput estimation errors as well as dynamic resource availability. By frequently adjusting worker mini-batches based on its throughput, we equalize compute time across all workers. For worker 'n' that calculates its update in time $t_n^{(e)}$ with local throughput $T_n^{(e)}$ at epoch 'e', we ideally want $t_i^{(e)} = t_j^{(e)}\ \forall\ (i,j)$ workers. With average computation time over the cluster as $\bar{t}^{(e)}$, dynamic-batching uses a simplistic proportional-control mechanism to equalize batch processing times, i.e., minimize error $\tau_n^{(e)}$ from Eqn. (3.4a) by updating worker batches as per Eqn. (3.4b). Slower workers thus have a positive error $\tau_n^{(e)}$ and need to reduce their local batch-size, while faster workers with a negative error should increase their batch size since they are capable of handling a heavier workload. Our approach essentially combines DL with black-box PID controllers [172]. Instead of tuning an arbitrary constant in PID controllers, we use the estimated throughput.

In dynamic-batching, workers are assigned variable batches based on static heterogeneity and any errors in throughput approximation are ironed out and fixed by the control mechanism. However, dynamic batching works well with any initial mini-batch size, so even if workers are allocated the same batches (i.e., uniform batching), `OmniLearn` can converge to the optimal batches in just a few adjustments.

***Control Stability in Dynamic-Batching**:*

In our implementation, batch-size adjustments are made at the end of each epoch, although this is a tunable parameter and the granularity can be changed easily, for instance, after every few iterations. Adjusting batch-sizes is not a zero-cost operation as it involves re-initializing the dataloader with

the updated batch-size. Additionally, the compute time of workers will rarely converge to the same exact average, thus there will always be some degree of error that proportional-control will try to chase. To control this perpetual batch adjustment, we leverage three techniques: *deadbanding*, *exponentially smoothed computation times* and *batch-size bounds*.

$$\Delta b_{min} \leq \left| \frac{b_n^{(e+1)} - b_n^{(e)}}{b_n^{(e)}} \right| \tag{3.5}$$

Deadbanding: We calculate the updated batches at the end of each epoch via proportional-control. We also implement deadbanding along with dynamic-control where batch-size on worker 'n' is updated only when the change is substantial and exceeds a predefined threshold Δb_{min}, shown in Eqn. (3.5). When change in mini-batch is lower than Δb_{min}, no adjustments are made. The chosen value of the threshold decides how sensitive we want the controller to be; an extremely small value may incur additional performance overhead from a frequently changing batch-size.

Exponential smoothing: With deadbanding, `OmniLearn` makes batch adjustments if a workers' underlying resource availability changes. To improve control stability and avoid spurious batch-size alterations, we compute the error between a workers' local compute time and average computation time across the cluster. We then apply exponentially weighted moving average, or EWMA such that more weights are assigned to recent iterations. This provides an integrator component in the controller and helps prevent outliers with respect to compute-time measurements.

Batch-size bounds: A potential risk of dynamic-batching is that slower workers may get assigned very small batches, while the batch-size on faster workers may get excessively large in highly heterogeneous settings. With smaller batches, the controller may degrade overall training throughput while exceedingly large-batches may increase the computation time even on the faster workers. Thus, we enforce lower and upper batch-size bounds (b_{min}, b_{max}) to preserve `OmniLearn`'s performance.

$$\triangle B = \sum_{j=1}^{N} b_j^{(e+1)} - Nb \quad \Rightarrow \quad b_n^{(e+1)} = b_n^{(e+1)} - \frac{\triangle B}{N} \tag{3.6}$$

Despite the min-max constraints on worker batches, the global batch-size can rise significantly over subsequent batch adjustments in BSP, especially when heterogeneity varies frequently. Thus, we enforce an additional rule in BSP to keep the global batch-size fixed to preserve parallel scaling,

prevent generalization loss and make fair comparison with uniform batching (where global batch $B = Nb$ for 'N' workers each with mini-batch 'b'). We do so by uniformly lowering the deviation between the adjusted batches from Eqn. (3.4) and global batch-size, as shown here in Eqn. (3.6).

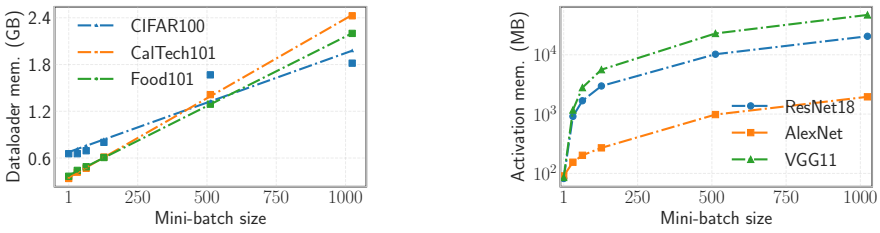
3.3.3 Memory Estimation Across Heterogeneous Systems

$$M_{total} = M_{model} + M_{gradient} + M_{optimizer} + M_{activation} + M_{batch} \tag{3.7}$$

Instead of compute cores, heterogeneity may even arise from the different types and amounts of memory available across workers. By Eqn. (3.7), the total memory required includes model parameters, gradients, intermediate activation maps, optimization-related memory and batch-memory [178, 179].

The model and gradient-size, i.e., M_{model} and $M_{gradient}$ is constant for a given DL model and can be estimated by knowing the product of total number of parameters and the byte-size of each parameter. For e.g., single-precision floating point representation takes 4 bytes, half-precision takes 2 bytes, etc. The optimizer memory $M_{optimizer}$ depends on the type and order of gradient information stored by the optimizer. This memory is also proportional to the model-size and stays constant throughout training. For instance, memory held by vanilla SGD is significantly lower than that needed by Nesterov's momentum and Adam optimization [178, 179]. The activation and batch-memory vary with the dataset and chosen mini-batch size and thus, may limit the maximum batch-size we can choose with `OmniLearn` when workers are highly heterogeneous with respect to the available memory. $M_{activation}$ is the memory consumed by the activation function while processing every sample in the forward pass, while M_{batch} is the memory held by the dataloader to process and store samples corresponding to the chosen batch-size.

In Fig. (3.6a), we fit the relationship between batch-memory and mini-batch size as a linear regression problem. For Canadian Institute For Advanced Research (CIFAR)100 [180], CalTech101 [181] and Food101 [182] datasets, the dashed line shows the memory estimated with our linear model while the scatter points show the actual dataloader memory consumption at batches $\{1, 32, 64, 128, 512, 1024\}$; this gives us accurate estimates on how much batch-memory will be needed for a given mini-batch size. Activation memory is also increases linearly with the batch-size, and can be readily estimated from a DNNs' structure and size. To validate the same, we run ResNet18,



(a) M_{batch} predicted vs. actual

(b) $M_{activation}$

Figure 3.6: (a) Batch memory is accurately predicted by our linear model for any batch-size. (b) Activation memory is directly proportional to the batch-size (y-axis is log-scale).

AlexNet and VGG11 on the aforementioned datasets with vanilla SGD optimizer. We deducted M_{model}, $M_{gradient}$, $M_{optimizer}$ and M_{batch} from the total memory consumed by the training process. The first three are the same as the model-size itself while batch-memory was retrieved from prior execution in Fig. (3.6a). For each mini-batch, this leaves only the activation memory which we plot in Fig. (3.6b). The y-axis is on the log-scale and shows how $M_{activation}$ rises linearly with its batch-size. With this, we can accurately predict the memory needed to run any configuration and set the cap on the min-max batch-size to use over heterogeneous memory systems in `OmniLearn`.

3.4 Experimental Evaluation

We now look at the parallel and statistical performance of `OmniLearn` in scenarios with static and dynamic heterogeneity. The end-to-end execution logic of `OmniLearn` in ASP and BSP is illustrated in Alg. (2) and (3) of Appendix (A.1). The cluster setup to simulate different *HL*s and complete training schedule is described in Appendix (A.2).

3.4.1 Weighted Gradient Aggregation in BSP Variable-Batching

In synchronous training, we perform weighted aggregation when workers are running different mini-batches. Here, the gradients are scaled in proportion to its local batch-size and reduced together instead of simply averaging the local updates. This attenuates the noise and improves the signal-to-noise ratio in the gradients [40, 25]. We compare convergence in BSP variable-batching with and without gradient scaling; the latter performs simple averaging by summing all the local gradients and dividing by the total number of workers. Table (3.2) shows how gradient scaling improves the

Configuration		Test accuracy (%)		
Model	**HL**	**Scaled grads.**	**Unscaled grads.**	**Difference**
ResNet18	*HL2*	**94.9%**	93.6%	+1.3%
	HL4	**94.54%**	93.76%	+0.78%
	HL8	**94.02%**	92.11%	+1.91%
ResNet50	*HL2*	**87.74%**	86%	+1.74%
	HL4	**85.94%**	84.33%	+1.61%
	HL8	**84.65%**	80.6%	+4.05%
AlexNet	*HL2*	**68.9%**	64.1%	+4.8%
	HL4	**68.38%**	60.67%	+7.71%
	HL8	**67.69%**	57.94%	+9.75%
VGG11	*HL2*	**82.38%**	79.4%	+2.98%
	HL4	**81.39%**	75.6%	+5.79%
	HL8	**78.98%**	71.19%	+7.79%

Table 3.2: Performance of weighted gradient aggregation in BSP variable-batching

convergence quality considerably over unscaled gradients. *HL1* is omitted since its equivalent to uniform-batching as all worker resources are similar, so both weighted and unweighted aggregation perform a simple mean operation. As heterogeneity rises, the accuracy improvement with gradient scaling improves even further, as seen from the increasing 'Difference' column that gives the delta between scaled and unscaled gradients' accuracy. Thus, we see that our weighted aggregation scheme assigns higher priority to less noisy gradients which improves the statistical efficiency of BSP.

3.4.2 Learning-Rate Scaling in ASP Variable-Batching

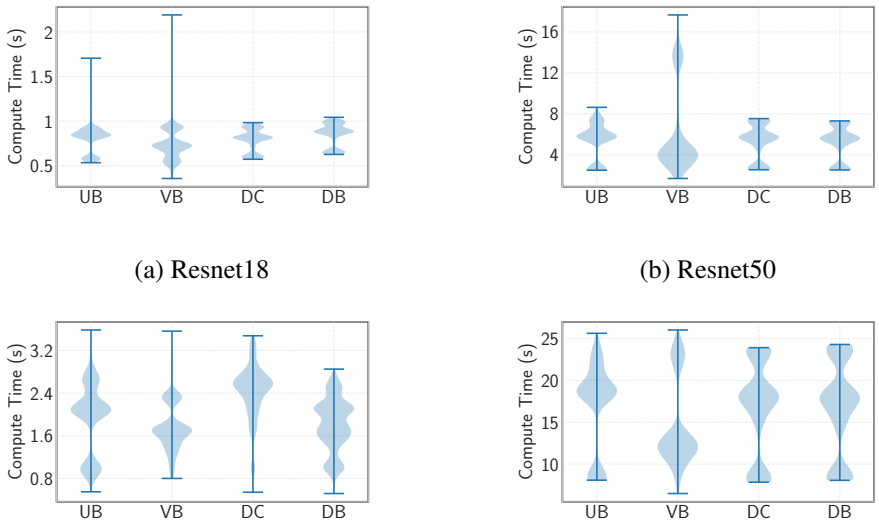
Learning-rate (LR) is another critical hyperparameter that needs careful tuning in data-parallel training [61, 152]. In ASP training, `OmniLearn` adjusts step-size relative to the global batch-size, if we had trained synchronously (i.e., processed 'Nb' samples per-iteration instead of 'b'). The intuition is that smaller batches prefer a smaller learning-rate to reach minima, and vice versa [40]. In Table (3.3), we record model accuracy at *HL4* and *HL8* when we scale LR with respect to a workers' local batch-size, or use the default LR schedule described in Appendix (A.2). For ResNet18, ResNet50 and AlexNet, we noticed a significant gain in statistical performance by performing LR scaling in `OmniLearn`. For an overparameterized network like VGG11, we observed marginal gains in test accuracy with LR scaling.

Configuration		Test accuracy (%)		
Model	**HL**	**with LR scaling**	**without LR scaling**	**Difference**
ResNet18	*HL4*	**78.76%**	77.2%	+1.56%
	HL8	**78.27%**	76.4%	+1.87%
ResNet50	*HL4*	**75.01%**	72.75%	+2.26%
	HL8	**61.39%**	54.5%	+6.89%
AlexNet	*HL4*	**66.41%**	64.25%	+2.16%
	HL8	**65.46%**	64.2%	+1.26%
VGG11	*HL4*	**81.92%**	81.86%	+0.06%
	HL8	**76.1%**	75.8%	+0.3%

Table 3.3: Performance of learning-rate scaling in ASP variable-batching

3.4.3 Compute Cost Analysis of Different Batching Techniques

In this section, we compare the worker computation times in `OmniLearn` with uniform-batching (UB), variable-batching (VB), dynamic-controller (DC) and deadband-controller (DB) over synchronous training. The results are shown in Fig. (3.7). As we move from variable-batching to dynamic and deadband-controllers, the quality of throughput estimation improves significantly and the average cluster computation time is reduced. The median or second quartile of density distribution gets smaller as we move from uniform to variable-batching. For e.g., in ResNet18 UB is most dense around 0.85 sec., while VB is around 0.65 sec. ResNet50 with UB is centered at 6 sec. and 4 sec. with VB. The improvement in parallel scaling with `OmniLearn` is most apparent with the largest model in our evaluation. UB on VGG11 is centered around 18.5 sec., and 12.5 sec. for VB. Across all DNNs, the maximum computation time with variable-batching is nearly as high as uniform-batching due to poor throughput estimation by statically allocating larger batches to workers with more CPU-cores. With VB, the maximum computation time was observed on workers with more computational resources (thus, running larger batches). With deadband-controller (DC), we apply dynamic-batching with the additional constraint of min-max values over the batch-size. DB and DC in `OmniLearn` mitigate outliers and improves the overall compute-density distribution of BSP training. This shows the effectiveness of our throughput-based proportional controller in balancing batches across workers and equalizing their computation times (as seen from the condensed density distributions of DB and DC in Fig. (3.7)).



(a) Resnet18

(b) Resnet50

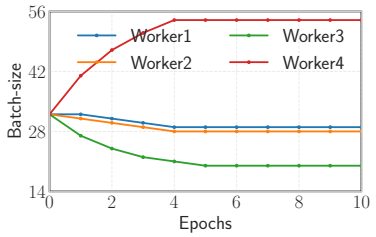
(c) AlexNet

(d) VGG11

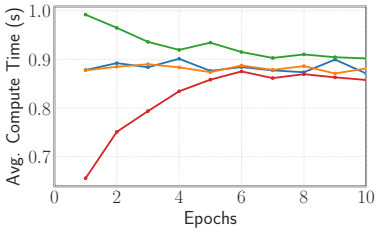
Figure 3.7: Distribution of worker compute times in `OmniLearn` with uniform-batching (UB), variable-batching (VB), dynamic controller (DC) and deadband controller (DB).

3.4.4 Deadband Controller under Static Heterogeneity

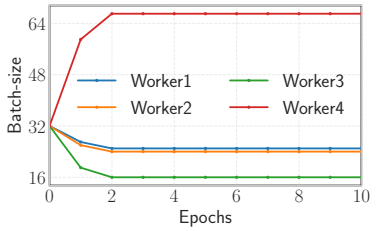
We now see how `OmniLearn`'s proportional-control works with any initial batch-size under '*static*' heterogeneity (i.e., systems' heterogeneity is fixed and does not change over time). Generally, we want to allocate worker mini-batches based on open-loop variable batching, and any throughput estimation errors are corrected over the course of training via control mechanism. However, an optimal initial point is not critical for batch-size estimation in `OmniLearn`. The farther away a workers' mini-batch is from the optimal, the more adjustments are needed to arrive at steady-state batches. We perform BSP dynamic-batching at *HL8* across 4 workers with per-worker mini-batch 32, and report the batch-size adjustments and corresponding computation times in Fig. (3.8). We set $b_{min} = 12$ and $b_{max} = 96$, $\Delta b_{min} = 15\%$ and perform batch adjustments as per Eqn. (3.4) (i.e., without any constraints on the global batch-size in Eqn. (3.6)). Thus, global batch-size may increase beyond $32 \times 4 = 128$ if numerous batch adjustments are made over the course of training. From Fig. (3.8), we see `OmniLearn` need not make several adjustments to reach equilibrium under static heterogeneity. With batch-size tuning triggered at the end of every epoch, we see that ResNet18, ResNet50 and VGG11 converged to their respective steady-state batches in about 2-4 adjustments, while AlexNet took around 10 adjustments before stabilizing. In AlexNet, worker #4



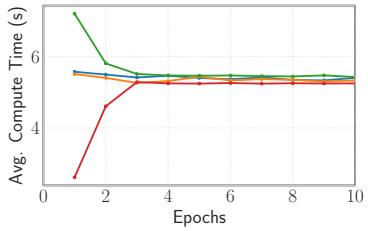
(a) ResNet18 batch-size



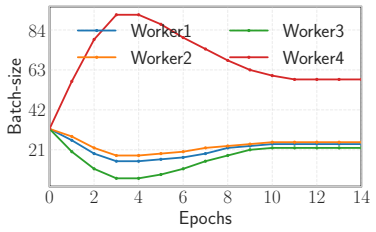
(b) ResNet18 compute times



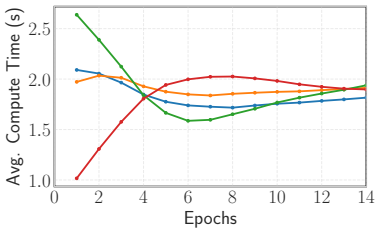
(c) ResNet50 batch-size



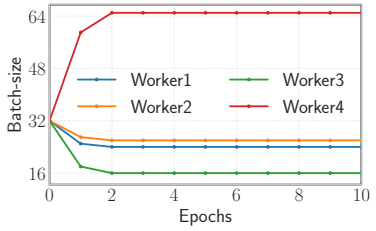
(d) ResNet50 compute times



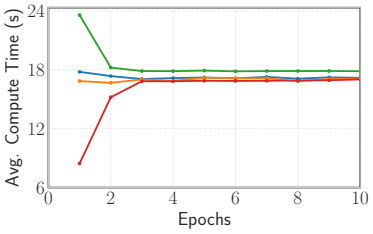
(e) AlexNet batch-size



(f) AlexNet compute times

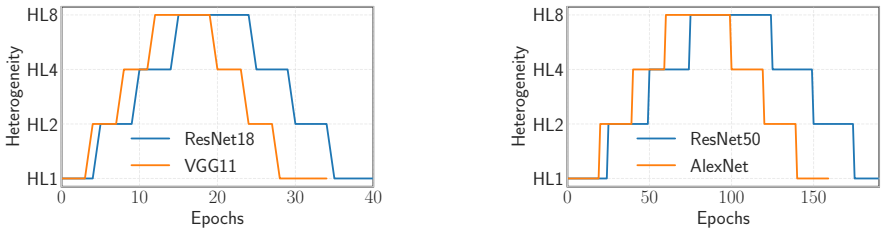


(g) VGG11 batch-size



(h) VGG11 compute times

Figure 3.8: Dynamic-batching in `OmniLearn` showing workers' batch-sizes and average compute times over the epochs under static heterogeneity. Average compute times are reported from epoch 1 since it is measured at the end of each epoch.



(a) ResNet18 and VGG11

(b) ResNet50 and AlexNet

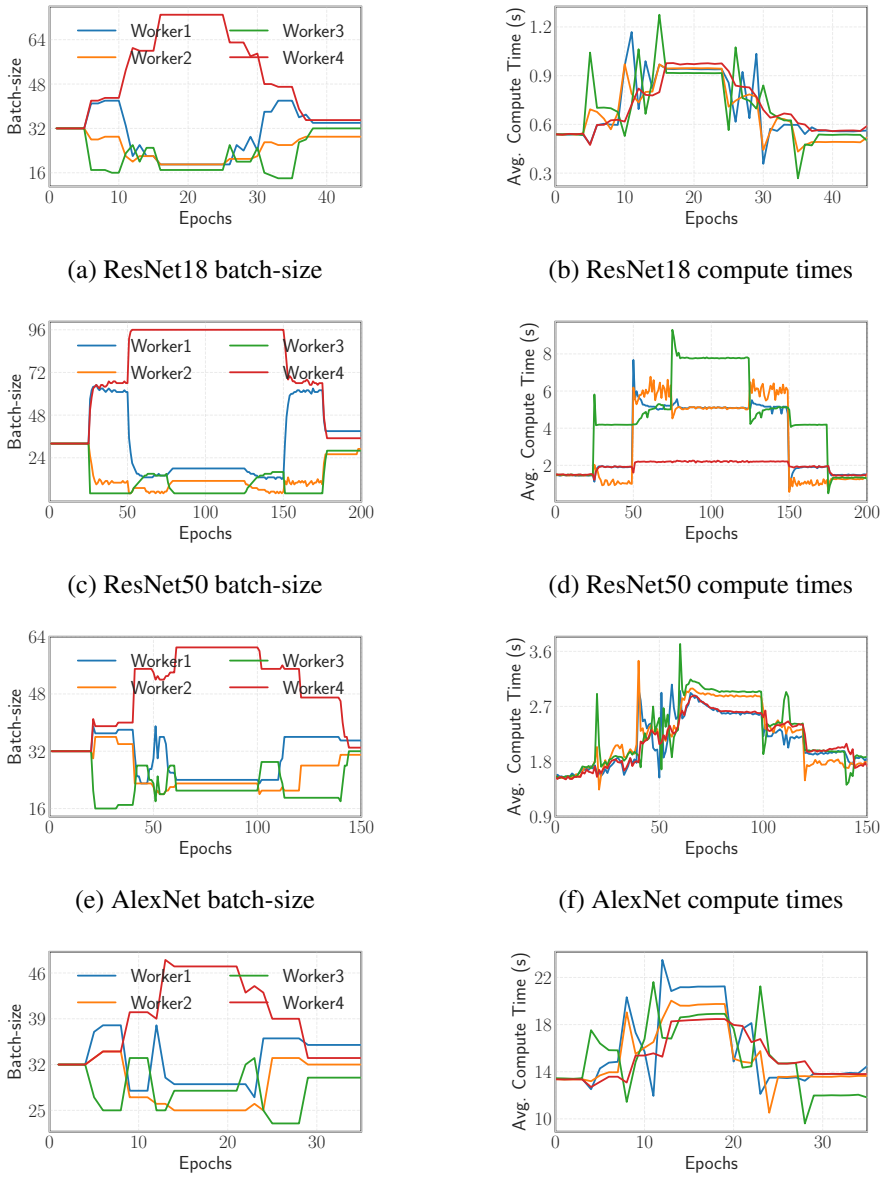
Figure 3.9: *HL*s are changed over the epochs to emulate dynamic heterogeneity in a cluster.

with the most resources (from Table (3.1)) peaked to maximum batch-size b_{max} as allowed by the batch-size bounds, then decreases and stabilizes to the nearest equilibrium state. Despite lack of global batch-size constraints, global batch-size was increased by only 12% for ResNet50, and under 2.5% for ResNet18, AlexNet and VGG11. When heterogeneity changes frequently, it becomes necessary to impose bounds and keep the global batch-size fixed to preserve final model accuracy.

3.4.5 Deadband Controller under Dynamic Heterogeneity

In the previous section, we saw how dynamic-batching works well under static heterogeneity. `OmniLearn` is also capable of responding to variations in available computational resources over time. We set the $b_{min} = 4$, $b_{max} = 96$, $\Delta b_{min} = 10\%$ and simulate dynamic heterogeneity in a step-wise manner between *HL1*, 2, 4 and 8. From Fig. (3.9), we vary *HL* at fixed intervals, triggered by a subprocess that tweaks the CPU-cores allocated to worker containers. For a specific *HL*, we allot CPU core-sets for the four workers as per Table (3.1). For ResNet18, we vary heterogeneity once every 5 epochs, and every 25 epochs in ResNet50. As for AlexNet and VGG11, we switch *HL*s after every 20 and 4 epochs respectively. By doing so, we can scale heterogeneity from *HL1* to *HL8*, and back to *HL1* over each DNNs training routine. This helps emulate scenarios when worker resource availability changes periodically in distributed training.

Fig. (3.10) shows the batch-size adjustments and worker compute times in `OmniLearn` with BSP under varying heterogeneity conditions. Across four worker containers, we set b_{min} = 4, b_{max} = 96, Δb_{min} = 10% and a global batch-size of 128. ResNet18 trains at *HL1* between epochs 0-4 and so, uses the same initial mini-batch 32 across every worker (i.e., starts with uniform batching). Although we switch to *HL2* at epoch 5, workers still train with the previously chosen batches since



(a) ResNet18 batch-size

(b) ResNet18 compute times

(c) ResNet50 batch-size

(d) ResNet50 compute times

(e) AlexNet batch-size

(f) AlexNet compute times

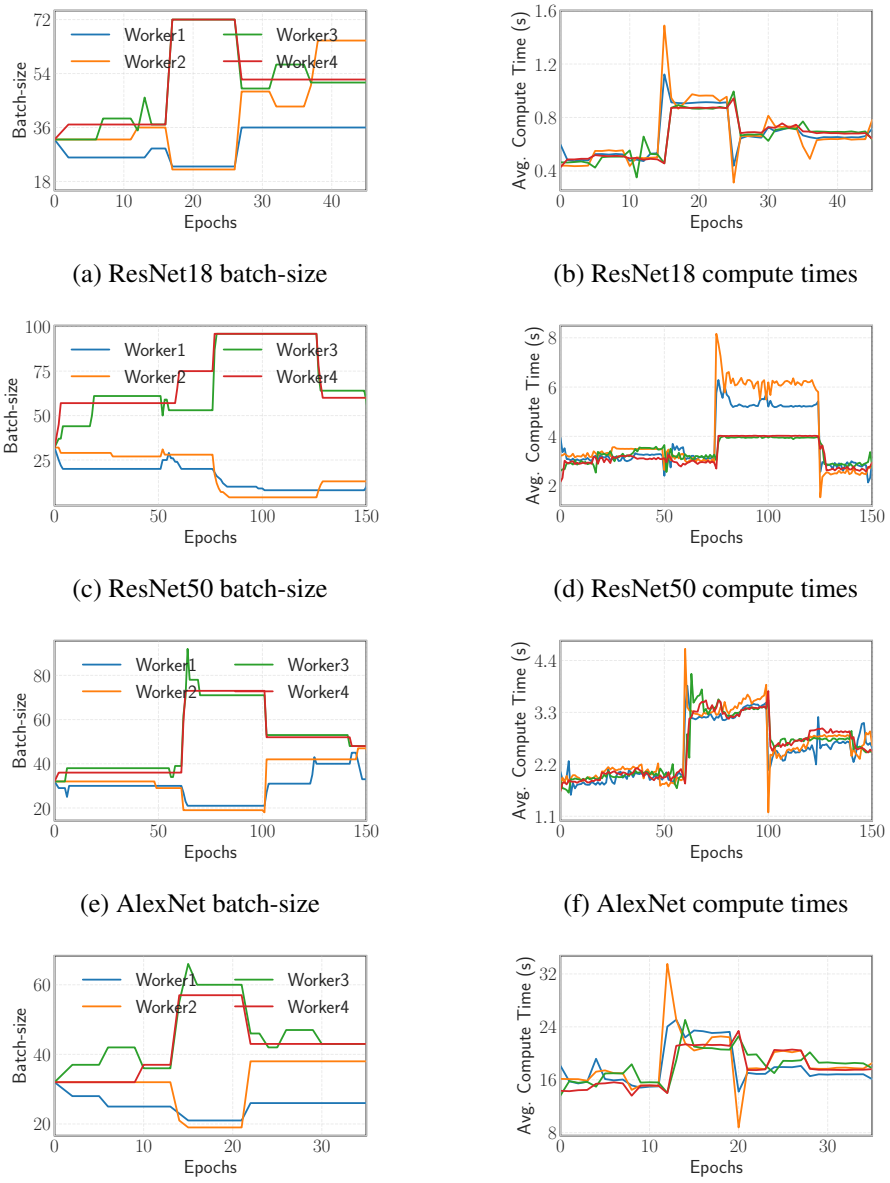
(g) VGG11 batch-size

(h) VGG11 compute times

Figure 3.10: `OmniLearn` BSP training under dynamic heterogeneity with $b_{min} = 4$, $b_{max} = 96$ and $\Delta b_{min} = 10\%$.

batch adjustments were made only at the end of an epoch. Thus, we observe a rapid variance in worker compute times when heterogeneity is increased (since some workers get faster and others get slower). Worker #3 has the least resources, so its the slowest in the cluster to calculate its gradients. The updated compute times are collected over the epoch and proportional-control (Eqn. (3.4) and (3.6)) is applied post epoch 5 and prior epoch 6. Hence, we see the spike in compute times at epoch 5, which then lowers and stabilizes after proportional-control for *HL2* between epochs 5-9. Heterogeneity is further increased to *HL4* at epoch 10, as evident from the abrupt change in compute times of that phase. `OmniLearn` is able to efficaciously detect the delta in worker performance and adjusts batches to reach steady-state equilibrium. For ResNet50 in Fig. (3.10c) and (3.10d), although controller stabilizes between epochs 50-150, worker compute times have noticeable variance. This is because workers #3 and #4 with the least and most resources respectively have already saturated to the smallest and largest mini-batch allowed as per the batch-size bounds, i.e., b_{min} and b_{max}. The batch-size of #3 cannot be further reduced, or that of #4 be increased in order to bring down the average cluster time. For AlexNet and VGG11, appropriate batch-adjustments are made soon after *HL* is changed as controller tries to best balance worker computation times given the constraints on batch-size bounds and Δb_{min}.

Next, we run `OmniLearn` under ASP and emulate dynamic heterogeneity. We use the same configuration as BSP, $b_{min} = 4$, $b_{max} = 96$ and $\Delta b_{min} = 10\%$. To keep the cumulative resources fixed over decentralized BSP and centralized ASP, total cores allocated to the four workers are lower in ASP due to additional resources needed for the PS container (from Table (3.1)). We then vary HL over the epochs for each model as per Fig. (3.9). With the same initial mini-batch 32, batches are tuned based on worker compute times logged over the epochs. It is important to note that the averaged estimates of computation times measured over the iterations in ASP may not be the same as BSP even if the mini-batch size was the same. This is because the total CPU-cores allotted to ASP workers is lower than BSP; cumulatively, ASP workers have 40 cores while BSP has 48 cores. Additionally, BSP has a synchronization barrier where every worker executes its updates and logs compute time at every iteration. In ASP however, frequency of updates performed by workers is not the same; containers with more resources execute a larger fraction of total updates on the PS than those with fewer resources. Smaller workers thus have higher variance in their measured compute times. Still, we see in Fig. (3.11) that `OmniLearn` under ASP is successfully able to



(a) ResNet18 batch-size

(b) ResNet18 compute times

(c) ResNet50 batch-size

(d) ResNet50 compute times

(e) AlexNet batch-size

(f) AlexNet compute times

(g) VGG11 batch-size

(h) VGG11 compute times

Figure 3.11: `OmniLearn` ASP training under dynamic heterogeneity with $b_{min} = 4$, $b_{max} = 96$ and $\Delta b_{min} = 10\%$.

tune its batches based on worker compute times. Akin to BSP, workers with more resources tend to have a lower computation overhead and thus, can afford larger batches to process. For instance, ResNet18 at *HL8* between epochs 15-24 has the most resources on worker #4. `OmniLearn` thus applies a larger batch-size on this worker in order to equalize worker compute times that minimizes staleness. Unlike BSP, ASP does not impose a global batch-size rule; however min-max bounds on the batch-size of individual workers still applies.

3.5 Conclusion

In this chapter, we introduced `OmniLearn`, a distributed learning framework to efficiently train DNNs over heterogeneous clusters. We proposed the *Heterogeneity-level* (*HL*) metric to account for the variance in computational capability between workers and show how the parallel and statistical efficiency is affected in BSP and ASP-based training. The former suffers from stragglers, while the latter faces staleness issues. With variable-batching, we attenuate these challenges to some extent by performing variable work based on the amount of computational resources available on each worker; compared to uniform-batching, this reduced training time by 14-85% in highly heterogeneous settings. We further fine-tune this approach with proportional-control to equalize compute across the entire cluster, thus ameliorating stragglers or staleness, and improving parallel scaling as well as convergence quality in distributed training. Last, we show how `OmniLearn` is able to work efficiently in an online manner with minimal overhead, and works well in settings with static or dynamic heterogeneity. On average, our system accelerates training across heterogeneous systems by 26% while attaining comparable convergence targets.

3.6 Related Publications

1. **Tyagi, Sahil** and Sharma, Prateek. "OmniLearn: A Framework for Distributed Deep Learning over Heterogeneous Clusters." (under review), 2024.

2. **Tyagi, Sahil** and Sharma, Prateek. "Taming Resource Heterogeneity In Distributed ML Training With Dynamic Batching." 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS) (2020): 188-194.

CHAPTER 4

BALANCING <COST, TIME> TRADE-OFFS OF DL IN THE CLOUD

4.1 Introduction

Cloud computing platforms like Amazon Web Services (AWS), Google Cloud Platform (GCP) and Microsoft Azure [183] can provide the necessary infrastructure and computational resources for training sota DNNs. However, DL techniques as well as their frameworks (such as TensorFlow [97], PyTorch [98]) are oblivious to costs associated with training on provisioned cloud resources. Additionally, cloud providers offer hundreds of different VM sizes and configurations, each with different cost-performance trade-offs that makes it extremely challenging to select the "right" type and quantity of cloud resources. As a result, DNNs are often trained on sub-optimally configured cloud resources, leading to cost overruns, slow performance and underutilized resources.

These challenges also persist when optimizing resource allocation for conventional distributed applications (such as map-reduce [19]) in the cloud [184]. But distributed training comes with unique execution and synchronization characteristics, as well as a plethora of configurable hyperparameters that impact performance and resource efficiency. In this chapter, we introduce `Scavenger`, a service developed on principled and practical techniques for optimizing time and cost of distributed training in the cloud. Both parallel and statistical efficiency need to be considered to select the optimal job configuration such as cluster-size (N) and global batch-size (B). By combining conventional parallel scaling concepts and insights into SGD noise, we develop models for estimating time and cost of training on different (N, B) configurations. Overall, we make the following contributions:

- Thoroughly investigate cost vs. time trade-offs in distributed training, and show how parallel and statistical efficiency impact performance.
- Show how gradient variance results in SGD noise, and how it can serve as a reliable scalability indicator for elastic horizontal or vertical scaling.
- Develop new analytical models that consider both parallel and statistical efficiency for pre-

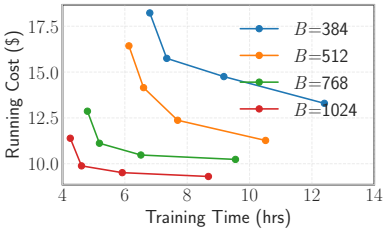


Figure 4.1: ResNet18 cost-time trade-offs over various (N, B) configurations. (Going from left-to-right) For any 'B', each point represents a cluster of 20, 16, 12 and 8 workers. A larger cluster-size has lower training time but higher costs.

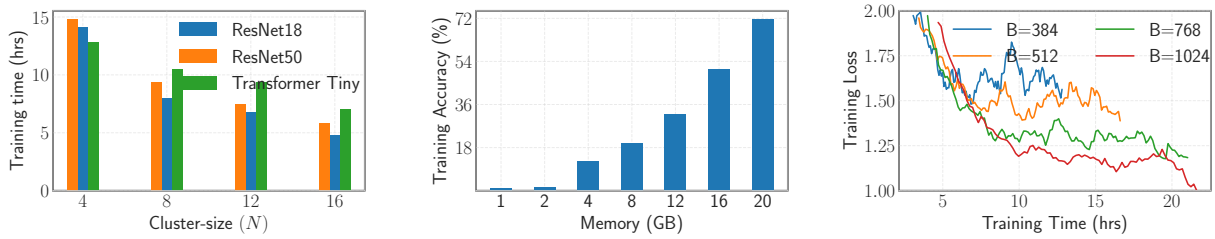
dicting DNNs' performance. Our model predicts time and cost of different job configurations (i.e., cluster-size and global batch-size), and constructs full trade-off curves (pareto-frontiers) with accuracy as high as 98%.

- Our models search for the optimum job configuration in a model agnostic and online manner, and reduce training time by up to 2× over a naively chosen configurations.

4.2 Motivation

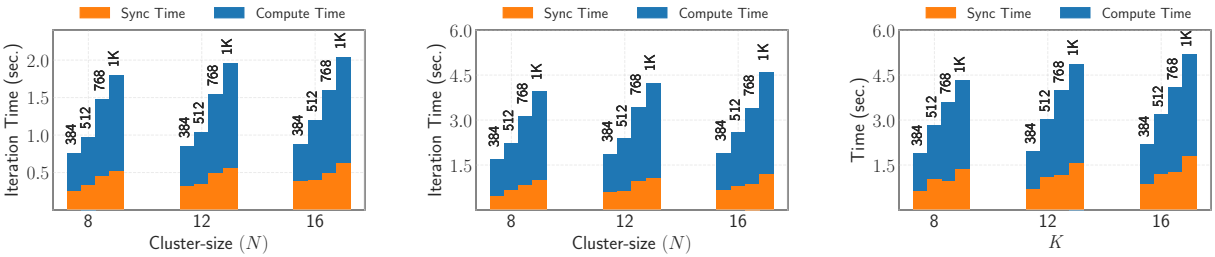
From Chapter (2), synchronous or BSP training (shown as Eqn. (2.2)) performs weak-scaling by processing 'b' samples on each of the 'N' workers, effectively training with a global batch-size $B = Nb$. Further, distributed training is resource *elastic*; DNNs can be trained across varying cluster-sizes and configurations that can even be changed at runtime. Training can either be scaled-out horizontally by adjusting the number of workers (i.e., by varying 'N'), or scaled-up vertically by changing the global batch-size 'B'. Frameworks like TensorFlow and PyTorch support model checkpointing, so we can change horizontal or vertical scaling dynamically by checkpointing the model state and resume training over a different configuration. Recent DL frameworks even come with fault-tolerant and elastic training capability right out-of-the-box [185].

Such elasticity makes DL a good fit for clouds, since we can easily scale the cluster by adding or removing VMs, and change the underlying VM size to increase the batch-size and intra-worker parallelism. `Scavenger` exploits this elasticity in search for the ideal cloud configuration. However, distributed training has a statistical efficiency aspect as well that introduces complex trade-offs [186] which makes it challenging to determine the ideal cluster configuration (N, B). Naively run-



(a) (b) (c)

Figure 4.2: (a) Training time decreases as cluster-size increases, but not proportionally for ResNet18, ResNet50 and Transformer-Tiny. (b) Training accuracy attained by Transformer Base before job fails due to memory constraints. (c) Training loss on ResNet18 with different batch-sizes. Larger B takes lesser time to reach the same model quality.



(a) ResNet18 (b) ResNet50 (c) Transformer-Base

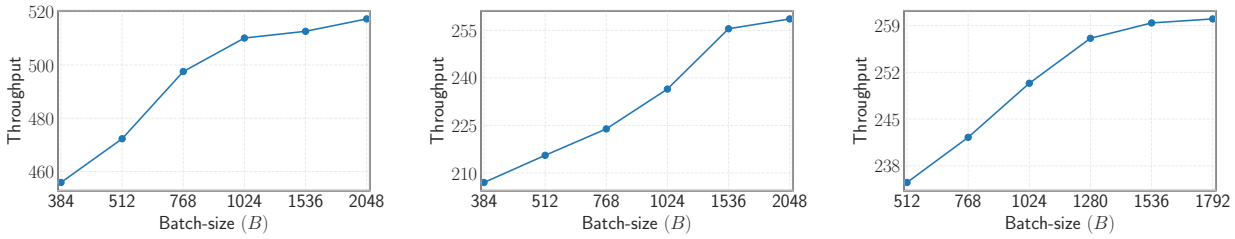
Figure 4.3: Gradient computation and synchronization breakdown across different cluster and batch-sizes, i.e., (N, B) configurations.

ning more workers or increasing the mini-batch size has diminishing returns. Fig. (4.1) shows the running cost and time of training ResNet18 at different cluster and batch sizes. Each point corresponds to a specific (N, B), and there is a pareto-relationship between cost and time (i.e., one improves at the detriment of the other with diminishing returns) which makes it non-trivial to choose the "ideal" configuration.

Horizontal Scaling by Changing Cluster-size:

Parallel scaling can be improved by adding more workers, or increasing 'N'. Fig. (4.2a) shows training time with rising cluster-size to reach the same accuracy levels in ResNet18, ResNet50 [9] and Transformer tiny [4]. As the number of workers increases, training time decreases with diminishing returns. Raising workers from 4 to 16 does not proportionally decrease training time by a factor of $4\times$.

This parallel inefficiency in distributed training arises from non-negligible communication or



(a) ResNet18 (b) ResNet50 (c) Transformer-Base

Figure 4.4: Training throughput (samples/sec.) on increasing the global batch-size B for $N = 16$. Throughput increases as we raise B up to a certain point, then plateaus.

synchronization overhead. As per Eqn. (2.2b), assuming negligible data-movement and I/O overheads, iteration cost is majorly composed of gradient computation and a synchronization step where updates are aggregated. Fig. (4.3) shows the breakdown of gradient computation and communication where the cumulative cluster capacity in terms of CPU-cores and memory allocated to workers is the same across different 'N'. We see the overhead of horizontal scaling in terms of higher synchronization cost as we increase the number of CPU workers communicating with PS strategy; bandwidth cost increases linearly with 'N' in star-topology [187]. For the same cluster-size in Fig. (4.3), computation increases with larger batches as more work is done (to compute loss) in each iteration.

The allocated memory-size of a VM is another factor that influences the choice of VMs to use. Insufficient memory only allows small-batch training, which limits parallel scaling by performing less work per-iteration and needs more iterations to complete an epoch, consequently attaining lower training accuracy in the same time (as large-batch BSP training). Fig. (4.2b) shows the training accuracy of a Transformer under a strict time deadline. From left-to-right, we increase batch-size 'B' and so, the corresponding memory consumed for training increases as well due to higher activation memory requirements [178]. This is because activation memory in a transformer model is directly proportional to the number of transformer layers, hidden dimensions, sequence length and batch-size. VMs below 4 GB memory failed from inadequate or OOM (out-of-memory) errors.

Vertical Scaling by Changing Batch-Size:

BSP communication overheads can be mitigated with larger batches, which reduces the number of iterations required to complete an epoch, thus improving parallel efficiency. Fig. (4.2c) shows the training loss in ResNet18 at different 'B' and shows that larger batches like 1024 achieve better loss value than smaller batches for the same duration of training. The gains in compute efficiency with larger batches can be seen for the three DNNs in Fig. (4.4), where the training throughput (i.e., number of samples processed per-unit of time) first grows as we increase the global batch-size, and then saturates after an inflection or knee-point. The plateau in throughput after a certain B occurs when the CPU utilization of the workers hits the ceiling and cannot increase further. The results are shown for a cluster of 16 workers where each VM is a GCP E2-standard with 4vCPUs and 16 GB memory.

