Tula: Optimizing Time, Cost, and Generalization in Distributed Large-Batch Training

Sahil Tyagi
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
tyagis@ornl.gov

Feiyi Wang
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
fwang2@ornl.gov

***Abstract*—Distributed training increases the number of batches processed per iteration either by scaling-out (adding more nodes) or scaling-up (increasing the batch-size). However, the largest configuration does not necessarily yield the best performance. Horizontal scaling introduces additional communication overhead, while vertical scaling is constrained by computation cost and device memory limits. Thus, simply increasing the batch-size leads to diminishing returns: training time and cost decrease initially but eventually plateaus, creating a knee-point in the time/cost vs. batch-size pareto curve. The optimal batch-size therefore depends on the underlying model, data and available compute resources. Large batches also suffer from worse model quality due to the well-known "generalization gap". In this paper, we present Tula, an online service that automatically optimizes time, cost, and convergence quality for large-batch training of convolutional models. It combines parallel-systems modeling with statistical performance prediction to identify the optimal batch-size. Tula predicts training time and cost within 7.5–14% error across multiple models, and achieves up to 20× overall speedup and improves test accuracy by ≈ 9% on average over standard large-batch training on various vision tasks, thus successfully mitigating the generalization gap and accelerating training at the same time.**

I. Introduction

Distributed processing is essential to develop accurate models trained over vast datasets [1]. In the age of big data and federated processing, training deep learning workloads spans a broad spectrum of hardware—from datacenter-scale Graphic Processing Units (GPUs) to edge and personal devices. This ecosystem is highly heterogeneous, with compute [3] and network capabilities [2] varying significantly across edge [43], cloud, and High Performance Computing (HPC) environments. Consequently, existing training techniques and systems are often unable to utilize/allocate resources efficiently for a given task. Choosing an appropriate training and system configuration has a pronounced impact on both the parallel and statistical performance of neural networks [4], [16].

The pareto relationship between parallel and statistical efficiency is influenced by total nodes, hardware, batch-size, data and execution characteristics of the model [5], [6], [16]. A sub-optimal configuration may lead to extended runtimes, increased cost, or under-utilization