Statistical Efficiency of Elastic Training in the Cloud:

Parallel training does not scale linearly with either horizontal or vertical scaling. Improving batch-size or cluster-size by a certain amount does not proportionately reduce the training time to reach similar convergence levels. In conventional parallel applications, computing work performed by all the processes is equally useful. However, as explained in §2.9.2, not all the work in distributed training is equally effective due to the stochastic nature of mini-batch gradient descent. The work done (i.e., gradients computed) by the individual nodes does not fully compose as total forward progress made does not equal to the sum of the progress made by each worker or across individual iterations. This 'statistical inefficiency' reflects how "aligned" the gradients are across different workers, and is a fundamental attribute of SGD family of optimization algorithms. Statistical efficiency can be captured by computing SGD noise, i.e., using gradient variance information in BSP training.

The gradients computed over smaller-batches tend to be noisier as they are more likely to differ from updates calculated over the entire training data. Increasing the cluster-size 'N' (with weak-scaling where per-worker batch-size 'b' is fixed) can increase the effective global batch-size and converge faster (due to better statistical efficiency), but may increase the associated training costs as well. On the other hand, increasing the batch-size has other associated costs like generalization loss, additional memory requirements or communication overheads. In terms of parallel efficiency, we

see the diminishing returns of large batches in Fig. (4.4) where throughput alone is not sufficient to capture performance. Due to statistical inefficiency, we need to consider the end-to-end wall-clock time reach a desired accuracy level with a specific (N, B) configuration.

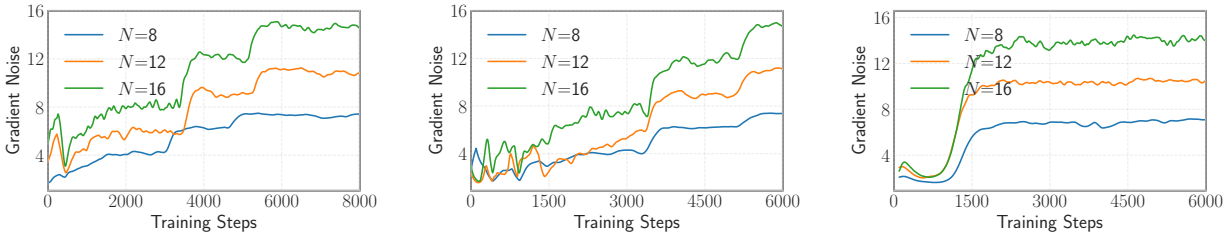
`Scavenger` builds a performance model to capture both parallel and statistical efficiency associated with horizontal and vertical scaling, and provides accurate estimates of training time for different (N, B) to estimate the optimal training configuration in the cloud.

4.3 `Scavenger` Design

From a bird's eye view, `Scavenger` tries to find the "ideal" configuration of a training job in an online manner with nominal profiling and minimal information about the DL model. Our end goal is to minimize the time and cost of training DNNs in the cloud. We build analytical performance models to estimate and construct the time-cost curves with different (N, B), and select the "right" configuration based on user preferences and constraints. However, predicting training time to reach a specific model accuracy is especially challenging due to statistical inefficiency in distributed SGD. To address this, we study and investigate general SGD noise indicators that serve as a proxy of statistical efficiency. With this, we develop a statistical performance model and combine it with a parallel performance model to estimate time and cost of training with different cloud configurations.

4.3.1 SGD Noise as a Scaling Indicator

In §4.2, we observed that simply adding more resources to a distributed training job does not proportionately reduce its training time and may lead to escalating execution costs in the cloud. We thus need appropriate scaling indicators to measure if adding more resources would be beneficial or inane. For classic parallel applications, communication or synchronization overheads typically serve as such scaling indicators, where we scale up or down if the scaling efficiency changes beyond a certain threshold. Scaling laws like Amdahl's [188] and Gustafson's [150] use this to measure the performance and scaling properties of an application. Although synchronization overheads are applicable in distributed training as well, it alone is not sufficient due to the statistical efficiency aspect of DL. We seek a general indicator of statistical efficiency that is independent of model and execution environment (such as number and type of workers). For instance, such a metric could indicate



(a) ResNet18 (b) ResNet50 (c) Transformer-Base

Figure 4.5: Gradient noise requires some iterations to stabilize in the early training phase, then eventually saturates to a value corresponding to cluster-size 'N'.

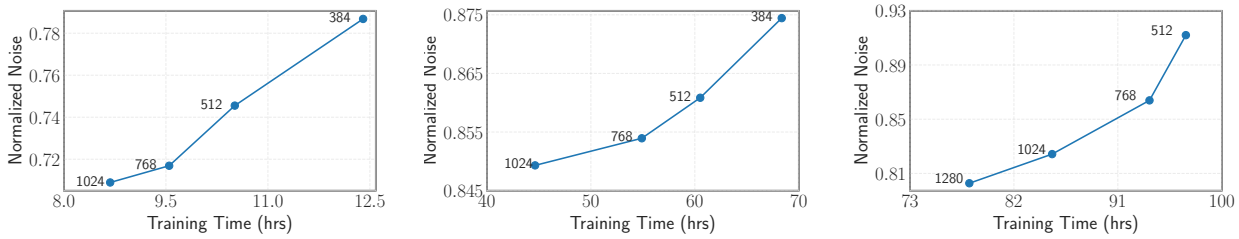
the batch-size threshold for a given 'N', beyond which scaling the application does not significantly reduce training time.

$$\text{Gradient Noise}_{(i)} = \frac{\mathbb{E}[\frac{1}{N}\sum_{n=1}^{N}||g_i^{(n)}||^2]}{\mathbb{E}[||\tilde{g}_i||^2]} \tag{4.1}$$

Statistical inefficiency arises from noise in the gradients computed by workers. From Eqn. (4.1), SGD noise can be captured by gradients variance, where $g_i^{(n)}$ is the gradient tensor computed on worker 'n' at iteration 'i', $\tilde{g}_i$ is the aggregated update across 'N' workers of the cluster, and noise is calculated by the mean of workers' local gradient norms and expected norm of the averaged gradients. Gradient variance has previously been investigated to understand batch-size [40] or cluster-size scaling [167], while we generalize it to both horizontal and vertical scaling.

Another advantage of SGD noise is that it can easily be integrated into the backpropagation routine. As a measure of divergence between the calculated and "true" gradients, it is computed by looking at the prior and post-aggregation gradients in synchronous training. The idea is that local gradients are computed over smaller batches on a worker, while aggregated gradients are more accurate as they are effectively calculated from larger batches in BSP (i.e., 'b' vs. 'Nb' samples per-iteration respectively). From Fig. (4.5), noise is low in the early training stages since variance in the gradients is on the same scale as the gradient itself. As DNNs converge, aggregated updates become smaller, thus increasing gradient noise before saturating to cluster-size 'N'.

As seen from Fig. (4.5), SGD noise does not stay constant throughout training, even with a static job configuration. This is a fundamental artifact of SGD-based optimization in DL. Additionally, gradient noise is affected by model hyperparameters such as learning-rate, regularization, order of



(a) ResNet18 (b) ResNet50 (c) Transformer-Base

Figure 4.6: Gradient noise (normalized by cluster-size 'N') affects the overall training time at different 'B' in ResNet18, ResNet50 and Transformer-Base trained to 80%, 90% accuracy and 18.0 BLEU score respectively.

optimization, etc. Although we want to select the optimal job configuration as soon as possible, noise in the early training epochs is volatile and unstable, so we let it stabilize before using it as a scalability indicator. `Scavenger` thus begins its exploration process for potential (N, B) configurations only after the noise stabilizes. Although this increases the overall profiling and search time, it provides reliable noise estimates for an accurate statistical performance model.

Fig. (4.6) shows end-to-end training time to desired accuracy level or BLEU score [189] on different batch-sizes across various DNNs. We normalize SGD noise by 'N' since we want to compare it across different cluster-sizes for statistical efficiency. We see an increase in gradient noise at smaller batches result in a higher training time. Another key observation is that noise saturates and does not significantly reduce any further beyond a certain batch-size, as seen from batches 768 and 1024 in ResNet18 and ResNet50. This is because gradients at larger batches are approximately the same and thus, increasing B does not considerably improve the quality of gradients and thus, may not necessarily accelerate training from a statistical standpoint (while negatively impacting parallel scaling).

4.3.2 Performance and Cost Model

$$T = t_{step} \times I \tag{4.2a}$$

$$I = \frac{ED}{B} \tag{4.2b}$$

$$C = T \times N \times p \tag{4.2c}$$

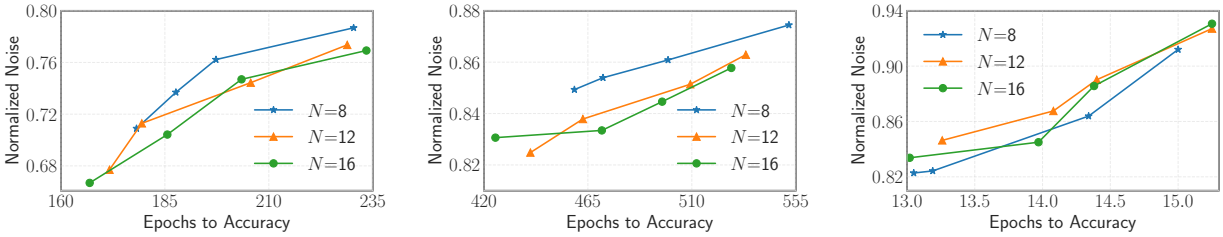
To develop an analytical model that predicts time-cost trade-off curves (like Fig. (4.1)) and guides cloud resource allocation policies, we use statistical and parallel scaling indicators in an online manner (i.e., without any prior profiling). The performance of a job depends on its batch and cluster-size (N, B), and `Scavenger`'s analytical model predicts the total training time 'T' of each configuration as per Eqn. (4.2a), where 'I' iterations are required to converge on a target metric for a given (N, B), and t_{step} is the per-iteration time.

Further, iterations 'I' depends on the total number of epochs 'E', size of the training dataset 'D' and global batch-size 'B' (Eqn. (4.2b)). E is the key unknown which depends on a DNNs' size and complexity, desired accuracy-level and its statistical efficiency. Given fixed resources, the per-iteration cost t_{step} stays the same since each iteration is identical from a computational standpoint. This is because the same amount of gradient computation and synchronization is performed over a mini-batch for every iteration. Last, the total training cost 'C' is simply the product of total training time, cluster-size 'N' corresponding to the total VMs allocated, and per-VM price 'p'. We estimate the number of epochs required for convergence with the statistical model, and predict the iteration costs with the parallel model.

Search and Profiling Techniques: Before constructing the trade-off curves in distributed cloud training, we run a search or exploration phase where we briefly deploy the job on a configuration, measure the corresponding parallel and statistical scaling metrics, and estimate time-cost based on Eqn. (4.2). `Scavenger`'s search cost involves running the job over a candidate (N, B), compute relevant metrics, restore model state via checkpointing, and then proceed to next configuration in the grid search. We refer to it as "*full*" or "*offline*" search since we first explore the configuration space and then run the job on the "best" available option.

To mitigate search cost and build time-cost estimates, we can also profile a small sample of (N, B) configurations, and build the rest of trade-off curves by fitting the learned performance models. This "*partial*" or "*online*" search significantly reduces the search cost, but may differ from offline search as the estimates from interpolation may be prone to a certain margin of error.

Last, we observed that many training jobs deployed in the cloud are similar and deployed as part of hyperparameter tuning, neural architecture search, etc. For e.g., hyperparameter tuning may involve dozens of jobs with the same model, but different activation functions, weight decay factor,



(a) ResNet18 (b) ResNet50 (c) Transformer-Base

Figure 4.7: Total epochs required to reach the same convergence target (80% in ResNet18, 90% in ResNet50 and 18.0 BLEU in Transformer) is *near-linear* with respect to the normalized SGD noise (plotted with different B for a given N).

etc. Thus, once the performance model of a job is learned, it can be stored and reused when a similar model configuration is trained in the future. We avoid the exploratory phase entirely in "*no-search*" scenario. In `Scavenger`, we develop *full*, *partial* and *no-search* techniques for both parallel and statistical performance models.

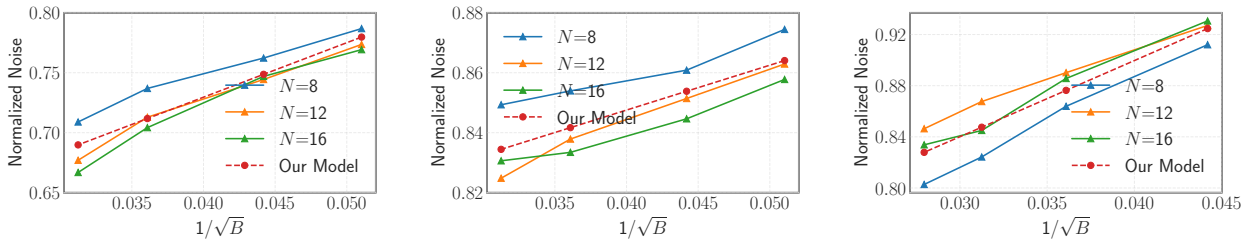
4.3.3 Statistical Performance Model

The gradient noise scaling indicator allows us to model statistical performance and estimate the epochs needed for convergence. SGD noise increases the amount of work required for convergence, i.e., total number of iterations or epochs. For any global batch-size 'B', epochs 'E' is directly proportional to gradient noise (Eqn. (4.3)). We see the empirical support for this in Fig. (4.7) which shows normalized noise vs. epochs taken to reach a specific test metric with different cluster and batch-size configurations.

$$E \propto noise_{(N,B)} \tag{4.3}$$

For a given 'N', each point corresponds to a particular value of 'B'. The linear model can be understood in relation to full gradient descent devoid of any noise where gradients are computed over the entire training data. If full gradient descent converges in epochs E^*, then batch-size 'B' would converge in $E = (E^* + \theta \times noise_{(N,B)})$ epochs, where θ is an unknown linear-model parameter that relates noise to statistical efficiency.

Fully Offline Search: Here, we profile DNNs on every (N, B) configuration and measure its corresponding noise. Both E^* and θ are properties of the model which are unaffected by its cluster



(a) ResNet18 (b) ResNet50 (c) Transformer-Base

Figure 4.8: Gradient noise for each (N, B) configuration to train ResNet18 to 80%, ResNet50 to 90% accuracy and Transformer-Base to 18.0 BLEU score. Normalized noise is not highly sensitive to cluster-size, and our averaged noise model can estimate noise for any (N, B) with relatively low error.

configuration, and can easily be calculated from prior profiling as shown in Fig. (4.7). We see that the epochs needed for convergence across DNNs is not sensitive to 'N'. It is a static property of a model and its training routine, and not influenced by job configuration parameters. Thus, if we have access to any prior executed configuration, we could theoretically estimate the epochs required reach the same model quality for any other configuration. In `Scavenger`, we only need to compare "relative" performance between different configurations for which we do not need prior profiling, and compare the epochs of different configurations based on their observed noise.

Partial Search: Instead of modeling noise and epochs as a linear relationship, we can model statistical efficiency using partial search that avoids exploring the entire configuration space. Eqn. (4.4) inversely relates gradient noise to batch-size, with the intuition that larger batches correspond to lower noise, while gradients computed on smaller batches are considerably noisier [40].

$$noise_{(N,B)} \propto \frac{1}{\sqrt{B}} \tag{4.4}$$

This allows us to estimate noise on different batch-sizes by profiling a subset of batches in the search space, measure the noise and build a linear model of noise and $1/\sqrt{B}$. The model can then be used to interpolate noise for unseen values of B. Fig. (4.8) illustrates the near-linear relationship between noise and $1/\sqrt{B}$.

Partial search is performed by running the job briefly on the extremums of B, i.e., the smallest and largest batch-size specified by the user, and then fitting data as per Eqn. (4.4). We could repeat this process for different values of N and improve the accuracy of our statistical model. However,

as seen in Fig. (4.8), the relation between SGD noise and B is not highly sensitive to cluster-size N. Thus, we can further simplify the statistical efficiency model and avoid cluster-size specific profiling by using an aggregated statistical model where we average the estimated noise for various cluster-sizes. This can be seen in Fig. (4.8) as the dashed line across DNNs.

No-search: If the same or a similar job is executed repeatedly, we can leverage its noise vs. batch-size relation without any additional profiling to model its statistical performance.

4.3.4 Parallel Performance Model

$$t_{compute} \propto b \tag{4.5a}$$

$$t_{sync} \propto N \tag{4.5b}$$

In this section, we build a parallel performance model to estimate the iteration costs by leveraging the repetitive nature of DL training where the computational performance characteristics of each iteration is nearly identical. This allows us to continue using the profiling-based search strategy. *Full-search* strategy runs each of the (N, B) configurations for a few iterations and logs down the corresponding parallel performance metrics.

In *partial-search*, we use a similar interpolation approach as described in the previous section. Assuming low I/O overhead, a single iteration in distributed synchronous training entails loss calculation, gradient computation and synchronization of model updates. From Eqn. (4.5a), gradient computation time increases proportionally with worker mini-batch size 'b'. For PS-based synchronous training, the communication or synchronization cost increases proportionally with cluster-size in a star-topology, while cost of decentralized communication collectives like ring/tree AR are nearly independent of 'N' [187]. With these, we can model the iteration cost by profiling the extreme points in the (N, B) configuration space and fit linear models to infer the computation and communication time for any unknown configuration. In *no-search*, repeated executions of the same job configuration return the actual performance metrics, avoiding the search phase altogether.

4.3.5 Resource Allocation Policies

`Scavenger` combines parallel and statistical performance models for job configuration and resource allocation policies in the cloud. Depending on prior information available, the optimal search strategy and cost may differ. Using profiling and modeling, we construct the time-cost trade-off curves for various (N, B) configurations. Once the trade-off curves are built, the "right" job configuration depends on a user's objectives and constraints. In `Scavenger`, we support optimizing for time, cost or a knee-point that selects the inflection point in the trade-off curves using the kneedle algorithm [190].

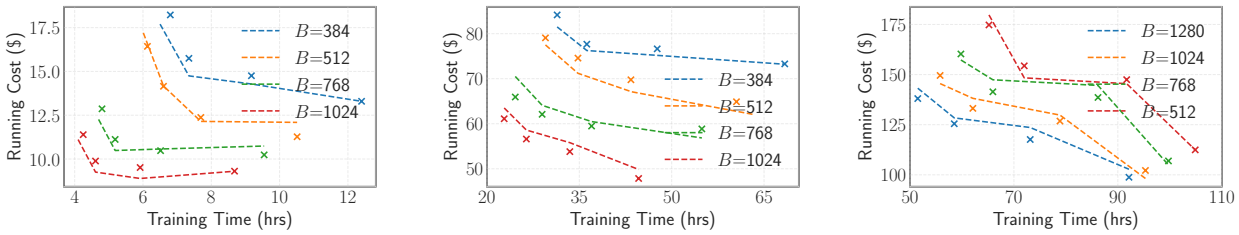
The maximum cost and time is bounded within the search space by limiting the number of workers or VMs to be allocated for a job, i.e., N_{min} and N_{max}. The bounds on the batch-size, B_{min} and B_{max} limit the maximum amount of noise in gradient updates (with its lower bound) or the min-max memory size of a VM (determined by the upper bound B_{max}).

4.4 Experimental Evaluation

`Scavenger` is implemented as a modular extension to TensorFlow v1.5 [97], where the training scalability indicators are built atop TensorFlow's Estimator framework [191]. Synchronous distributed training on general purpose CPUs is implemented via PS strategy [49] where a central node aggregates the gradients and computes SGD noise. Gradients can be fluctuating so we apply EWMA smoothing over the inter-iteration updates.

All the scalability indicators: gradient noise, worker computation and synchronization time are sent to an external model service at each iteration. The service uses these metrics to update the performance model if operating in the initial exploratory phase. The user selects the full or partial search mode based on their search-cost and performance-model's prediction accuracy requirements. By default, we use partial search since its results were comparable to full-search with much lower search costs. `Scavenger` saves all performance models on persistent storage, and no-search strategy is used if a DL model has been trained before. Once the trade-off curves are built, we choose the appropriate configuration and stop all profiling.

We interface `Scavenger` with standard cloud APIs to manage VMs for training. The partial-search approach starts with the smallest (N, B) configuration and adds VMs subsequently to reach



(a) ResNet18 (b) ResNet50 (c) Transformer-Base

Figure 4.9: Cost-Time trade-offs of different DNNs to reach the same convergence quality with different job configurations on Google E2-standard-4 VMs. We show cost-time pareto-frontiers for a fixed 'B' across decreasing cluster-sizes (from left-to-right): [20, 16, 12, 8]. Dashed line shows predicted cost of `Scavenger`'s full-search performance model.

the largest configuration. We use lightweight checkpointing where new workers in an updated configuration can pull the latest parameters from the PS and resume training. We switch to different configurations only on iteration boundaries and reuse existing machines to avoid excessive VM churns and startup-shutdown overheads.

We evaluate three popular DNNs: skip-connection based ResNet18/50 and attention-based Transformer, across different VM size and price configurations over Google Cloud [69]. We demonstrate the efficacy of SGD noise as an indicator of statistical efficiency, show the accuracy of our performance and cost model across different job configurations, investigate the performance and cost trade-offs of different cloud computing cost models, and overall time-cost savings with our job configuration and resource allocation policies.

While DL is faster over GPUs, we evaluate `Scavenger` over CPU VMs. GPUs simply reduce the iteration costs (i.e., parallel performance), and all aspects of `Scavenger` such as the analytical model and service are unaffected by the underlying hardware parallelism. Standard CPU VMs can also be sized in a fine-grained manner and we can configure them with arbitrary amounts of CPU and memory. In contrast, GPUs have fixed and limited memory, and not all classes of GPUs can be virtualized [192]. Additionally, we only consider worker cost and assume sufficient PS nodes are available. PS allocation tackled by other systems like Optimus [164] is orthogonal and complementary to our work.

4.4.1 Cost and Time Trade-Offs

From §4.3.2, we can predict the running cost and time in distributed training across different cluster and batch-size configurations. Fig. (4.9) shows the cost vs. time trade-offs in ResNet18 and ResNet50 to reach 80% and 90% accuracy, and Transformer-Base to converge to 18.0 BLEU score. For a given 'B', we spawn 8, 12, 16 and 20 Google E2-standard-4 VMs for distributed training, along with an additional PS node. Each scatter point shows the results from full runs on each (N, B) configuration, while the dashed line shows the predicted time and cost with the offline performance model. The rental cost of each allocated VM is fixed, about $0.13402 per-hour. Going from left to right, the points on each 'B' curve represent decreasing cluster-size, going from 20, 16, 12 to 8 workers respectively.

In Fig. (4.9), we clearly see the cost vs. time trade-offs for a given 'B'. The per-worker compute hardware is the same in this case, and cost of compute is proportional to 'N' under this pricing model. Thus, training on larger clusters costs more, but also have the lowest training times. Decreasing the worker count may lower the cost slightly, but may considerably increase the running time.

Both ResNet18 and ResNet50 have a single inflection or knee-point for each 'B' curve, after which we see diminishing returns on cost (Fig. (4.9a) and (4.9b)). For ResNet18 at batch-size 384, (16, 384) represents the optimal (N, B) configuration as it corresponds to the knee-point for that batch-size. With B=512, inflection point occurs at N=12 so that's the ideal configuration at that batch-size. For Transformer in Fig. (4.9c), we observed two inflection points at cluster-size 12 and 16 at any batch-size. For this model, we observed a notable decrease in per-iteration cost from N=12 to 16 since for the same hardware and global batch-size, the larger N=16 results in a smaller mini-batch on each worker. For e.g., B=768 changes the local batch-size from 64 to 48 when cluster-size increased from 12 to 16. We thus see a considerable difference in running times between N=12 and 16, resulting in two distinct inflection points.

The trade-off curves can be a crucial tool for judicious resource allocation in the cloud running distributed training jobs. The dashed line in Fig. (4.9) shows the cost predicted by our analytical model using full-search strategy, which relies on profiling different (N, B) configurations to estimate gradient noise and iteration time. Compared to the actual job running time, our offline

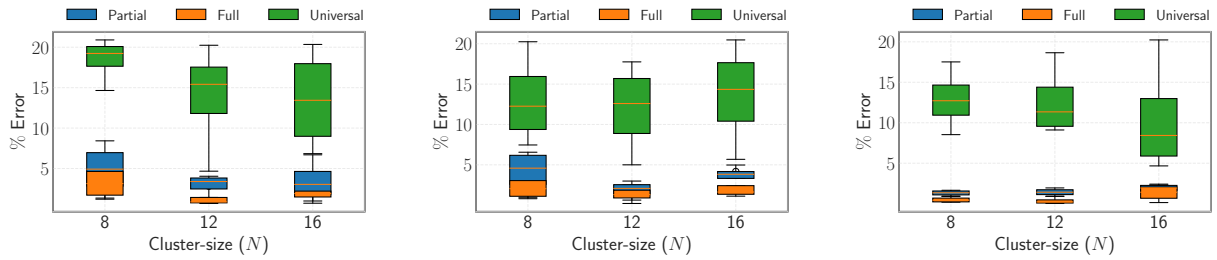
performance model reported an error of only 1-5% across the three DNNs, and various cluster and batch-sizes.

4.4.2 Performance of Partial Search Strategy

We now see the performance of partial search strategy to construct cost-time pareto-fronts by profiling only a few configurations. We use the profile logs to construct the linear models to estimate time and cost of running different job configurations. In our evaluation, we set $8 \leq N \leq 20$ and $384 \leq B \leq 1024$ for residual networks, while we set min-max batches to 512 and 1280 in case of Transformers. We increment the cluster-size by 4 to compare results with offline approach (from Fig. (4.9) that uses N = [8, 12, 16, 20]). The initial configurations deployed with partial search are (N_{min}, B_{min}) and (N_{min}, B_{max}), until gradient noise has stabilized. With exponential smoothing, SGD noise for ResNet18, ResNet50 and Transformer-Base stabilized at *2000*, *3000* and *10000* iterations respectively. The overhead from profiling on these extreme configurations with checkpoint-restore approach was minimal, about *37* seconds for ResNet18, *40* for ResNet50 and *127* seconds for Transformer. Each job configuration with partial search was deployed for around 20 iterations, which took around *17-35* sec. across the three DNNs. Compared to an "oracle" scenario of running on the optimal configuration all along (i.e., bypassing the search phase altogether), our approach increases the running time by *0.83* hours and adds *$0.89* to the final cost. This represents roughly 13% increase in running time and 9% increase in total costs over the oracle approach. Compared to an arbitrary job configuration, `Scavenger` can reduce training time by a factor of $2\times$ and reduces cost by up to 40%.

4.4.3 Accuracy of Analytical Performance Model

As seen from Fig. (4.10), both full and partial-search are able to accurately predict the total training time. We evaluate three configurations: full-search (orange), partial-search (blue) and a worst-case no-search strategy that does not consider any model-specific information (shown in green). The plot shows the error distribution between the predicted running time vs. the empirical job running time across different (N, B) configurations. The average error for partial-search was about 4% for residual models and less than 2% for Transformer. The error rates of full-search were even lower, between 0.5-3.5% across the three DNNs. The no-search strategy has the same performance as



(a) ResNet18 (b) ResNet50 (c) Transformer-Base

Figure 4.10: Error in the predicted vs. actual training time across all job configurations. The error of full-search strategy is the lowest, followed by partial-search. Even a universal model that does not consider any model-specific details gives modest results.

full-search if an identical model configuration has been deployed before. However, we construct a scenario where a globally averaged model is used that averages the statistical and parallel performance across all three DNNs, referred to as the universal performance model. Even a naive approach such as this predicts job running time within an error range of 4-20%. Such a universal model does not require any search or suffers its associated costs, and can thus be used in a fully online manner with zero overhead.

The exact prediction of running time is not highly critical to our overall objective of discovering the optimal configuration. We are primarily interested in the relative running times since we profile configurations over a brief period and finally deploy the job on the best predicted configuration. It is likely for the best predicted configuration to stay the same even if the prediction error was higher, or the sub-optimal configuration chosen due to errors is very close to the optimal configuration in the trade-off curve.

4.5 Conclusion

Our work falls under the category of adaptive DL on distributed infrastructure such as cloud and shared clusters [193]. `Scavenger` uses noise scale proposed in AdaScale [167], which is similar to the gradient noise or gain proposed in [40, 168, 169]. Other works use SGD noise to monitor training performance and dynamically adjust resource allocation. We further extend these elastic and adaptation mechanisms by adding profiling that also takes training cost into account.

Commercial offerings of 'model-as-a-service' (MaaS) such as AWS Sagemaker [194] have used simple performance models, and did not consider statistical efficiency or pareto-optimal allocation.

Searching for optimal model hyperparameters is an important cloud workload, and reducing this search cost using parallel search techniques and early stopping can provide significant time and cost savings [166, 195]. But unlike hyperparameter optimization which focuses only on a "bag of jobs", `Scavenger` focuses on optimizing time-cost of a "single" job. Efficient elasticity mechanisms and policies for machine learning [196, 197] can also be easily incorporated into `Scavenger`.

Scheduling and resource allocation in shared clusters is challenging for distributed training jobs due to complex performance trade-offs and large-scale computing needs. Although resource contention is not an issue in the cloud, but cost optimization is indeed crucial. `Scavenger` is a cloud service that uses online profiling and novel performance models for estimating the training performance of different cloud configurations, with accuracy as high as 95%, and reduces training time by up to $2\times$ over a naive configuration.

4.6 Related Publications

1. **Tyagi, Sahil** and Sharma, Prateek. "Scavenger: A Cloud Service For Optimizing Cost and Performance of ML Training." 2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid) (2023): 403-413.

2. **Tyagi, Sahil**. Poster "Scavenger: A Cloud Service for Optimizing Cost and Performance of DL Training." 2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW) (2023): 349-350.

CHAPTER 5

ADAPTIVE GRADIENT COMPRESSION IN DISTRIBUTED LEARNING

5.1 Introduction

The exponential growth in the size of sota DNNs as well as the massive datasets used to train them [198] necessitate distributed execution of training. DDL techniques increase training throughput over multiple machines to train a globally shared model. Synchronous data-parallelism aggregates local model updates at the end of each iteration, so per-iteration communication cost increases with the growing size of DNNs. While the size and computational requirements of sota models doubles between 3.5 to 6 months [12, 13], corresponding gains in chip design and improvements in telecommunications grow every 24 and 18 months on-average respectively [199, 200]. As a result, the infrastructure needed to train DNNs often falls behind their computing and networking demands. The latter is especially challenging since upgrading the network stack in the cloud or HPC clusters can be infrequent compared to appending new accelerators (to grow compute) in pre-existing systems. Thus, communication tends to be a significant bottleneck in DDL [201].

Although a multitude of gradient compression techniques have been proposed in recent years (as described in §2.8), the "ideal" *compression factor (CF)* that gives maximum communication savings while attaining high convergence quality requires careful tuning for each learning task. A small CF may not lower the communication cost significantly while a high CF may considerably degrade the final model quality compared to *uncompressed* or *DenseSGD* training (since more gradient information is lost at high CFs). Additionally, the optimal CF depends on a model itself (its architecture, size, structure and depth), network bandwidth and the overhead of compression itself. As model sensitivity varies over the course of training [27, 28], the statistical impact of compression varies as well. Thus, gradient compression in DDL has a parallel and statistical efficiency aspect associated with it; a high CF improves training throughput by reducing communication cost, but increases gradient information loss resulting in slower or insignificant updates, and vice versa. The two metrics are inversely related as one improves at the detriment of the other. In this chapter, we introduce `GraVAC`, an adaptive technique to dynamically adjust the degree of compression or CF

Model	Layers	Size (MB)	Dataset	Test target
ResNet101	101	170	CIFAR10	80% Top-1 accuracy
VGG16	16	528	CIFAR100	90% Top-5 accuracy
LSTM	2	252	Penn Treebank (PTB)	22.0 perplexity

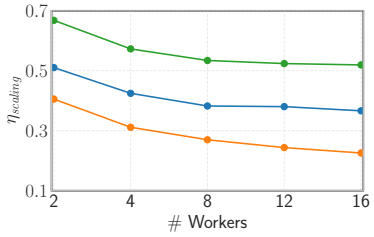
Table 5.1: DNNs, datasets and the corresponding convergence targets used.

by comparing information loss between prior and post-compression gradients. `GraVAC` evaluates various CFs in a given search space and selects the one that best balances the parallel and statistical efficiency. We test `GraVAC` over a variety of DNNs and compare with various other static and adaptive compression techniques.

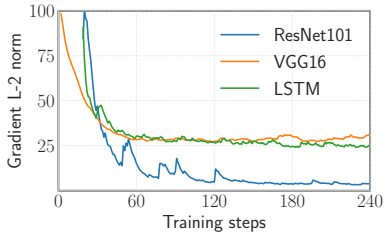
5.2 Motivation

As described in §2.5, parallel scaling in distributed training is bounded by communication and I/O overheads. Assuming the latter to be negligible, the 'scaling efficiency' $\eta_{scaling}$ is mainly constrained by its communication cost, i.e., $\eta_{scaling} = \frac{T_N}{N \cdot T_1}$. Here, T_N is the actual throughput of a distributed application across 'N' workers, while the *ideal* throughput is N times the throughput of a single worker T_1 (assuming homogeneous workers). In an ideal setting, $\eta_{scaling} = 1$ irrespective of N. To measure the scaling efficiency of real-world applications, we train DNNs like ResNet101 [9], VGG16 [8] and Long Short-Term Memory (LSTM) [202] over different datasets and convergence targets (as shown in Table (5.1)). For each model, we first measure the overhead of inter-node communication latency and its effect on the overall scaling efficiency. Fig. (5.1a) shows how $\eta_{scaling}$ decreases as cluster-size increases for nodes communicating over a 2.5Gbps Network Interface Card (NIC). In addition to the number of workers, scaling is also influenced by the size of model updates to be aggregated; VGG16 is the largest model in the set and has the lowest $\eta_{scaling}$. It is also affected by the tensor-size distribution across model layers as well. For e.g., LSTM has a better $\eta_{scaling}$ than ResNet101 despite being a larger model, as listed in Table (5.1). This is because we are communicating updates one layer at a time, and the parameters in LSTM are spread only over 2 layers, compared to 101 in ResNet101.

In addition to parallel scaling, we previously discussed the statistical efficiency aspect of DNNs where gradient sensitivity varies over the course of training. Fig. (5.1b) shows training sensitivity in the early phase [149, 27] of the aforementioned models where we plot gradient $\mathbb{L}$-2 norm over

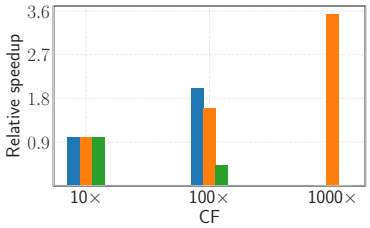


(a) Scaling efficiency

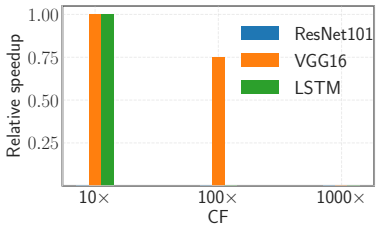


(b) Initial training sensitivity

Figure 5.1: (a) Inter-node scaling limited by communication overhead over a 2.5Gbps ethernet interface (b) Gradient sensitivity in the early training phase.



(a) Top-*k* compression



(b) Random-*k* compression

Figure 5.2: Training speedup of various CFs relative to 10× for each model over a given compressor. A speedup of 0 implies failure to converge to Table (5.1) convergence targets. The CF with maximal speedup varies for each model and compression technique used.

the initial iterations and see how updates change and saturate over training.

Although lossy gradient compression techniques improve scaling efficiency in distributed learning, they incur a fundamental trade-off between communication cost and information loss; we can reduce synchronization overhead by reducing the size of gradient updates and losing more information (i.e., by compressing more), or preserve data quality by keeping most of the original bits intact (i.e., compressing less). In the context of DDL, higher CF generally reduces communication time at the cost of accuracy degradation, or takes more iterations/epochs to reach the same convergence target. For instance, compressing gradients to 1% volume to attain 100× CF would accelerate training, but may attain lower final accuracy compared to DenseSGD.

What should be the ideal CF for gradient compression in deep learning?

The "ideal" CF is one that compresses enough to considerably minimize the communication cost, but avoids trimming excessive gradient information which may degrade the final model quality.

Additionally, compression has its own associated costs depending on target CF and computational complexity of the compression-mechanism itself. These factors thus affect both parallel and statistical efficiency, the latter due to information loss on account of compression. Fig. (5.2) aptly demonstrates this trade-off between parallel and statistical efficiency where the CF that offers maximum speedup to successfully meet the convergence target varies across DNNs for different compressors like Top-k and Random-k [111]. With Top-k, ResNet101 achieves maximal speedup at CF 100×, while VGG16 and LSTM attain peak parallel performance at CFs 1000× and 10× respectively. With Random-k, ResNet101 fails to converge at any CF, while VGG16 and LSTM converged to maximal speedup at 10×. Although the typical ML practitioner may not necessarily need to consider a plethora of compressors, choosing the right CF and configuration that minimizes training time or even converges successfully is non-trivial.

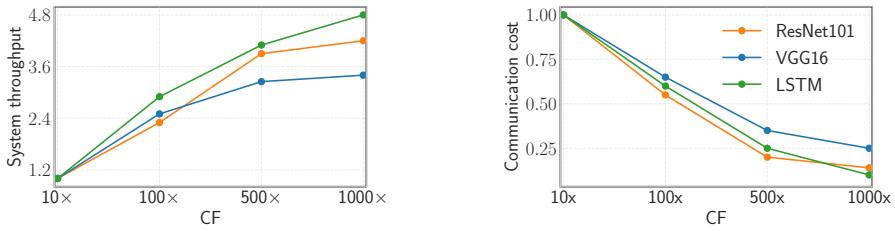
Previously, dynamic compression techniques like AdaQS [203] performed quantization using gradient mean-to-standard deviation ratio (MSR), while Accordion [131] switched between low and high CF based on critical regime detection. In `GraVAC`, we tackle the ideal CF exploration problem in a gradient-driven manner by comparing information loss between prior and post-compression gradients to dynamically adjust CF over each iteration. Here, prior compression refers to the original gradient tensors computed in backpropagation, while post-compression implies compressed tensors. Starting with a low CF initially, we gradually increase compression as training progresses. During volatile periods, `GraVAC` switches to a lower CF value that least degrades convergence.

5.3 `GraVAC` Design

In this section, we further explain the trade-offs between parallel and statistical efficiency in distributed training in the context of gradient compression. We then combine the two using metrics like 'compression gain' and 'compression throughput' to propose an adaptive compression regime.

5.3.1 Parallel Efficiency in Gradient Compression

Parallel efficiency is contingent upon the collective and communication library used (like MPI [24], NCCL [56], Gloo [57]), cluster-size and message-size (i.e., model-size). Improvement in network bandwidth improves communication, but to only a certain extent, as shown in [201] where ResNet50



(a) Relative system throughput

(b) Relative communication cost

Figure 5.3: Relative to CF 10× in DGC compression, throughput increases and communication time decreases as we increase CF up to a certain point, then begins to saturate.

peaked to 75% scaling efficiency on a 25Gbps NIC and failed to improve any further, even at 100Gbps bandwidth. Even though cloud providers like Google Cloud [69] can provide 10-32Gbps bandwidth depending on machine-type and VM size, they may not be utilized to their full potential. Assuming negligible I/O overheads, the iteration cost of gradient compression at CF 'c' in Eqn. (2.7) approximates as:

$$t_{step}^{(c)} \approx t_{compute} + t_{sync}^{(c)} + t_{comp-decomp}^{(c)}$$

where the latter two terms are the synchronization and compression-decompression overhead for a specific CF and compression technique. Compression overhead, denoted by $t_{comp-decomp}^{(c)}$ must be as low as possible for a compressor to be viable.

Fig. (5.3) shows how the communication cost and training throughput reduces only up to a certain point even with compression. The DNNs are trained with layerwise Deep Gradient Compression (DGC) [115] over a 32 GPU cluster, and results are normalized with respect to CF 10×. Overall system throughput is influenced by compression overhead and communication time, since the time $t_{compute}$ of backpropagation is fixed across all CFs. Based on the compression technique, the latency of compression and decompression operation may vary with the target CF. For e.g., $t_{comp-decomp}^{(c)}$ for $c = 100/k$ decreases with larger CF in Top-k using max-heap to sort the top k% across M elements in $\mathcal{O}(M + k \log M)$ time. The throughput of ResNet101 and VGG16 saturates at CF 500× and plateaus thereafter, while LSTM increases even at 1000× (from Fig. (5.3a)). Communication savings also diminish at higher CFs due to smaller message size in compressed updates and dominating latency costs, as seen in Fig. (5.3b).

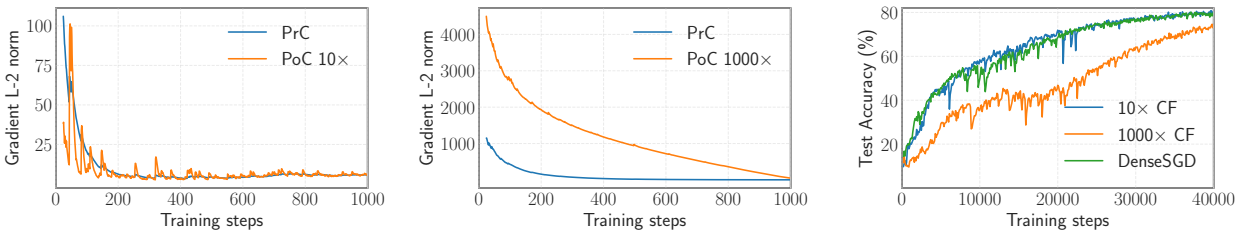
5.3.2 Statistical Efficiency in Gradient Compression

As explained in §2.8, gradient compression relies on error-feedback in the form of delayed updates. Residual gradients in error-feedback help preserve model features and attain good accuracy [112, 115], but can sometimes degrade model generalizability on account of stale updates. In spite of error-feedback, training at high CFs can exacerbate overall training time, convergence quality, or both if the compressed updates fail to update the model in any useful way. *It is thus crucial to have a training indicator that quantifies information loss between compressed and original gradients*. We do so by comparing the variance of compressed and original gradient tensors on every iteration and see how it correlates to actual model convergence. Denoting original gradients computed in backpropagation as "*Prior-Compression (PrC)*" and the compressed, error-fed tensors as "*Post-Compression (PoC)*", we compare PrC and PoC tensors in two separate configurations at CFs 10× and 1000× for each model in Fig. (5.4). We compare the convergence curves for the two CFs with DenseSGD to see how much the convergence quality degrades with compression.

For ResNet101, compressed gradients PoC at 10× are nearly identical to the original gradients PrC (in Fig. (5.4a)), while there is considerably more information loss between PrC and PoC gradients at 1000×, as seen from Fig. (5.4b). This information divergence between prior and post-compression gradients translates to their convergence curves as well in Fig. (5.4c), where CF 10× and DenseSGD have similar accuracy curves while 1000× converges to lower accuracy in the same iterations. VGG16 follows a similar trend with CF 10× where PrC and PoC gradient variance is nearly identical (Fig. (5.4d)) and so are the convergence curves of 10× and DenseSGD in Fig. (5.4f). We observe a slight deviation between PrC and PoC 1000× initially in Fig. (5.4e), which also corresponds to slow convergence in the early iterations of 1000×. As models converge and gradients get smaller, the deviation between PrC and PoC gradients decreases for both CFs and they converge to nearly the same accuracy levels as DenseSGD in Fig. (5.4f).

As seen in Fig. (5.4g) and (5.4h) of LSTM, PoC 10× and 1000× gradients lie on similar scales as their respective PrC gradients, with CF 1000× having slightly higher deviation from its PrC counterpart. From Fig. (5.4i), DenseSGD has the least perplexity (and thus, better model quality), followed by CF 10× and 1000× respectively.

To compare information loss between PrC and PoC gradients compressed to CF 'c', we propose



(a) ResNet101 10×

(b) ResNet101 1000×

(c) ResNet101 convergence

(d) VGG16 10×

(e) VGG16 1000×

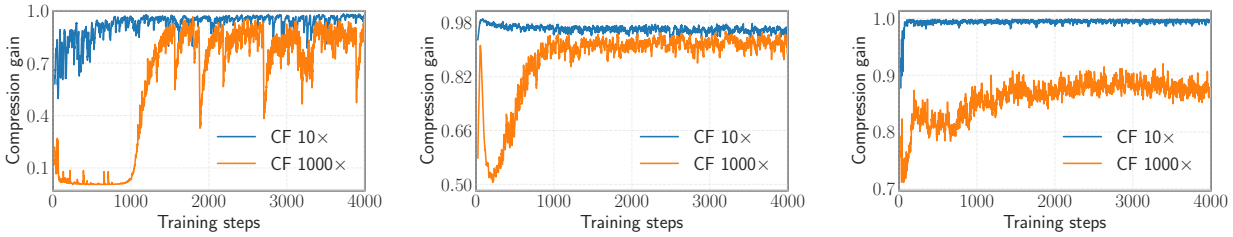
(f) VGG16 convergence

(g) LSTM 10×

(h) LSTM 1000×

(i) LSTM convergence

Figure 5.4: Comparing prior and post compression gradients at CF 10× and 1000×, as well as their respective convergence curves along with DenseSGD.



(a) ResNet101 (b) VGG16 (c) LSTM

Figure 5.5: Compression gain over the iterations at CFs 10× and 1000×. Gain is low initially when the model is volatile, but gradually rises.

a simplistic metric called '*Compression gain*'. As per error-feedback from Eqn. (2.8), gradients are updated such that $g_{ef}^{(i)} = g_{o}^{(i)} + \texttt{residual}^{(i-1)}$, where $g_{o}^{(i)}$ are the gradients computed at iteration 'i' in backpropagation, while $\texttt{residual}^{(i-1)}$ are accumulated from the previous iterations and added back to produce the error-fed gradients $g_{ef}^{(i)}$. With compressor $\mathcal{C}$, gradients are compressed to $g_{c}^{(i)} = \mathcal{C}[g_{ef}^{(i)}]$. Compression gain is then measured as the ratio of the expected norm of compressed gradients and the gradients updated with error-feedback, as shown in Eqn. (5.1).

$$\text{Compression gain} = \frac{\mathbb{E}[||g_{c}^{(i)}||^2]}{\mathbb{E}[||g_{ef}^{(i)}||^2]} \tag{5.1}$$

Thus, `GraVAC` accounts for statistical efficiency by comparing the deviation between original and compressed gradient tensors. As updates computed over each iteration can be noisy, we keep a moving average of the gradients to measure compression gain. The computational and memory footprint of this approach is low since the window-size in EWMA is finite and only a single-precision float corresponding to gradient variance is stored at each iteration. Compression gain is also bounded between $\{0, 1]$, where it is close to 0 when the compressor trims excessive information, and $\approx$ 1.0 when it retains most data. As DNNs converge, gradients get smaller and saturate (i.e., close to 0) and higher compression thus becomes viable in the later stages of training. This is because gain increases in later training phases (up to a maximum value of 1.0) as compressed tensors become more aligned with dense gradients. Intuitively, DenseSGD will always have a compression gain of 1.0 since the hypothetical compression operator in that case is effectively an all-pass filter, making prior and post-compression gradients the same.

For CFs 10× and 1000× runs in Fig. (5.4), we plot the corresponding compression gain using

Eqn. (5.1) in Fig. (5.5). Across all models, 10× has a higher gain than 1000× since more gradient information is preserved at the lower CF. The 10× gain trajectory is close to 1.0 for all DNNs, which interestingly also corresponds to the near-similar convergence curves of 10× and DenseSGD in Fig. (5.4c), (5.4f) and (5.4i). For ResNet101 in Fig. (5.5a), gain of CF 1000× is low initially, then rises and oscillates between 0.4 and 0.8. The low compression gain in the first thousand iterations correlates to the considerable gap between PrC and PoC gradients in Fig. (5.4b) and lower accuracy at CF 1000× in Fig. (5.4c). A larger model like VGG16 is more robust to a high CF of 1000× and achieves nearly the same accuracy as DenseSGD in the end (Fig. (5.4f)). Correspondingly, the compression gain of VGG16 1000× is high as well; up to 0.9 as the model converges. For LSTM in Fig. (5.5c), compression gain for 10× is nearly 1.0, and oscillates between 0.8-0.9 for 1000×. The high gain values of the two CFs is close to its DenseSGD counterpart (with a gain of 1.0), which also correlates to their perplexity curves in Fig. (5.4i). From the results in Fig. (5.4) and (5.5), we thus see how compression gain serves as a viable indicator of statistical efficiency at different CFs in gradient compression, which in turn correlates to its convergence quality.

5.3.3 Fusing Parallel and Statistical Efficiency in Gradient Compression

From §5.2, choosing a CF stochastically may not necessarily yield improvements in training time and attain high model quality at the same time. To account for both parallel and statistical efficiency in gradient compression, we combine '*system throughput*' and '*compression gain*' into a single metric '*Compression Throughput*', given by Eqn. (5.2).

$$T_{compression} = T_{system} \times \text{Compression gain} \tag{5.2}$$

If CF is very high, system throughput will be high as well but the corresponding compression gain would be relatively smaller, thus lowering the resulting $T_{compression}$. On the other hand, gain will be high if CF is low, but the system throughput of this configuration will be smaller due to higher communication overhead. *With Compression Throughput, we capture this pareto-relationship between parallel and statistical efficiency (i.e., system throughput and compression gain) of gradient compression in DDL.*

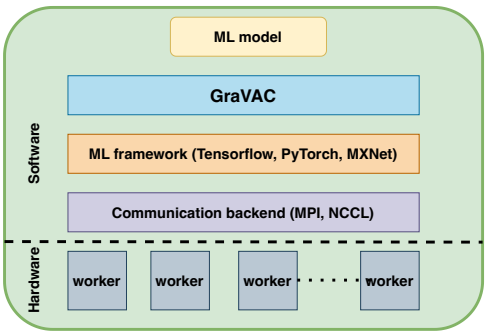


Figure 5.6: GraVAC is implemented between the training framework and DL model.

5.4 Experimental Evaluation

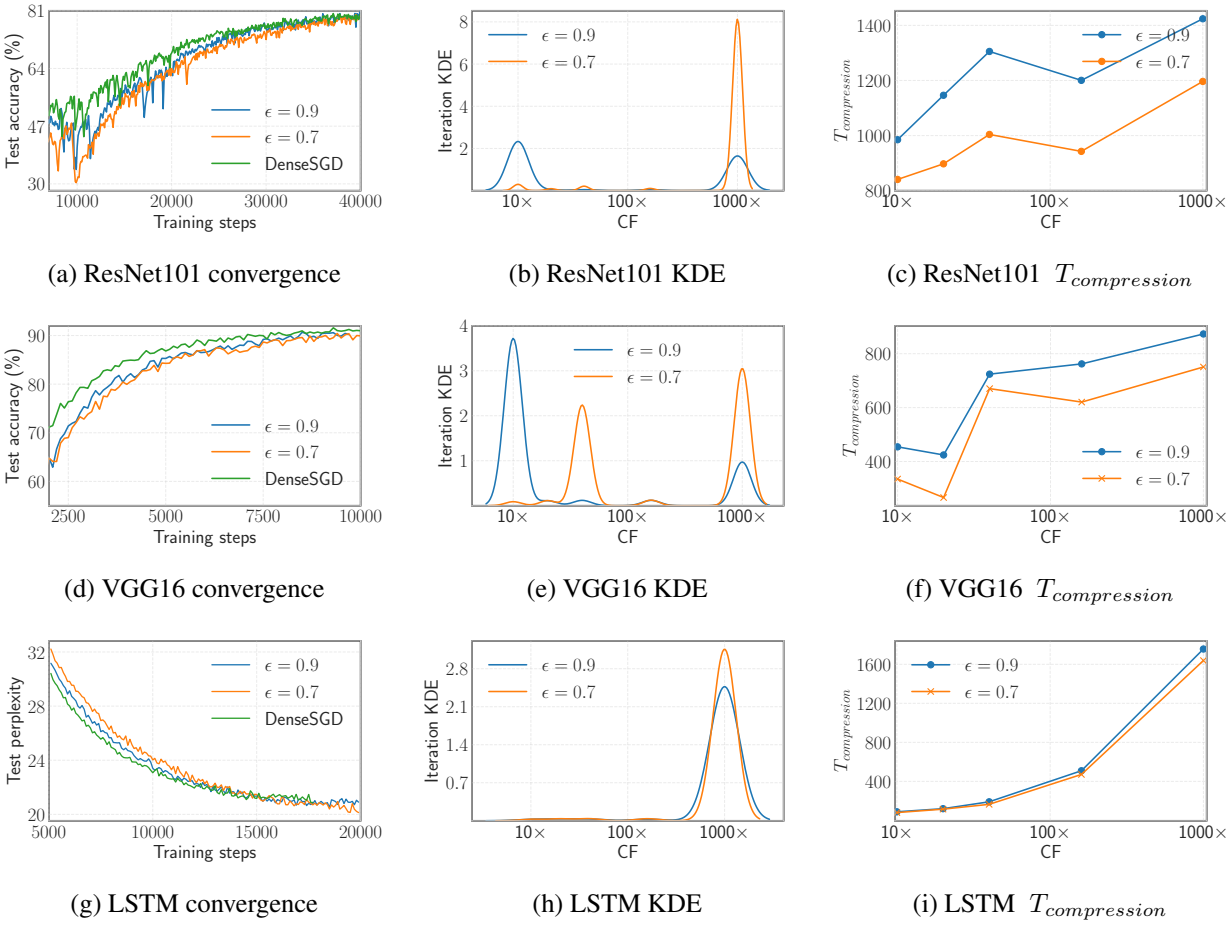
GraVAC is developed as a modular extension to PyTorch's [98] DDP package [204, 64], as shown in Fig. (5.6). We implement four prior compression techniques that we evaluate with GraVAC: Top-*k* [114], DGC [115], Redsync [127] and Random-*k* [111]. The details of GraVAC's adaptive compression mechanism are described in §B.1 of Appendix (B). The convergence targets and training routines are described in Table (5.1) and Appendix (B.2). We evaluate GraVAC with different scaling policies and look at the convergence curves (i.e., test accuracy or perplexity), average compression throughput of candidate CFs and kernel density estimates (KDE) of iterations with different CFs. KDE plots the iteration distribution used by candidate CFs on the log-scale with a smoothing bandwidth of 0.1 passed to a gaussian kernel.

5.4.1 GraVAC's Adaptive Compression Policies

We now look at GraVAC's parallel and statistical performance with different scaling policies for CFs bounded in the range $[\theta_{min}, \theta_{max}]$, with gain threshold ϵ, evaluation frequency 'window' and saturation threshold ω. The step-size θ_s is determined by the scaling policy, such that θ_s is either upscaled exponentially, or increased gradually as a geometric progression.

Exponential Scaling Policy:

Here, we implement the ScalingPolicy function in Alg. (4) such that CFs are scaled up in exponents of base 2 with respect to the initial θ_{min}. On top of DGC, we set θ_{min}=10×, θ_{max}=1000×, window=500 iterations and ω=1%. Between 10× and 1000×, we thus raise θ_s in factors of 2^1, 2^2,



(a) ResNet101 convergence (b) ResNet101 KDE (c) ResNet101 $T_{compression}$

(d) VGG16 convergence (e) VGG16 KDE (f) VGG16 $T_{compression}$

(g) LSTM convergence (h) LSTM KDE (i) LSTM $T_{compression}$

Figure 5.7: Performance of GraVAC with exponential scaling policy with ϵ-thresholds 0.7 and 0.9 vs. DenseSGD, iteration densities and compression throughput for various CFs.

2^4, 2^8, etc. The candidate CFs thus evaluated with current parameters under this policy are 10×, 20×, 40×, 160× and 1000×. We deploy GraVAC on two configurations, $\epsilon = 0.7$ and $\epsilon = 0.9$. The former threshold value relaxes the constraint on compression gain for larger CFs to be eligible for communication, consequently achieving higher compression. A larger ϵ (i.e., close to 1.0) allows synchronization of compressed gradients only if they are highly representative of the original gradients. We first compare the statistical performance of the two ϵ-thresholds with DenseSGD since the latter represents the ideal convergence scenario. Then, we compare GraVAC with multiple compression techniques using static CFs and compare final model accuracy or perplexity, total communication savings and overall speedup.

For ResNet101, Fig. (5.7a) shows that GraVAC achieves the same test accuracy as DenseSGD in the same number of iterations. The low and high gain thresholds reduce the overall communi-

cation volume by 163× and 19× over DenseSGD. Here, we measure the communication volume as the total single-precision floats exchanged among workers in GraVAC relative to DenseSGD. With compression, convergence is more volatile, as seen from accuracy drop due to learning-rate decay after 9000 iterations. The accuracy degradation is more apparent for $\epsilon = 0.7$ as we train with larger CFs due to the lower threshold. Comparatively, the higher gain threshold is more robust to hyperparameter tuning like learning-rate decay as we train with smaller CFs. This is seen in Fig. (5.7b) which plots the distribution of iterations using different CFs over training. For $\epsilon = 0.9$, we train in similar frequency at 10× and 1000×, while we train at 1000× for majority of the iterations with the lower threshold. For the larger threshold in Fig. (5.7c), although $T_{compression}$ is maximum for 1000× and minimum at 10×, we still evenly train with the two CFs (in Fig. (5.7b)). This is due to the higher threshold value and because θ_{min} did *not* scale up and remained at 10× for ResNet101. Thus, when gain of any candidate CF failed to meet the threshold, we communicated at 10× compression. For $\epsilon = 0.7$, compression throughput was maximum for 1000× and GraVAC continued training at this CF for majority of the iterations as the corresponding gain easily met the set threshold in this case.

Fig. (5.7d) compares the convergence for VGG16 with GraVAC's exponential policy and DenseSGD. Similar to ResNet101, VGG16 reaches similar accuracy-levels in the same number of iterations, where $\epsilon = 0.7$ and 0.9 reduce communication volume by 80× and 13.5× over DenseSGD. For $\epsilon = 0.9$, although $T_{compression}$ was maximum at CF 1000×, the compression gain corresponding to this CF was not as high to meet the required threshold. We thus often switched back to θ_{min} and trained at CF 10× for most iterations (as seen from density plots in Fig. (5.7e)). For the smaller $\epsilon = 0.7$, GraVAC found CF 40× to meet that threshold, and thus trained with 40× and 1000× for majority of the training phase. The compression throughput of CF 40× at $\epsilon = 0.7$ was also the second-largest in our exploration space. As training progresses, model converges and gradients saturate. Thus, compression gain improves even further and we attain maximum $T_{compression}$ corresponding to CF 1000×, as seen in Fig. (5.7f).

GraVAC with LSTM converged to nearly the same perplexity on both gain thresholds, shown in Fig. (5.7g), while reducing communication volume by 279× for $\epsilon = 0.9$ and 289× for $\epsilon = 0.7$. For the given dataset and training hyperparameters of LSTM, we already saw from Fig. (5.5c) that the compression gain was high for both CFs 10× and 1000×. Thus, we see that most iterations

train at 1000× in LSTM (from Fig. (5.7h)) as the gain for this CF easily met both ϵ-thresholds. The compression throughput is maximum at 1000× as well (Fig. (5.7i)), making it the optimal CF for this configuration.

Next, we compare GraVAC's exponential scaling policy with static compression techniques, i.e., compressors using a fixed CF. We evaluate against Top-*k*, DGC, Redsync and Random-*k* at 10× and 1000× CFs. Each compressor is trained until the target metric does not improve any further, and we report test accuracy/perplexity and training speedup relative to Top*k* 10× CF in Table (5.2). In ResNet101, compressors Redsync, DGC and Top-*k* at CF 1000× achieved considerable speedup over Top-*k* 10×, albeit with lower final accuracy. At CF 1000×, Top*k*, DGC and Redsync converged to 76.4%, 78.6% and 77.4% test accuracy. Random-*k* failed to converge at both CFs and the final accuracy did not improve beyond 20%. With GraVAC's adaptive scheme, ResNet101 converged to 80.2% accuracy while still reducing training time by 4.32×.

We previously saw that VGG16 is robust to high compression (as seen from Fig. (5.4f)). VGG16 at 1000× on Top-*k*, DGC and Redsync achieves $>$ 90% accuracy with about 3.22×, 3.35× and 3.6× speedup over Top*k* 10×. At CF 10×, Random-*k* converged to 87.8% accuracy and failed to improve beyond 30% accuracy at CF 1000×. GraVAC attained 90.48% accuracy with 1.95× overall speedup, so other compression schemes were faster since they used a higher CF for a learning task robust to compression.

For LSTM, GraVAC obtains the least perplexity 21.25 while achieving maximal speedup of 6.67× over Top-*k* 10×. Random-*k* at CF 1000× failed to converge, while 10× reached a perplexity of 24.15. Aside from GraVAC, only Top-*k*, DGC and Redsync at CF 10× achieved similar or better convergence quality than DenseSGD.

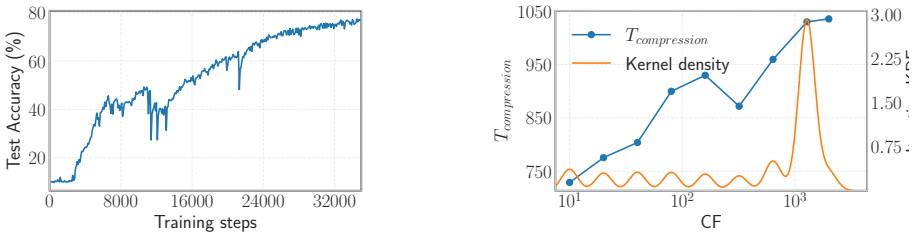
Hence, we see that GraVAC is able to train DNNs like ResNet101 and LSTM to high quality while still considerably reducing its training time. Static compression schemes achieve high accuracy at smaller CFs but result in lower speedups due to higher communication costs, while larger CFs significantly reduce communication at the cost of lower model quality. On the other hand, over-parameterized models like VGG16 can be robust to compression and successfully converge even at high CFs.

Model	Compression	Target metric	Speedup
ResNet101	Top-k 10x	80.14%	1×
	Top-k 1000x	76.4%	3.02×
	DGC 10x	80.4%	1.23×
	DGC 1000x	78.6%	5.19×
	Redsync 10x	79.4%	1.2×
	Redsync 1000x	77.4%	6.94×
	Random-k 10x	-	-
	Random-k 1000x	-	-
	GraVAC	**80.2%**	**4.32**×
VGG16	Top-k 10x	91.2%	1×
	Top-k 1000x	90.68%	3.22×
	DGC 10x	90.8%	0.935×
	DGC 1000x	90.4%	3.35×
	Redsync 10x	90.45%	0.99×
	Redsync 1000x	90.3%	3.6×
	Random-k 10x	87.8%	0.7×
	Random-k 1000x	-	-
	GraVAC	**90.48%**	**1.95**×
LSTM	Top-k 10x	22.0	1×
	Top-k 1000x	26.78	3.36×
	DGC 10x	21.67	1.23×
	DGC 1000x	25.14	6.25×
	Redsync 10x	21.65	1.17×
	Redsync 1000x	24.24	6.9×
	Random-k 10x	24.15	1.3×
	Random-k 1000x	-	-
	GraVAC	**21.25**	**6.67**×

Table 5.2: Parallel and statistical performance of GraVAC over static compression

Geometric Scaling Policy:

Here, we propose a gradual scaling policy where the ScalingPolicy function (from Alg. (4)) increments CFs as a geometric progression with a common ratio of 2.0. We deploy GraVAC over Redsync compression with $\theta_{min} = 10\times$, $\theta_{max} = 2000\times$, $\epsilon = 0.7$, 'window' = 2000 iterations and $\omega = 1\%$. Thus, candidate CFs evaluated with this policy are 10×, 20×, 40×, 80×, 160×, 320×, 640×, 1280× and 2000×. Fig. (5.8a) shows the accuracy of ResNet101 over the course of training. GraVAC reduced the overall communication volume by 76× over DenseSGD. Compared to exponential policy, convergence is slower since each candidate CF is evaluated over a larger window-size. As a result, gradients get smaller and saturate as we approach higher CFs, and gain of candidate CFs easily meets the ϵ-threshold and gives similar iteration densities from CF 10×-640×



(a) Model convergence

(b) $T_{compression}$ and KDE

Figure 5.8: ResNet101 deployed with GraVAC on geometric scaling policy. We evaluate more candidate CFs under this policy as seen from the iteration density distributions.

Model	Method	Direct (ms)	Multi-level (ms)	Speedup
ResNet101	Top-k	606	332	1.83×
	DGC	90	59	1.52×
	Redsync	33	29.8	1.1×
	Random-k	23	14	1.64×
VGG16	Top-k	181	121	1.49×
	DGC	122	95.5	1.27×
	Redsync	101.4	87.7	1.16×
	Random-k	41.6	31	1.34×
LSTM	Top-k	200	126	1.59×
	DGC	88	63	1.4×
	Redsync	69.4	46.4	1.5×
	Random-k	56.4	37.4	1.5×

Table 5.3: GraVAC's mulit-level compression speedup

(Fig. (5.8b)). After that, we mostly train with CF 1280× instead of 2000× since the compression throughput at these CFs was 1029.9 and 1035.4, which falls well within the bounds of $\omega = 1\%$. This is because GraVAC does not scale the CF beyond the point where the parallel and statistical efficiency of compression stops improving.

5.4.2 Multi-level Compression in GraVAC

Alg. (4) in Appendix (B) explains how GraVAC scales up compression from θ_{min} to θ_{max} with a step-up factor θ_s. Compressing the original gradients twice, once over θ_{min} and then to candidate CF θ_c may incur considerable overhead. The latency of compression operation may be higher for large DNNs with proportionally larger tensors, as well as the target CF used for compression. If

Model	Method	Floats sent	Comm. savings	Time savings
Resnet101	Accordion	4.17×10^{11}	$1\times$	$1\times$
	GraVAC	$\mathbf{9.38 \times 10^{9}}$	$\mathbf{44.5\times}$	$\mathbf{1.94\times}$
VGG16	Accordion	3.83×10^{11}	$1\times$	$1\times$
	GraVAC	$\mathbf{1.7 \times 10^{10}}$	$\mathbf{22.4\times}$	$\mathbf{5.63\times}$
LSTM	Accordion	4.2×10^{11}	$1\times$	$1\times$
	GraVAC	$\mathbf{4 \times 10^{9}}$	$\mathbf{104.2\times}$	$\mathbf{2.06\times}$

Table 5.4: Communication and time savings with GraVAC

operator $\mathcal{C}$ compresses tensor $\mathcal{X}$ to CFs θ_1 and θ_2:

$$\mathcal{X}_1 = \mathcal{C}(\theta_1, \mathcal{X}) \text{ and } \mathcal{X}_2 = \mathcal{C}(\theta_2, \mathcal{X})$$

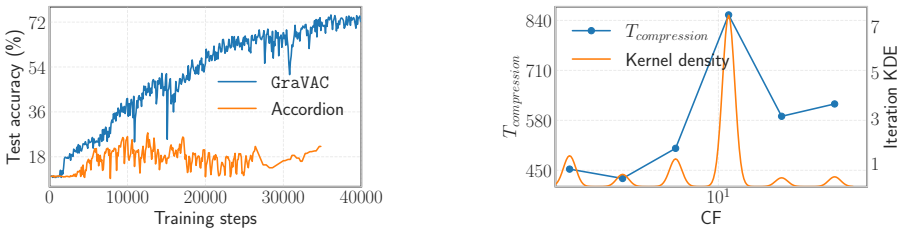
where $\theta_2 > \theta_1$ and compressed tensors $\mathcal{X}_1$ and $\mathcal{X}_2$ are produced by compressing the original tensor $\mathcal{X}$. To reduce this overhead, we apply multi-level compression where we first compress $\mathcal{X}$ to $\mathcal{X}_1$, and then further compressing $\mathcal{X}_1$ to CF $\theta_2^{'}$ and produce $\mathcal{X}_2^{'}$.

$$\mathcal{X}_1 = \mathcal{C}(\theta_1, \mathcal{X}) \implies \mathcal{X}_2^{'} = \mathcal{C}(\theta_2^{'}, \mathcal{X}_1) \ : \ \theta_2^{'} = \frac{\theta_2}{\theta_1}$$

The resulting tensor $\mathcal{X}_2^{'} \equiv \mathcal{X}_2$ for $\theta_2^{'} = \theta_2/\theta_1$. The time savings come from the second compression operation applied on the smaller tensor $\mathcal{X}_1$ instead of $\mathcal{X}$. We tabulate these savings in Table (5.3) for CF $1000\times$ across various compression techniques. In multi-level compression, we first compress to CF $10\times$ (i.e., θ_1), and then compress the former's output to CF $100\times$ (i.e., $\theta_2^{'}$) to get tensors compressed to $1000\times$ (i.e., θ_2) relative to original tensors. In direct approach, we tabulate the cumulative cost of first compressing to $10\times$ and then compressing the original tensors to $1000\times$. From these results, we see that multi-level compression in GraVAC is about 1.1-1.83$\times$ faster than directly compressing the original gradients twice.

5.4.3 Comparing GraVAC with prior art

We compare GraVAC with another adaptive scheme, Accordion [131], which uses rank-1 and rank-4 compression as the lower and upper bounds. Table (5.4) tabulates the communication cost and time savings in GraVAC and normalizes it with respect to Accordion's performance. For ResNet101, GraVAC reduces communication volume by $44.5\times$ and speeds up training by $1.94\times$ over Accor-



(a) Model convergence

(b) $T_{compression}$ and KDE

Figure 5.9: ResNet101 on Random-k showing convergence of GraVAC and Accordion, along with GraVAC's compression throughput and iteration densities at different CFs.

dion. For larger models like VGG16, GraVAC accelerates training by 5.63×, and twice as fast to train LSTM by reducing communication volume by over hundred fold.

Accordion uses two CFs during training; a small CF during sensitive phases, and a larger CF otherwise. On the other hand, GraVAC looks at information loss due to compression and adjusts CFs in a fine-grained manner throughout training. Accordion looks at inter-iteration gradients, while GraVAC looks at intra-iteration gradients to perform compression, thus cognizant of the compression quality of a compressor.

We further see the convergence capability of GraVAC by training with Random-k compression and compare with Accordion. Previously, we observed in Table (5.2) that ResNet101 on Random-k failed to converge successfully at either 10× or 1000× CF. Although the compression quality of Random-k is lower than other compressors, we use it as a special case to demonstrate GraVAC's dynamic compression policy and ability to operate at a fine granularity. We set $\theta_{min} = 1.5\times$, $\theta_{max} = 1000\times$, 'window' = 2000 and $\epsilon = 0.7$, while CFs are scaled-up via geometric policy. Accordion was deployed with the same min-max CF bounds, i.e., lower CF of 1.5× and higher CF 1000×. Fig. (5.9a) plots the convergence curves of the two, and see that unlike Random-k at fixed CF 10× that failed to converge (in Table (5.2)), GraVAC achieved 78% test accuracy on ResNet101. The candidate CFs evaluated here were 1.5×, 3×, 6×, 12×, 24× and 48×; CFs beyond this point were ignored as they did not meet the minimum gain threshold requirement. From Fig. (5.9b), CF 12× has the largest density, implying that most iterations used this CF for training. The compression throughput is maximal at this CF, and smaller for later CFs due to lower compression gain. Compared to DenseSGD, GraVAC reduced the overall communication volume by 18×.

Accordion on Random-k attained ≈ 20% test accuracy only as it does not consider the efficacy

of the compression technique (i.e., statistical efficiency in gradient compression) and switches CFs if the uncompressed, inter-iteration gradients change beyond a certain measure. The information loss in the gradients with these min-max bounds was thus too high to update the model in any meaningful way.

5.5 Conclusion

Gradient information has previously been studied to detect critical training regions [27, 28, 149] or as a scalability indicator [168, 169, 167] in DL. Adaptive compression schemes such as Accordion switch between two compression levels based on critical or sensitive regions measured through inter-iteration gradients. The key insight of `GraVAC` is to adaptively vary compression-rate throughout training while balancing the pareto-relationship between parallel and statistical efficiency of gradient compression. We use 'compression gain' to measure information loss from compression and adjust CFs accordingly. `GraVAC` converged 1.95-6.67$\times$ faster over static CF training, while converging in nearly the same number of iterations and accuracy-levels as DenseSGD. Compared to Accordion, we attained up to 5.63$\times$ reduction in overall training time.

`GraVAC`'s performance is influenced by parameters like min-max CFs used, the window-size for each candidate CF, scaling policy used and gain threshold ϵ. Setting too small a window-size may result in poor convergence as all candidate CFs may be exhausted while the model is still in early training stages and gradients are volatile. For the gain threshold, choosing a small ϵ may achieve high compression from the beginning, but also lead to model degradation since highly compressed gradients would likely fail to update the model in a significant manner.

5.6 Related Publications

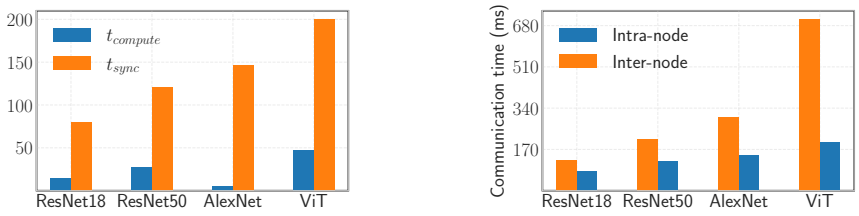
1. **Tyagi, Sahil** and Swany, Martin. "GraVAC: Adaptive Compression for Communication-Efficient Distributed DL Training." 2023 IEEE 16th International Conference on Cloud Computing (CLOUD) (2023): 319-329.

CHAPTER 6
FLEXIBLE COMPRESSION-COMMUNICATION OVER UNPREDICTABLE NETWORKS

6.1 Introduction

Gradient compression techniques alleviate expensive communication either by transmitting fewer values, or reducing the bit-width of updates, or approximating gradient tensors to a lower dimensional space. Training with low CFs ensures good convergence quality, but suffers from poor parallel scaling due to high communication costs. On the other hand, high CFs improve parallel efficiency but may degrade statistical performance as model parameters are updated with compressed gradients [31]. Additionally, the overall speedup attained in distributed training depends on a plethora of factors like cluster-size, model-size, target CF, and network-specific parameters like device placement, latency, effective bandwidth and the collective implementation used for communication. Even smaller DNNs like ResNet18 can suffer from frequent communication in high latency, low-bandwidth settings. With the exception of low-rank approximators, most compression techniques involve sharing a subset of total values and their corresponding indices using collectives like All-Gather (AG) [187]. In some cluster and network configurations, cost of AG can be higher than All-Reduce (AR) for the same message-size. Thus, distributed DL may exhibit varying communication costs contingent upon various factors.

In this chapter, we propose `AR-Top`*k* or *All-Reducible Top-k* [32] compression that works with different topologies of AR (like ring and tree) and describe its two variants, followed by their respective communication cost analysis. For fluctuating and unpredictable networks, we develop heuristics to choose the optimal collective between `AR-Top`*k* (with ring or tree AR), AG-based compression or AR with DenseSGD. Lastly, to balance parallel and statistical efficiency, we model gradient compression as a multi-objective optimization (MOO) problem to assess the optimal CF at different training stages.



(a) Communication cost

(b) Intra vs. Inter-node latency

Figure 6.1: (a) Computation and synchronization times across 8 intra-node GPUs. (b) Communication cost when intra-node device placement comprises of 8 GPUs/node on 1 node, while inter-node setting contains 1 GPU/node on 8 nodes.

6.2 Motivation

Following Moore's law, growth in computational capabilities of modern silicon has accelerated compute in DL applications. We corroborate this in Fig. (6.1a), where the computation cost of ResNet18, ResNet50 [9], AlexNet [1] and ViT [205] is substantially lower than the communication cost of aggregating updates among 8 V100 GPUs residing on a single node. As model-size increases (from left to right), so does its communication time.

Inter-node vs. Intra-node scaling:

In BSP training, synchronization cost between a cluster of nodes depends on the physical distance between them. Intra-node GPUs on a single node are connected via high-speed interconnects like PCIe (Peripheral Component Interconnect express) Gen4 (at speeds 64GB/s), Gen5 (128GB/s) [58] or NVLink 4.0 with up to 900GB/s bandwidth [206]. Scaling on a single system is limited by finite interconnect density or limited PCIe lanes. For e.g., NVIDIA DGX systems can accommodate up to 8 GPUs per-node [207] and the NVSwitch can connect up to 256 GPUs in a cluster. Inter-node systems can scale even further, but suffer the high communication cost to transfer large updates over networks with comparatively lower bandwidths. The average bandwidth available to general-purpose cloud VMs is about 10Gbps [208], while high tier VMs can get up to 25Gbps egress bandwidth [69]. In Fig. (6.1b), we compare the synchronization cost of various DNNs for intra and inter-node configurations. 8 V100s are co-located on a single node in the former (i.e., 8 GPUs per-node) with peer-to-peer (P2P) transport enabled to allow CUDA direct access, while the

Operation	Bandwidth Complexity	Communication cost
PS (Star)	$\mathcal{O}(MN)$	$2\alpha + 2(N-1)M\beta$
Ring-AR	$\mathcal{O}(M)$	$2(N-1)\alpha + 2\frac{(N-1)}{N}M\beta$
Tree-AR	$\mathcal{O}(M\log(N))$	$2\alpha\log(N) + 2\log(N)M\beta$
Broadcast	$\mathcal{O}(M\log(N))$	$\alpha\log(N) + \log(N)M\beta$
Allgather	$\mathcal{O}(MN)$	$\alpha\log(N) + (N-1)M\beta$

Table 6.1: Bandwidth Complexity and Cost of Communication Primitives

latter is composed of 8 machines with 1 GPU per-node, connected over a 10Gbps interface. We see that inter-node communication is manyfolds more expensive than intra-node AR over NVIDIA's NCCL communication backend [56].

Communication Cost of MPI Collectives:

Depending on network topology and algorithmic implementation, the cost of aggregating updates in distributed DL may vary. Centralized systems use one or multiple-sharded PS based on star-topology, while decentralized systems may use different versions of AR like reduce-broadcast, scatter-reduce-allgather (SRA), ring, tree, etc. Table (6.1) lists the total cost and bandwidth complexity of different collectives based on the α-β communication model [62, 30]. Here, the α-term is network latency, $1/\beta$ is the network bandwidth and 'M' is size of the model or gradients exchanged among 'N' workers. Synchronization cost in PS increases with both cluster and model-size. Ring-AR is dominated by latency cost (i.e., the α-term) as its bandwidth-optimal, i.e., β-term is nearly independent of N and only increases with model-size M. Although Tree-AR is logarithmic bandwidth-optimal, it has a lower latency cost than ring; α-term is proportional to $\log(N)$ in Tree and $(N-1)$ in Ring-AR. Thus, cost of different collectives vary for a given network configuration, cluster and model-size, and implementation. In the latter for instance, AR over CUDA GPUs is faster over NCCL than Gloo, while Gloo performs better than MPI over CPU-based training.

Gradient Compression to Mitigate Communication Overhead:

Lossy gradient compression techniques like LWTop-*k* [112] and MSTop-*k* [209] retrieve the largest $k\%$ gradients and transfer its respective values and indices. As explained in Eqn. (2.8), error-feedback mechanism preserves updates and improves the statistical efficiency of lossy compressors. By Eqn. (2.7), parallel efficiency depends on the compression overhead $t^{(c)}_{comp-decomp}$ and synchro-

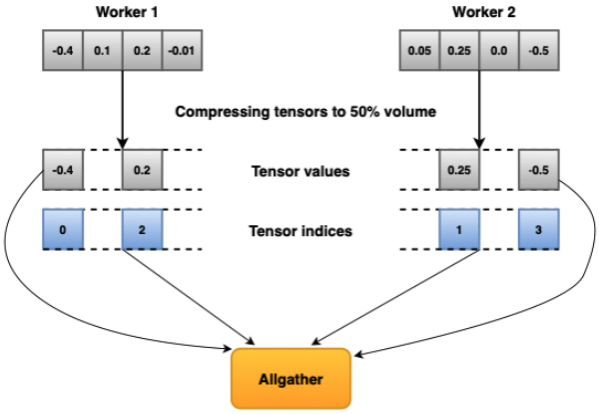
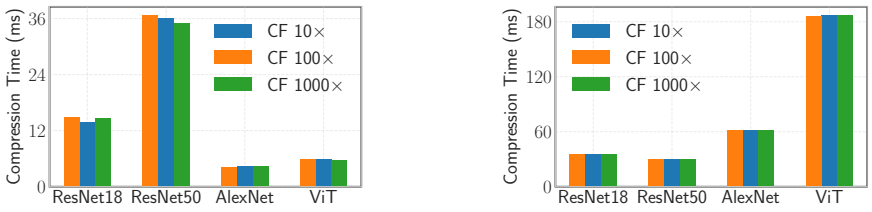


Figure 6.2: Workers communicate gradient values and indices via AG in sparsification.

nization cost $t_{sync}^{(c)}$ at CF 'c'. If the communication cost of DenseSGD is t_{sync}, then for a compressor to be viable over DenseSGD: $t_{comp-decomp}^{(c)} + t_{sync}^{(c)} < t_{sync}$; updates in the latter are aggregated via centralized PS or decentralized AR, where each may have different latency-bandwidth costs associated with it. However, sparse updates are not compatible with AR, since updates are collected via AG by transmitting select values and its indices from each worker, as a tensor index chosen on one worker may not necessarily be chosen for communication on another. We see this in Fig. (6.2) where the top 50% values in the two workers correspond to different indexes; we thus communicate both tensor values and indices via AG. A gradient tensor of size 'M' compressed to top-k% values thus needs to transmit $\lceil 2kM/100 \rceil$ data-points (i.e., $\lceil kM/100 \rceil$ gradient values and $\lceil kM/100 \rceil$ indices). Although some compressors like Random-*k* (a sparsification technique) and PowerSGD (a low-rank approximation method) are AR friendly, the former has poor convergence quality while the latter has considerable compression overhead and has been shown to attain lower CFs than other techniques [124].

The computation cost of compression itself may also vary with each technique and target CF. Fig. (6.3) shows the compression overhead in LWTop-*k* and MSTop-*k* compression at CFs 10, 100 and 1000×. The latter has a higher compression overhead than LWTop-*k* as it needs multiple rounds to compute its threshold. Thus, the largest model ViT has the largest compression cost in MSTop-*k*. On the other hand, LWTop-*k* retrieves the top-k% gradients on a layerwise basis, so a model with the more layers tends to have a higher compression time. We see this for ResNet50 in Fig. (6.3a) with its higher compression overhead over larger models like AlexNet and ViT.



(a) LWTop-k compression

(b) MSTop-k compression

Figure 6.3: Overhead of different compression techniques. For the same target CF, MSTop-k has higher cost than LWTop-k due to its multi-round threshold estimation.

Latency and Bandwidth Variability in Distributed Systems:

Variable and unpredictable network performance is prevalent on the edge, cloud and even HPC. Different classes of nodes or VMs have varying networking tiers available to them in terms of ingress/egress bandwidth [69, 70]. In mobile computing and IoT, edge devices tend to have slower networks compared to fog serves. Network heterogeneity can also arise from topology (e.g., star, mesh, peer-to-peer, ring, flat tree, etc.), type of links (ethernet, infiniband) and physical mediums (like fiber or copper). Latency and bandwidth can even vary on the same link with time due to various factors. Network bandwidth, which measures the data-transfer rate of a connection can fluctuate with time because of:

- *Network Congestion*: Effective bandwidth can decrease over high network traffic, increasing data-transfer time on account of saturated links.
- *Quality of Service (QoS) prioritization*: Jobs deployed on a cluster of nodes may have different QoS priorities, where high-priority jobs are allocated a bigger chunk of the total bandwidth. Thus, low-priority jobs may suffer from slow communication.
- *Resource Sharing/Contention*: Even without explicit prioritization, concurrent jobs in shared environments must share finite resources, lowering bandwidth per-job.
- *Network Scheduling*: Switches and routers may use different scheduling algorithms to allocate bandwidth for different traffic flows, which in turn influences how networking resources are shared among applications.

Configuration		Time cost (ms)		
Tensor-size	$(\alpha, 1/\beta)$	**AG 10×**	**AG 1000×**	**Ring-AR**
10^8	(10, 40)	189	66	286
	(10, 25)	262	67	374
	(10, 10)	525	70	716
	(10, 5)	976	74	1271
	(10, 1)	4568	111	5773
	(100, 40)	460	336	1548
	(100, 25)	529	338	1638
	(100, 10)	798	340	1975
	(100, 5)	1248	345	2530
	(100, 1)	4830	380	7028
10^9	(10, 40)	1668	442	**1552**
	(10, 25)	2320	453	2402
	(10, 10)	5010	482	5774
	(10, 5)	9507	534	11380
	(10, 1)	45355	898	56190
	(100, 40)	1920	711	2808
	(100, 25)	2604	720	3654
	(100, 10)	5280	745	7024
	(100, 5)	9805	791	12621
	(100, 1)	45645	1154	57442

Table 6.2: Top-*k* compression + communication cost at CFs 10/1000× with AG vs. Ring-AR on uncompressed tensors with 100 million and 1 billion parameters respectively.

The latency or communication delay over a network can fluctuate as well. Local area network (LAN) has a very low latency, ranging from a few micro- to milliseconds, inter-rack latency in a data-center can be in the order of tens of milliseconds, while inter-node latency is even worse (as seen in Fig. (6.1b)). Network load and congestion can also cause latency-surge under high traffic flows due to queuing delays as multiple jobs may perform simultaneous data transfers. The network routing that decides the data-path of a packet also influences its latency. Although routing protocols dictate the optimal path for transmission, different routes can result in varying latency due to network congestion and physical distance. As a result, variable latency and bandwidth over a cluster of nodes can affect a distributed job's completion time.

On a cluster of 8 inter-node V100 GPUs connected over a 40Gbps network, we vary the latency (i.e., α) and bandwidth ($1/\beta$) using linux' '`tc`' module [210] and measure the time it takes to synchronize tensors with 100 million and 1 billion parameters represented as single-precision or 32-bit floating points. Latency is controlled via '*netem*' or network emulation qdisc (queuing discipline)

while the bandwidth is adjusted via '*htb*' or hierarchical token bucket qdisc. We compare the time taken to reduce the original, uncompressed gradients with Ring-AR vs. the cumulative compression *and* communication time of Top-*k* using AG to collect compressed updates. For e.g., a 1 billion tensor compressed to CF 1000× exchanges 2 million data-points with AG (values + indices), while Ring-AR communicates all 1 billion values. The results are tabulated in Table (6.2) for different (α, β) configurations. *Although compressed communication with AG for any CF 'c' is faster than ring-AR for the same (α, β) due to its smaller message size, AR call does not take $c\times$ more time to communicate its updates*. This is because the cost of different collectives varies with message-size, cluster-size and network parameters. Ring-AR is bandwidth-optimal but sensitive to high-latency, as evidently seen from the wider gap between AG and ring-AR when latency is as high as 100 milliseconds. Synchronization overhead in AG rises considerably in low-bandwidth networks even if latency remains constant. Thus, it is not always clear if compression is viable with AG-communication, or synchronize original updates with optimized collectives instead.

Measuring Statistical Efficiency in Gradient Compression:

We now look at the effect of a chosen compression technique and CF on the statistical performance of DNNs. LWTop-*k* and MSTop-*k* train at CFs = [10×, 100×, 1000×] on a cluster of 8 V100s over a (4ms, 20Gbps) network. For MSTop-*k* we use 25 rounds to accurately estimate its threshold value. Table (6.3) records the iteration time, test accuracy and convergence difference compared to DenseSGD for the two compressors. The training routine and hyperparameters for each model is described in §6.4. Here, the iteration cost includes computation time, I/O overhead, compression and communication cost to transmit updates via AG, while DenseSGD has no compression overhead but communicates entire model state via AR. Regardless of CF, the iteration cost of MSTop-*k* is more than LWTop-*k* due to its multi-round estimation (as also corroborated by its higher compression cost in Fig. (6.3)). In some cases like ViT at CF 10×, MSTop-*k* has a higher iteration time than even DenseSGD due to its high compression overhead.

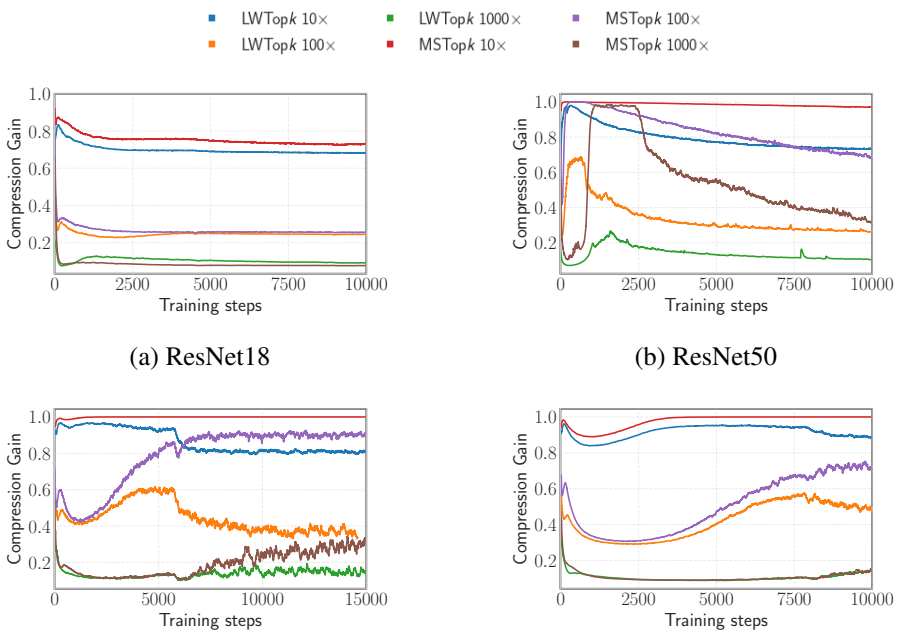
From a statistical point (seen from the 'Accuracy' and 'Difference' column of Table (6.3)), MSTop-*k* attains similar or better accuracy than LWTop-*k* since its threshold estimates are computed over the entire model, while the latter compresses on a layer-by-layer basis. In LWTop-*k*, DNNs with non-uniform layers and skewed gradients may thus lose critical updates since the fraction of

Model	Method	$t_{iteration}$	Accuracy	Difference
ResNet18	DenseSGD	98.7	90.8%	0.0%
	LWTop-*k* 10×	62	90.15%	-0.65%
	LWTop-*k* 100×	38.4	88.49%	-2.31%
	LWTop-*k* 1000×	36.8	87.2%	-3.6%
	MSTop-*k* 10×	83.22	90.59%	-0.21%
	MSTop-*k* 100×	60.22	89.83%	-0.97%
	MSTop-*k* 1000×	58	88.69%	-2.11%
ResNet50	DenseSGD	152.5	98.75%	0.0%
	LWTop-*k* 10×	130.24	98.65%	-0.1%
	LWTop-*k* 100×	79.4	98.3%	-0.45%
	LWTop-*k* 1000×	72.7	96.55%	-2.2%
	MSTop-*k* 10×	124.35	98.64%	-0.11%
	MSTop-*k* 100×	72.63	98.49%	-0.26%
	MSTop-*k* 1000×	67.33	97.73%	-1.02%
AlexNet	DenseSGD	230.6	82.41%	0.0%
	LWTop-*k* 10×	156.8	81.68%	-0.71%
	LWTop-*k* 100×	33.45	81.45%	-0.96%
	LWTop-*k* 1000×	21.3	78.9%	-3.51%
	MSTop-*k* 10×	214.5	81.8%	-0.61%
	MSTop-*k* 100×	91.4	81.7%	-0.71%
	MSTop-*k* 1000×	79	78.92%	-3.49%
ViT	DenseSGD	475	80.4%	0.0%
	LWTop-*k* 10×	362.4	79%	-1.4%
	LWTop-*k* 100×	94.64	78.75%	-1.65%
	LWTop-*k* 1000×	67.7	78.25%	-2.15%
	MSTop-*k* 10×	543.6	80.47%	+0.07%
	MSTop-*k* 100×	275.24	79.87%	-0.53%
	MSTop-*k* 1000×	248.8	79.76%	-0.64%

Table 6.3: Iteration time (ms), accuracy and difference with respect to DenseSGD.

gradients selected from each layer is fixed (proportional to layer-size). Important updates from larger layers may not even be eligible for communication if majority of the critical updates are cluttered in only a few layers.

ResNet18 reached 90.8% accuracy with DenseSGD, while LWTop-*k* attained 88.49% and 87.2% accuracy at 100× and 1000× respectively. MSTop-*k* on ResNet18 achieved 89.83% and 88.69% for the same CFs. Other DNNs exhibit a similar pattern; as CF is increased, communication cost is reduced but model accuracy degrades as well. In Chapter (5), we defined *Compression Gain* to measure this information loss between prior and post-compression gradients. From Eqn. (5.1), gain



(a) ResNet18

(b) ResNet50

(c) AlexNet

(d) ViT

Figure 6.4: Compression gain of LWTop-k and MSTop-k at various CFs measures statistical efficiency that correlates to final model accuracy achieved by each configuration.

is computed by comparing the deviation between error-fed and compressed gradients:

$$\mathbb{E}[||g_c^{(i)}||^2]/\mathbb{E}[||g_{ef}^{(i)}||^2]$$

.

The overhead of calculating compression gain is low as it only needs to calculate the norm of first-order updates computed in backpropagation. Fig. (6.4) plots the compression gain over the iterations at various CFs of LWTop-k and MSTop-k compression. Across all DNNs, the gain tends to be smaller for high CFs as more gradient information is lost. Gain also changes in the early training stages and during certain critical regimes, and eventually saturates [31, 149, 27, 131]. By comparing with Table (6.3) for each compressor and CF configuration, we notice a correlation between compression gain and model convergence. For e.g., ResNet50 at CF 1000× attained 96.55% accuracy with LWTop-k and a higher 97.73% with MSTop-k; at the same time, MSTop-k has a higher compression gain than LWTop-k. Additionally, the generalization performance between different compressors and CFs can also be inferred from their respective compression gains. E.g., AlexNet with LWTop-k at CF 10× and MSTop-k at 100× have roughly the same 81.68% and 81.7% test

accuracy, while their gains saturate to the same level as well (Fig. (6.4c)). Thus, compression gain is a practical and effective heuristic to evaluate the efficacy of different compressors and CFs, and measures the statistical efficiency of lossy gradient compression.

6.3 AR-Top*k* Compression

We now propose an AR-compatible Top-*k* compression-communication mechanism called AR-Top$k$, along with its worker selection mechanism and communication cost analysis to determine the optimal collective in DDL training. Although optimized AR collectives have low bandwidth-cost compared to AG, using the latter to communicate the same tensors may have lower cost under certain networking and training-specific conditions. Hence, we propose a flexible compression-communication scheme that uses the optimal collective based on the α-β communication model. To determine the optimal CF and collective that best balances the parallel and statistical efficiency of lossy compression, we model gradient compression as a multi-objective optimization (MOO) problem.

6.3.1 AR-compatible Top-*k* Compression

We implement a Top-*k* compressor using max-heap and retrieve the top $k\%$ values across fused gradient tensors, i.e., compression is applied over the entire model instead of a layerwise manner (as done in LWTop-*k*). Compared to MSTop-*k* which is based on multi-round threshold estimation, the overhead of max-heap based sorting is much lower. AR-Top$k$ takes the compressed values and indices from Top-*k* sorting and initiates a "Broadcast" from a worker to disperse its corresponding indices, followed by an "AR" call to aggregate values on the indexes that were broadcast in the first step. The AR call in the second step can be ring, tree or any other implementation of allreduce.

After the broadcast of compressed indices from a worker, gradient values reduced at those positions may not necessarily correspond to the top $k\%$ gradients across all workers. To choose which worker shares its local top-*k* updates with other workers in cluster, we explore two worker selection mechanisms in AR-Top$k$ compression:

1. **Variance-based** or **VAR-Top*k*** selects worker with the largest gradient variance and disperses the indexes corresponding to its local top-*k* values, followed by calling AR on those elements.

Algorithm 1: AR-Top$k$ Compression

1 **Parameters:** CF c, cluster-size N, worker-rank r, gradient value g at index ix
2 **function** `train()`**:**
3 $\quad$ residual$_{(0,r)} = 0$
4 $\quad$ **for** $i = 1, 2, ...$ **do**
5 $\quad\quad$ $\mathrm{G}_{(i,r)} = \nabla\mathcal{F}(x_{(i,r)}, w_{(i,r)}) + \mathrm{residual}_{(i-1,r)}$
6 $\quad\quad$ $\mathrm{g}_{(i,r)}$, $\mathrm{ix}_{(i,r)}$ = Top$k$ $(G_{(i,r)},\ c)$
7 $\quad\quad$ **if** STAR-Top$k$ **then**
8 $\quad\quad\quad$ $\tilde{r}_{(i)} = i\ \%\ N$
9 $\quad\quad$ **else if** VAR-Top$k$ **then**
10 $\quad\quad\quad$ var [N] = 0
11 $\quad\quad\quad$ var [r] = $||g_{(i,r)}||^2$
12 $\quad\quad\quad$ var = AllGather(var)
13 $\quad\quad\quad$ $\tilde{r}_{(i)} = \mathrm{var.indexOf}(\max(\mathrm{var}))$
14 $\quad\quad$ $\tilde{\mathrm{ix}}_{(i,r)} = \mathrm{Broadcast}\ (ix_{(i,r)}, src = \tilde{r}_{(i)})$
15 $\quad\quad$ $\tilde{\mathrm{g}}_{(i,r)} = \mathrm{G}_{(i,r)}[\tilde{\mathrm{ix}}_{(i,r)}]$
16 $\quad\quad$ $\mathrm{residual}_{(i,r)} = \mathrm{G}_{(i,r)} - \tilde{\mathrm{g}}_{(i,r)}$
17 $\quad\quad$ $\tilde{\mathrm{g}}_{(i,r)} = \mathrm{AllReduce}\ (\tilde{g}_{(i,r)})$
18 $\quad\quad$ $w_{(i+1,r)} = w_{(i,t)} - \eta \cdot \tilde{\mathrm{g}}_{(i,r)}$
19 $\quad\quad$ i++

Since the significance of model updates can be measured by its variance across workers, the worker with largest gradient variance is chosen to communicate its local updates.

2. **Staleness-based** or **STAR-Top*k*** chooses top-*k* indexes from each worker in a round-robin fashion, which are then broadcast to call AR on values at those indices. This way, each worker transmits its top-*k* compressed update once every 'N' iterations (i.e., in a cluster of 'N' workers). While error-feedback effectively works as delayed or stale updates, STAR-Top*k* further adds an implicit staleness of N training steps.

Alg. (1) illustrates how workers are chosen in AR-Top$k$. Error-fed gradients $\mathrm{G}_{(i,r)}$ at iteration 'i' are compressed to CF 'c' on each worker with rank or worker-id 'r' in the first step. STAR-Top*k* selects a worker based on the current iteration count and cluster-size (at line 8), while VAR-Top*k* involves additional steps where an array of length 'N' stores the gradient norm of worker 'r' at index r (line 11). An AG call over the array synchronizes variance across all workers (line 12), and the worker with the largest gradient norm is chosen for communication (line 13). *The overhead of this step is relatively low as the message-size exchanged among workers is just 'N'*

single-precision floats. Once a worker (denoted by rank $\tilde{r}_{(i)}$) is chosen, the corresponding indexes are sent via broadcast (line 14). The values at these indices are then selected on other workers, residual gradients are updated and aggregated via AR (lines 15-17). Finally, model parameters are updated and training proceeds to the next iteration (lines 18-19).

6.3.2 Communication Cost Analysis of `AR-Topk`

$$t^{(c)}_{ART-ring} = \alpha[\, 2(N-1) \;+\; \log(N)\,] + Mc\beta[\, 2\frac{(N-1)}{N} \;+\; \log(N)\,] \tag{6.1a}$$

$$t^{(c)}_{ART-tree} = 3\alpha\log(N) + 3Mc\beta\log(N) \tag{6.1b}$$

The cumulative communication overhead of AG to exchange '$2Mc$' bytes of compressed gradient values and indices is: $\alpha\log(N) + 2Mc\beta(N-1)$, for latency α and bandwidth $1/\beta$ to transmit 'M' bytes of gradients compressed to CF 'c'. On the other hand, communication in `AR-Topk` is a two-step process: one Broadcast call to distribute a worker's top-*k* indices, followed by calling AR to reduce values at the corresponding indexes. `AR-Topk` may use different implementations of AR such as ring or tree-based reduction. From Table (6.1), each has a different cost in terms of the α-β model. The overall message exchange cost of `AR-Topk` thus depends on the implementation of AR algorithm, shown in Eqn. (6.1a) and (6.1b) for ring and tree-based reduction respectively.

Since communication cost can vary based on the AR collective used, we develop rough heuristics to determine which reduction algorithm is ideal for `AR-Topk` based on cluster and network configurations, target CF and DNNs' parameters. Looking at the α-β costs of ART-Ring and ART-Tree in Eqn. (6.1), the former should be used if $t^{(c)}_{ART-ring} < t^{(c)}_{ART-tree}$ and vice versa. After solving, ART-Ring is faster than ART-Tree if the α-β ratio follows Eqn. (6.2a). In other cases, it may be optimal to use AG instead of `AR-Topk`. For DNNs with 'M' bytes of gradients compressed to CF 'c', ART-Ring or ART-Tree are faster than AG if their latency-bandwidth ratio satisfies Eqn. (6.2b) or (6.2c) respectively, otherwise communication is faster with AG.

$$(\frac{\alpha}{\beta})_{\text{ART-Ring\_over\_ART-Tree}} < \Big(\frac{\log(N) - (N-1)/N}{N - 1 - \log(N)}\Big) Mc \tag{6.2a}$$

$$(\frac{\alpha}{\beta})_{\text{ART-Ring\_over\_AG}} < \Big(1 - \frac{1}{N} - \frac{\log(N)}{2(N-1)}\Big) Mc \tag{6.2b}$$

$$(\frac{\alpha}{\beta})_{\text{ART-Tree\_over\_AG}} < \Big(\frac{(N-1)}{\log(N)} - \frac{3}{2}\Big) Mc \tag{6.2c}$$

6.3.3 Compression as a Multi-Objective Optimization Problem

The parallel and statistical efficiency of lossy gradient compression in DDL is inversely related, i.e., one improves at the detriment of the other. This can be seen from the iteration costs and accuracy achieved in Table (6.3) where larger CFs result in lower iteration times (thus, improving parallel efficiency) but at the cost of lower final accuracy (i.e., degrading statistical efficiency). As gradient sensitivity varies throughout training, it is intuitive to change CF as training progresses as well. For instance, we could use low CFs in early stages when gradients are volatile and higher CFs when the updates are saturated and stop changing significantly (thus, training is robust to compression in this stage).

To accelerate training while still achieving DenseSGD-like convergence quality, we propose an adaptive compression scheme on top of `AR-Top`k that models compression as a *multi-objective optimization* (MOO) problem. Here, we identify three key metrics that affect the relationship between parallel and statistical efficiency:

1. **Compression time:** The overhead to compress and decompress gradients depends on the compression technique itself and may vary with the target CF as well. For e.g., Top-k sorting based on max-heap for G gradients has complexity $\mathcal{O}(G + k \log G)$. In this case, compression cost decreases as target CF increases.

2. **Communication time:** Among AG, ART-Ring and ART-Tree, the optimal collective with the least communication cost depends on α, β, M, N and target CF (as described earlier in Eqn. (6.2)).

3. **Compression gain:** Depends on the compressor quality, CF, stage of training, and indicates statistical efficiency from the gradient information lost in compression.

$$c_{optimal} = \underset{c}{\text{argmin}}\{\mathcal{F}(\ t^{(c)}_{comp-decomp}, t^{(c)}_{sync}, gain^{-1(c)})\} \tag{6.3}$$

In Eqn. (6.3), the optimal CF $c_{optimal}$ can be determined as a solution to the MOO problem that aims to minimize the compression and communication overheads (i.e., $t^{(c)}_{comp-decomp}$ and $t^{(c)}_{sync}$), while maximizing compression $gain$ (or minimizing $1/gain$). Since both network performance and gain can vary throughout training, so does $c_{optimal}$. Additionally, we bound the candidate CFs in a min-max exploration space $[c_{low}, c_{high}]$ so as to avoid losing important updates at excessively high CFs or incur considerable communication cost at very low CFs. The search for ideal CF at the current model and training state is triggered when the average latency or bandwidth changes beyond a certain threshold. A background process measures inter-node bandwidth using `iperf` [211] and latency via `traceroute` [212]. The same process also controls α and β via traffic control utility `tc` to emulate different network settings (as explained later in §6.4). Based on α, β, M, N and CF c, the compression-communication mechanism with least cost is chosen as per Eqn. (6.2). Compression gain for each candidate CF is re-evaluated only in sensitive periods when the inter-iteration gain at current CF changes beyond a specified threshold. In order to do this, we first preserve the global model state via checkpoint-restore, select a candidate CF and run for a few iterations, log the average gain and compression time, then reload saved state from the checkpoint to evaluate the next candidate CF. The checkpoint-restore technique ensures that model quality does not degrade from evaluating very high CFs during the exploration phase.

6.4 Experimental Evaluation

We implement `AR-Top`k compression-communication mechanism in PyTorch v1.12.1 with NCCL v2.10.3 to enable communication between 8 inter-node GPUs. ResNet18 was trained over CIFAR100 for 50 epochs with per-worker batch-size 32, momentum 0.1, weight decay 5e-4, learning-rate 0.1 that decays by 10$\times$ at epochs 15, 30 and 45. ResNet50 trained on Food101 [182] for 100 epochs with batch-size 64, momentum 0.1, learning-rate 0.1 and weight decay 1e-4. AlexNet trained on CalTech101 [181] for 50 epochs with batch-size 32, momentum 0.9, weight decay 5e-4, learning-

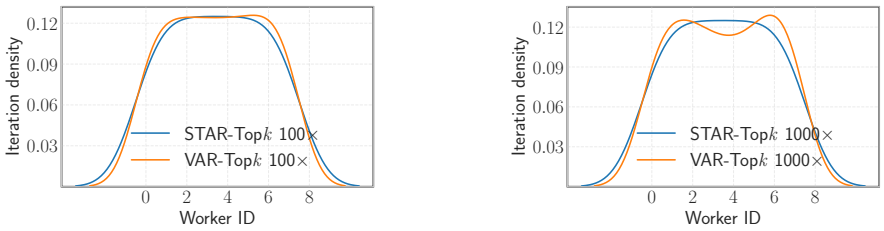
Model	Method	t_{step} (ms)	Acc.	Diff.
ResNet18	DenseSGD	146.21	90.79%	0.0%
	STAR-Top$k$ 10×	64.83	90.19%	-0.6%
	STAR-Top$k$ 100×	49.68	89.89%	-0.9%
	STAR-Top$k$ 1000×	48.17	88.81%	-1.98%
	VAR-Top$k$ 10×	77.2	90.15%	0.64%
	VAR-Top$k$ 100×	62.1	89.5%	-1.29%
	VAR-Top$k$ 1000×	60.5	88.7%	-2.09%
ResNet50	DenseSGD	294.35	98.72%	0.0%
	STAR-Top$k$ 10×	100.8	98.68%	-0.04%
	STAR-Top$k$ 100×	67.68	98.57%	-0.15%
	STAR-Top$k$ 1000×	64.37	98.38%	-0.34%
	VAR-Top$k$ 10×	113.4	97.65%	-1.07%
	VAR-Top$k$ 100×	80	98.2%	-0.52%
	VAR-Top$k$ 1000×	76.5	97.33%	-1.39%
AlexNet	DenseSGD	614	82.4%	0.0%
	STAR-Top$k$ 10×	129.5	81.74%	-0.66%
	STAR-Top$k$ 100×	50.3	80.41%	-1.99%
	STAR-Top$k$ 1000×	42.38	79.57%	-2.83%
	VAR-Top$k$ 10×	142	83.15%	+0.75%
	VAR-Top$k$ 100×	62.6	81.78%	-0.62%
	VAR-Top$k$ 1000×	54.4	78.06%	-4.34%
ViT	DenseSGD	1348.5	80.42%	0.0%
	STAR-Top$k$ 10×	276.32	79.72%	-0.7%
	STAR-Top$k$ 100×	104.13	79.18%	-1.24%
	STAR-Top$k$ 1000×	86.91	79.23%	-1.19%
	VAR-Top$k$ 10×	289.2	80.12%	-0.3%
	VAR-Top$k$ 100×	117	79.2%	-1.22%
	VAR-Top$k$ 1000×	99.7	79.8%	-0.62%

Table 6.4: Test accuracy, iteration time and difference over DenseSGD in STAR-Top$k$ and VAR-Top$k$ over a (4ms, 20Gbps) network.

rate 0.01 and decay factor 0.1 at epoch 25. Last, ViT was trained over CalTech256 [213] for 50 epochs with batch-size 32, momentum 0.9, weight decay 5e-4, learning-rate 0.01 and decay-rate 0.2 at epoch 40. While comparing different compression techniques and target CFs, we report the top-1 test accuracy and the time taken to complete a single training iteration (in milliseconds).

6.4.1 Training Speedup and Convergence Quality

Table (6.4) tabulates model convergence quality and time taken per-iteration in STAR-Top$k$, VAR-Top$k$ and DenseSGD. We train over a network limited to 4ms latency and 20Gbps bandwidth through `tc`, and configure NCCL to use tree-AR in DenseSGD by setting the environment vari-



(a) ViT with CF 100× (b) AlexNet with CF 1000×

Figure 6.5: Iteration density of 8 nodes (ranks 0-7) to broadcast top-*k* indices with STAR and VAR-Top*k* compression. STAR-Top*k* has similar distribution across all workers due to its round-robin nature, while some nodes have higher density than others in VAR-Top*k*.

able `NCCL_ALGO` to '`TREE`'. From the table, we see that accuracy decreases as CF increases across STAR and VAR-Top*k*. At the same CF, STAR-Top*k* has better performance than VAR-Top*k*. For e.g., ResNet50 at CF 1000× has 0.34% lower accuracy than DenseSGD with STAR-Top*k*, and about 1.39% lower accuracy with VAR-Top*k*. To understand the difference in generalization performance between the two (even at the same CF), we look at the iteration density (KDE) of each worker that broadcasts its top*k* indices during training. For the 8 workers with ranks or id between $[0, N-1]$ in Fig. (6.5a), we see similar iteration densities in ViT at CF 100× for the two `AR-Top`k variants, which also correlates to their similar accuracy levels of 79.18% and 79.2% in Table (6.4). For AlexNet at CF 1000×, STAR and VAR-Top*k* converged to 79.57% and 78.06% respectively (a considerable difference of 1.51%). In this case, iteration density of VAR-Top*k* is more skewed and non-uniform compared to STAR-Top*k*, shown in Fig. (6.5b). Worker ranks 1 and 6 broadcast considerably more updates than other workers, while rank 4 transmits its top-*k* gradients sporadically. The infrequent updates from rank 4 are stale and may degrade model quality. On the other hand, STAR-Top*k* has a uniform distribution due to its round-robin approach.

Another important aspect of VAR-Top*k* to consider is its higher iteration time than STAR-Top*k* due to the additional computational overhead associated with it (lines 10-13 in Alg. (1)). VAR-Top*k* computes gradient variance and calls AG to synchronize it across all workers, exchanging '$4N$' bytes of data in the process. The synchronization overhead of this call is low due to its small message size. *Although STAR-Topk performs better in terms of parallel performance (due to its lower compute overhead) and statistical efficiency (from its higher test accuracy), we conjecture that VAR-Topk can be viable when training on unbalanced and non-IID data, or when training over a*

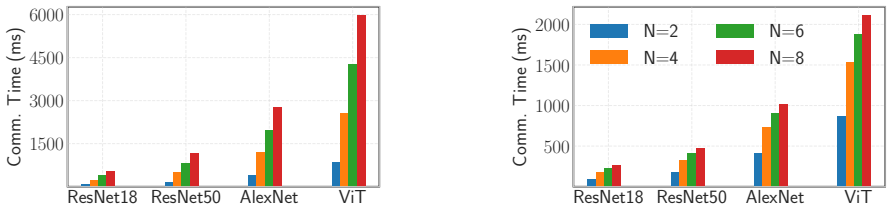
DNNs config.		t_{step} (ms)			Accuracy (%)		
Model	**CF**	**STAR**	**VAR**	**LWTop-*k***	**STAR**	**VAR**	**LWTop-*k***
ResNet18	10×	64.83	77.2	**62**	**90.19**	90.15	90.15
	100×	49.68	62.1	**38.4**	**89.89**	89.5	88.49
	1000×	48.17	60.5	**36.8**	**88.81**	88.7	87.2
ResNet50	10×	**100.8**	113.4	130.24	**98.68**	97.65	98.65
	100×	**67.68**	80	79.4	**98.57**	98.2	98.3
	1000×	**64.37**	76.5	72.7	**98.38**	97.33	96.55
AlexNet	10×	**129.5**	142	156.8	81.74	**83.15**	81.68
	100×	50.3	62.6	**33.45**	80.41	**81.78**	81.45
	1000×	42.38	54.4	**21.3**	**79.57**	78.06	78.9
ViT	10×	**276.32**	289.2	362.4	79.72	**80.12**	79
	100×	104.13	117	**94.64**	79.18	**79.2**	78.75
	1000×	86.91	99.7	**67.7**	79.23	**79.8**	78.25

Table 6.5: Comparing STAR-Top*k*, VAR-Top*k* and LWTop-*k* on ResNet18/50, AlexNet and ViT. AR-Top*k* variants use AR while LWTop-*k* uses AG for communication.

large number of workers. With unbalanced data, collecting updates from workers with high variance prioritizes critical updates. In massive clusters where 'N' is large, the round-robin updates of STAR-Topk may add excessive staleness and degrade DNNs' convergence quality.

Comparing `AR-Topk` *with other compressors:*

Table (6.5) shows the performance of the two variants of `AR-Topk` and LWTop-*k* at different CFs. The former uses Broadcast + AR while the latter uses AG to communicate its compressed updates. We limit the latency to 4ms while bandwidth is capped at 20Gbps. From the results, we see that STAR-Top*k* has a lower iteration time than LWTop-*k* in some cases, and higher in others. For e.g., STAR-Top*k* is faster across all CFs in ResNet50, and 10× in AlexNet and ViT. On the other hand, LWTop-*k* is faster across all CFs of ResNet18, 100× and 1000× of AlexNet and ViT. VAR-Top*k* has higher iteration costs across all configurations due to its additional compute overhead. Thus, AG is faster than `AR-Topk` in some cases and vice versa, based on network topology, latency, bandwidth, CF, model-size and cluster-size. `AR-Topk` applies *tensor fusion* prior compression, i.e., gradients are compressed as a whole across all layers while LWTop-*k* compresses on a layer-by-layer basis. By selecting the most significant updates across entire model instead of each layer, tensor fusion improves the convergence quality of `AR-Topk`. This is seen in STAR-Top*k* across all CFs of ResNet18 and ResNet50, while VAR-Top*k* converges to higher accuracy for most CFs in



(a) Allgather communication

(b) AR−Top$k$ communication

Figure 6.6: Communication cost increases steeply with 'N' in AG compared to AR−Top$k$, shown for CF 10× on 5ms latency and a low 1Gbps bandwidth network.

case of AlexNet and ViT.

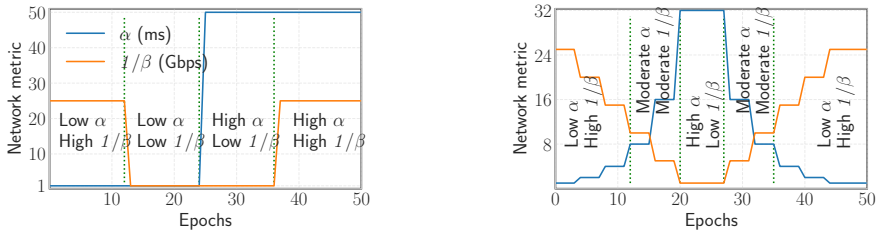
6.4.2 Measuring Communication Overhead

In this section, we measure the communication cost of AR−Top$k$ with ring and tree-based AR (i.e., ART-Ring and ART-Tree) and compare its synchronization costs with AG-based communication. The latency is limited to 1ms while bandwidth $1/\beta$ is varied between 10, 5 and 1Gbps on a cluster of 8 V100 GPUs. We fused tensors into buckets of 64 MB with PyTorch' gradient-bucketing and tabulate the communication cost in Table (6.6). At 10Gbps bandwidth and CF 10×, ART-Ring has the least communication overhead across all DNNs. On the other hand, AG is optimal at high CFs like 1000× over moderate bandwidths like 5 and 10Gbps. In low-bandwidth settings like 1Gbps, AR−Top$k$ is faster than AG since the AR collective in the former is bandwidth-optimal. ART-Ring is the optimal collective in low-latency, moderate-bandwidth (10Gbps) and low CF 10× settings as well. Thus, we see that the optimal compression-communication operation in distributed training depends on network latency, bandwidth, model-size, cluster-size and target CF. AG generally performs better when model-size is relatively small (like ResNet18) with moderate-to-high bandwidth or high CFs. In contrast, AR−Top$k$ with Ring or Tree-based reduction performs better on larger models in low-bandwidth settings, as well as low CFs on high-bandwidth networks.

Additionally, we compare the *scale-out* cost between AR−Top$k$ and AG as cluster-size increases. Fig. (6.6) shows the results for CF 10× as 'N' increases from 2 to 8 nodes on a network with an average latency of 5ms and a low 1Gbps bandwidth. The communication cost of AR−Top$k$ has a gradual incline, while that of AG collective increases much more sharply as N increases; in the former, the bandwidth-cost of broadcast and AR-tree increases logarithmically with N, while

Configuration		Communication Time (ms)			
Model	$(\alpha, 1/\beta)$	**CF**	**AG**	**ART-Ring**	**ART-Tree**
ResNet18	(1,10)	10×	54	**35**	43.2
		100×	**7.66**	18.1	12.2
		1000×	**3.28**	16.7	9
	(1,5)	10×	107.76	**52.5**	76.3
		100×	**13.83**	20.8	16.1
		1000×	**4.25**	17.9	10.1
	(1,1)	10×	526.3	**194.7**	345.6
		100×	51.93	**34.1**	41.9
		1000×	**8.86**	19.5	12.8
ResNet50	(1,10)	10×	115.1	**52.9**	83.4
		100×	**14.35**	20.3	15.9
		1000×	**4.65**	18.1	10
	(1,5)	10×	232	**94.7**	156.2
		100×	28.1	26.1	**24.2**
		1000×	**5.3**	17.8	10.5
	(1,1)	10×	1148	**405.5**	745
		100×	126.5	**58.8**	83.7
		1000×	**14.35**	21	16.1
AlexNet	(1,10)	10×	271.8	**106.8**	180.4
		100×	32.73	**25.2**	25.8
		1000×	**6**	18.6	11.1
	(1,5)	10×	544.5	**200.4**	354.8
		100×	61.75	**34.8**	42.6
		1000×	**8.92**	19.3	13.1
	(1,1)	10×	2718.7	**964.4**	1778
		100×	282.7	**111.8**	186.8
		1000×	31.33	**27**	27.3
ViT	(1,10)	10×	592.77	**238.6**	401.2
		100×	68.48	**36.2**	46.2
		1000×	**9.15**	19.2	12.9
	(1,5)	10×	1206	**424.3**	779.1
		100×	127.45	**58**	86.2
		1000×	**15.3**	21.4	16.9
	(1,1)	10×	5973	**2047**	3852
		100×	601.8	**222.8**	385.2
		1000×	59.68	**36.7**	44.4

Table 6.6: Communication cost in fixed latency (α) and variable bandwidth ($1/\beta$) for AR-Top*k* with Ring-AR and Tree-AR, as well as communicating updates via AG.



(a) Network configuration C_1 (b) Network configuration C_2

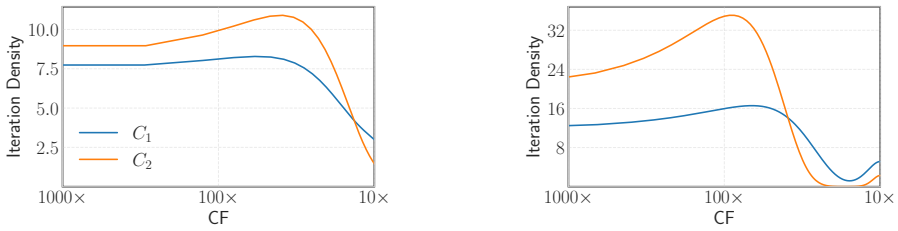
Figure 6.7: Latency and bandwidth is varied over the epochs via traffic control (`tc`) to emulate different network settings with low, moderate and high α-β parameters.

AR-ring is near-independent of N. On the other hand, bandwidth overhead of AG increases linearly with cluster-size.

6.4.3 MOO Configuration and Performance

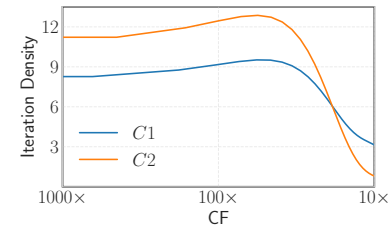
We assign $c_{low} = 10\times$, $c_{high} = 1000\times$, and increase CFs by a factor of 3.0 starting from c_{low}. Thus, the candidate CFs are $[10\times, 30\times, 90\times, 270\times, 810\times, 1000\times]$ such that each configuration is launched briefly in order to measure its compression gain, $t_{compress}$ and t_{sync} to formulate the MOO problem. The overhead of this exploration phase is low as its trained for only a few iterations and checkpoint-restore is performed in-memory (thus avoiding expensive disk read-writes). Evaluation is triggered when the inter-iteration gain of current CF changes by at least 10% (i.e., gain-threshold = 10%). Following this, we initiate the search for $c_{optimal}$ *only* if the network latency or bandwidth changes. This is done since changes in α and β influence the communication cost of a CF configuration. Based on the current network and training parameters, the optimal collective with least communication cost is chosen. If `AR-Top`k is chosen over AG, we save the current model state and select ART-Ring or ART-Tree (based on which is faster as per Eqn. (6.2a)) by setting `NCCL_ALGO` environment variable to `RING` or `TREE`, and deploying the model from last saved state. We use the NSGA-II algorithm (Non-dominated Sorting Genetic Algorithm) implemented in *pymoo* MOO framework [214] to find $c_{optimal}$ using the data gathered across all candidate CFs.

Fig. (6.7) shows configurations C_1 and C_2, such that each training epoch has a specific latency and bandwidth set via `tc`. In particular, we emulate four scenarios in C_1: (*low*-α, *high*-$1/\beta$) from epoch 1-12, (*low*-α, *low*-$1/\beta$) from 13-24, (*high*-α, *low*-$1/\beta$) from 25-36 and (*high*-α, *high*-$1/\beta$) thereafter. In our evaluation, we consider 1ms as low and 50ms as high latency, while inter-node

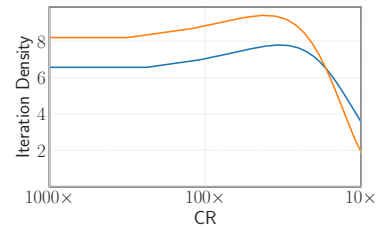


(a) ResNet18

(b) ResNet50



(c) AlexNet



(d) ViT

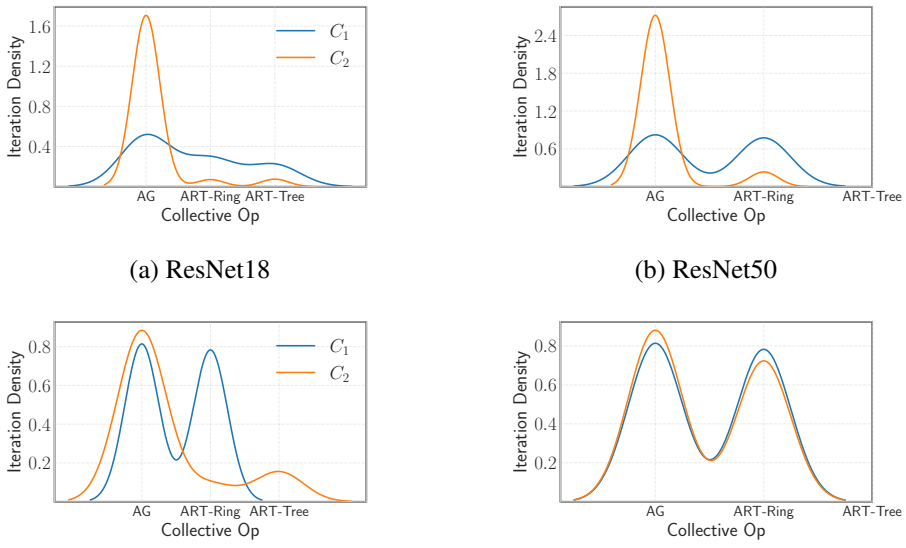
Figure 6.8: Iteration density distributions of different CFs used when training with C_1 and C_2 network configurations when gain-threshold is 10%.

bandwidth of 1 and 25Gbps is regarded as low and high respectively. Configuration C_2 simulates settings of (*moderate*-α, *moderate*-$1/\beta$) between epochs 12-19 and 28-35, (*high*-α, *low*-$1/\beta$) between 20-27 and (*low*-α, *high*-$1/\beta$) between epochs 0-11 and 36 onwards. These schedules are applicable for ResNet18, AlexNet and ViT that train for 50 epochs. For ResNet50 that runs for 100 epochs, we scale the epoch length for each (α, $1/\beta$) configuration by 2×. In this way, C_1 applies (*low*-α, *high*-$1/\beta$) state from epochs 1-24 (instead of 1-12). Similarly, C_2 on ResNet50 exhibits (*high*-α, *low*-$1/\beta$) between epochs 40-54, and so forth.

Performance of MOO-based Compression:

For C_1 and C_2, ResNet18 converged to 89.88% and 90.1% accuracy respectively, while ResNet50 reached 98.56% and 98.62%. AlexNet reached 82.4% accuracy on C_1 and 83.6% on C_2, while ViT attained 80.75% and 81.2% test accuracy across the two configurations.

By comparing convergence quality with 'static' CFs in `AR-Topk` (from Table (6.4)) or AG-based LWTop-*k* and MSTop-*k* (from Table (6.3)), we see that dynamically changing CF while cognizant of the compression gain achieves good convergence quality. In the larger AlexNet and ViT models, MOO-based training achieves slightly better accuracy than even DenseSGD as flexible



(a) ResNet18

(b) ResNet50

(c) AlexNet

(d) ViT

Figure 6.9: Density distributions of AG, ART-Ring and ART-Tree collectives used over training with configurations C_1, C_2 and a gain-threshold of 10%.

CFs allow for more exploration of local minima on workers which improves generalization performance. Fig. (6.8) plots iteration KDEs of different CFs used over training across the two network configurations. The CFs chosen by MOO are the three-dimensional knee-point or pareto-front of compression time, communication time and compression gain, and lies in the min-max range [c_{low}, c_{high}]. Configuration C_2 has a higher density than C_1 as search for $c_{optimal}$ is triggered only when network latency or bandwidth changes over the epochs, and C_2 changes more often than C_1 (as seen in Fig. (6.7)). The iteration density peaks between CFs 100× and 1000× for all DNNs, implying most iterations used CFs in that range for majority of the training phase.

Based on the α-β cost model, communication collectives have different associated costs based on network configuration, target CF, DL model and cluster-size. While optimizing for $c_{optimal}$, we dynamically choose the collective with least communication overhead as well. Fig. (6.9) shows the iteration densities of different collectives used throughout training as network performance varies in C_1 and C_2. For smaller models like ResNet18 and ResNet50, training under C_2 configuration uses AG over `AR-Top`k for most iterations. A major chunk of C_2 comprises of (*low*-α, *high*-$1/\beta$) and *moderate*-(α, $1/\beta$) phase. In Table (6.6), we observed that both residual models use AG at CFs 100× and 1000× over a 1ms, 10Gbps network (a low-latency, moderate-to-high bandwidth

configuration). For ResNet50 and ViT, ART-Ring is chosen over ART-Tree since the latter has a higher synchronization overhead. As a result, iteration density of ART-Tree is zero in Fig. (6.9b) and (6.9d). Larger models like AlexNet and ViT use both AG and ART-Ring, as similarly observed from the communication overhead of different (α, β) settings in Table (6.6).

6.5 Conclusion

Unlike conventional parallel applications, not all work performed in training DNNs holds equal contribution towards final model quality. This opens up new avenues for adaptive communication to attain high training speedups [31, 33, 34]. Gradient compression offers trade-offs between speedup and convergence quality based on the degree of compression. We model this relationship as a multi-objective optimization problem to minimize compression and communication cost (parallel efficiency) while maximizing compression gain (i.e., statistical efficiency) in DDL. Communication cost is influenced by factors like network topology, latency, bandwidth, model-size, cluster-size and the degree of compression (CF). Network performance may also fluctuate due to congestion, contention, scheduling, etc. and thus degrade training performance. This chapter presents an adaptive mechanism to predict the optimal compression-communication configuration over variable and unpredictable networks. Based on the α-β cost-model and other factors, we intelligently switch between AG and `AR-Top`k (with different versions of AR), and dynamically choose the optimal CF to attain high convergence quality and training speedups.

The average DL practitioner may not necessarily be aware of the network topology when deploying DNNs over a cluster of nodes, or the network configuration may vary with training infrastructure (for instance, training in a private cluster vs. the cloud). Our system keeps track of the current network latency and available bandwidth to determine the ideal collective operation that incurs the least overhead based on the latency-bandwidth communication cost-model. Apart from Top-*k*, our approach is compatible with other gradient compressors (such as DGC [115], SIDCo [117], etc.) and enables adaptive training with other sota techniques.

6.6 Publications

1. **Tyagi, Sahil** and Swany, Martin. “Flexible Communication for Optimal Distributed Learning over Unpredictable Networks.” 2023 IEEE International Conference on Big Data (BigData) (2023): 925-935.

CHAPTER 7
REAL-TIME LEARNING OVER STREAMING DATA

7.1 Introduction

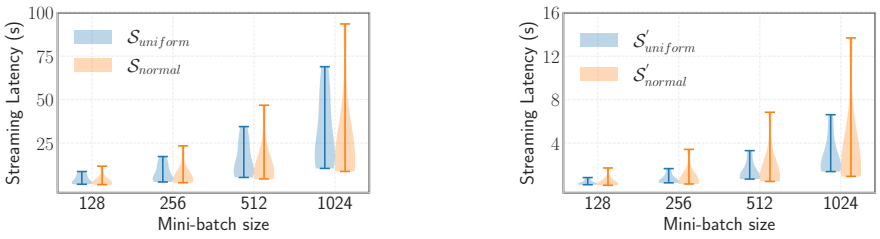
Distributed learning in real-time over streaming data faces challenges like heterogeneous compute resources and streaming-rates, limited network bandwidth, finite memory and on-device storage, as well as unbalanced and non-IID data. In this chapter, we propose `ScaFLES`: {**Sca**}lable {**F**}ederated {**L**}earning over {**S**}treaming data at the {**E**}dge, a FL framework for streaming data in an online manner that addresses the aforementioned issues to provide significant training speedups and achieve high convergence quality.

7.2 Motivation

To understand the challenges of synchronous distributed training (i.e., BSP) over streaming data, we simulate heterogeneous data-streams over Apache Kafka [215] and PyTorch [98, 204], and explain its impact on training time under the effects of data-skewness, limited compute resources and expensive communication from aggregating updates of large models over slow networks.

7.2.1 Heterogeneous Data-Streams

By Eqn. (2.2a) of BSP data-parallel training, workers with local batch-size 'b' cumulatively run a global batch-size 'Nb' when training across 'N' devices. However, for real-time execution over streaming data [216, 217, 218], clients may have different streaming-rates. Devices with high-volume streams may instantaneously collect 'b' samples, while low-stream clients need to wait until 'b' samples are accumulated before performing model updates. A client with an inflow-rate of 's' samples/sec. would thus need to wait about (b/s) seconds. We regard such variances in device stream-rates as a source of *streaming heterogeneity*. These variances can be dynamic as well, varying based on application traffic, usage, time of day, etc. To understand how streaming heterogeneity can affect wall-clock training time in DL due to latency incurred while gathering a



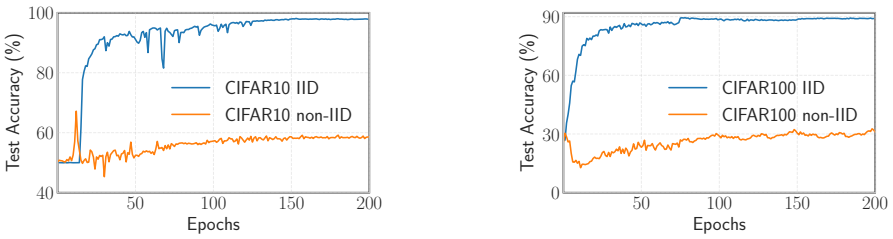
(a) (b)

Figure 7.1: Streaming latency incurred to aggregate different batches when worker stream-rates are sampled from uniform and normal distributions with a mean and standard deviation of (a) 50 (b) 500.

specific mini-batch, we sample stream-rates from different distributions and measure the latency incurred to gather a mini-batch.

We use two sets of uniform and normal distributions and sample possible device streaming-rates with varying degrees of heterogeneity. These distributions capture inflow heterogeneity commonly observed in real-world settings. $\mathcal{S}_{uniform}$ and $\mathcal{S}'_{uniform}$ sample stream-rates from a uniform distribution with a mean/standard deviation of 50 and 500 respectively. On the other hand, $\mathcal{S}_{normal}$ is sampled from a normal distribution with a mean and standard deviation of 50, while $\mathcal{S}'_{normal}$ has a mean and standard deviation of 500. If enough samples stream into a device to meet its training batch-size, then there will not be any wait times. However, heterogeneous streams with varying inflow-rates incur additional latency to accumulate sufficient samples corresponding to its batch-size. For the chosen distributions, We plot the variance in streaming latency at different local batch-sizes for [$\mathcal{S}_{uniform}$, $\mathcal{S}_{normal}$] and [$\mathcal{S}'_{uniform}$, $\mathcal{S}'_{normal}$] in Fig. (7.1). As batch-size increases, more time is spent accumulating the samples and thus, higher is the latency. Normal distributions tend to be centered around the mean while uniformly sampled rates are spread evenly over the min-max range of the distribution.

On account of heterogeneous data-streams, devices with lowest streaming-rate (and thus, most latency) effectively act as stragglers, causing other devices to halt training while it gathers a mini-batch, performs computations and initiates aggregation across clients.



(a) CIFAR10 on ResNet152

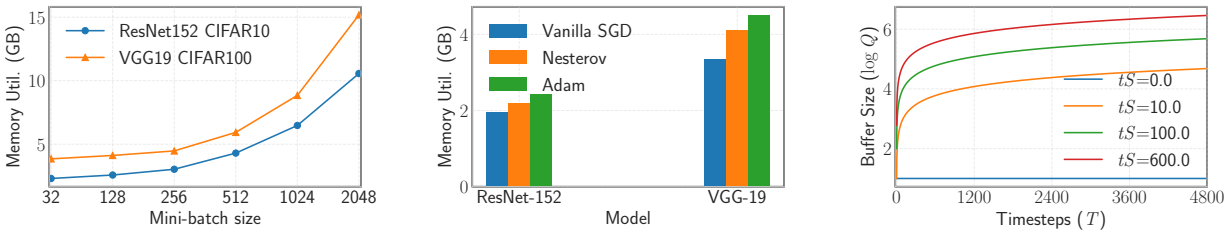
(b) CIFAR100 on VGG19

Figure 7.2: Convergence quality with training over IID and non-IID data. Test accuracy falls substantially over non-IID data.

7.2.2 Skewed and Unbalanced Training Data

In collaborative learning, data across the devices can be skewed either in volume, properties, or both. For e.g., data imbalance due to volume can arise in a traffic surveillance system where different cameras capture IID data such as frames of individual vehicles (like cars, bicycles, trucks, etc.) but the volume of data on each device varies with the traffic density on the route where the camera is installed. Skewness arises when the distribution of local device data deviates considerably from the overall distribution. For instance, a vehicle recognition model deployed over a video surveillance feed installed in a subway system collects images of trains, while devices installed on the airport may only cover aeronautical vehicles. In such scenarios, training over non-IID data degrades convergence quality as each device contains only partial labels (like only trains or planes in the previous example). Privacy and security concerns, as well as its large volume make it unfeasible to move data to a centralized location, especially over constrained networks.

We run two image classification tasks, ResNet152 [9] and VGG19 [8] over non-IID data and see its impact on convergence quality compared to IID data. Skewness is introduced in CIFAR10 and CIFAR100 datasets [180] by mapping a subset of labels to a unique device. We train over a cluster of 10 devices such that each one of the 10 labels of CIFAR10 resides on a different device, while the 100 labels of CIFAR100 are mapped such that each device contains 10 unique labels. As seen in Fig. (7.2), test accuracy degrades considerably when training on unbalanced and non-IID data; ResNet152 on IID CIFAR10 converged to 98% while non-IID version saturated to only 59% accuracy. VGG19 on IID CIFAR100 data attained 90.3% while non-IID CIFAR100 did not improve beyond 32% test accuracy.



(a) Cumulative memory (b) Optimizer memory (c) Buffering queue

Figure 7.3: (a) Memory utilization increases with batch-size (b) Memory consumption changes with the type of optimization used (c) Buffering queue grows over time when computations cannot be performed at line-rate.

7.2.3 Limited Memory and Storage

Under high-volume data-streams, further processing and storage can get costly or downright unfeasible due to limited resources. Finite memory presents challenges even in datacenters when the available CPU or GPU memory is significantly lower than what is needed. Training DNNs requires memory to store model parameters, gradients (same size as trainable parameters), intermediate activation maps and training batches [178, 219]. With the model, dataset and optimizer-type fixed, only the activation and batch memory change with batch-size. We see in Fig. (7.3a) the GPU memory utilization on NVIDIA V100 [220] with different batches as we trained ResNet152 on CIFAR100 and VGG19 on CIFAR100. The memory consumption to train these models rises with larger batches. Additionally, training on limited memory systems gets even more challenging as we use heavier optimization techniques. For e.g., as we move from vanilla-SGD to keeping moving average of previous iterations' gradients in momentum-SGD [134], to tracking gradient variance with Adam optimizer [136], the memory utilization rises as well. Fig. (7.3b) plots the total memory utilization of ResNet152 and VGG19 for a fixed batch-size with different SGD optimization algorithms.

As DL can be compute-intensive, it is difficult to train models over streaming data at 'line-rate'. As a consequence, data can quickly accumulate if the device stream-rate is higher than its processing rate. The size of streaming or buffering queue can grow exponentially when learning over real-time streams, either on-disk or in-memory. For instance, client 'c' among 'N' devices has a streaming-rate of S_c samples/sec. and processes a batch $b_c = \sum_{j=1}^{N}(\frac{b_j}{N})$ on-average in synchronous training. In a setting where the device stream-rate is larger than the average batch-size, we have $|S_c| > b_c$.

Each training iteration is comprised of computing loss and gradients, aggregating and applying model updates; let's denote this iteration time as t_c on device 'c'. At the initial timestep $ts = 0$, S_c number of samples flow in every second on client 'c', which then processes b_c samples from it. In the time t_c that client 'c' completes an iteration, about $(t_c \cdot S_c)$ more samples arrive in addition to the residual $(S_c - b_c)$ that were not processed in the last iteration. Thus, there are $(S_c - b_c) + t_c \cdot S_c$ samples enqueued in the streaming buffer at timestep $ts = 1$. At $ts = 2$, there are $2(t_c + 1)S_c - 2b_c$ samples in the buffering queue, and so on.

$$Q_c = (t_c \cdot S_c - b_c) \cdot T + S_c \quad \forall \quad t_c \cdot S_c \geq b_c \tag{7.1}$$

The queue size thus increases over time due to residual samples accumulated over the previous timesteps. We can generalize the total number of accumulated samples Q_c on device 'c' after T timesteps as per Eqn. (7.1) and note that Q_c grows linearly with T.

As DNNs typically run for many iterations before reaching acceptable accuracy levels, the buffer size can rise dramatically when T is large. To limit Q_c from flaring up, we could theoretically set b_c to $t_c \cdot S_c$. As seen by Eqn. (7.1), $Q_c = S_c$ irrespective of T in that case. However, b_c is a sensitive training hyperparameter that may require careful tuning. Further, using $t_c \cdot S_c$ as the batch-size can be impractical if the stream-rate is too high, or inefficient if stream-rate is too low. This is because a small batch does not leverage parallelism while a larger batch may hurt generalization performance [140, 141].

$$Q_c = (T \cdot t_c \cdot S_c + S_c) \quad \text{if} \quad (t_c \cdot S_c) \gg b_c \tag{7.2}$$

Assuming high stream-rates and considerable iteration cost in distributed synchronous training due to limited compute and bandwidth, Eqn. (7.1) can be reduced to Eqn. (7.2). Using this, we simulate how the buffering queue Q_c rises with T. For different 'tS', we plot queue-size on the log-scale as a function of timesteps in Fig. (7.3c). As 'tS' gets larger, so does its corresponding buffering queue. When $tS \approx 0$, the buffer holds exactly S samples at any given time; a hypothetical setting where total iteration time is negligible regardless of the device's streaming-rate.

To gauge the buffer requirements of inflow streams in real-world settings, we measure the space needed to store (32×32) colored images for training ResNet152 and VGG19. With mini-batch

DNNs	t (s)	S (img/s)	Data accumulated at T steps (GB)		
			$T = 10^3$	$T = 10^4$	$T = 10^5$
ResNet152	1.2	100	0.35	3.5	34.33
		600	2.06	20.6	200.6
VGG19	1.6	100	0.47	4.69	46.8
		600	2.75	27.5	274.83

Table 7.1: Data accumulated in the streaming buffer over time

size 64, the models have average iteration times of 1.2 and 1.6 seconds respectively on the same hardware. Table (7.1) tabulates how many samples are accumulated after 10^3, 10^4 and 10^5 timesteps for the two DNNs. As T increases, so does the storage requirements for data persistence. Distributed stream processing platforms like Kafka [215] reduce memory footprint by storing messages on-disk as partitions, then deleting the data based on some retention policy once the messages are successfully consumed. However, persisting data even on-disk may be expensive (due to high I/O overhead) or become unfeasible under limited storage as data may keep accumulating over time.

7.2.4 Synchronization Cost Over Limited Bandwidth

Although training over fast processors or accelerators can significantly reduce computation time, periodic aggregation of model updates incurs significant overhead. In a client-server setting, communication cost tends to increase with the total number of devices participating in training. Improving network bandwidth lowers the synchronization cost to only a certain extent, and saturates thereafter [201].

7.3 Related Work

Training DL models with a focus on privacy, unbalanced and non-IID data has been studied in FL, where devices train on local data and a globally shared model is learned by periodically aggregating updates [25, 89]. STC [128] combines gradient sparsification [112] with quantization [121] to reduce communication overhead and is shown to perform better than popular federated algorithms under IID and non-IID data settings. Data skewness has prior been studied as a weight divergence problem between local models and measured as the distance between device-local and overall data distribution [83].

The compute requirements grows in proportion to the size of sota DNNs. To reduce the memory footprint, [120] proposed automatic mixed-precision training and use 16-bit floating point representation to store gradients. Gradient checkpointing [221] trades memory for computation by flushing intermediate data from the computation graph to reduce memory utilization, although this comes at the cost of increased computation as flushed activation maps need to be recomputed as needed. Memory consumption can also be lowered by using smaller batches, although this affects parallelizability and increases the overall training time. DL frameworks like PyTorch implement `DatasetFolder` and `ImageFolder` [98] to read multiple samples of batches from the disk at a time and avoid loading entire training data into memory. However, such dataloaders adhere to a rigid format for specifying labels in the underlying directory structure. Thus, training DNNs on a specific dataset may require elaborate preprocessing and even the on-disk storage can quickly fill up on massive datasets.

Federated algorithms like FedAvg [25] and FedProx [89] minimize communication overhead by choosing a low-frequency, high-volume communication strategy that periodically aggregates model updates. On the other hand, gradient compression techniques use a high-frequency, low-communication approach to reduce the tensor volume for data exchange among clients [115, 128, 121, 112].

7.4 `ScaFLES` Design

We propose `ScaFLES`: {**Sca**}lable {**F**}ederated {**L**}earning over {**S**}treaming data at the {**E**}dge, a FL framework to address the aforementioned challenges and accelerate convergence over IID and non-IID heterogeneous data-streams in near-real time.

7.4.1 Improving Parallel and Statistical Performance over Heterogeneous Streams

The works described in §7.3 either consider systems or statistical heterogeneity; the former from variance in computational resources or stream-rates between the devices and the latter from skewed and non-IID data. Synchronous training on devices with heterogeneous stream-rates can result in stragglers depending on the chosen batch-size and devices with low stream-rates. The bottleneck arises from streaming latency to accumulate b_c samples on client 'c' to perform SGD update. At the

edge, stream-rates can vary at an intra-device level as well, depending on factors like battery level, usage, time of day, etc. The wait-time incurred from streaming latency can thus slow down training.

To mitigate the effects of low-inflow streams, we perform variable computations on devices such that we set $b_c \propto S_c \ \forall \ c$. That means we mitigate latency by setting device batch-size to its streaming-rate. Thus, devices with high-volume streams train on larger batches while low-volume streams use relatively smaller batches, eliminating wait times due to streaming latency. Since local batch-sizes are different, the amount of work performed on each device is different as well. To preserve statistical performance, device with larger batches should contribute more towards the final model update (due to large-batches being less noisy than small-batches [40]). Hence, we perform weighted aggregation instead of simply averaging the model updates.

$$r_c^{(i)} = \frac{S_c^{(i)}}{\sum_{j=1}^{N} S_j^{(i)}} \quad \text{such that} \quad \sum_{j=1}^{N} r_j^{(i)} = 1.0 \tag{7.3a}$$

$$\tilde{g}_i = \sum_{j=1}^{N} r_j^{(i)} \cdot g_j^{(i)} \tag{7.3b}$$

$$\eta_{scaled}^{(i)} = \eta \cdot \gamma_{scaled}^{(i)} \quad \text{where} \quad \gamma_{scaled}^{(i)} = \frac{\sum_{j=1}^{N} S_j^{(i)}}{B} \tag{7.3c}$$

$$w_{i+1} = w_i - \eta_{scaled}^{(i)} \cdot \tilde{g}_i \tag{7.3d}$$

At iteration 'i', client 'c' trains with a batch-size corresponding to its streaming-rate $S_c^{(i)}$, scales its local gradients $g_c^{(i)}$ by factor $r_c^{(i)}$ and updates its parameters as shown in Eqn. (7.3). Here, the global batch-size for the current iteration is $\sum_{j=1}^{N} S_j^{(i)}$. As stream-rates can vary both inter- and intra-device, so does the global batch-size. To ensure that high stream-rates don't excessively increase the batch-size and degrade final model accuracy, we add a linear scaling rule for the learning-rate parameter [61, 156]. A learning-rate schedule η corresponds to global batch-size B, so the step-size routine in case of heterogeneous data-streams is adjusted proportional to the batch-size (Eqn. (7.3c)). When the batch-size grows by some factor, we multiply the learning-rate with the same scalar in order to keep the scale of SGD update fixed (relative to the global batch-size and preserve model generalization quality).

Even with the linear scaling rule, DNNs quality may still suffer from large-batch training under high volume streams. We thus set a threshold on the maximum batch-size to use for training, b_{max}.

7.4.2 Handling Limited Memory and Storage

The buffering queue can grow over time under continuous data-streams and due to non-negligible iteration costs at the edge. The accumulated data can either reside in memory like a buffered queue or on-disk. By default, we can persist all the data flowing in until its processed successfully, and then discarded. We refer to this policy as '*Stream Persistence*'. However, as we saw in Table (7.1), accumulated samples grow proportionally with timesteps under this policy (i.e., $\mathcal{O}(S_cT)$ complexity). Stream persistence is especially viable when devices have large memory or storage to hold the data, like in data-centers, cloud, or high-end fog devices. However, in '*Stream Truncation*' policy, we discard the residuals and hold just enough data corresponding to the device stream-rate. Thus, storage requirements with truncation is just $\mathcal{O}(S_c)$ at any given time, significantly lower than stream persistence.

7.4.3 Convergence Quality over Unbalanced and Non-IID Data

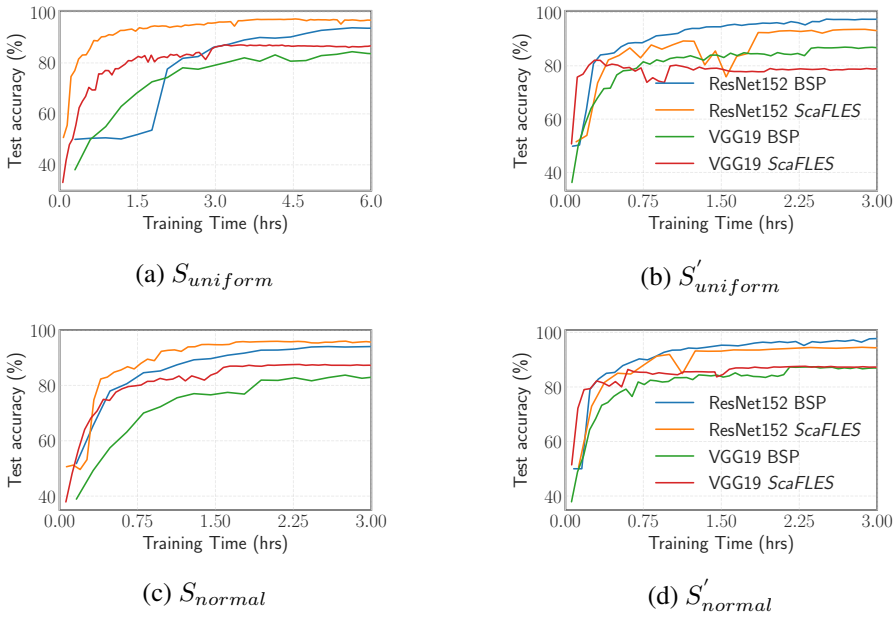
Fig. (7.2) showed how model quality degrades when the device-local data is not representative of the overall data distribution. As a result, features learned by the local replicas are skewed. To improve convergence over unbalanced and non-IID data, we propose '*stochastic data-injection*' where a subset of devices share partial training samples with other devices in the network. Specifically, at every iteration a client randomly chooses fraction ϕ of the total set of devices $\{C\}$ (i.e., degree of nodes) and shares ψ fraction of its streaming data $\psi S_c \rightarrow [\phi C]$ (i.e., degree of data). Together, variables (ϕ, ψ) determine the degree of nodes sharing a certain fraction of their data with other devices in the network. Data-injection thus improves the overall data distribution in the cluster by making device-local datasets more representative of the overall distribution. However, this implies a trade-off between high model quality from better data distribution and privacy concerns arising from moving data between clients. But privacy violations are greatly minimized by choosing only a subset of devices randomly (based on ϕ) at every iteration and sharing only partial training samples (based on ψ).

7.4.4 Handling High Communication Cost

Gradient compression with a larger, static CF may not necessarily yield the same accuracy levels as BSP. As described previously, DNNs' sensitivity can be high in the early phases of training and at certain critical regions [27, 28, 149, 131]. Based on this, we use a dynamic compression technique for FL that uses low CF over sensitive phases, and a high CF otherwise. Extending from Chapter (5), we compare the information or entropy loss between prior and post-compression gradients to detect these phases on top of Top-*k* compression. If the deviation between compressed and uncompressed gradients is below a user-defined threshold ξ, we communicate compressed updates, otherwise send original gradients. The intuition is that if the largest $k\%$ gradients retain as much information as the original gradients within the margin of ξ, remaining gradients can likely be ignored as they don't contribute greatly towards the overall model update. If the compressor is removing excessive information that may not update the model in a significant way or even detriment training, it is fitting to send the original gradients instead.

$$\text{send}(\text{Top}k(g_c)) \quad \text{if} \quad \left|\frac{|g_c|^2 - |\text{Top}k(g_c)|^2}{|g_c|^2}\right| \leq \xi \quad \text{else} \quad \text{send}(g_c) \tag{7.4}$$

In Eqn. (7.4), we keep a moving average of the $\mathbb{L}$-2 norm of the original gradients g_c and its compressed update on client 'c'. The *compression threshold*, denoted by ξ decides the degree of relaxation we want to impose for the compressed tensors to be eligible for communication. A small value of ξ penalizes compressed updates more severely and performs reduction only when the compressed tensors capture most of the information, while constraints are loose with a larger ξ so it communicates compressed updates even if the information loss from original gradients is higher. With this high-frequency, low-volume adaptive communication approach, we can vary the trade-offs between parallel and statistical efficiency in FL by adjusting the threshold. A small ξ thus favors statistical efficiency and prioritizes convergence quality, while a large ξ improves parallel scaling by reducing communication overhead at the risk of lower model accuracy (from compressed updates).



(a) $S_{uniform}$ (b) $S'_{uniform}$

(c) S_{normal} (d) S'_{normal}

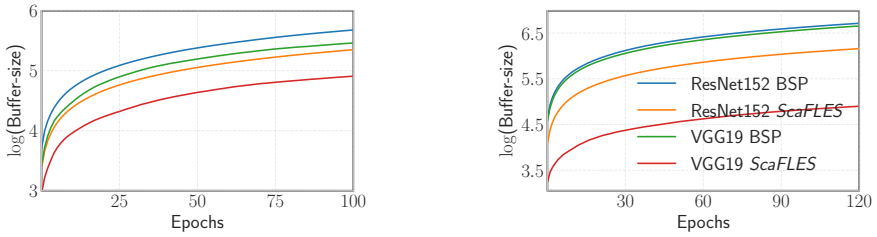
Figure 7.4: Convergence in BSP and ScaFLES with weighted gradient aggregation based on stream-rates and learning-rate scaling. Stream-rates are sampled from the distributions described in §7.2.1.

7.5 Performance and Evaluation

In this section, we evaluate the parallel and statistical performance of ScaFLES under varying streaming configurations commonly seen in FL on the edge. The cluster setup and training schedule is described in Appendix (C.1), while the simulator to emulate heterogeneous data-streams is described in Appendix (C.2).

7.5.1 Convergence Quality over Heterogeneous Streams

We now look at the convergence quality of ResNet152 and VGG19 where we set the training batch-size corresponding to a devices' current stream-rate. To minimize loss of generalization, we enforce batch-size bounds by setting the maximum to 1024. We evaluate the convergence of ScaFLES' weighted aggregation and learning-rate scaling over heterogeneous streams, and compare with BSP training where updates are averaged and communicated across all clients at every iteration. The cluster setup and hyperparameters are described in Appendix (C.1). In BSP training, we set client batch-size to 64, and a global batch-size $B = 64 \times N$ for learning-rate scaling across $N = 16$ clients in ScaFLES. We sample stream-rates for the N clients from the different uniform and normal



(a) $S_{uniform}$ (b) $S^{'}_{uniform}$

(c) S_{normal} (d) $S^{'}_{normal}$

Figure 7.5: Buffer-size grows over the iterations for different streaming distributions.

distributions highlighted in §7.2.1. Generally, uniform distributions tend to be more heterogeneous than normal, as the former are even distributed over a specified range while 67% values lie within 1 standard deviation from the mean in the latter.

Fig. (7.4a) illustrates the convergence quality of ScaFLES and BSP where device stream-rates are sampled from $S_{uniform}$. We see that ResNet152 and VGG19 converge up to 3.33× and 1.92× faster in ScaFLES to achieve the same accuracy-levels as BSP, and even attaining slightly higher accuracy in the end. With $S^{'}_{uniform}$ in Fig. (7.4b), final model quality of ResNet152 and VGG19 was worse than BSP due to large-batch training performed in ScaFLES under this distribution; global batch-size was $\approx$ 4500 in ScaFLES compared to only 1024 in BSP. Learning-rate scaling on batches this large did not improve convergence quality over BSP in this case. Streaming-rates sampled from S_{normal} (Fig. (7.4c)) were 3.6-4× faster to converge with ScaFLES' weighted aggregation and reached higher accuracy. Large batches with LR scaling was beneficial from a convergence standpoint in this case. Last, Fig. (7.4d) observed similar accuracy levels between ScaFLES and BSP for VGG19, while ResNet152 over BSP attained slightly better accuracy than ScaFLES over the normally distributed data-streams $S^{'}_{normal}$.We see good convergence quality with ScaFLES in normal distribution since most of the sampled rates are close to the mean in this case.

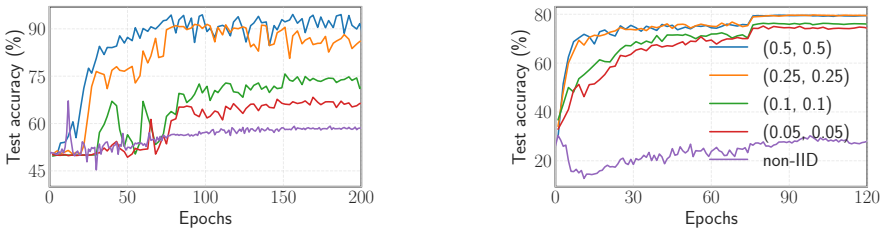
Distribution	Model	Persistence	Truncation	Reduction
$S_{uniform}$	ResNet152	2.9×10^5	129	2238×
	VGG19	1×10^5	118	848×
$S'_{uniform}$	ResNet152	4.36×10^6	633	6889×
	VGG19	4×10^6	523	7830×
S_{normal}	ResNet152	6.2×10^5	143	4340×
	VGG19	3.7×10^5	129	2861×
S'_{normal}	ResNet152	3.6×10^6	384	9429×
	VGG19	2.5×10^6	360	6956×

Table 7.2: Buffer-size reduction with stream truncation policy

7.5.2 Space Savings with Storage Policies

We now look at the accumulation of streaming data into a device with the default persistence policy in ScaFLES and compare with BSP which uses a local batch-size of 64. In Fig. (7.5), we plot the number of samples accumulated (on the logscale) over the iterations and see that buffer-size is smaller in ScaFLES just by using batches proportional to stream-rates, while BSP used a fixed batch-size. The buffering queue in $S'_{uniform}$ and S'_{normal} is larger than $S_{uniform}$ and S_{normal} as the former represent streams with higher inflow-volumes. BSP occupies 2× and 3.5× more space in ResNet152 and VGG19 under $S_{uniform}$ distribution. ScaFLES holds 3.6× and 640× less storage than BSP for $S'_{uniform}$. On comparing $S'_{uniform}$ in Fig. (7.4b) and (7.5b), we see that training with larger batches in ScaFLES keeps the buffer queue small at the cost of lower final accuracy. For S_{normal}, we have 4.7-5× lesser storage footprint than conventional BSP, and about 3.9-42× smaller buffers in ScaFLES under S'_{normal} distribution.

From Fig. (7.5), we saw that ScaFLES accumulates considerably fewer samples than BSP in distributed training. We can further reduce the occupied buffers by stream truncation where the residual samples are discarded and only samples that can be processed at line-rate are processed. The buffer-size with truncation policy is thus constant and stays the same as long as the streaming-rate is continuous. On the other hand, queue size under persistence policy linearly grows over the iterations. Table (7.2) shows the final buffer-size with stream persistence and truncation policies for ResNet152 to achieve 95% accuracy, and for VGG19 to achieve 84% accuracy. Additionally, the table reports the reduction with truncation policy with respect to stream persistence. With stream truncation, we observed buffer reductions anywhere from 850-9400× depending on the streaming



(a) ResNet152 (b) VGG19

Figure 7.6: Model convergence with stochastic data-injection on various (ϕ, ψ) configurations. We add non-IID convergence curve for reference.

distribution.

7.5.3 Convergence with Data-Injection over Non-IID Data

Stochastic data-injection improves overall model quality over non-IID training while cognizant of the privacy risk. The degree of data-injection is determined by node and data sharing parameters, (ϕ, ψ) to select a subset of devices that share partial training samples. We evaluate four configurations of (ϕ, ψ) in `ScaFLES`: (0.5, 0.5), (0.25, 0.25), (0.1, 0.1) and (0.05, 0.05). A tuple of (0.5, 0.5) implies 50% of the devices are chosen randomly to share 50% of their batched samples at every iteration. Fig. (7.6) plots the convergence curves of different (ϕ, ψ) configurations in `ScaFLES` and shows how the convergence quality improves over non-IID training. We also observe that there is an inverse relationship between model quality and privacy risk in data-injection. A small value preserves privacy by sharing minimal data between devices at the cost of lower accuracy, and vice versa. Additionally, data-injection is not a zero-cost operation as there is additional network overhead incurred to exchange training samples (details covered in Appendix (C.3)).

7.5.4 Parallel and Statistical Performance of Dynamic Compression

Using the dynamic compression scheme described in §7.4.4, we measure performance gains by reduction in overall communication volume, i.e., cumulative single-precision floating points exchanged to achieve target accuracy. We evaluate ResNet152 and VGG19 with 230 MB and 548 MB of original gradient updates, and compress them to either CF 10$\times$ or 100$\times$. CF 10$\times$ reduces tensor size of the two models to 23 and 55 MB respectively, while CF 100$\times$ reduces gradient-size of ResNet152 to 2.3 and VGG19 to 5.5 MB.

DNN	CF	ξ	CNC	Accuracy	Floats exchanged
ResNet152	10×	0.1	0.29	97.55%	4.43×10^{11}
		0.2	0.99	96.81%	0.56×10^{11}
		0.3	1.0	98.41%	0.4×10^{11}
		0.4	1.0	98.57%	0.4×10^{11}
	100×	0.1	0	97.39%	6.02×10^{11}
		0.2	0.17	97.47%	4.99×10^{11}
		0.3	0.43	96.72%	2.56×10^{11}
		0.4	0.99	94.97%	6.32×10^{8}
VGG19	10×	0.1	0	85.45%	1.3×10^{12}
		0.2	0.08	84.74%	1.19×10^{12}
		0.3	1.0	81.91%	1.3×10^{10}
		0.4	1.0	81.78%	1.3×10^{10}
	100×	0.1	0	84.68%	1.3×10^{12}
		0.2	0	83.98%	1.3×10^{12}
		0.3	0	83.94%	1.3×10^{12}
		0.4	0.004	84.39%	1.29×10^{12}

Table 7.3: Communication savings with dynamic compression in ScaFLES

$$\text{CNC ratio} = \left(\frac{steps_{compressed}}{steps_{compressed} + steps_{uncompressed}} \right) \tag{7.5}$$

Since some iterations exchange compressed gradients, while others aggregate original updates, we measure overall communication savings with compression through *Compression to No-Compression ratio (CNC)*. From Eqn. (7.5), CNC ratio measures the fraction of iterations using compressed updates to the total iterations throughout training. A CNC of 0.0 means compressed updates were *not* used for even a single iteration during training, while a value of 1.0 implies that devices communicated compressed gradients at *every* iteration. We tabulate the results of ScaFLES with dynamic compression in Table (7.3) for various CF and ξ configurations. To recall, the compression threshold metric ξ decides the relaxation to impose on compressed gradients to be eligible for communication. A small ξ is more stern towards compressed updates, while a large ξ loosens up the constraints over the compressed tensors.

ResNet152 with CF 10× and any $\xi \geq 0.2$ is fast as it noticeably reduces the number of floats exchanged among devices. For $\xi = 0.1$, ResNet152 with either CFs communicated original gradients for most iterations; CF 10× used compression for 29% of the total steps, while CF 100× used uncompressed gradients throughout training. From a strictly parallel performance standpoint, CF 100× with $\xi = 0.4$ is the optimal configuration for ResNet152 as its uses higher compression and

DNNs	Distribution	Accuracy drop	Buffer-size (GB)	Speedup
ResNet152	$S_{uniform}$	-0.06%	0.6	$\mathbf{1.89}\times$
	$S^{'}_{uniform}$	-0.32%	5.9	$\mathbf{1.15}\times$
	S_{normal}	-0.13%	0.8	$\mathbf{3.29}\times$
	$S^{'}_{normal}$	-0.21%	4.03	$\mathbf{1.42}\times$
VGG19	$S_{uniform}$	-1.93%	0.26	$\mathbf{1.56}\times$
	$S^{'}_{uniform}$	-4.18%	3.91	$\mathbf{2.83}\times$
	S_{normal}	-2.03%	0.35	$\mathbf{2.06}\times$
	$S^{'}_{normal}$	-1.59%	2.58	$\mathbf{2.13}\times$

Table 7.4: ScaFLES parallel and statistical performance gains over BSP training.

makes it eligible for majority of the iterations (with its high ξ). However, we also observed a loss of statistical efficiency (i.e., model accuracy) at that configuration. This pattern of low communication cost accompanied with lower final accuracy was also observed in VGG19 for CF $10\times$ at $\xi = 0.3$ and 0.4. A high CNC in both the cases implies most of the iterations exchanged compressed updates and so the accuracy drop is attributed to model degradation from sharing sparse updates in gradient compression. An interesting observation was VGG19 with CF $100\times$ where communication volume was the same across all configurations. This is because training iterations at that CF did not meet any of the compression thresholds ξ and exchanged original gradients throughout training. Thus, CNC is ≈ 0.0 for all ξ at CF $100\times$ in VGG19.

7.5.5 Overall Performance of ScaFLES

In this section, we look at the overall parallel and statistical efficiency of ScaFLES by combining weighted aggregation of heterogeneous streams, buffering queue reduction with stream truncation, reducing communication overhead with dynamic compression, and performance in non-IID data settings. We use CF $10\times$ with $\xi = 0.3$ for dynamic compression in ScaFLES, which we compare with BSP training at fixed batch-size 64, stream persistence policy and the training routine described in Appendix (C.1).

For a given streaming distribution, we compare the accuracy drop in ScaFLES with BSP, reduction in buffer-size (in gigabytes) and overall training speedup until accuracy does not improve any further. A negative value of accuracy drop implies that ScaFLES achieved lower accuracy than BSP by that margin. Table (7.4) shows the results for IID training, where ResNet152 over ScaFLES observed a maximum accuracy loss of only 0.32%. Convergence was also faster with ScaFLES,

ranging from 1.15-3.29× over BSP. For high-volume streams $S^{'}_{uniform}$ and $S^{'}_{normal}$, stream truncation policy saved up to 5.9 GB in the buffering queue. For VGG19, ScaFLES observed up to 4.18% accuracy drop over BSP in high-volume stream $S^{'}_{uniform}$ from large-batch training as well as dynamic compression. However, ScaFLES with VGG19 reduced the buffer-size by up to 3.91 GB and reduced overall training time by 1.56-2.83× over BSP.

For non-IID training, BSP was unable to reach the same accuracy levels as ScaFLES' data-injection strategy. ResNet152 and VGG19 over BSP saturated to 56% and 35% accuracy respectively. On the other hand, ScaFLES over non-IID data trained ResNet152 to 93.6% and VGG19 attained 77.8% accuracy (i.e., accuracy improvements of 37.6% and 42.8% respectively over BSP).

7.6 Conclusion

ScaFLES presents the notion of FL over streaming data at the edge, which presents both parallel and statistical challenges in distributed learning. Our system addresses heterogeneous stream-rates among devices through variable computing with additional optimizations. Unable to process data at line-rate, we present a simplistic truncation policy to keep the buffer-size in check over devices with limited memory and storage. To deal with unbalanced data distribution in non-IID settings, we present stochastic data-injection that considerably improves DNNs' convergence quality. Last, we present a dynamic technique that compares entropy loss in compression to attenuate high communication cost over limited bandwidth networks. We simulate varying degrees of streaming heterogeneity by sampling from uniform and normal distributions, and show that ScaFLES can converge anywhere from 1.15-3.29× faster than BSP. At the same time, we observe a reduction of 848-9429× in the buffering queue with our truncation policy. In non-IID settings, ScaFLES achieves up to 40% more accuracy than BSP.

7.7 Related Publications

1. **Tyagi, Sahil** and Swany, Martin. "ScaDLES: Scalable Deep Learning over Streaming data at the Edge." 2022 IEEE International Conference on Big Data (Big Data) (2022): 2113-2122.

2. Widanage, Chathura and Li, Jiayu and **Tyagi, Sahil** and Teja, Ravi and Peng, Bo and Kamburugamuve, Supun and Baum, Dan and Smith, Dayle and Qiu, Judy and Koskey, Jon.

"Anomaly Detection over Streaming Data: Indy500 Case Study." 2019 IEEE 12th International Conference on Cloud Computing (CLOUD) (2019): 9-16.

CHAPTER 8

FEDERATED LEARNING WITH SEMI-SYNCHRONOUS COMMUNICATION

8.1 Introduction

In FL systems, multiple clients collaboratively train a globally shared model by computing updates on local data and periodically aggregating them over a centralized server. In addition to challenges like data privacy and locality, FL on the the edge is constrained by limited compute and network bandwidth. Synchronous or BSP training performs more work per-iteration by simultaneously computing updates over multiple model replicas, but suffers from the high communication overhead of aggregating updates at each step. To attenuate this slowdown, different techniques either reduce the frequency or volume of communication, or relax the constraints on synchronization itself. We refer to such techniques that enforce a loose form of synchronization or apply infrequent communication based on some rule or heuristic as "*semi-synchronous*".

In this chapter, we propose `SelSync`, a semi-synchronous FL technique that switches between local and synchronous updates by assessing the importance of each iteration across clients. The objective of this partial-synchronous mechanism is to achieve BSP-level convergence and high training speedups by eliminating communication and applying local updates only whenever possible. In our evaluation, `SelSync` is able to train sota models up to 14$\times$ faster. In summary, we make the following contributions:

- Implement semi-synchronous training in a centralized PS [49] setting over PyTorch RPC framework [204].
- Develop a low-overhead technique to measure the significance of each iteration and apply a communication rule based on it.
- Design a custom data-partitioning scheme optimal for semi-synchronous training.
- Empirically validate that parameter reduction outperforms gradient aggregation in semi-synchronous training. This is because local and global model states tend to diverge more in the latter when

clients train locally for longer and updates are aggregated less frequently.

- Improve `SelSync`'s convergence in non-IID settings by extending stochastic data-injection [33] in the context of semi-synchronous training which outperforms other federated algorithms in terms of statistical performance.

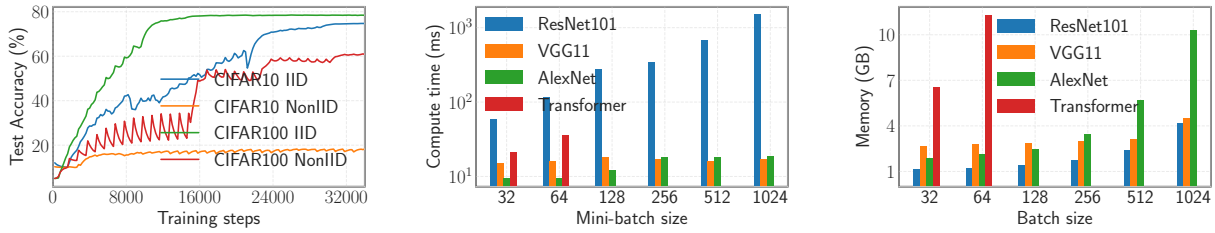
- Extensive evaluation with other FL methods over multiple DNNs to validate the efficacy of `SelSync`.

8.2 Motivation

8.2.1 Semi-synchronous training mechanisms

From §2.5, BSP does not parallelize linearly due to Amdahl's scaling limitations [188]. This is further exacerbated in FL on the edge where latency is high and bandwidth is low. To attenuate parallel scaling challenges, federated techniques like FedAvg allows DNNs to be deployed on numerous clients and aggregate model updates after a fixed number of steps or epochs. With its low-frequency, high-volume communication strategy, FedAvg collects updates from fraction 'c' out of 'N' clients in periods of '$freq$' times every epoch, such that the synchronization factor $f = 1/freq$. For e.g., $c = 0.5$ and $f = 0.25$ implies that FedAvg aggregates updates among half of the total devices 4 times in each epoch (uniformly spaced). Based on the chosen (c, f) configuration, the communication frequency and client participation may affect the overall training time or convergence quality. Thus, there are complex trade-offs between the two based on the deployment configuration in FedAvg. Another critical challenge in FL systems is unbalanced and skewed data distribution across decentralized clients. In spite of large model-sizes, statistical performance in FL is stumped due to non-IID data. Fig. (8.1a) shows the accuracy degradation between balanced and unbalanced data in FedAvg. ResNet101 [9] is trained on CIFAR10 [180] and VGG11 [8] on CIFAR100 over a cluster of 10 NVIDIA V100 GPUs [220] such that non-IID versions of the two datasets are partitioned with 1 and 10 labels per-client respectively. Here, we set $(c, f) \rightarrow (1, 0.1)$ so that updates are collected from all clients 10 times over each epoch (i.e., every 1/10-th of an epoch).

Other techniques like *Stale-synchronous parallel (SSP)* are based on a centralized PS strategy where workers mostly train independently of each other, but synchronization blocks are enforced



(a) FedAvg IID vs. non-IID data (b) SSP compute time (c) SSP memory consumption

Figure 8.1: Limitations of FedAvg and SSP. (a) ResNet101/VGG11 trained over IID/non-IID CIFAR10/100. (b) and (c) Compute time and memory of SSP rises with batch-size.

when fast clients exceed the slow ones by some limit, denoted by user-specified *staleness threshold* 's'. Once the staleness threshold is crossed, fast workers pause and wait for the slow nodes to catch up. For e.g., setting $s = 100$ implies that the fastest and slowest clients cannot exceed each other by more than 100 iterations. With bounded asynchrony, SSP ensures the divergence between local and global model parameters is bounded as well, thus mitigating staleness. For a cluster of 'N' clients each processing mini-batch 'b', SSP trains local models on fewer samples compared to BSP (b vs. Nb respectively). Theoretically, one could increase the local batch-size to Nb in SSP and perform as much work as BSP in each iteration. However, this raises both the computational and resource requirements of training DNNs, as seen from Fig. (8.1b) and (8.1c). The first plot shows the computation time of SSP training as we gradually increase the batch-size on NVIDIA's K80 GPU for different image and language models. ResNet101 has the largest computation time due to its 101-layer depth. This is because feeding more samples with larger b increases the number of forward passes made through the layers of the network, thus increasing the compute cost of forward-pass. The second figure illustrates how the memory consumption on a client rise with the mini-batch. In Fig. (8.1c), Transformer [4] training over WikiText-103 [222] failed to scale beyond b = 64 due to OOM (out-of-memory) errors as its memory requirements exceeded the GPU's 12 GB capacity. AlexNet [1] training over ImageNet-1K [154] had a noticeably high memory utilization with large b while using PyTorch's `torchvision.datasets.ImageFolder` [223] for data loading and batching. In general, we see that for a large cluster size 'N', setting a per-client batch-size Nb would greatly increase the compute cost and memory on each client, making SSP impractical or even prone to failures in some cases.

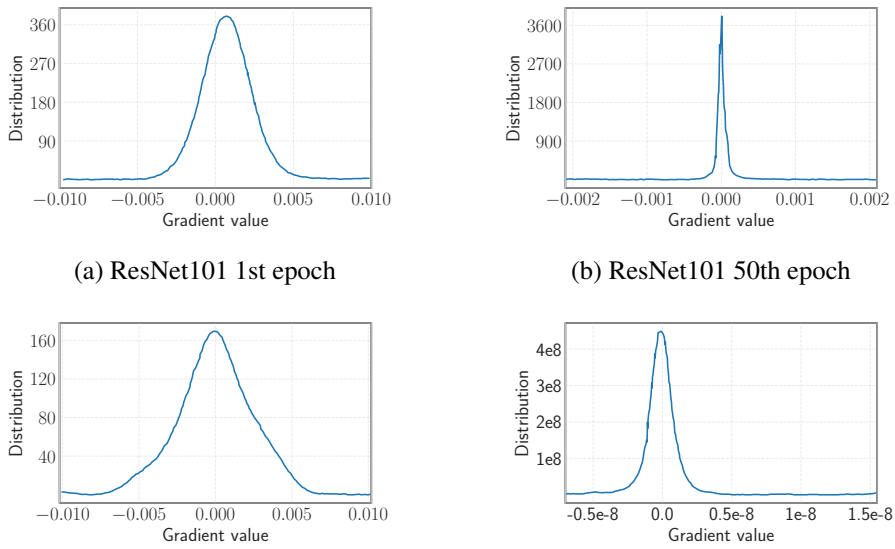
To summarize, FL methods maximize useful work by frequently aggregating client updates

(BSP), or speedup training by lowering the communication volume and frequency (FedAvg), or loosen the constraints on synchronization itself (SSP). The latter two attain considerable speedup over BSP in some cases, but observe considerable detriment with respect to convergence quality in others. This is because they only consider the parallel aspect and ignore the statistical aspect in distributed and federated learning settings.

8.2.2 Gradient Sensitivity in DNN Training

Prior work has explored that gradients computed in backpropagation are large and change drastically, especially in the early stages of training [149]. A randomly initialized model adjusts its parameters aggressively towards the minima during this stage. With good conditioning and optimal hyperparameters, a model would eventually converge after many iterations. Updates computed in the later stages become smaller as the model approaches convergence and gradients stop changing significantly. Gradients computed on a randomly sampled mini-batch are not representative of the true gradients (calculated over full-batch) and tend to be noisier [40, 167, 168, 169]. Fig. (8.2) illustrates how updates vary over the course of training by plotting density or KDE of the gradients for a single convolutional layer in ResNet101, and an encoder layer in Transformer. As training continues, gradients get smaller and closer to 0.0 at higher epochs compared to initial epoch 1.

In addition to early phases, training is sensitive at certain critical or sensitive regions [27, 28]. For a given model, second-order information or Hessian measures the gradient divergence [142], while [28] observed that changes in the eigenvalues of the Hessian acts as an indicator of critical learning periods. However, computing second-order information is highly compute-intensive. Fortunately, first-order gradient information effectively approximates the Hessian eigenvalues to detect critical periods [131]. Fig. (8.3) validates the same for ResNet101 and VGG11, where the largest eigenvalue of the Hessian calculated at every iteration follows a similar course as first-order $\mathbb{L}$-2 gradient norm. Even though their magnitudes lie on different scales, the relative inter-iteration changes are similar. Additionally, gradient norm is significantly cheaper to compute and can easily be integrated as part of backpropagation. Although the exact magnitude and gradient trajectory varies for each model in SGD-based optimizations, first-order information can be a useful indicator of statistical efficiency in FL systems.



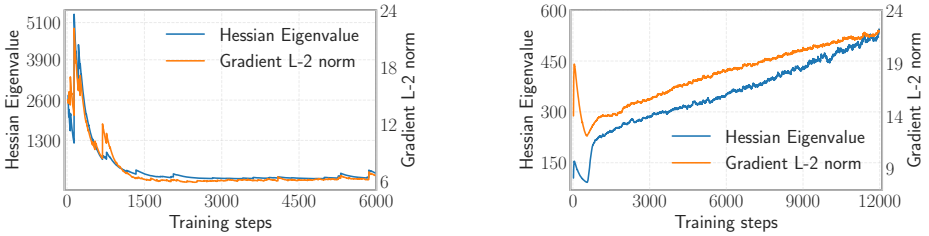
(a) ResNet101 1st epoch

(b) ResNet101 50th epoch

(c) Transformer 1st epoch

(d) Transformer 4th epoch

Figure 8.2: Distribution of gradient values (i.e., KDE on the y-axis) over the epochs for ResNet101's 4th layer weights and Transformer's 1st encoder layer weights. Gradients are large and volatile in the early epochs, but get smaller and saturate as training progresses.



(a) ResNet101

(b) VGG11

Figure 8.3: Largest eigenvalue of the gradient Hessian along with gradient norm. Both have similar trajectory, thus we can also detect critical periods with first-order information which is cheaper to compute.

8.3 `SelSync` Design

`SelSync` develops a flexible communication strategy to perform either synchronous or local-SGD update, based on the significance of its gradients. This approach improves parallel efficiency in FL via local updates which avoids the synchronization overhead altogether and reduces iteration time. During critical updates, statistical efficiency is prioritized by synchronizing gradients across all clients to have a global, consistent model state.

8.3.1 Detecting Crucial Updates in DNN Training

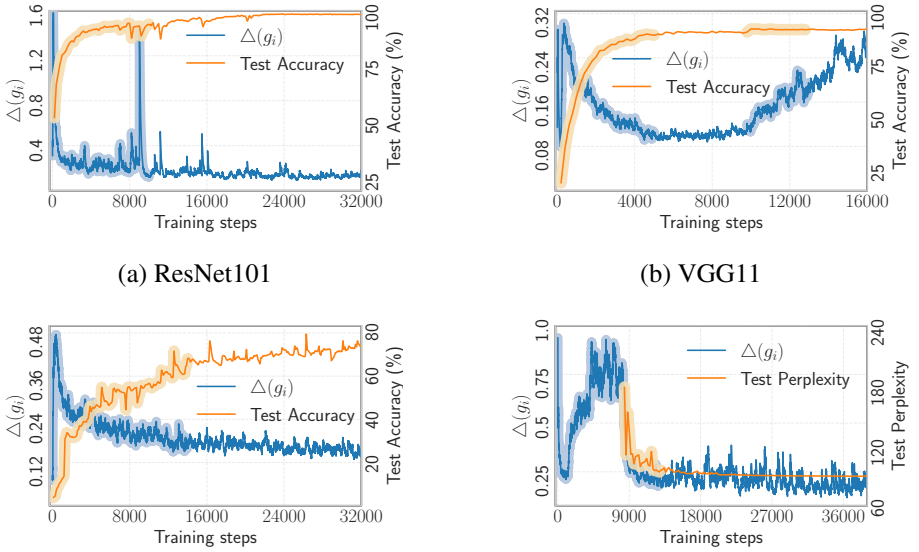
Distributed learning extends metrics like throughput and scale-out factor from the fields of HPC and parallel computing to measure the parallel efficiency of training a model in a distributed setting. However, training DNNs' comes with a statistical efficiency aspect as well that is "non-linear"; some updates are highly critical while others provides marginal or negligible gains in terms of a models' learning and generalization ability.

We exploit first-order gradient information to measure the significance of model updates on client 'c' by tracking changes in the gradient norm at every iteration 'i'. The *gradient change* metric, denoted by $\triangle(g_i^{(c)})$, can be estimated from the $\mathbb{L}2$-norm of gradients $\nabla\mathcal{F}$ between consecutive iterations '$(i-1)$' and 'i', as shown in Eqn. (8.1).

$$\triangle(g_i^{(c)}) = \left| \frac{\mathbb{E}[||\nabla\mathcal{F}_{(i)}^{(c)}||^2] - \mathbb{E}[||\nabla\mathcal{F}_{(i-1)}^{(c)}||^2]}{\mathbb{E}[||\nabla\mathcal{F}_{(i-1)}^{(c)}||^2]} \right| \tag{8.1}$$

With EWMA smoothing, we plot $\triangle(g_i^{(c)})$ alongside the convergence curves for various DNNs in Fig. (8.4). The convergence plots correspond to test accuracy of ResNet101, VGG11 and AlexNet, and test perplexity of Transformer across 16 clients. Higher accuracy is better while a lower perplexity is preferred. The figures highlight the correlation between critical updates as measured by its gradient change metric and DNNs' convergence quality. The excessively volatile phases are shaded for clarity. We omit the perplexity for the first 8000 steps in Transformer as it was high initially and fell sharply in that phase.

In Fig. (8.4a), $\triangle(g_i^{(c)})$ changes sharply as accuracy rises in the first 10000 steps of ResNet101. The spike in gradient change after 10000 steps corresponds to learning-rate decay in model training.



(a) ResNet101

(b) VGG11

(c) AlexNet

(d) Transformer

Figure 8.4: Correlation between gradient change ($\triangle(g_i^{(c)})$) and model convergence in synchronous training. A rise or decline in accuracy/perplexity is accompanied by changes in the gradients as well. As convergence plateaus, so does $\triangle(g_i^{(c)})$.

With Transformer in Fig. (8.4d), perplexity fell sharply in the first 10000 iterations while $\triangle(g_i^{(c)})$ metric rose and fell swiftly in that phase as well. The perplexity fell gradually after 13000 steps, as did the gradient change. AlexNet in Fig. (8.4c) shows how $\triangle(g_i^{(c)})$ varies gradually when the accuracy gain over the iterations in the same phase is gradual as well. The gradients of VGG11 in Fig. (8.4b) change drastically while the accuracy rises sharply in the first 5000 iterations. As gradients saturate between 5000 and 10000 iteration, $\triangle(g_i^{(c)})$ metric flattens out in that phase and the accuracy plateaus as well. After 10000 steps, the learning-rate is decayed and we see how the accuracy and $\triangle(g_i^{(c)})$ rise concurrently. Thus, there is a strong correlation between gradient change and model convergence in FL and DL systems.

8.3.2 δ-based Selective Synchronization

Gradient changes are prominent over significant updates and stabilize as training saturates. The degree to which the gradients vary depends on factors like training hyperparameters, model-size and its complexity. Denoting this extremum as $\mathcal{M} = max([\triangle(g_i^{(c)})]_{i=1}^{i=I})$ for a model on client 'c' running for 'I' iterations, $\triangle(g_i^{(c)})$ lies in the range $[0, \mathcal{M}]$.

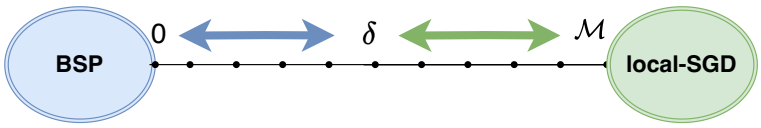


Figure 8.5: Adjusting threshold δ on relative gradient change metric in `SelSync`. Choose BSP if $\triangle(g_i^{(c)}) \geq \delta$ and local-SGD if $< \delta$. Setting δ=0 implies BSP training, while a very high δ mostly performs local updates.

With this intuition, `SelSync` synchronizes client updates only when $\triangle(g_i^{(c)})$ changes considerably, and perform local updates otherwise to mitigate expensive communication costs. By choosing intelligently between synchronous and local updates, `SelSync` strives to balance between the parallel and statistical efficiency. Additionally, incorporating local-SGD with BSP allows for more exploration of a client's latent space which in turn can improve model generalization [83]. This also bounds the divergence between the local and global model states (i.e., on clients and PS respectively) which is necessary for good convergence quality [90]. `SelSync` implements a practical and low-overhead technique to measure the significance of inter-iteration updates by imposing a threshold, denoted as δ, on the gradient change metric as follows: *use local-SGD if gradient change is below δ and synchronize updates if any of the clients'* $\triangle(g_i^{(c)}) \geq \delta$. The threshold δ is set prior training such that $\delta = 0$ implies fully synchronous or BSP training while setting a sufficiently high threshold (i.e., $\delta \geq \mathcal{M}$) trains locally. Fig. (8.5) demonstrates how we can slide δ between 0 and $\mathcal{M}$ to adjust the proportion of training between local-SGD and BSP updates.

8.3.3 Gradient vs. Parameter Aggregation in FL

In BSP, aggregation can either be performed on the gradients or model parameters. They are equivalent in BSP assuming all clients were initialized with the same model state prior training. Clients pull the initial model state from the central PS prior training, while decentralized systems broadcast the initial parameters from one client (for e.g., rank-0 process) across the cluster. Local model states are thus consistent with the global model in BSP at every iteration since the same aggregated updates are applied or attained on each client. However, gradient aggregation is more common as gradient updates can be compressed via sparsification, quantization or low-rank approximations [115, 112, 117]. Model parameters can be compressed via pruning as well [99], although the rate of compression achieved may be considerably lower.

In the context of semi-synchronous FL, gradient and parameter averaging can have different effects; reducing parameters instead of gradients makes more generalizable DNNs in semi-synchronous training. As `SelSync` collects updates infrequently, based on $\triangle(g_i^{(c)})$ and chosen δ, gradient averaging can cause the local models to diverge from the global state on the PS and result in inconsistent models across clients. The more iterations are applied locally, the more local replicas can thus deviate. This is because even though the same averaged gradients are applied on each client in gradient averaging, the resulting parameters post-SGD update will be different across clients and may diverge over time. This increased deviation can thus deteriorate the test performance in semi-synchronous methods.

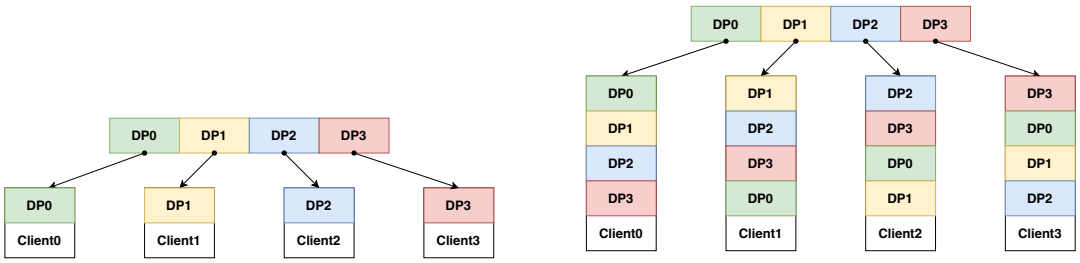
Local training is not necessarily detrimental; local updates promote exploration of a clients' local minima [90]. FL with parameter averaging ensures local model weights are bounded with respect to the global weights, and client models does not fall into a local minima too far from the global state. Additionally, aggregating parameters during the synchronization phase ensures model consistency across clients. In §8.4, we empirically compare gradient and parameter averaging in `SelSync` over multiple DNNs.

8.3.4 Data-Partitioning in Semi-Synchronous FL

***For IID Data**:* BSP maximizes per-iteration work by processing different batches on every client and averaging updates computed from those partitions. For e.g., training dataset $\mathcal{D}$ is partitioned among 4 clients as $\mathcal{D} \supseteq \{$*DP0*, *DP1*, *DP2*, *DP3*$\}$, such that each partition resides on a single client. We refer to this as 'default partitioning' or *DefDP*, depicted in Fig. (8.6a) where *client0* samples data from partition *DP0*, *client1* from *DP1*, and so forth. This approach is ideal for BSP as every sample is processed at least once every epoch.

However, such a partitioning scheme might not work well in semi-synchronous training. Consider a scenario where a model mostly applies local instead of synchronous updates. In that case, each client optimizes its model replica over its local subset only and fails to aggregate client updates computed over other partitions. Thus, local replicas fail to learn features from data residing on other clients when there is low-to-negligible communication. Even if aggregation is triggered occasionally, averaging a diverge model can produce a resultant model of low quality [25].

`SelSync` proposes a custom data-partitioning scheme called *SelDP* that is tailored for tech-



(a) Default data-partitioning (b) `SelSync` data-partitioning

Figure 8.6: Data-partitioning schemes in FL over IID data. DefDP splits training data into as many partitions as the number of clients. SelDP partitions as a circular queue whose head is rotated to a unique chunk on each client.

niques that use a hybrid of local and synchronous SGD. Instead of splitting data into unique and separate chunks, dataset $\mathcal{D}$ is shuffled on every client based on its cluster identifier (in decentralized systems, this corresponds to device rank). This way, *client0* trains on partitions ordered as $\{$*DP0*, *DP1*, *DP2*, *DP3*$\}$, *client1* on $\{$*DP1*, *DP2*, *DP3*, *DP0*$\}$, *client2* on $\{$*DP2*, *DP3*, *DP0*, *DP1*$\}$, while *client3* processes its samples as $\{$*DP3*, *DP0*, *DP1*, *DP2*$\}$. The shuffle operation on $\mathcal{D}$ is a one-time overhead incurred prior training. Thus, training samples are available across every client in SelDP, as shown in Fig. (8.6b). From a statistical standpoint, SelDP ensures that local model replicas will not get skewed if training with local-SGD for majority of the iterations (for instance, training with `SelSync` under a high δ-setting). On the other hand, when clients do synchronize, each client's updates comes from a unique chunk and no client redundantly processes the same set of training samples at any iteration. For e.g., if the very first iteration qualifies for communication, then *client0* processes a mini-batch from chunk *DP0*, *client1* from *DP1*, *client2* from chunk *DP2* and *client3* processes from *DP3*.

***For Non-IID Data**:* As seen from Fig. (8.1a), FL suffers from model degradation due to unbalanced and skewed data distribution. To ameliorate this issue, we extend stochastic data-injection from Chapter (7) and apply it in the context of semi-synchronous FL. In data-injection, a random subset of clients communicate and share partial training samples at every iteration, and it is configured as a tuple (ϕ, ψ), where 'ϕ' is fraction of clients chosen and 'ψ' is the proportional batch-size to be shared. For e.g., a value of $(\phi, \psi) \rightarrow (0.5, 0.5)$ implies half the clients are selected randomly to share half of their training samples, thus improving the overall data distribution in FL and consequently, its statistical performance. Privacy risk is also greatly minimized by sharing data with a

small fraction of the clients randomly at each iteration.

Since mini-batch size is an important training hyperparameter, applying data-injection naively can be detrimental for model generalizability if the cluster-size is large since the per-client batch-size would become considerably larger [143, 140]. To solve this issue, `SelSync` adjusts client batch-size to $b^{'}$ such that cumulative batch-size post data-injection operation is same as original batch-size $|b|$ (specified prior training had we trained with IID data), shown in Eqn. (8.2) for a cluster of 'N' devices.

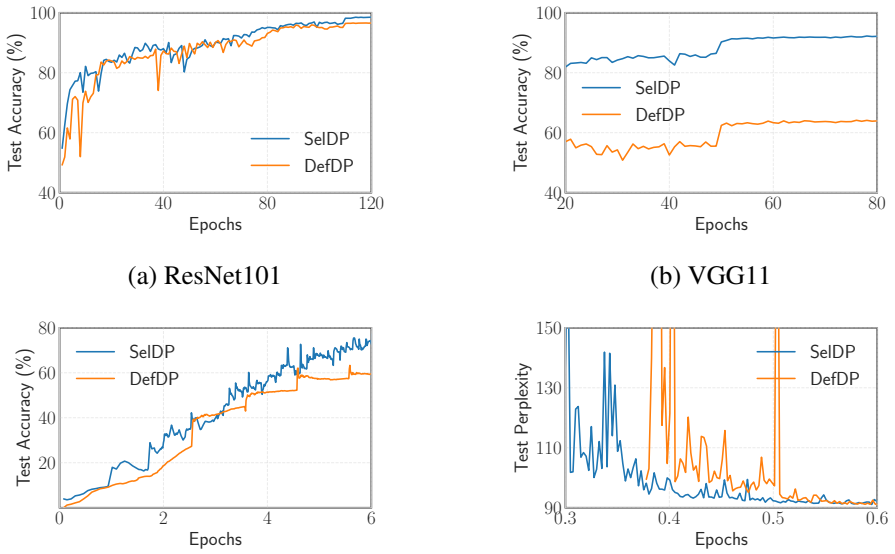
$$|b^{'}| = \frac{|b|}{(1 + \phi \cdot \psi \cdot N)} \tag{8.2}$$

The communication overhead of data-injection is low as well; each iteration communicates only $(\phi \cdot \psi \cdot N \cdot |b^{'}|)$ times the size of an individually datum. In vision datasets like CIFAR100 and ImageNet-1K, each image is about 3Kb (kilobytes) and 10-150Kb respectively. Thus, data-injection on a cluster of 16 clients and $b = 32$ in a $(\phi, \psi) \rightarrow (0.5, 0.5)$ configuration incurs a transfer cost of only 132Kb on CIFAR100 and 440-6600Kb in ImageNet-1K. This overhead is negligible compared to the exchange cost of model updates that span hundreds of megabytes or gigabytes at every iteration. `SelSync` appends δ-threshold to the data-injection configuration, making it (ϕ, ψ, δ). Another advantage of stochastic data-injection in FL is that privacy is ensured with K-anonymity [224] by not knowing from which $\lceil \phi \cdot K \rceil$ of the 'K' clients the training samples came from.

8.4 Experimental Evaluation

We implement `SelSync` over PyTorch, with details of its training algorithm described in Appendix (D.1). The cluster setup, evaluation testbed, DNNs and training configuration is described in full detail in Appendix (D.2).

We compare `SelSync` with BSP, FedAvg and SSP by comparing parallel performance in terms of overall training speedup and the proportion of updates applying synchronous vs. local-SGD, while statistical performance is measured in terms of test accuracy or perplexity of a model. BSP is trained with DefDP scheme where updates are aggregated on the PS at the end of each iteration. FedAvg is deployed with four (c, f) configurations: (1, 0.25), (1, 0.125), (0.5, 0.25) and (0.5, 0.125);



(a) ResNet101

(b) VGG11

(c) AlexNet

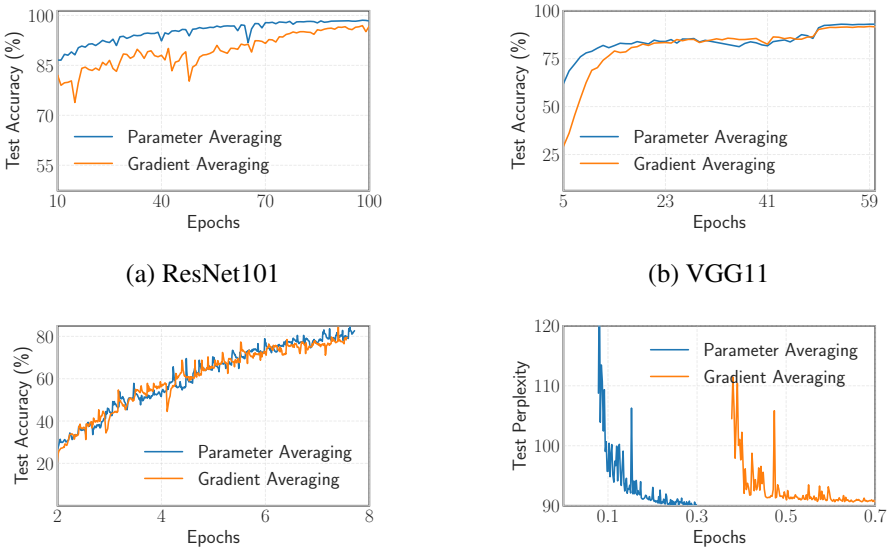
(d) Transformer

Figure 8.7: Test performance of `SelSync` = 0.25 with gradient averaging on default and custom data-partitioning schemes. For the same epochs, SelDP achieves better generalization performance than DefDP.

thus updates are collected either from all or half of the participating clients, while aggregation is performed either 4 or 8 times in uniform intervals at every epoch. We deploy SSP with staleness thresholds of 100 and 200 iterations respectively. The chosen thresholds 's' are large enough to not be a frequent synchronization barrier, while still small enough to not let clients model replicas diverge too far from the global model. Lastly, `SelSync` is deployed with SelDP over different δ-thresholds like 0.25, 0.3 or 0.5.

8.4.1 Default and `SelSync` Partitioning Schemes

We train `SelSync` with $\delta = 0.25$ and see how different DNNs converge with its custom partitioning compared to the default partitioning scheme in distributed training. The results are shown in Fig. (8.7) as a plot of test accuracy/perplexity and epochs. During the synchronization phase, we perform gradient aggregation on the PS and return the averaged gradients back to the clients so as to apply locally on their model replicas. ResNet101 converged to 97.6% accuracy on SelDP while DefDP attained 96.8% after 125 epochs. CIFAR100 partitioned with the default scheme performed poorly on VGG11 with merely 64.1% accuracy, while SelDP achieved 90.86% after 90 epochs (from Fig. (8.7b)). CIFAR100 saw worse accuracy than CIFAR10 with DefDP since the former had more



(a) ResNet101

(b) VGG11

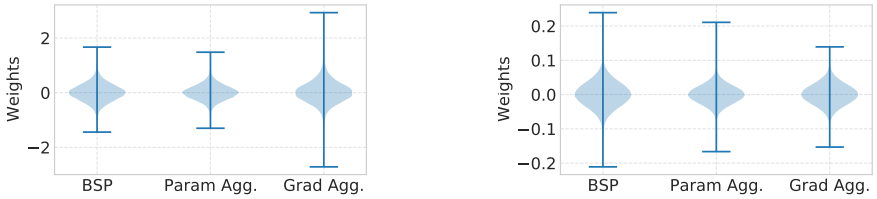
(c) AlexNet

(d) Transformer

Figure 8.8: Convergence curves for parameter and gradient averaging in `SelSync` at $\delta = 0.25$. Parameter averaging attains better or similar accuracy than gradient averaging for the same epochs.

labels to evaluate. Although VGG11 is larger, ResNet101 is a skip-connection based architecture that generalizes better than simpler convolution networks [9], thus reaching higher accuracy in the residual network. AlexNet on SelDP reached 81.1% accuracy while DefDP did not improve beyond 61.2%, as shown in Fig. (8.7c). Training Transformer with the custom and default partitioning schemes attained 92.6 and 94.91 test perplexity after training for around 50 million tokens of the WikiText-103 dataset. The perplexity before 0.3 and 0.38 epochs with the two schemes is omitted as it was very high and fell sharply in the initial phases.

We thus see that SelDP outperforms DefDP in semi-synchronous training since many updates are performed locally and are not aggregated among clients. In mostly-local training, model replicas only process their local samples with the default scheme and fail to learn from partitions residing on other clients. As a result, local models tend to diverge from the global model over time. With SelDP, each client has access to the entire IID training data, albeit in different orders. This ensures the clients do not reach a local minima that is too far from the centralized model on the PS.



(a) Epoch 25 (b) Epoch 50

Figure 8.9: Density distribution of model parameters at different training stages in BSP, and `SelSync` with parameter and gradient aggregation. Comparatively, BSP and `SelSync` with parameter averaging have a more similar distribution than gradient averaging.

8.4.2 Gradient vs. Parameter Averaging in `SelSync`

In this section, we look at the impact of averaging parameters instead of gradients in a semi-synchronous technique like `SelSync`, where we set $\delta = 0.25$ and partition the data with SelDP. Fig. (8.8) shows the convergence curves for the two aggregation schemes. By the end of 100 epochs in ResNet101, gradient averaging attained 96.2% while parameter averaging reached 98.52%, an improvement of 1.72% by aggregating model parameters. For VGG11, gradient aggregation attained 90.86% accuracy and 91.42% with parameter averaging after 60 epochs (i.e., 0.56% better accuracy with parameter aggregation). Transformer in Fig. (8.8d) reached nearly the same perplexity of 90.0; however, `SelSync` with parameter averaging reached that target in 0.3 epochs, while gradient averaging took about 130% iterations to converge. AlexNet, the only model trained with a fixed learning-rate, had a similar convergence trajectory with the two averaging methods.

The better generalization performance of `SelSync` with parameter averaging can be seen from the distribution of model parameters with the two aggregation approaches and how well they align with BSP training. In Fig. (8.9), we run ResNet101 with BSP, and `SelSync` on parameter and gradient averaging at different training stages. On the 25th epoch, BSP and parameter averaging have similar distributions (Fig. (8.9a)). As training continues to its 50th epoch, density of gradient averaging becomes even more narrower than BSP or parameter averaging. Parameter aggregation is more aligned with BSP because synchronizing parameters bounds the divergence between local and global model state, and ensures all clients have a globally consistent model.

8.4.3 Speedup and Convergence over IID and Non-IID Data

IID data:

To measure the cumulative portion of training using local updates relative to iterations that are synchronized among clients, we propose a metric called *Local-to-Synchronous Step Ratio (LSSR)*, as shown in Eqn. (8.3).

$$\text{LSSR} = \left(\frac{steps_{local}}{steps_{local} + steps_{sync}}\right) \tag{8.3}$$

If a model trains entirely in local mode, $steps_{sync} = 0$ and LSSR = 1.0, while LSSR = 0.0 for BSP since there are no local updates (i.e., $steps_{local} = 0$). LSSR essentially measures communication reduction with respect to BSP as $1/(1 - \text{LSSR})$ for the same number of epochs. For e.g., LSSR = 0.9 implies a communication reduction of $10\times$ over fully-synchronous training. Although this metric applies to semi-synchronous methods like `SelSync` and FedAvg, it does not apply to SSP as clients push their updates to the PS asynchronously and do not synchronize them. When staleness threshold is crossed, faster clients halt training and wait for the slower devices to catch up. Since there is no explicit aggregation operation, LSSR metric is not applicable here.

Table (8.1) evaluates `SelSync` against BSP, FedAvg and SSP. We run the models until its test accuracy/perplexity does not improve any further, and note the corresponding iterations taken and LSSR scores (if applicable). We measure convergence and speedup with respect to BSP as many semi-synchronous methods do provide training speedups, but not necessarily meet the same BSP-level convergence. *Results from column 'Speedup' in Table (8.1) are omitted for settings that failed to meet the same accuracy/perplexity as BSP*.

For ResNet101, `SelSync` and some FedAvg configurations achieved higher accuracy than BSP. The statistical gains over synchronous training come from ample exploration of the local minima on individual clients which results in better generalization performance. However, applying too many local updates or partial aggregation from only a subset of total clients can increase the divergence between local and global model, as seen in FedAvg with (c, f) values (0.5, 0.25) and (0.5, 0.125). Here, updates were collected from only half the clients during synchronization phase (4 and 8 times per-epoch respectively). FedAvg with $c = 1$ performed well despite high LSSR scores

Model	Method	Iterations	LSSR	Acc./Perplexity	Speedup
ResNet101	BSP	22.8K	0.0	98.5%	1×
	FedAvg (1, 0.25)	63.4K	0.979	98.55%	2.59×
	FedAvg (1, 0.125)	57.1K	0.959	98.6%	2.55×
	FedAvg (0.5, 0.25)	66.8K	0.979	94.4%	-
	FedAvg (0.5, 0.125)	62.8.1K	0.959	95.5%	-
	SSP s = 100	54.8K	-	93.8%	-
	SSP s = 200	59.4K	-	93.33%	-
	SelSync δ = 0.3	**42.2K**	**0.829**	**98.62%**	**2.03×**
	SelSync δ = 0.5	**53.2K**	**0.846**	**98.56%**	**1.67×**
VGG11	BSP	10.8K	0.0	90.9%	1×
	FedAvg (1, 0.25)	17.4K	0.979	87.56%	-
	FedAvg (1, 0.125)	20.2K	0.959	88.2%	-
	FedAvg (0.5, 0.25)	19.6K	0.979	84.6%	-
	FedAvg (0.5, 0.125)	21.1K	0.959	85.16%	-
	SSP s = 100	11.8K	-	68.72%	-
	SSP s = 200	12.2K	-	68.78%	-
	SelSync δ = 0.3	**72.4K**	**0.912**	**91.33%**	**6.16×**
	SelSync δ = 0.5	**81.2K**	**0.937**	**91.32%**	**13.75×**
AlexNet	BSP	106.5K	0.0	85.15%	1×
	FedAvg (1, 0.25)	118K	0.997	77.15%	-
	FedAvg (1, 0.125)	124K	0.993	78.2%	-
	FedAvg (0.5, 0.25)	115.6K	0.997	75.6%	-
	FedAvg (0.5, 0.125)	121.4K	0.993	76.67%	-
	SSP s = 100	111.5K	-	86.52%	4.53×
	SSP s = 200	113.2K	-	85.8%	4.47×
	SelSync δ = 0.3	**95.8K**	**0.954**	86.72%	**4.62×**
	SelSync δ = 0.5	**106.8K**	**0.966**	**85.93%**	**4.63×**
Transformer	BSP	28K	0.0	90.02	1×
	FedAvg (1, 0.25)	112.1K	0.998	89.96	1.99×
	FedAvg (1, 0.125)	104.2K	0.999	90.01	2.14×
	FedAvg (0.5, 0.25)	118K	0.999	91.67	-
	FedAvg (0.5, 0.125)	111.6K	0.998	91.2	-
	SSP s = 100	110K	-	89.98	2.02×
	SSP s = 200	116.2K	-	90.0	1.92×
	SelSync δ = 0.3	**38.6K**	**0.725**	**89.91**	**2.42×**
	SelSync δ = 0.5	**46.1K**	**0.733**	**89.97**	**2.06×**

Table 8.1: Speedup and convergence (with respect to BSP) in FedAvg, SSP and SelSync.

($\approx$ 0.95-0.97) since ResNet101 is an over-parameterized network for the CIFAR10 dataset, and was thus more robust to local training. `SelSync` with $\delta = 0.3$ and 0.5 had LSSR scores between 0.82-0.85. As a result, FedAvg had a higher speedup than `SelSync` over BSP training. SSP failed to beat BSP because pushing-pulling updates asynchronously to-from the PS on a layer-by-layer basis introduced additional staleness which degraded final accuracy.

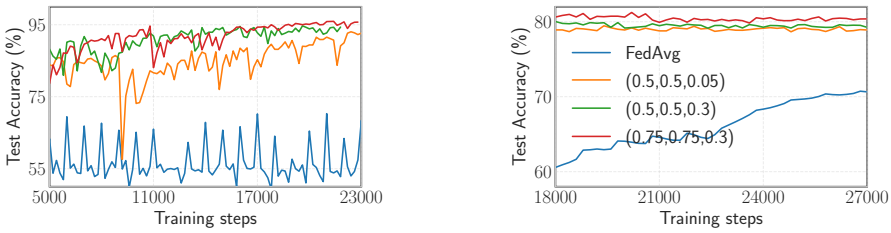
The convolutional network VGG11 failed to reach BSP-level convergence with FedAvg or SSP. `SelSync` attained up to 0.43% more accuracy than BSP, while lowering training time by up to 13.75×, courtesy of highly localized training as seen from the high LSSR scores. `SelSync`'s LSSR scores are however lower than FedAvg as it switches between local and synchronous updates based on their significance.

AlexNet on Imaget-1K failed to converge across all FedAvg configurations. For a given (c, f) configuration, 'f' determines the aggregation frequency in an epoch, so the number of synchronization calls made with FedAvg were too low on account of the large dataset size of 1.28 million samples. This is also corroborated from the extremely high LSSR scores ($\approx$ 0.99) and hence, the model trained locally for almost all its iterations and learned only from its local state. Due to the smaller size and fewer layers of AlexNet, staleness was not an issue in SSP training at either threshold. By asynchronously pushing-pulling updates to-from the PS, clients were also able to learn features on other clients' partitions, thus improving its test accuracy. SSP achieved about 1.37% better accuracy than BSP while accelerating training by 4.53×. On the other hand, `SelSync` was 4.63× faster and attained 1.57% better accuracy than BSP. The speedup comes from communicating only the crucial updates and performing 95-96% of the iterations locally (as seen from the LSSR scores).

For c = 1 in FedAvg, Transformer converged to a better perplexity than BSP while being 2.14× faster, while SSP converged 2× faster than BSP due to its asynchronous execution. `SelSync` ran between 39K-46K training iterations and synchronized for about 26% of those updates. By doing so, it reduced training time by up to 2.42× while reaching an even better perplexity score than BSP.

Non-IID data:

We now compare `SelSync` with FedAvg under non-IID data settings. For `SelSync`, we choose three different (ϕ, ψ, δ) configurations: (0.5, 0.5, 0.05), (0.5, 0.5, 0.3) and (0.75, 0.75, 0.3). As per



(a) ResNet101 on CIFAR10 (b) VGG11 on CIFAR100

Figure 8.10: `SelSync` with stochastic data-injection strategy at different (ϕ, ψ, δ) configurations vs. FedAvg over non-IID data.

Eqn. (8.2), we set $b^{'} = 11$ to keep $b = 32$ for the first two configurations, and $b^{'} = 6$ for the last. For non-IID CIFAR10, ResNet101 with FedAvg oscillated between 55-70% test accuracy. For VGG11 training over skewed CIFAR100, FedAvg saturated to 70% test accuracy. As the value of (ϕ, ψ, δ) increases, the corresponding data distribution improves along with the test accuracy. `SelSync` configuration (0.75, 0.75, 0.3) attained the maximum accuracy, followed by (0.5, 0.5, 0.3) and (0.5, 0.5, 0.05) respectively. Data-injection thus improves the statistical efficiency and works well in the context of semi-synchronous training.

8.5 Conclusion

From Table (8.1), we see that BSP takes the least number of steps to converge since it performs more work per-training iteration by combining updates from all clients. However, it suffers from high communication cost of aggregating updates from distributed clients. On the other hand, federated algorithms like FedAvg provides appreciable speedups courtesy of its high LSSR scores; however, the (c, f) configuration that provides optimal speedup or even convergence guarantees varies for each model, dataset and training hyperparameters. As already seen, naively aggregating updates in fixed intervals does not always yield the same convergence as BSP. Although clients communicate asynchronously in SSP, exchanging updates for larger and deeper models with a centralized server can introduce staleness issues that may slowdown or saturate DNNs. Both FedAvg and SSP take more iterations to converge than BSP as they perform lesser work per-iteration (by using smaller batches in SGD update). Across all DNNs, `SelSync` attained either the same or better statistical performance than BSP by selectively synchronizing crucial updates, and applying local updates

otherwise. The gradient change metric serves as an effective indicator of the significance of each update in FL. With its high LSSR scores, `SelSync` eliminates considerable communication cost in the training phase while allowing sufficient exploration of the local optimization landscape and bounds the divergence between the local and global model states from infrequent synchronization.

8.6 Related Publications

1. **Tyagi, Sahil** and Swany, Martin. "Accelerating Distributed ML Training via Selective Synchronization." 2023 IEEE International Conference on Cluster Computing (CLUSTER) (2023): 1-12.

2. **Tyagi, Sahil** and Swany, Martin. "Accelerating Distributed ML Training via Selective Synchronization (Poster Abstract)." 2023 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops) (2023): 56-57.

CHAPTER 9
SUMMARY AND OUTLOOK

The following sections summarize the contributions of this thesis and discuss its potential impact, along with directions for future research.

9.1 Optimizing Compute Cost in Deep Learning

In this section, we studied the impact of distributed training on heterogeneous clusters, and proposed a simple workload-balancing framework called `OmniLearn` in Chapter (3) to equalize the computation cost in a cluster. The impact of 'stragglers' in synchronous training, and 'staleness' in asynchronous training is considerably attenuated, although not completely eliminated by our *variable-batching* technique. For more accurate throughput estimation, as well as adapting to dynamic heterogeneity in a cluster, `OmniLearn` develops proportional-control [172] based *dynamic-batching*. Implemented in TensorFlow [97] and PyTorch [98], `OmniLearn` significantly improves the parallel and statistical efficiency in both BSP and ASP training. The results of this research were published in [66, 29].

Our system is ideal for learning in federated settings as well, since heterogeneity is ubiquitous across edge devices [225]. It is also applicable for low-cost training in the cloud with transient or spot instances [226] and burstable VMs [174], where reclaimed VMs may be considered as heterogeneous nodes with 'zero' compute capability and adjust the training process on the remaining nodes accordingly. Our framework can be integrated with other systems [227, 228, 229, 173] to leverage transient VM availability for low-cost, high-reliability training in the cloud.

Some critical hyperparameters specific to distributed DL are *cluster-size* and *batch-size*, where training on a particular hardware can either be scaled horizontally by adding more nodes, and scaled vertically by raising the batch-size. Choosing the right configuration involves large and complex trade-off space between the running 'cost' and 'time' when training in the cloud. In fact, cost-time are pareto-related as one improves at the detriment of the other before reaching a point of saturation where adding significantly more resources spikes up the cost but only marginally reduces

the training time. Chapter (4) proposes `Scavenger` to address the challenge of exploring the "right" resource configuration in DDL. It develops a *parallel* and a *statistical* performance model to estimate and construct cost-time curves of DNNs. We predict parallel performance as a linear-relationship between computation time and training batch-size, and model communication time as a function of cluster-size in a centralized PS strategy. To measure statistical efficiency in DL, we use 'gradient noise' to predict the total epochs or iterations needed by DNNs to meet convergence targets. Using these performance models, `Scavenger` estimates the total running cost and training time of a specific cluster and batch-size configuration under different pricing policies in the cloud. This work was published in [230, 231].

With its adaptive scaling approach, our system is ideal for elastic training [185, 232] to intelligently tweak training configurations as worker availability changes. ML-as-a-service offerings like AWS (Amazon web services) SageMaker [194] that use rudimentary performance models can also leverage the statistical efficiency of `Scavenger` to determine the pareto-optimal resource configuration. Concurrently, other elasticity mechanisms and training policies can also be integrated with `Scavenger` [233, 197]. Using gradient noise as a scalability indicator can also be extended to other DL schedulers [234, 235] for better resource allocation and job placement of DNNs in training clusters.

9.2 Mitigating Communication Cost in Deep Learning

Another challenge in DDL is the expensive communication cost, especially when the compute, size and space requirements of sota DNNs grows at an exponential rate [12]. BSP training aggregates model parameters or gradients at every iteration, resulting in low parallel efficiency. This is further exacerbated in limited network settings where latency is high and bandwidth is low. Although high communication overhead can be attenuated via asynchronous training, ASP typically results in lower model quality compared to BSP. Lossy gradient compression techniques have emerged as viable mechanisms to improve parallel performance in DDL by reducing its communication overhead, depending on the chosen degree of compression (i.e., its CF). Due to its lossy nature, there is an inverse relationship between parallel and statistical efficiency of DL; higher CFs reduce the communication cost further but likely result in lower quality model from applying partial or approximated

updates. Chapter (5) presents GraVAC to balance parallel and statistical efficiency by adjusting CF throughout training in order to attain high training speedups while meeting high convergence targets. To account for statistical efficiency in lossy compression, we propose '*Compression Gain*' and empirically validate how it affects a DNNs' convergence quality, followed by describing '*Compression Throughput*' that combines parallel and statistical performance metrics into one. Under multiple scaling policies, GraVAC successfully switches CF over different phases of training and converges up to 6.67× faster than a static compression technique for similar accuracy-levels. We published our results in [31].

In our evaluation, we implement GraVAC's adaptive compression scheme over Top-k [114], DGC [115], Random-k [111] and RedSync [127]. But GraVAC is compatible with other lossy compression techniques as well, and can easily be integrated into the training routine. For compression in FL over non-IID data, *compression gain* can measure the significance of device-local updates so as to tune CF on a per-device level.

In decentralized settings, model updates are aggregated using different implementations of AR collective such as ring, tree, recursive doubling, scatter-reduce-allgather, etc. But the lossy compression techniques described as part of GraVAC use AG collective to communicate its update among workers. Based on the latency-bandwidth cost model, MPI collectives have different communication overheads depending on network infrastructure and topology, cluster-size, message-size, network latency and bandwidth. Communication overhead also varies based on job placement in the cluster; minimal between intra-node workers with high-bandwidth interconnects, and higher over inter-node workers due to limited bandwidth over the network. Further, network bandwidth may vary over time due to congestion and contention. Thus, compression with AG may not always necessarily be the optimal collective to aggregate updates. Chapter (6) proposes AR-Top$k$, a compression-communication mechanism compatible with AR collective. In our evaluation, we showed how the different worker selection mechanisms in AR-Top$k$ achieve similar or better statistical performance than AG-based compressors using a static CF. To balance the parallel and statistical efficiency in gradient compression, and choose the optimal collective when network performance fluctuates, we model compression as a MOO problem to find a CF that minimizes compression and communication time, and maximizes compression gain in DDL. We extensively evaluate our proposed approach over heterogeneous networks by emulating varying degrees of latency

and bandwidth, and show how AR-Top$k$ with MOO attains good convergence quality by dynamically adjusting its CF and collective operation throughout training. The results of this research were published in [32].

As we validate our approach over heterogeneous networks, we conjecture that our adaptive approach can be leveraged in shared clusters with transient jobs, which affects the inter-node or intra-node bandwidth available to currently running DL jobs. The latency-bandwidth cost model heuristics presented here could also be extended to resource and job schedulers for DDL applications, where larger DNNs or high-priority jobs may be placed on higher-bandwidth nodes in order to minimize the makespan or job completion time. In future, we plan to evaluate AR-Top$k$ over large language models (LLMs) to assess the trade-offs between parallel and statistical performance over massive attention-based models as well. As part of integration with other distributed training paradigms, we expect our adaptive compression mechanisms to work with model-parallelism as well, where gradients or feedforward maps between subsequent layers are compressed to different CFs depending on the layer's sensitivity.

9.3 Accelerating Convergence in Federated Learning

FL presents several challenges in terms of parallel and statistical aspect of training, especially at the edge. Parallel efficiency suffers due to limited compute and bandwidth, while statistical inefficiency stems from unbalanced and non-IID data. These challenges are further amplified when learning in an online manner over transient streaming data. Chapter (7) introduced ScaFLES, a system for scalable FL over streaming data at the edge that addresses the problem of heterogeneous stream-rates, limited memory and storage on devices, expensive synchronization overhead and improving convergence over non-IID data. Parallel performance in heterogeneous streams is improved via 'variable' computing by processing batches proportional to a device's local stream-rate, while the exponentially-growing buffering queue is kept in check by different queuing policies. Communication over limited networks is improved via adaptive compression using a lower and upper threshold on the compression factor. For non-IID training, we propose 'stochastic data-injection' that significantly improves convergence quality over the baseline. To further boost its statistical performance, ScaFLES performs weighted gradient aggregation and learning-rate scaling based

on device stream-rates. With extensive evaluation by simulating concurrent heterogeneous streams over Apache Kafka [215], we reduce the overall training time by up to 3.3×, attenuate buffering queue and communication cost while attaining good convergence quality under IID and non-IID settings. Our research pertaining to learning over streaming data is published in [33, 3]. While we explained how data-injection proposed in this chapter ensures K-anonymity [224], we plan to do a deep theoretical analysis in terms of its privacy guarantees in the future.

Distributed training algorithms like FedAvg and SSP minimize communication overhead at the cost of lower model accuracy compared to BSP. Chapter (8) introduced `SelSync`, a statistically-aware federated learning algorithm that avoids excessive synchronization while attaining BSP-level accuracy targets. The 'gradient change' metric that we described here works as an effective indicator of statistical performance and detects significant updates. Based on this, `SelSync` employs an opportunistic synchronization mechanism where important updates are communicated among client devices, while trivial updates are applied locally. For semi-synchronous training, we empirically validate the efficacy of aggregating model parameters over gradients, and show empirically why the former performs better. We further improve `SelSync`'s statistical performance with a custom data-partitioning scheme and extend the data-injection strategy (from Chapter (7)) in semi-synchronous training. With rigorous experimental evaluation, we show that `SelSync` is up to 13.75× faster than BSP and successfully converges to similar accuracy-levels. This results of this work were published in [34, 35].

In future, we plan to extend the adaptive strategy of `SelSync` and introduce it in the context of differential privacy [236], where we vary privacy coefficients (ϵ, δ) based on the significance of each update in FL. We expect this to provide robust privacy guarantees for training DNNs in federated settings.

APPENDIX A

TRAINING OVER HETEROGENEOUS CLUSTERS WITH `OMNILEARN`

A.1 Adaptive Training in `OmniLearn`

Algorithm 2: `OmniLearn` over BSP

Input: N, B, E, η, $w_{(0)}$, Δb_{min}, b_{min}, b_{max}

procedure `Train()`:

- $b_{(0,n)} = B/N$
- **for** *epoch* e *in* `range(0, E)` *on worker n* **do**
  - $\lambda_n = b_{(e,n)}/B$
  - $\mathcal{B} =$ `batchTrainingData(`$b_{(e,n)}$`)`
  - **for** *iteration* i *in* $0, 1, 2, ...$ **do**
    - $t^{(init)} =$ `time()`
    - $g_{(i,n)} = \frac{\partial}{\partial w_i}(\frac{1}{|b_{(e,n)}|}\sum_{\mathcal{B}_{(i,n)}} \mathcal{F}(\mathcal{B}_{(i,n)}, w_i))$
    - $\bar{t}^{(compute)}_{(i,n)} =$ `EWMA(`$($`time() -` $t^{(init)})$`)`
    - $\bar{g}_{(i)} =$ `Aggregation(`$\lambda_n \odot g_{(i,n)}$`, SUM)`
    - $w_{(i+1)} = w_{(i)} - \eta \cdot \bar{g}_{(i)}$
  - $\bar{t}^{(avg)}_{(e)} =$ `getAvgComputeTime(`$\bar{t}^{compute}_{(i,n)}$`)`
  - $b_{(e+1,n)} =$ `evalBatchSize(`$b_{(e,n)}, \bar{t}^{(compute)}_{(i,n)}, \bar{t}^{(avg)}_{(e)}$`)`

procedure `evalBatchSize(`$b_{(e,n)}, \bar{t}^{(compute)}_{(i,n)}, \bar{t}^{(avg)}_{(e)}$`)`:

- $T_{(e,n)} = b_{(e,n)} \,/\, \bar{t}^{(compute)}_{(i,n)}$
- $b^{(*)}_{(e+1,n)} = b_{(e,n)} - T_{(e,n)}(\bar{t}^{(compute)}_{(i,n)} - \bar{t}^{(avg)}_{(e)})$
- **if** $\frac{|b^{(*)}_{(e+1,n)} - b_{(e,n)}|}{b^{(*)}_{(e+1,n)}} \geq \Delta b_{min}$ **then**
  - **if** $b^{(*)}_{(e+1,n)} < b_{min}$ **then**
    - $b_{(e+1,n)} = b_{min}$
  - **else if** $b^{(*)}_{(e+1,n)} > b_{max}$ **then**
    - $b_{(e+1,n)} = b_{max}$
  - **else**
    - $b_{(e+1,n)} = b^{(*)}_{(e+1,n)}$
- **return** $b_{(e+1,n)}$

Algorithm 3: `OmniLearn` over ASP

Input: K, B, E, η, $w_{(0)}$, Δb_{min}, b_{min}, b_{max}

On Workers:

procedure `Train()`:

- $b_{(0,n)} = B/N$
- **for** *epoch* e *in* `range(0, E)` *on worker* k **do**
  - $\eta_n = \eta \cdot b_{(e,n)} \;/\; B$
  - $\mathcal{B} =$ `batchTrainingData(`$b_{(e,n)}$`)`
  - **for** *iteration* i *in* $0, 1, 2, ...$ **do**
    - $w_{(i)} =$ `async.pullFromPS()`
    - $t^{(init)} =$ `time()`
    - $g_{(i,n)} = \frac{\partial}{\partial w_i}(\frac{1}{|b_{(e,n)}|}\sum_{\mathcal{B}_{(i,n)}} \mathcal{F}(\mathcal{B}_{(i,n)}, w_i))$
    - $\bar{t}^{(compute)}_{(e,n)}$ = `EWMA(`$($`time() -` $t^{(init)}$`)`$)$
    - $\theta_{(i+1,n)} = \theta_{(i)} - \eta_n \cdot g_{(i,n)}$
    - `async.pushToPS(`$\theta_{(i+1,n)}$`)`
  - $b_{(e+1,n)}$ = `evalBatchSize(`$k, b_{(e,n)}, \bar{t}^{(compute)}_{(e,n)}$`)`

On Parameter Server:

procedure `async.pullFromPS()`:

- **return** $w_{(i)}$

procedure `async.pushToPS(`$w_{(i+1,n)}$`)`:

- $w_{(i+1)} = w_{(i+1,n)}$

procedure `evalBatchSize(`$n, b_{(e,n)}, \bar{t}^{(compute)}_{(e,n)}$`)`:

- `wrkB[`n`]` = $b_{(e,n)}$
- `wrkTime[`n`]` = $\bar{t}^{(compute)}_{(e,n)}$
- **if** `sizeOf(wrkB)` = N **then**
  - $\bar{t}^{(avg)}_{(e)}$ = `mean(wrkTime.values())`
  - **for** `w_id`, b *in* `wrkB.items()` **do**
    - $\bar{t}^{(compute)}_{(e,n)}$ = `wrkTime[w_id]`
    - $b_{(e+1,n)}$ = `getBsz(`b, $\bar{t}^{(compute)}_{(e,n)}$, $\bar{t}^{(avg)}_{(e)}$`)`
    - `wrkB[w_id]` = $b_{(e+1,n)}$
  - **return** `wrkB[`n`]`

Alg. (2) illustrates how `OmniLearn` integrates control stability with proportional-control in BSP training. Initially, all workers use a mini-batch of B/N for global batch-size B across N workers and train for a total of E epochs. The '`batchTrainingData`' function preprocesses and batches the training data with the specified batch-size. After computing gradients, we apply exponential smoothing function `EWMA` over compute-time at every iteration 'i' of worker 'n' to avoid sporadic and erroneous readings. This is followed by gradient aggregation via '`Aggregation`', where updates can be synchronized in a centralized manner over a PS or in decentralized topology via Allreduce collective. Each workers' updates are scaled by 'λ_n' and summed so as to perform weighted aggregation in BSP. In our implementation, we trigger batch adjustment at the end of every epoch, although it can even be triggered to execute after certain iterations as well. We then synchronize compute time across all workers with '`getAvgComputeTime`' to get the cluster's average computation time $\bar{t}_e^{(avg)}$. The overhead of this operation is low as just a 32-bit float is sent by each worker, followed by execution of '`evalBatchSize`' procedure to calculate the optimal batch-size as per in Eqn. (3.4). Batch adjustments are made only if it exceeds the preset threshold Δb_{min}, and worker batches are chosen according to the min-max batch-size bounds and fixed B.

For ASP training, `OmniLearn`'s pseudocode is shown in Alg. (3). Each worker 'n' begins training at epoch 0 with the same initial mini-batch $b_{(0,n)}$. Unlike BSP, we do not scale per-worker gradients since only one worker updates the parameters on the central PS in a given iteration. Instead, we scale learning-rate relative to the local batch-size on each worker (line 6). The latest model state is pulled from the PS at the beginning of every iteration, gradients are calculated over the training data, followed by applying SGD update on local replica and sending the updated model back to the PS. In ASP, each worker repeats these steps independently and asynchronously of each other. The batch-controller resides on the PS and triggered remotely by the workers via `evalBatchSize` function that takes worker ID, local batch-size and average compute time as arguments. The PS holds the last two in a hashmap or dictionary corresponding to every worker ID, calculates the average cluster compute-time and applies control stability for batch adjustments.

The average computation time for a worker measured over an epoch may vary slightly between BSP and ASP, as the number of samples processed by each worker varies between the two. In BSP, compute-time is logged at every iteration across all workers, while high-resource workers in ASP execute more iterations for accurate measures compared to low-resource workers.

A.2 Cluster Setup and Training Configuration

`OmniLearn` in built atop both Tensorflow's Estimator framework [97, 191] and PyTorch's DDP and RPC modules v.1.12.1 [64, 98, 204]. ASP is implemented over centralized PS over Tensorflow's 'ParameterServerStrategy' and PyTorch's RPC framework, while BSP is implemented in a decentralized manner via MPI [24] over PyTorch DDP.

We evaluate `OmniLearn`'s performance in different scenarios of static and dynamic heterogeneity by comparing with conventional distributed training with uniform-batching in ASP or BSP, and report final model accuracy and reduction in worker compute times. We use the following training schedule and model hyperparameters in our evaluation:

- *ResNet18* [9] trains on Food101 dataset [182] with per-worker batch-size 32, momentum SGD with factor 0.9 and weight decay 5e-4, initial learning-rate 0.1 that decays by a factor of 10 after 15, 30 and 45 epochs respectively.
- *AlexNet* [1] runs with worker batch-size of 32 over the CalTech101 dataset [181] using momentum 0.9, weight decay 5e-4, and initial learning-rate 1e-2 decaying by a factor of 10 at epochs 80, 120 and 160.
- *ResNet50* [9] trains on CalTech256 [213] with per-worker batch-size 32, momentum 0.9, weight decay 1e-4, initial learning-rate 0.1, and decay factor 10 after 100, 150 and 200-th epoch.
- *VGG11* [8] is trained on the Places365 dataset [237] with per-worker batch-size 32, momentum 0.9, weight decay 5e-4, initial learning-rate 1e-2 and decay factor of 5.0 at epochs 15, 25 and 35.

We simulate heterogeneity via Docker v23.0.5 over 48-core Intel Xeon E5-2670 @2.3GHz Ubuntu v20.04.6 servers with 384 GB memory on each. Spawning workers as docker containers ensures minimum interference, resource isolation and helps emulate various heterogeneity scenarios. We simulate different HLs with respect to the allocated CPU cores by mapping and limiting a container to a specific set of CPU-cores [238] and allocating 32 GB memory to each. PS is allocated 8 cores and 48 GB memory in ASP, and containers communicate via docker swarm over 5Gbps NIC.

APPENDIX B

ADAPTIVE GRADIENT COMPRESSION WITH GRAVAC

B.1 GraVAC's Compression Algorithm

Alg. (4) illustrates GraVAC's adaptive compression technique with compressor $\mathcal{C}$ to scale CFs in the exploration space $[\theta_{min}, \theta_{max}]$, such that each candidate CF is evaluated for iterations corresponding to 'window' and incremented by step-size θ_s with respect to θ_{min}. For e.g., scaling CF from $10\times$ to $20\times$ implies $\theta_s = 20/10 = 2\times$. The gain threshold '$\epsilon$' denotes the minimum compression gain required for any CF to be eligible for communication, while 'ω' is used to measure the saturation in compression throughput $T_{compression}$ and for scaling up θ_{min} as well. At every iteration, we compute gradients $g_o^{(i)}$ for a DL model with parameters w_i on training sample $x^{(i)}$ in time t_o, and append the residual gradients from previous iterations (line 4). In the first stage, $\mathcal{C}$ compresses gradients at minimum CF θ_{min} to $g_{min}^{(i)}$ and calculates the corresponding compression gain δ_{min} (line 5). As the gradients computed across iterations can be noisy, so can the compression gain. We thus apply EWMA to smoothen out the inter-iteration gain; we set EWMA smoothing factor as a percentage of the cluster-size, N/100. GraVAC then evaluates the next candidate CF by further compressing $g_{min}^{(i)}$ by factor θ_s (line 6). Thus, the candidate CF evaluated in this case effectively becomes $(\theta_{min} \cdot \theta_s)$. This is done as part of GraVAC's multi-level compression strategy to avoid compressing the larger, original gradients $g_o^{(i)}$ twice; once to CF θ_{min}, and a second time to candidate CF θ_c $(= \theta_{min} \cdot \theta_s)$. GraVAC then calculates the compression gain for candidate CF $(\theta_{min} \cdot \theta_s)$ and measures the total compression time $t_{compress}$ (line 7). The total compression cost is comprised of t_{min}, compressing gradients $g_o^{(i)}$ to $g_{min}^{(i)}$ and t_c, time taken to compress $g_{min}^{(i)}$ to $g_c^{(i)}$.

Based on the compression gains obtained corresponding to θ_{min} and θ_c (i.e., $\theta_{min} \cdot \theta_s$), we select the appropriate gradients to communicate among workers. If the compression gain of our current candidate CF meets gain threshold ϵ (line 8), we aggregate compressed gradients $g_c^{(i)}$ and update the residuals (line 9). With the total iteration time t_{iter} (line 10), GraVAC updates the system and compression throughput for θ_c (as shown in the UpdateMetrics function). Here, $T_{compreession}$ is a hashmap or a dictionary that stores compression throughput for each candidate CF, the min-max

Algorithm 4: Adaptive Compression with `GraVAC`

Input: θ_{min}, θ_{max}, ϵ, θ_s, ω, window, compressor $\mathcal{C}$

w_o : initial model state, N: total nodes, b: per-worker batch-size, $\mathrm{T}_{sys} = 0, \mathrm{T}_{compress}$ = `empty()`, `residual` = `torch.zeros(shape=`w_o`)`

for *i = 1,2,3...* **do**

- $g_o^{(i)}, t_o = \nabla f(x^{(i)}, w_i)$ and $g_o^{(i)} = g_o^{(i)} +$ residual
- $g_{min}^{(i)}, t_{min} = \mathcal{C}(g_o^{(i)}, \theta_{min})$ and $\delta_{min} = \text{EWMA}(\frac{||g_{min}^{(i)}||^2}{||g_o^{(i)}||^2})$
- $g_c^{(i)}, t_c^{(i)} = \mathcal{C}(g_{min}^{(i)}, \theta_s)$ and $\delta_c = \text{EWMA}(\frac{||g_c^{(i)}||^2}{||g_o^{(i)}||^2})$
- $t_{compress} = t_{min} + t_c$
- **if** $\delta_c \geq \epsilon$ **then**
  - $\tilde{g}^{(i)}$, t_s = `Synchronize(`$g_c^{(i)}$`)` and `residual` = $g_o^{(i)} - g_c^{(i)}$
  - t_{iter} = $t_o + t_{compress} + t_s$
  - `UpdateMetrics(`$\theta_s \cdot \theta_{min}, \delta_c, t_{iter}$`)`
- **else if** $\delta_c < \epsilon$ *and* $\delta_{min} \geq \epsilon$ **then**
  - $\tilde{g}^{(i)}$, t_s = `Synchronize(`$g_{min}^{(i)}$`)` and `residual` = $g_o^{(i)} - g_{min}^{(i)}$
  - t_{iter} = $t_o + t_{compress} + t_s$
  - `UpdateMetrics(`$\theta_{min}, \delta_{min}, t_{iter}$`)`
- **else**
  - $\tilde{g}^{(i)}$, t_s = `Synchronize(`$g_o^{(i)}$`)` and `residual` = 0
  - t_{iter} = $t_o + t_{compress} + t_s$
  - `UpdateMetrics(`$1, 1, t_{iter}$`)`
- $w_{i+1} = w_i - \eta \cdot \tilde{g}^{(i)}$
- θ_s = `EvaluateGraVAC(`*i,* $\theta_s, \delta_{min}, \delta_c$`)`

procedure `UpdateMetrics(`θ, δ, t_{iter}`)`**:**

- $\mathrm{T}_{sys} = \frac{(\mathrm{N} \cdot \mathrm{b})}{t_{iter}}$ and $\mathrm{T}_{compress}[\theta] = \mathrm{T}_{sys} \cdot \delta$

procedure `EvaluateGraVAC(`*i,* $\theta_s, \delta_{min}, \delta_c$`)`**:**

- **if** *i* % window $== 0$: **then**
  - $\theta_s = \text{ScalingPolicy}(\theta_s)$
  - **if** $\omega \geq \frac{|\delta_{min} - \delta_c|}{\delta_{min}}$: **then**
    - $\theta_{min} = \theta_s \cdot \theta_{min}$
  - ct = sort($\mathrm{T}_{compress}$.values())
  - **if** $|\frac{ct[\text{-}1] \; - \; ct[\text{-}2]}{ct[\text{-}2]}| \leq \omega$: **then**
    - $\theta_{ideal} = \mathrm{T}_{compress}.get(\text{ct}[\text{-}2])$
    - return $\theta_{ideal}/\theta_{min}$
  - **else**
    - return θ_s

CFs in the exploration space, as well as for DenseSGD (i.e., CF 1×).

If the compression gain of candidate CF does not meet the gain threshold, but that of θ_{min} does (line 12), we synchronize $g_{min}^{(i)}$ corresponding to θ_{min}, update residuals, calculate total iteration time and update compression throughput for θ_{min} (line 13-15). It is important to note that the communication cost of transmitting $g_{min}^{(i)}$ is more than sending $g_c^{(i)}$ due to the former's lower CF. *The trade-off* `GraVAC` *makes is to incur higher communication overhead for a more accurate representation of the original gradients (as measured by compression gain) and vice versa*.

If neither θ_{min} or θ_c meet the gain threshold ϵ, we transmit the original, dense tensors (line 17). We also set the residual gradients to 0 as there are no accumulated gradients from error-feedback in this case. Even though we exchange dense tensors, we still incur $t_{compression}$ with DenseSGD as compression cost was incurred for evaluation (lines 4-6) in `GraVAC` (with total iteration cost shown at line 18). The CF and compression gain are both 1.0, and set accordingly in UpdateMetrics function (line 19).

Following SGD update (line 20), `GraVAC` evaluates the parallel and statistical performance of candidate CFs after 'window' iterations. Here, the scale-up factor θ_s raises the amount of compression performed according to some ScalingPolicy, giving the next candidate CF to evaluate (line 26). `GraVAC` explores two compression scaling policies, described in further detail in §5.4.1. In addition to θ_c, this policy also raises the minimum compression value θ_{min} by factor θ_s, until it reaches the upper-bound θ_{max}. The intuition behind raising θ_{min} is that as training progresses, model converges and we can use higher CFs even for the minimum CF in later training stages. `GraVAC` increases θ_{min} if the candidate CF gain δ_c is within $\omega\%$ of the current minimum CF's gain δ_{min} (line 27-28). Once ample CFs are evaluated, we look at the two largest compression throughputs (line 29) and fetch the smaller CF if they are within the bounds of ω, implying that compression throughput has saturated and thus, set the appropriate step-size of θ_s for the optimal CF (lines 29-32). If threshold ω is not met, we use θ_s as it is (line 34).

When does compression scale-up? From Alg. (4), compression scale-up occurs when step-size θ_s is increases according to a scaling policy. At the same time, we raise the minimum CF θ_{min} to current candidate CF if the two gains are within $\omega\%$ of each other.

When does compression scale-down? Compression scale-down is determined by ϵ (shown via conditional statements between lines 8-19). If the current candidate CF loses considerable gradient

information, we use a smaller CF θ_{min}. If the latter also fails to meet the minimum gain threshold, we communicate the original, uncompressed gradients $g_o^{(i)}$.

B.2 Cluster Setup and Training Hyperparameters

`GraVAC` is evaluated on a 32-GPU cluster on the Google cloud platform (GCP) [69] across 8 VMs. Each VM is a `n1-standard-8` with 8vCPUs, 30GB system memory and 4 NVIDIA V100s [220] with 16GB vRAM each. The machines are configured with PyTorch 1.10.1 [98], CUDA 11.3 [239], CUDA driver 465.19.01 and NCCL 2.10.3 [56]. We evaluate the three DNNs described in Table (5.1) in the following configurations:

- ResNet101 [9] is trained on CIFAR10 [180] with per-worker mini-batch 32, momentum 0.9, weight decay 1e-4 and SGD optimizer with initial learning-rate 0.1 decayed by a factor of 10 at iterations 9000 and 14000 respectively.
- VGG16 [8] is trained on CIFAR100 [180] with per-worker batch-size 32, weight decay 5e-4, momentum 0.9 and SGD with fixed learning-rate 0.1.
- LSTM [202] is measured for test perplexity (i.e, exponential of the test loss) on PTB dataset [240] with per-worker batch-size 20, momentum 0.9, weight decay 1e-4 and SGD with fixed learning-rate 0.1. The model is intialized with 1500 embedding dimensions and 2 hidden layers with 35 bptt (backpropagation through time) steps.

APPENDIX C

FEDERATED LEARNING OVER STREAMING DATA WITH SCAFLES

C.1 Cluster and Training Setup

We simulate data-streams over Apache Kafka [215] by sampling streaming rates from uniform and normal distributions as described in §7.2.1. Our training cluster is composed of 4 servers each with a 48-core Intel Xeon E5-2650, 128 GB system memory and 8 NVIDIA K80 GPUs [241] connected over 5Gbps ethernet. We mimic CUDA-aware [239] edge devices by spawning them as `nvidia-docker` containers [177] on CentOS Linux 7.9.2009 with docker engine 20.10.17. Each device running as a container is allocated 4vCPUs, 12 GB memory and 1 K80 GPU running NVIDIA driver 465.19.01 on CUDA 11.3 and PyTorch 1.10.1 [98]. We create a docker swarm network on the 5Gbps NIC to enable communication for gradient synchronization among the containers.

We train two popular computer vision models across different streaming distributions. ResNet152 [9] is trained on CIFAR10 [180] using Nesterov's optimizer with momentum 0.9, weight decay 1e-4 and a learning-rate schedule with initial step-size 0.1 that decays by a factor of 0.2 after epoch 75, 150 and 225 respectively. VGG19 [8] trains on CIFAR100 [180] with momentum SGD of factor 0.9, weight decay 5e-4 and initial step-size 0.01 that decays by 0.3 after 75, 150 and 200 epochs. We use test accuracy as a measure of model quality when trained under IID and non-IID data settings, the cluster setup for which is outlined in Table (C.1). For IID data, we train over a cluster of 16 devices where each container is equipped with a K80 GPU and training labels are evenly partitioned across all devices, while non-IID data is partitioned by mapping a device to a unique subset of labels. Non-IID CIFAR10 is trained on 10 devices where each device contains a single label, while non-IID CIFAR100 runs on 25 devices with 4 labels per-device to partition the 100 labels.

C.2 Simulating Streaming Data in ScaFLES

We spawn an additional docker container running Kafka v3.1.0 to launch a broker and multiple producers. The container has 16vCPUs and 32GB system memory, but does not contain any GPUs

Model	# Parameters	Dataset	Devices	# Labels per-device
ResNet152	60.2×10^6	IID CIFAR10	16	10
		non-IID CIFAR10	10	1
VGG19	143.7×10^6	IID CIFAR100	16	100
		non-IID CIFAR100	25	4

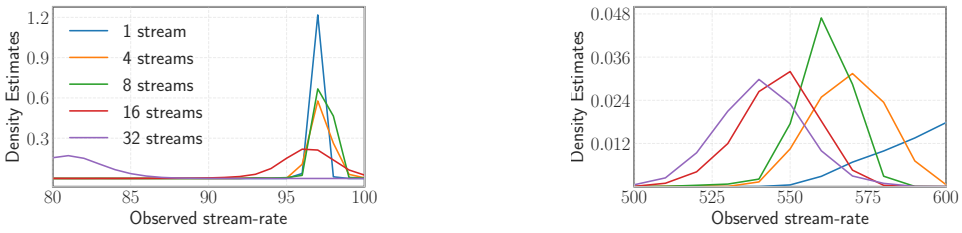
Table C.1: Cluster setup for training over IID and non-IID data

since it does not partake in model training. We configure the broker with 8 network threads, 4 I/O-threads and 1 partition per-topic. The purpose of this container is to host the Kafka broker and launch multiple producers such that each process publishes to a unique topic corresponding to a device. Thus, there are as many topics as the number of training devices. A producer simulates and controls data streams at varying rates corresponding to a devices' topic. For data consumption, training devices have a running Kafka consumer process, which implements a custom PyTorch dataloader to batch training data and feed it to the main training process.

As we spin multiple producers from a single container, we measure the effective stream-rates achieved with our streaming simulator to ensure that we meet the target stream-rates in our evaluation. For e.g., running 32 streams in Fig. (C.1a) implies 32 producers are publishing across 32 topics at the rate of 100 samples/sec. We scale up the number of concurrent producers (i.e., topics for data-streams) and measure the observed streaming-rate with the objective to attain close to 100 and 600 samples/sec. rates. Fig. (C.1) shows the total density estimates of the target rates when 1, 4, 8, 16 and 32 producers are deployed in parallel. With the target rate of 100 samples/sec., we see in Fig. (C.1a) that we nearly achieve the target rate across all producers. For the 600 samples/sec. target, the effective streaming-rate achieved per-topic is slightly below target, particularly beyond 16 parallel streams. Although higher stream-rates can be attained by increasing the number of network threads and partitions per-topic, this cluster and setup is sufficient to simulate heterogeneous data-streams and execute training over streaming data in `ScaFLES`.

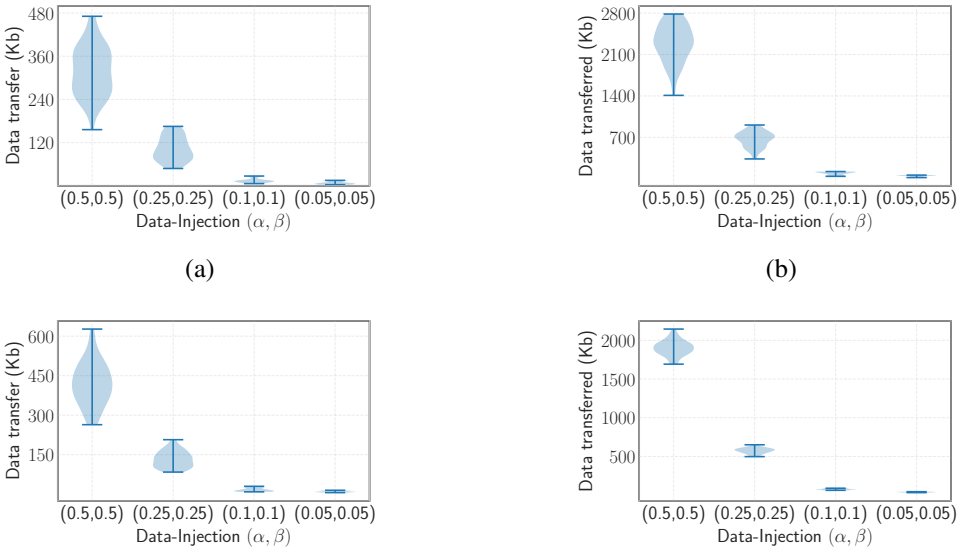
C.3 Communication Overhead of Stochastic Data-Injection

Although data-injection improves the statistical efficiency by augmenting the overall data distribution, the operation itself has an associated network overhead as some devices transfer few training samples to other devices in the network. For CIFAR10 and CIFAR100, each training sample is an



(a) (b)

Figure C.1: Effective streaming-rate achieved while simulating multiple heterogeneous streams at (a) 100 samples/sec. (b) 600 samples/sec.



(a) (b)

(c) (d)

Figure C.2: Data-transfer overhead in a single iteration with stochastic data-injection for various (α, β) in (a) $S_{uniform}$ (b) $S'_{uniform}$ (c) S_{normal} (d) S'_{normal}.

image with an average size of 3 Kb (kilobytes). For the four streaming distributions, we look at the density distribution of total communication incurred (in Kb) for various (α, β) configurations. Compared to the network overhead of synchronizing model updates, this overhead is negligible and ranges from only 150-2000 Kb in a single iteration. As we prioritize privacy and lower (α, β) parameters, the amount of data transferred over the network is subsequently smaller. Thus, data-transfer overhead gets lower as we reduce the (α, β) configuration. Transfer overhead is higher in $S'_{uniform}$ and S'_{normal} as the inflow stream-rates are higher compared to $S_{uniform}$ and S_{normal}.

APPENDIX D

SEMI-SYNCHRONOUS FEDERATED LEARNING WITH `SELSYNC`

D.1 Training with `SelSync`

`SelSync` improves parallel efficiency by avoiding communication with local-SGD, while synchronizing crucial model updates across clients under IID and non-IID conditions. We fashion `SelSync` over the PS architecture where worker updates are coordinated through a centralized server, as described in Alg. (5). Updates are pulled from the PS via `pullFromPS` function, while updates are sent and averaged on the PS with the `pushToPS` call. This can also be implemented in decentralized training where clients communicate via MPI-based collectives. Training starts from line 3 when clients pull initial model parameter state from the PS with `pullFromPS` call.

Gradients and its $\mathbb{L}$-2 norm is computed on a mini-batch sampled from $\mathcal{D}_c$ on client 'c' at lines 7-8. The `DetectGradientChange` function calculates and stores client c's gradient norm, applies EWMA smoothing and computes the $\triangle(g_i^{(c)})$ metric on every iteration (i.e., from Eqn. (8.1)). The local gradients then update the model locally on the client (line 9). To keep track of $\triangle(g_i^{(c)})$ *across all clients*, we initialized a flags array of length 'N' (i.e., corresponding to the total clients participating in training). Each index in the array is initially set to 0 at the beginning of each iteration (line 5); a bit-value of 0 at index 'c' in `flags` implies $\triangle(g_i^{(c)})$ *of client id 'c' is below the δ-threshold and thus, choose to perform SGD update locally*. Similarly, a bit-value of 1 at index 'c' means that *client c's relative gradient change metric exceeds the δ-threshold and chooses to communicate its update with other clients*. Lines 10-11 show that if $\triangle(g_i^{(c)})$ on a client is greater than δ, we flip its corresponding bit at index 'c' to 1. To aggregate the decision of all clients about whether to synchronize updates or apply locally, we call `AllGather` on `flags` where each client stores its decision at an appropriate index (line 12). Since we exchange only a single flag-bit from each client, the total communication volume of this step is negligible, $(N-1)$ bits across 'N' clients. The overhead of this operation is negligible as well, about 2-4 ms in a 16-node cluster configured over a 5Gbps NIC. If any of the synchronization bits in `flags` is 1 post-AG call, we call the synchronization function to collect and aggregate the model state. The `pushToPS` function receives the latest

Algorithm 5: {**Sel**}ective {**Sync**}hronization

Input: learning rate η, gradient change threshold δ, total number of clients 'N', training data $\mathcal{D}_c$ on client 'c'

procedure `train()`:

  $w_{(n,0)}$ = `pullFromPS()` ▷ initialize parameters

  for *iteration i = 0,1,...I on client device c* **do**

    `bit` [N] `flags` = 0 ▷ synchronization status

    $d_{(i,c)} \in \mathcal{D}_c$ ▷ sample mini-batch from data

    $g_i^{(c)} = \nabla \mathcal{F}(d_{(i,c)}, w_{(c,i)})$ ▷ compute gradient at step 'i'

    $\triangle(g_i^{(c)})$ = `DetectGradientChange` ($||g_i^{(c)}||^2$)

    $w_{(c,i+1)} = w_{(c,i)} - \eta \cdot g_i^{(c)}$ ▷ apply local updates

    if $\triangle(g_i^{(c)}) \geq \delta$: **then**

      `flags` [c] = 1 ▷ synchronize called by worker 'c' as its gradient change exceeds δ

    `flags` = `AllGatherStatus(flags)` ▷ call AllGather on `flags` such that index 'c' holds worker c's synchronization status bit

    if 1 $\in$ `flags` : **then**

      `pushToPS`($w_{(c,i+1)}$) ▷ push local updates

      $w_{(c,i+1)}$ = `pullFromPS()` ▷ pull global

model from all clients and pushes back the averaged parameters: $w_{(c,i+1)} = \sum_{c=1}^{N} w_{(c,i)}/N$ (i.e., parameter averaging). Thus, local replicas stay consistent with the global model state as synchronization is triggered across all clients if even a single one chooses synchronization over local-SGD. On the other hand, `SelSync` performs local-SGD if all the bits of the `flags` array are 0.

The dataset $\mathcal{D}_c$ from which mini-batch $d_{(i,c)}$ is sampled at iteration 'i' on client 'c' may have IID or non-IID properties. With IID data, `SelSync` implements SelDP by reordering and partitioning chunks as a circular queue where the head is rotated on each client corresponding to its id 'c'. For non-IID data, `SelSync` performs stochastic data-injection by randomly choosing a subset of clients and pulling training samples, depending on the chosen (ϕ, ψ, δ) configuration. To limit the size of local batches, we reduce client mini-batch to $b^{'}$ as per Eqn. (8.2). `SelSync` performs data-injection via point-to-point (P2P) communication, i.e., via `send/push` and `recv/pull` calls.

D.2 Cluster Setup and Training Specifications

We evaluate `SelSync` on a system of 4 servers, each equipped with 48-core Intel Xeon E5-2560 CPU, 128GB system memory and 4 NVIDIA V100 GPUs. To ensure resource isolation and min-

imum interference, we deploy a cluster of 16 CUDA-compatible nvidia-docker containers where each is allocated 8vCPUs, 24GB memory and 1 V100 GPU. The client containers push-pull their updates to-from a PS container with 12vCPUs and 32GB system memory. Communication between PS and client containers is facilitated via docker swarm configured over a 5Gbps ethernet NIC.

For IID data settings, CIFAR10/100 [180] contains 50000 training and 10000 test images with 10 and 100 labels respectively. ImageNet-1K [154] contains about 1.28 million training images and 50000 test images across 1000 class labels. WikiText-103 [222] contains 100 million tokens of verified and featured articles from Wikipedia. To emulate non-IID data settings, we split CIFAR10/100 across 10 clients with 1 and 10 labels residing on each device respectively. The training routine of each of model is described as follows:

- ResNet101 [9] trained over CIFAR10 with local mini-batch size 32, momentum optimizer with initial learning-rate 0.1, weight decay 4e-4 and momentum factor 0.9. We progressively decay the learning-rate by a factor of 10 after 110 and 150 epochs, and report top-1 test accuracy.

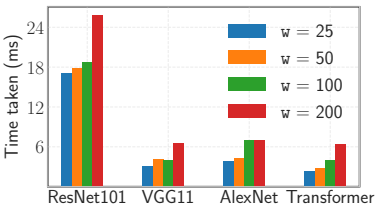
- VGG11 [8] trained over CIFAR100 with batch-size 32 and momentum SGD with factor 0.9, weight decay 5e-4 and initial learning-rate 0.01 decayed by $10\times$ at epochs 50 and 75. We report top-1 test accuracy here as well.

- AlexNet [1] trained over ImageNet-1K and reports top-5 test accuracy with local batch-size 128, Adam optimizer [136] and fixed learning-rate 1e-4.

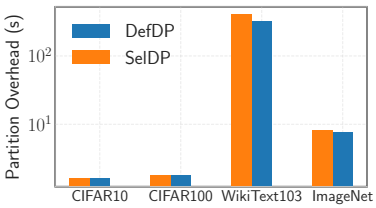
- Transformer trained over WikiText-103 where the encoder contains 2 hidden layers of dimension 200, dropout factor 0.2 and 35 bptt (backpropagation through time) steps. We set local batch-size to 20 and use SGD optimizer with initial learning-rate 2.0 that decays by a factor of 0.8 every 2000 training iterations and report the test perplexity, i.e., the exponential of test loss.

D.3 Analyzing `SelSync`-specific Overheads

The `DetectGradientChange` function in Alg. (5) applies EWMA to smoothen out the noise in the inter-iteration gradients. This overhead is exclusive to `SelSync` as $\triangle(g_i^{(c)})$ is not com-



(a) $\triangle(g_i^{(c)})$ computation overhead



(b) Data-partitioning overhead

Figure D.1: `SelSync` overheads: (a) computing gradient norm and applying EWMA smoothing on different windows. (b) SelDP overhead to partition and reorder data.

puted in BSP, FedAvg or SSP. As shown in Fig. (D.1a), the cost of this operation rises as the window-size for applying EWMA smoothing becomes larger. Compute time increases from 17 to 26ms in ResNet101 as the window-size increases from 25 to 200 steps, a jump of about 53%. A window-size of 25 samples takes only about 3.1, 3.9 and 2.3ms on VGG11, AlexNet and Transformer respectively. By increasing this window to 200 samples, the latency of respective DNNs increases by 110%, 79% and 178%. Although this overhead rises with the window-size evaluated, it is much lower than the typical computation and communication overheads in distributed training. Besides, we observed in our evaluation that a window-size of 25 samples sufficed and was able to successfully detect inter-iteration gradient changes. We use this window-size in our evaluation of `SelSync`.

We additionally evaluate the overhead of data-partitioning with `SelSync` in this section. The overhead to partition and reorder chunks to generate DefDP and SelDP is shown in Fig. (D.1b). Instead of splitting data into unique, equal-sized chunks, SelDP rearranges partitions based on client ID in the cluster. For CIFAR10/100 datasets, this additional operation does not incur any significant overhead as the loading time of the two schemes are nearly identical. This is because the size of these datasets is not large enough to pose a considerable overhead to reorder and shuffle operations. For larger datasets like ImageNet-1K and WikiText-103, SelDP had a slightly higher processing time over the default partitioning scheme. However, the margin between the two was only a few seconds and the partitioning operation is a one-time cost incurred only once before the training phase begins.

REFERENCES

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Communications of the ACM* 60 (2012), pp. 84 –90.

[2] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proc. IEEE* 86 (1998), pp. 2278–2324.

[3] Chathura Widanage et al. "Anomaly Detection over Streaming Data: Indy500 Case Study". In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)* (2019), pp. 9–16.

[4] Ashish Vaswani et al. "Attention is All you Need". In: *Neural Information Processing Systems*. 2017.

[5] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *North American Chapter of the Association for Computational Linguistics*. 2019.

[6] *Gwern: The Scaling Hypothesis*. `https://gwern.net/scaling-hypothesis`.

[7] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323 (1986), pp. 533–536.

[8] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556 (2014).

[9] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015), pp. 770–778.

[10] Rich Sutton. *The Bitter Lesson*. `http://www.incompleteideas.net/IncIdeas/BitterLesson.html`.

[11] *OpenAI*. `https://openai.com/`.

[12] OpenAI. *AI and Compute*. `https://openai.com/research/ai-and-compute`.

[13] Jaime Sevilla et al. “Compute Trends Across Three Eras of Machine Learning”. In: *2022 International Joint Conference on Neural Networks (IJCNN)* (2022), pp. 1–8.

[14] John Warner Backus. “Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs”. In: *Commun. ACM* 21 (1978), pp. 613–641.

[15] Norman P. Jouppi et al. “In-datacenter performance analysis of a tensor processing unit”. In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (2017), pp. 1–12.

[16] Chen Zhang et al. “Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks”. In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2015).

[17] Gordon E. Moore. “Cramming More Components Onto Integrated Circuits”. In: *Proceedings of the IEEE* 86 (1998), pp. 82–85.

[18] Tracy D. Braun et al. “A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems”. In: *J. Parallel Distributed Comput.* 61 (2001), pp. 810–837.

[19] Muthu Dayalan. “MapReduce: simplified data processing on large clusters”. In: *Commun. ACM* 51 (2008), pp. 107–113.

[20] Matei A. Zaharia et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *Symposium on Networked Systems Design and Implementation*. 2012.

[21] Ankit Toshniwal et al. “Storm@twitter”. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (2014).

[22] Leonardo Dagum and Ram Menon. “OpenMP: an industry standard API for shared-memory programming”. In: 1998.

[23] *OpenSHMEM*. `http://openshmem.org/site/`.

[24] Message Passing Interface Forum. “MPI: A message - passing interface standard”. In: 1994.

[25] H. B. McMahan et al. “Communication-Efficient Learning of Deep Networks from Decentralized Data”. In: *International Conference on Artificial Intelligence and Statistics*. 2016.

[26] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: *ArXiv* abs/1609.04747 (2016).

[27] Alessandro Achille, Matteo Rovere, and Stefano Soatto. “Critical Learning Periods in Deep Neural Networks”. In: *ArXiv* abs/1711.08856 (2017).

[28] Stanislaw Jastrzebski et al. “On the Relation Between the Sharpest Directions of DNN Loss and the SGD Step Length”. In: *arXiv: Machine Learning* (2018).

[29] Sahil Tyagi and Prateek Sharma. “OmniLearn: A Framework for Distributed Deep Learning over Heterogeneous Clusters”. In: *under review* (2024).

[30] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. “Optimization of Collective Communication Operations in MPICH”. In: *The International Journal of High Performance Computing Applications* 19 (2005), pp. 49 –66.

[31] Sahil Tyagi and Martin Swany. “GraVAC: Adaptive Compression for Communication Efficient Distributed DL Training”. In: *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)* (2023), pp. 319–329.

[32] Sahil Tyagi and Martin Swany. “Flexible Communication for Optimal Distributed Learning over Unpredictable Networks”. In: *2023 IEEE International Conference on Big Data (BigData)* (2023), pp. 925–935.

[33] Sahil Tyagi and Martin Swany. “ScaDLES: Scalable Deep Learning over Streaming data at the Edge”. In: *2022 IEEE International Conference on Big Data (Big Data)* (2022), pp. 2113–2122.

[34] Sahil Tyagi and Martin Swany. “Accelerating Distributed ML Training via Selective Synchronization”. In: *2023 IEEE International Conference on Cluster Computing (CLUSTER)* (2023), pp. 1–12.

[35] Sahil Tyagi and Martin Swany. “Accelerating Distributed ML Training via Selective Synchronization (Poster Abstract)”. In: *2023 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)* (2023), pp. 56–57.

[36] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. “Activation functions in deep learning: A comprehensive survey and benchmark”. In: *Neurocomputing* 503 (2021), pp. 92–108.

[37] Dami Choi et al. “On Empirical Comparisons of Optimizers for Deep Learning”. In: *ArXiv* abs/1910.05446 (2019).

[38] Qi Wang et al. “A Comprehensive Survey of Loss Functions in Machine Learning”. In: *Annals of Data Science* (2020), pp. 1–26.

[39] Siddharth Krishna Kumar. “On weight initialization in deep neural networks”. In: *ArXiv* abs/1704.08863 (2017).

[40] Sam McCandlish et al. “An Empirical Model of Large-Batch Training”. In: abs/1812.06162 (2018).

[41] Francis R. Bach and Éric Moulines. “Non-Asymptotic Analysis of Stochastic Approximation Algorithms for Machine Learning”. In: *Neural Information Processing Systems*. 2011.

[42] Xiangru Lian et al. “Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent”. In: *Neural Information Processing Systems*. 2017.

[43] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. *Neural networks for machine learning: Overview of Mini-batch gradient descent*.

[44] Christopher J. Shallue et al. “Measuring the Effects of Data Parallelism on Neural Network Training”. In: *ArXiv* abs/1811.03600 (2018).

[45] Jeff Rasley et al. “DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters”. In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2020).

[46] R.O Rogers and David B. Skillicorn. “Using the BSP cost model to optimise parallel neural network training”. In: *Future generations computer systems*. 1998.

[47] Jeffrey Dean et al. “Large Scale Distributed Deep Networks”. In: *Neural Information Processing Systems*. 2012.

[48] Yanping Huang et al. “GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism”. In: *Neural Information Processing Systems*. 2018.

[49] Mu Li et al. “Scaling Distributed Machine Learning with the Parameter Server”. In: *USENIX Symposium on Operating Systems Design and Implementation*. 2014.

[50] Yimin Jiang et al. “A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters”. In: *USENIX Symposium on Operating Systems Design and Implementation*. 2020.

[51] M. A. Ganaie et al. “Ensemble deep learning: A review”. In: *ArXiv* abs/2104.02395 (2021).

[52] Ofer Dekel et al. “Optimal Distributed Online Prediction Using Mini-Batches”. In: *ArXiv* abs/1012.1367 (2010).

[53] Alekh Agarwal and John C. Duchi. “Distributed delayed stochastic optimization”. In: *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)* (2011), pp. 5451–5452.

[54] Michael Blot et al. “Distributed optimization for deep learning with gossip exchange”. In: *Neurocomputing* 330 (2019), pp. 287–296.

[55] *Message passing via OpenMPI*. `https://www.open-mpi.org`.

[56] *NCCL: NVIDIA Collective Communication Library*. `https://developer.nvidia.com/nccl`.

[57] *PyTorch Gloo*. `https://github.com/facebookincubator/gloo`.

[58] Ang Li et al. “Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect”. In: *IEEE Transactions on Parallel and Distributed Systems* 31 (2019), pp. 94–110.

[59] Alexander Sergeev and Mike Del Balso. “Horovod: fast and easy distributed deep learning in TensorFlow”. In: *ArXiv* abs/1802.05799 (2018).

[60] Xianyan Jia et al. “Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes”. In: *ArXiv* abs/1807.11205 (2018).

[61] Priya Goyal et al. “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”. In: *ArXiv* abs/1706.02677 (2017).

[62] Shriram Sarvotham, Rudolf H. Riedi, and Richard Baraniuk. “Connection-level analysis and modeling of network traffic”. In: *International Memory Workshop*. 2001.

[63] Shaohuai Shi et al. “A Distributed Synchronous SGD Algorithm with Global Top-k Sparsification for Low Bandwidth Networks”. In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)* (2019), pp. 2238–2247.

[64] *PyTorch Distributed Communication Package*. `https://pytorch.org/docs/stable/distributed.html`.

[65] Aakash Sharma et al. “Stash: A Comprehensive Stall-Centric Characterization of Public Cloud VMs for Distributed Deep Learning”. In: *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)* (2023), pp. 1–12.

[66] Sahil Tyagi and Prateek Sharma. “Taming Resource Heterogeneity In Distributed ML Training With Dynamic Batching”. In: *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)* (2020), pp. 188–194.

[67] Wei Zhang et al. “Staleness-Aware Async-SGD for Distributed Deep Learning”. In: *ArXiv* abs/1511.05950 (2015).

[68] Nuwan S. Ferdinand et al. “Anytime Minibatch: Exploiting Stragglers in Online Distributed Optimization”. In: *ArXiv* abs/2006.05752 (2020).

[69] *Google Cloud Platform (GCP)*. `https://cloud.google.com/compute/docs/networking/configure-vm-with-high-bandwidth-configuration`.

[70] *Google Cloud Egress BW*. `https://cloud.google.com/compute/docs/network-bandwidth`.

[71] Srikanth Kandula et al. “The nature of data center traffic: measurements & analysis”. In: IMC ’09. Chicago, Illinois, USA: Association for Computing Machinery, 2009, 202–208. ISBN: 9781605587714.

[72] Mu Li, David G. Andersen, and Alex Smola. *Delayed Proximal Gradient Methods*. Available from `https://www.cs.cmu.edu/~muli/file/ddp.pdf`. 2013.

[73] Benjamin Recht et al. “Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent”. In: *Neural Information Processing Systems*. 2011.

[74] Xiangru Lian et al. “Asynchronous parallel stochastic gradient for nonconvex optimization”. In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*. NIPS’15. Montreal, Canada: MIT Press, 2015, 2737–2745.

[75] Xiangru Lian et al. *Asynchronous Decentralized Parallel Stochastic Gradient Descent*. 2018. arXiv: 1710.06952 [math.OC].

[76] Qirong Ho et al. “More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server”. In: *Advances in neural information processing systems* 2013 (2013), pp. 1223–1231.

[77] James Cipar et al. “Solving the Straggler Problem with Bounded Staleness”. In: *USENIX Workshop on Hot Topics in Operating Systems*. 2013.

[78] Xing Zhao et al. “Dynamic Stale Synchronous Parallel Distributed Training for Deep Learning”. In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)* (2019), pp. 1507–1517.

[79] Hang Shi et al. “Effective Parallel Computing via a Free Stale Synchronous Parallel Strategy”. In: *IEEE Access* 7 (2019), pp. 118764–118775.

[80] Shiqiang Wang et al. “Adaptive Federated Learning in Resource Constrained Edge Computing Systems”. In: *IEEE Journal on Selected Areas in Communications* 37 (2018), pp. 1205–1221.

[81] Jakub Konecný et al. “Federated Learning: Strategies for Improving Communication Efficiency”. In: *ArXiv* abs/1610.05492 (2016).

[82] Tian Li et al. “Federated Learning: Challenges, Methods, and Future Directions”. In: *IEEE Signal Processing Magazine* 37 (2019), pp. 50–60.

[83] Yue Zhao et al. “Federated Learning with Non-IID Data”. In: *ArXiv* abs/1806.00582 (2018).

[84] Sebastian Caldas et al. “LEAF: A Benchmark for Federated Settings”. In: *ArXiv* abs/1812.01097 (2018).

[85] Chaoyang He et al. “FedML: A Research Library and Benchmark for Federated Machine Learning”. In: *ArXiv* abs/2007.13518 (2020).

[86] S. U. Stich. “Local SGD Converges Fast and Communicates Little”. In: abs/1805.09767 (2018).

[87] Eduard A. Gorbunov, Filip Hanzely, and Peter Richtárik. “Local SGD: Unified Theory and New Efficient Methods”. In: abs/2011.02828 ().

[88] Martin A. Zinkevich et al. “Parallelized Stochastic Gradient Descent”. In: *Neural Information Processing Systems*. 2010.

[89] Anit Kumar Sahu et al. “Federated Optimization in Heterogeneous Networks”. In: *arXiv: Learning* (2018).

[90] Sixin Zhang, Anna Choromańska, and Yann LeCun. “Deep learning with Elastic Averaging SGD”. In: *Neural Information Processing Systems*. 2014.

[91] Peter H. Jin et al. “How to scale distributed deep learning?” In: *ArXiv* abs/1611.04581 (2016).

[92] H. Zhang et al. “Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters”. In: *ArXiv* abs/1706.03292 (2017).

[93] Shaohuai Shi and Xiaowen Chu. “MG-WFBP: Efficient Data Communication for Distributed Synchronous SGD Algorithms”. In: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications* (2018), pp. 172–180.

[94] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning internal representations by error propagation”. In: 1986.

[95] Yanghua Peng et al. “A generic communication scheduler for distributed DNN training acceleration”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019).

[96] Yixin Bao et al. “Preemptive All-reduce Scheduling for Expediting Distributed DNN Training”. In: *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications* (2020), pp. 626–635.

[97] Martín Abadi et al. “TensorFlow: A system for large-scale machine learning”. In: *USENIX Symposium on Operating Systems Design and Implementation*. 2016.

[98] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *ArXiv* abs/1912.01703 (2019).

[99] Suraj Srinivas and R. Venkatesh Babu. “Data-free Parameter Pruning for Deep Neural Networks”. In: *British Machine Vision Conference*. 2015.

[100] Sian Jin et al. “DeepSZ: A Novel Framework to Compress Deep Neural Networks by Using Error-Bounded Lossy Compression”. In: *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (2019).

[101] Xin Liang et al. “Error-Controlled Lossy Compression Optimized for High Compression Ratios of Scientific Datasets”. In: *2018 IEEE International Conference on Big Data (Big Data)* (2018), pp. 438–447.

[102] Zhenheng Tang et al. “Communication-Efficient Distributed Deep Learning: A Comprehensive Survey”. In: *ArXiv* abs/2003.06307 (2020).

[103] Hang Xu et al. “Compressed Communication for Distributed Deep Learning: Survey and Quantitative Evaluation”. In: 2020.

[104] Yann LeCun, John S. Denker, and Sara A. Solla. “Optimal Brain Damage”. In: *Neural Information Processing Systems*. 1989.

[105] Nitish Srivastava et al. “Dropout: a simple way to prevent neural networks from overfitting”. In: *J. Mach. Learn. Res.* 15 (2014), pp. 1929–1958.

[106] Song Han, Huizi Mao, and William J. Dally. “Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding”. In: *arXiv: Computer Vision and Pattern Recognition* (2015).

[107] Jan van Leeuwen. “On the Construction of Huffman Trees”. In: *International Colloquium on Automata, Languages and Programming*. 1976.

[108] Sai Praneeth Karimireddy et al. “Error Feedback Fixes SignSGD and other Gradient Compression Schemes”. In: *ArXiv* abs/1901.09847 (2019).

[109] Shuai Zheng, Ziyue Huang, and James Tin-Yau Kwok. “Communication-Efficient Distributed Blockwise Momentum SGD with Error-Feedback”. In: *ArXiv* abs/1905.10936 (2019).

[110] Yu Cheng et al. “A Survey of Model Compression and Acceleration for Deep Neural Networks”. In: *ArXiv* abs/1710.09282 (2017).

[111] Sebastian U. Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. “Sparsified SGD with Memory”. In: *ArXiv* abs/1809.07599 (2018).

[112] Dan Alistarh et al. “The Convergence of Sparsified Gradient Methods”. In: abs/1809.10505 ().

[113] Alham Fikri Aji and Kenneth Heafield. “Sparse Communication for Distributed Gradient Descent”. In: *ArXiv* abs/1704.05021 (2017).

[114] Shaohuai Shi et al. “Understanding Top-k Sparsification in Distributed Deep Learning”. In: *ArXiv* abs/1911.08772 (2019).

[115] Yujun Lin et al. “Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training”. In: *ArXiv* abs/1712.01887 (2017).

[116] Yoshua Bengio, Patrice Y. Simard, and Paolo Frasconi. “Learning long-term dependencies with gradient descent is difficult”. In: *IEEE transactions on neural networks* 5 2 (1994), pp. 157–66.

[117] Ahmed Mohamed Abdelmoniem et al. “An Efficient Statistical-based Gradient Compression Technique for Distributed Training Systems”. In: *ArXiv* abs/2101.10761 (2021).

[118] Dan Alistarh et al. “QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding”. In: *Neural Information Processing Systems*. 2016.

[119] Frank Seide et al. “1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs”. In: *Interspeech*. 2014.

[120] Paulius Micikevicius et al. “Mixed Precision Training”. In: *ArXiv* abs/1710.03740 (2017).

[121] Wei Wen et al. “TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning”. In: *Neural Information Processing Systems*. 2017.

[122] Jeremy Bernstein et al. “signSGD: compressed optimisation for non-convex problems”. In: *International Conference on Machine Learning*. 2018.

[123] Jeremy Bernstein et al. “signSGD with Majority Vote is Communication Efficient and Fault Tolerant”. In: *International Conference on Learning Representations*. 2018.

[124] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. “PowerSGD: Practical Low-Rank Gradient Compression for Distributed Optimization”. In: abs/1905.13727 (2019).

[125] Hongyi Wang et al. “ATOMO: Communication-efficient Learning via Atomic Sparsification”. In: *ArXiv* abs/1806.04090 (2018).

[126] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. “PowerGossip: Practical Low-Rank Communication Compression in Decentralized Deep Learning”. In: abs/2008.01425 (2020).

[127] Jiarui Fang et al. “RedSync : Reducing Synchronization Traffic for Distributed Deep Learning”. In: *J. Parallel Distributed Comput.* 133 (2018), pp. 30–39.

[128] Felix Sattler et al. “Robust and Communication-Efficient Federated Learning From Non-I.I.D. Data”. In: *IEEE Transactions on Neural Networks and Learning Systems* 31 (2019), pp. 3400–3413.

[129] S. W. Golomb. “Run-length encodings”. In: *IEEE Trans Info Theory* (1966), 12(3):399.

[130] Chia-Yu Chen et al. “AdaComp : Adaptive Residual Gradient Compression for Data-Parallel Distributed Training”. In: *AAAI Conference on Artificial Intelligence*. 2017.

[131] Saurabh Agarwal et al. “Accordion: Adaptive Gradient Communication via Critical Learning Regime Identification”. In: *ArXiv* abs/2010.16248 (2020).

[132] Jan Kukacka, Vladimir Golkov, and Daniel Cremers. “Regularization for Deep Learning: A Taxonomy”. In: *ArXiv* abs/1710.10686 (2017).

[133] *Keras Loss functions*. `https://keras.io/api/losses/`.

[134] Yurii Nesterov. “A method for solving the convex programming problem with convergence rate $\mathcal{O}(1/k^2)$”. In: *Proceedings of the USSR Academy of Sciences* 269 (1983), pp. 543–547.

[135] John C. Duchi, Elad Hazan, and Yoram Singer. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”. In: *J. Mach. Learn. Res.* 12 (2011), pp. 2121–2159.

[136] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *CoRR* abs/1412.6980 (2014).

[137] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. Cambridge, MA, USA: MIT Press, 2016.

[138] Zhewei Yao et al. “Hessian-based Analysis of Large Batch Training and Robustness to Adversaries”. In: *Neural Information Processing Systems*. 2018.

[139] Tao Lin et al. “Extrapolation for Large-batch Training in Deep Learning”. In: abs/2006.05720 (2020).

[140] Nitish Shirish Keskar et al. “On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima”. In: *ArXiv* abs/1609.04836 (2016).

[141] Yang You, Igor Gitman, and Boris Ginsburg. “Large Batch Training of Convolutional Networks”. In: *arXiv: Computer Vision and Pattern Recognition* (2017).

[142] Zhewei Yao et al. “Large batch size training of neural networks with adversarial training and second-order information”. In: *ArXiv* abs/1810.01021 (2018).

[143] Elad Hoffer, Itay Hubara, and Daniel Soudry. “Train longer, generalize better: closing the generalization gap in large batch training of neural networks”. In: *ArXiv* abs/1705.08741 (2017).

[144] Matthew D. Zeiler. “ADADELTA: An Adaptive Learning Rate Method”. In: abs/1212.5701 (2012).

[145] Laxmidhar Behera, S. Kumar, and Awhan Patnaik. “On Adaptive Learning Rate That Guarantees Convergence in Feedforward Networks”. In: *IEEE Transactions on Neural Networks* 17 (2006), pp. 1116–1125.

[146] Tomoumi Takase, Satoshi Oyama, and Masahito Kurihara. “Effective neural network training with adaptive learning rate based on training loss”. In: *Neural networks : the official journal of the International Neural Network Society* 101 (2018), pp. 68–78.

[147] Noam M. Shazeer and Mitchell Stern. “Adafactor: Adaptive Learning Rates with Sublinear Memory Cost”. In: *ArXiv* abs/1804.04235 (2018).

[148] Chien-Cheng Yu and Bin-Da Liu. “A backpropagation algorithm with adaptive learning rate and momentum coefficient”. In: *Proceedings of the 2002 International Joint Conference on Neural Networks. IJCNN’02 (Cat. No.02CH37290)* 2 (2002), 1218–1223 vol.2.

[149] Jonathan Frankle, David J. Schwab, and Ari S. Morcos. “The Early Phase of Neural Network Training”. In: *ArXiv* abs/2002.10365 (2020).

[150] John L. Gustafson. “Reevaluating Amdahl’s law”. In: *Commun. ACM* 31 (1988), pp. 532–533.

[151] Valeriu Codreanu, Damian Podareanu, and Vikram A. Saletore. “Scale out for large minibatch SGD: Residual network training on ImageNet-1K with improved accuracy and reduced time to train”. In: *ArXiv* abs/1711.04291 (2017).

[152] Alex Krizhevsky. “One weird trick for parallelizing convolutional neural networks”. In: *ArXiv* abs/1404.5997 (2014).

[153] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. “Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes”. In: *ArXiv* abs/1711.04325 (2017).

[154] Jia Deng et al. “ImageNet: A large-scale hierarchical image database”. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255.

[155] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *ArXiv* abs/1502.03167 (2015).

[156] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. “Don’t Decay the Learning Rate, Increase the Batch Size”. In: *ArXiv* abs/1711.00489 (2017).

[157] Yang You et al. “ImageNet Training in Minutes”. In: *Proceedings of the 47th International Conference on Parallel Processing* (2017).

[158] Aditya Devarakonda, Maxim Naumov, and Michael Garland. “AdaBatch: Adaptive Batch Sizes for Training Deep Neural Networks”. In: *ArXiv* abs/1712.02029 (2017).

[159] Samuel L. Smith and Quoc V. Le. “A Bayesian Perspective on Generalization and Stochastic Gradient Descent”. In: *ArXiv* abs/1710.06451 (2017).

[160] Jonathan Frankle and Michael Carbin. “The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks”. In: *arXiv: Learning* (2018).

[161] Jonathan Frankle et al. “Linear Mode Connectivity and the Lottery Ticket Hypothesis”. In: *International Conference on Machine Learning*. 2019.

[162] Guy Gur-Ari, Daniel A. Roberts, and Ethan Dyer. “Gradient Descent Happens in a Tiny Subspace”. In: *ArXiv* abs/1812.04754 (2018).

[163] R. A. Fisher. “Theory of Statistical Estimation”. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 22.5 (1925), 700–725.

[164] Yanghua Peng et al. “Optimus: an efficient dynamic resource scheduler for deep learning clusters”. In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys ’18. Porto, Portugal: Association for Computing Machinery, 2018. ISBN: 9781450355841.

[165] Haoyue Zheng et al. “Cynthia: Cost-Efficient Cloud Resource Provisioning for Predictable Distributed Deep Neural Network Training”. In: ICPP ’19. Kyoto, Japan: Association for Computing Machinery, 2019. ISBN: 9781450362955.

[166] Richard Liaw et al. “HyperSched: Dynamic Resource Reallocation for Model Development on a Deadline”. In: SoCC ’19. Santa Cruz, CA, USA: Association for Computing Machinery, 2019, 61–73. ISBN: 9781450369732.

[167] Tyler B. Johnson et al. “AdaScale SGD: A User-Friendly Algorithm for Distributed Training”. In: *International Conference on Machine Learning*. 2020.

[168] Luo Mai et al. “KungFu: Making Training in Distributed Machine Learning Adaptive”. In: *USENIX Symposium on Operating Systems Design and Implementation*. 2020.

[169] Aurick Qiao et al. “Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning”. In: *ArXiv* abs/2008.12260 (2020).

[170] Jimmy Ba, Roger Baker Grosse, and James Martens. “Distributed Second-Order Optimization using Kronecker-Factored Approximations”. In: *International Conference on Learning Representations*. 2016.

[171] Kazuki Osawa et al. “Large-Scale Distributed Second-Order Optimization Using Kronecker-Factored Approximate Curvature for Deep Convolutional Neural Networks”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2019), pp. 12351–12359.

[172] Kiam Heong Ang, G. Chong, and Yun Li. “PID control system analysis, design, and technology”. In: *IEEE Transactions on Control Systems Technology* 13.4 (2005), pp. 559–576.

[173] Shijian Li, Robert J. Walls, and Tian Guo. “Characterizing and Modeling Distributed Training with Transient Cloud GPU Servers”. In: *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)* (2020), pp. 943–953.

[174] *EC2 Burstable Instances,* `https://aws.amazon.com/blogs/aws/low-cost-burstable-ec2-instances`. 2014.

[175] Cheng Wang et al. “Exploiting Spot and Burstable Instances for Improving the Cost-efficacy of In-Memory Caches on the Public Cloud”. In: *Proceedings of the Twelfth European Conference on Computer Systems* (2017).

[176] Ali Bakhoda et al. “Analyzing CUDA workloads using a detailed GPU simulator”. In: *2009 IEEE International Symposium on Performance Analysis of Systems and Software* (2009), pp. 163–174.

[177] Dirk Merkel. “Docker: lightweight Linux containers for consistent development and deployment”. In: *Linux Journal* 2014 (2014), p. 2.

[178] Samyam Rajbhandari et al. “ZeRO: Memory Optimization Towards Training A Trillion Parameter Models”. In: *ArXiv* abs/1910.02054 (2019).

[179] Quentin G. Anthony et al. “ScaMP: Scalable Meta-Parallelism for Deep Learning Search”. In: *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)* (2023), pp. 391–402.

[180] *CIFAR10 and CIFAR100 datasets*. `https://www.cs.toronto.edu/~kriz/cifar.html`.

[181] Li Fei-Fei, Rob Fergus, and Pietro Perona. “Learning Generative Visual Models from Few Training Examples: An Incremental Bayesian Approach Tested on 101 Object Categories”. In: *2004 Conference on Computer Vision and Pattern Recognition Workshop* (2004).

[182] Lukas Bossard, Matthieu Guillaumin, and Luc Van Gool. “Food-101 - Mining Discriminative Components with Random Forests”. In: *European Conference on Computer Vision*. 2014.

[183] Michael Armbrust et al. “A view of cloud computing”. In: *Commun. ACM* 53 (2010), pp. 50–58.

[184] Omid Alipourfard et al. “CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics”. In: *Symposium on Networked Systems Design and Implementation*. 2017.

[185] *PyTorch Distributed Elastic package*. `https://pytorch.org/docs/stable/distributed.elastic.html`.

[186] Suyog Gupta, Wei Zhang, and Fei Wang. “Model Accuracy and Runtime Tradeoff in Distributed Deep Learning: A Systematic Study”. In: *2016 IEEE 16th International Conference on Data Mining (ICDM)* (2015), pp. 171–180.

[187] Saurabh Agarwal et al. “On the Utility of Gradient Compression in Distributed Training Systems”. In: *ArXiv* abs/2103.00543 (2021).

[188] Gene M. Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities”. In: *AFIPS ’67 (Spring)*. 1967.

[189] Kishore Papineni et al. “Bleu: a Method for Automatic Evaluation of Machine Translation”. In: *Annual Meeting of the Association for Computational Linguistics*. 2002.

[190] Ville A. Satopaa et al. “Finding a ”Kneedle” in a Haystack: Detecting Knee Points in System Behavior”. In: *2011 31st International Conference on Distributed Computing Systems Workshops* (2011), pp. 166–171.

[191] Heng-Tze Cheng et al. “TensorFlow Estimators: Managing Simplicity vs. Flexibility in High-Level Machine Learning Frameworks”. In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2017).

[192] *GPU virtuialization*. `https://www.nvidia.com/en-us/data-center/graphics-cards-for-virtualization/`.

[193] Ruben Mayer and Hans-Arno Jacobsen. “Scalable Deep Learning on Distributed Infrastructures”. In: *ACM Computing Surveys (CSUR)* 53 (2019), pp. 1 –37.

[194] Edo Liberty et al. “Elastic machine learning algorithms in Amazon SageMaker”. In: *SIGMOD/PODS 2020*. 2020.

[195] Ujval Misra et al. “RubberBand: cloud-based hyperparameter tuning”. In: *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys ’21. Online Event, United Kingdom: Association for Computing Machinery, 2021, 327–342. ISBN: 9781450383349.

[196] Aurick Qiao et al. “Litz: Elastic Framework for High-Performance Distributed Machine Learning”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 631–644. ISBN: 978-1-939133-01-4.

[197] Dhruv Mahajan et al. *Towards Resource-Elastic Machine Learning*. 2013.

[198] *Common Crawl Dataset*. https://commoncrawl.org/.

[199] R. R. Schaller. “Moore’s law: past, present and future”. In: *IEEE Spectrum* 34 (1997), pp. 52–59.

[200] Steven M. Cherry. “Edholm’s law of bandwidth”. In: *IEEE Spectrum* 41 (2004), pp. 58–60.

[201] Zhen Zhang et al. “Is Network the Bottleneck of Distributed Training?” In: *Proceedings of the Workshop on Network Meets AI & ML* (2020).

[202] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9 (1997), pp. 1735–1780.

[203] Jinrong Guo et al. “Accelerating Distributed Deep Learning By Adaptive Gradient Quantization”. In: *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2020), pp. 1603–1607.

[204] Shen Li et al. *PyTorch Distributed: Experiences on Accelerating Data Parallel Training*. 2020. arXiv: 2006.15704 [cs.DC].

[205] Alexey Dosovitskiy et al. “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”. In: *ArXiv* abs/2010.11929 (2020).

[206] *NVLink*. https://www.nvidia.com/en-us/data-center/nvlink/.

[207] *NVIDIA DGX*. https://www.nvidia.com/en-us/data-center/dgx-systems/.

[208] *AWS bandwidth*. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-network-bandwidth.html.

[209] Shaohuai Shi et al. “Towards Scalable Distributed Training of Deep Learning on Public Cloud Clusters”. In: *ArXiv* abs/2010.10458 (2020).

[210] *Linux Traffic Control*. `https://man7.org/linux/man-pages/man8/tc.8.html`.

[211] *iPerf test tool for TCP, UDP and SCTP*. `https://iperf.fr/`.

[212] *Linux traceroute*. `https://linux.die.net/man/8/traceroute`.

[213] Gregory Griffin, Alex Holub, and Pietro Perona. “Caltech-256 Object Category Dataset”. In: 2007.

[214] Julian Blank and Kalyanmoy Deb. “Pymoo: Multi-Objective Optimization in Python”. In: *IEEE Access* 8 (2020), pp. 89497–89509.

[215] Jay Kreps. *Kafka : A Distributed Messaging System for Log Processing*. 2011.

[216] Shilpa Chaturvedi, Sahil Tyagi, and Yogesh L. Simmhan. “Collaborative Reuse of Streaming Dataflows in IoT Applications”. In: *2017 IEEE 13th International Conference on e-Science (e-Science)* (2017), pp. 403–412.

[217] Shilpa Chaturvedi, Sahil Tyagi, and Yogesh L. Simmhan. “Cost-Effective Sharing of Streaming Dataflows for IoT Applications”. In: *IEEE Transactions on Cloud Computing* 9 (2019), pp. 1391–1407.

[218] Judy Qiu et al. “Real-Time Anomaly Detection from Edge to HPC-Cloud”. In: 2018.

[219] Samyam Rajbhandari et al. “ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep learning”. In: *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis* (2021), pp. 1–15.

[220] *NVIDIA V100*. `https://www.nvidia.com/en-us/data-center/v100/`.

[221] Andreas Griewank and Andrea Walther. “Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation”. In: *ACM Trans. Math. Softw.* 26.1 (2000), 19–45.

[222] Stephen Merity et al. “Pointer Sentinel Mixture Models”. In: *ArXiv* abs/1609.07843 (2016).

[223] *PyTorch ImageFolder dataloader*. pytorch.org/vision/0.12/generated/torchvision.datasets.ImageFolder.html.

[224] Latanya Sweeney. "k-Anonymity: A Model for Protecting Privacy". In: *Int. J. Uncertain. Fuzziness Knowl. Based Syst.* 10 (2002), pp. 557–570.

[225] Huansheng Ning et al. "Heterogeneous edge computing open platforms and tools for internet of things". In: *Future Gener. Comput. Syst.* 106 (2020), pp. 67–76.

[226] *AWS EC2 Spot Instances*. https://aws.amazon.com/ec2/spot/.

[227] Aaron Harlap et al. "Proteus: agile ML elasticity through tiered reliability in dynamic resource markets". In: *Proceedings of the Twelfth European Conference on Computer Systems* (2017).

[228] Shijian Li et al. "Speeding up Deep Learning with Transient Servers". In: *2019 IEEE International Conference on Autonomic Computing (ICAC)* (2019), pp. 125–135.

[229] Xiaoxi Zhang et al. "Machine Learning on Volatile Instances". In: *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications* (2020), pp. 139–148.

[230] Sahil Tyagi and Prateek Sharma. "Scavenger: A Cloud Service For Optimizing Cost and Performance of ML Training". In: *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)* (2023), pp. 403–413.

[231] Sahil Tyagi. "Scavenger: A Cloud Service for Optimizing Cost and Performance of DL Training". In: *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)* (2023), pp. 349–350.

[232] *Elastic Horovod*. https://horovod.readthedocs.io/en/latest/elastic_include.html.

[233] Mingzhen Li et al. "EasyScale: Accuracy-consistent Elastic Training for Deep Learning". In: *ArXiv* abs/2208.14228 (2022).

[234] Juncheng Gu et al. "Tiresias: A GPU Cluster Manager for Distributed Deep Learning". In: *Symposium on Networked Systems Design and Implementation*. 2019.

[235] Diandian Gu et al. “ElasticFlow: An Elastic Serverless Training Platform for Distributed Deep Learning”. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (2023).

[236] Cynthia Dwork. “Differential Privacy”. In: *Automata, Languages and Programming*. Ed. by Michele Bugliesi et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–12. ISBN: 978-3-540-35908-1.

[237] Bolei Zhou et al. “Places: A 10 Million Image Database for Scene Recognition”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40 (2018), pp. 1452–1464.

[238] Docker. *https://docs.docker.com/config/containers/resource constraints/.*

[239] Jie Cheng. “CUDA by Example: An Introduction to General-Purpose GPU Programming”. In: *Scalable Comput. Pract. Exp.* 11 (2010).

[240] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. “Building a Large Annotated Corpus of English: The Penn Treebank”. In: *Comput. Linguistics* 19 (1993), pp. 313–330.

[241] *NVIDIA K80*. `https://www.nvidia.com/en-gb/data-center/tesla-k80/`.

CURRICULUM VITAE

Sahil Tyagi

sahilt.tyagi@gmail.com

Webpage ORCID LinkedIn Google Scholar GitHub

Education

– Ph.D. in Intelligent Systems Engineering

- Indiana University Bloomington, USA
- Concentration: Computer engineering
- Advisor: Martin Swany
- Duration: August 2018 - January 2025
- Thesis: *Towards Building Efficient Computation and Communication Models for Deep Learning Systems*

– B.Tech. in Electrical and Electronics Engineering

- Guru Gobind Singh Indraprastha University
- Duration: September 2009 - June 2013

Research Experience

– ISE, Luddy School of Indiana University Bloomington

- Role: Research Assistant
- Projects: Research at the intersection of deep learning, distributed systems, machine learning and cloud computing. Worked on topics like distributed deep learning, federated learning, compression and stream processing.
- Duration: August 2018 - August 2024

– DREAM:Lab, Indian Institute of Science Bengaluru

- Role: Research Staff
- Project: Developed and analysed distributed stream processing systems. Built a validation system to authenticate streaming data as part of the SmartCity project in the Robert Bosch Center for Cyber-Physical systems.
- Duration: March 2017 - May 2018

Teaching Experience

– Course: Cloud Computing (E516)

- Role: Associate Instructor
- Department: ISE, Luddy school, Indiana University Bloomington
- Term: Fall 2019, Fall 2020, Fall 2021

– Course: Distributed Systems (ENGR-E510, CSCI-B534)

- Role: Associate Instructor
- Department: ISE, Luddy school, Indiana University Bloomington
- Term: Spring 2021, Spring 2022

– Course: Computer Networks (ENGR-E318/518, CSCI-P438/538)

- Role: Associate Instructor
- Department: ISE, Luddy school, Indiana University Bloomington
- Term: Fall 2022, Fall 2023, Fall 2024

– Course: Operating Systems (ENGR-E319/519, CSCI-P436/536)

- Role: Associate Instructor
- Department: ISE, Luddy school, Indiana University Bloomington
- Term: Spring 2023

– Course: High-Performance Computing (ENGR-E317/517)

- Role: Associate Instructor
- Department: ISE, Luddy school, Indiana University Bloomington
- Term: Spring 2024

Industry Experience

– Hindustan Times Media Limited

- Role: Data Scientist
- Description: Developed a mobile app analytics tool based on lambda architecture, implementing Spark on the batch layer, Spark Streaming on the speed layer and HBase to store data views on the serving layer. Migrated the legacy code from Hadoop Cascading to Spark (Java + Scala). Built a prototype data management platform for smart wearables and health devices, and implemented an advertising engine based on complex event-driven processing.
- Duration: January 2016 - December 2016

– Stayzilla

- Role: Big Data Engineer
- Description: Set up and deployed Hadoop, MemSQL , Elasticsearch and wrote complex mapreduce programs to solve key business problems. Performed sentiment analysis, by aggregating and storing data on historical basis from Facebook, Twitter and Instagram. Implemented a minimization model using optimization techniques to get the least cost of bid while getting maximum clicks.
- Duration: July 2015 - December 2015

– Tatras Data (formerly Algorithmic Insight)

- Role: Software Engineer
- Description: Worked on ETL (extract, transform, load) pipelines for Google Adwords API, Bing Adwords API, Amazon MarketPlace Web Services. Automated the inventory sheet update process from the client database, and upload the report on corresponding Amazon seller account using the Amazon MarketPlace Web Services. Wrote crawlers on competitor websites to fetch the price of products, compare and analyze the collected data.
- Duration: July 2014 - July 2015

Publications

– Journal Articles

1. Tyagi, S., & Sharma, P. (2024). `OmniLearn`: A Framework for Distributed Deep Learning over Heterogeneous Clusters (under review).
2. Chaturvedi, S., Tyagi, S., & Simmhan, Y.L. (2019). Cost-Effective Sharing of Streaming Dataflows for IoT Applications. IEEE Transactions on Cloud Computing, 9, 1391-1407.

– Conferences

1. Tyagi, S., & Swany, M. (2023). Flexible Communication for Optimal Distributed Learning over Unpredictable Networks. 2023 IEEE International Conference on Big Data (BigData), 925-935 (**Acceptance rate 17.5%**).
2. Tyagi, S., & Swany, M. (2023). Accelerating Distributed ML Training via Selective Synchronization. 2023 IEEE International Conference on Cluster Computing (CLUSTER), 1-12 (**Acceptance rate 25%**).
3. Tyagi, S., & Swany, M. (2023). GraVAC: Adaptive Compression for Communication-Efficient Distributed DL Training. 2023 IEEE 16th International Conference on Cloud Computing (CLOUD), 319-329 (**Acceptance rate 20%**).
4. Tyagi, S., & Sharma, P. (2023). Scavenger: A Cloud Service For Optimizing Cost and Performance of ML Training. 2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid), 403-413 (**Acceptance rate 21%**).
5. Tyagi, S., & Swany, M. (2022). ScaDLES: Scalable Deep Learning over Streaming data at the Edge. 2022 IEEE International Conference on Big Data (Big Data), 2113-2122 (**Acceptance rate 19.2%**).
6. Tyagi, S., & Sharma, P. (2020). Taming Resource Heterogeneity In Distributed ML Training With Dynamic Batching. 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), 188-194 (**Acceptance rate 25%**).
7. Widanage, C., Li, J., Tyagi, S., Teja, R., Peng, B., Kamburugamuve, S., Baum, D., Smith, D., Qiu, J., & Koskey, J. (2019). Anomaly Detection over Streaming Data: Indy500 Case Study. 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), 9-16 (**Acceptance rate 20%**).

8. Chaturvedi, S., Tyagi, S., & Simmhan, Y.L. (2017). Collaborative Reuse of Streaming Dataflows in IoT Applications. 2017 IEEE 13th International Conference on e-Science (e-Science), 403-412 (**Acceptance rate 36%**).

– Poster presentations

1. Tyagi, S. (2023). Scavenger: A Cloud Service for Optimizing Cost and Performance of DL Training. 2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW), 349-350. (*Best poster award*)
2. Tyagi, S., & Swany, M. (2023). Accelerating Distributed ML Training via Selective Synchronization (Poster Abstract). 2023 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops), 56-57.

Skills

- Programming: Python, C, Shell scripting, OpenMP, CUDA, Java, Scala, R, MATLAB, SQL
- Databases: MySQL, HBase
- Containerization: Docker, Kubernetes
- Distributed computing: Apache Hadoop, Spark, Storm, Kafka, Slurm, MPI
- ML libraries: PyTorch, TensorFlow, Keras, MXNet, DeepSpeed, Hugging Face
- Cloud computing: Google Cloud Platform (GCP), Amazon Web Services (AWS)

Awards and Services

- NSF Student Grant: To present at IEEE CLUSTER 2023, Santa Fe, New Mexico.
- Luddy Dean's Graduate Student Award: In Fall 2023 for outstanding research.
- NSF Travel Award: To present at IEEE/ACM CCGrid 2023, Bengaluru, India.
- Best early-career student poster award: Awarded at IEEE/ACM CCGrid 2023.
- Google Cloud Student Researcher (2021, 2022): Received GCP credits for research.
- Student Research Award: Funded via NSF grant Data Infrastructure Building Blocks (DiBBS) 17-500, for academic year 2018-2019.
- Reviewer: USENIX OSDI '24, USENIX ATC '24, IEEE CLUSTER '24, JPDC.

Talks and Presentations

- **09/24**: Research talk, “Improving Communication in Federated Learning via Adaptive Gradient Compression”, INRIA, France.
- **09/24**: Invited talk, “Optimizing Compute and Communication in Deep Learning Systems”, Oak Ridge National Laboratory (ORNL), Tennessee, USA.
- **04/24**: Guest lectures, ”Parallel Computing with GPUs for Distributed ML Applications”, High-Performance Computing (HPC) course, Indiana University Bloomington, USA.
- **12/23**: Paper presentation, ”Flexible Communication for Optimal Distributed Learning over Unpredictable Networks.” 2023 IEEE International Conference on Big Data, Sorrento, Italy.
- **11/23**: Paper presentation, ”Accelerating Distributed ML Training via Selective Synchronization.” 2023 IEEE International Conference on Cluster Computing, Santa Fe, New Mexico, USA.
- **11/23**: Poster presentation, ”Accelerating Distributed ML Training via Selective Synchronization.” 2023 IEEE International Conference on Cluster Computing, Santa Fe, New Mexico, USA.
- **09/23**: Invited talk, “Towards building efficient computation and communication models for distributed deep learning systems.” Mathematics and Computer Science (MCS) division, Argonne National Laboratory, Illinois, USA.
- **07/23**: Paper presentation, ”GraVAC: Adaptive Compression for Communication-Efficient Distributed DL Training.” 2023 IEEE International Conference on Cloud Computing, Chicago, Illinois.
- **05/23**: Paper presentation, ”Scavenger: A Cloud Service for Optimizing Cost and Performance of ML Training.” 2023 IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, Bengaluru, India.
- **05/23**: Poster presentation, ”Scavenger: A Cloud Service for Optimizing Cost and Performance of ML Training.” 2023 IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, Bengaluru, India.
- **12/22**: Paper presentation, ”ScaDLES: Scalable Deep Learning over Streaming Data at the Edge.” 2022 IEEE International Conference on Big Data, Osaka, Japan.
- **07/20**: Paper presentation, ”Taming Resource Heterogeneity in Distributed ML Training with

Dynamic Batching." 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems, virtual.

- **11/18**: "Real-Time Anomaly Detection from Edge to HPC-Cloud", Intel Speakerships at SC18 (Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis 2018), Dallas, Texas, USA.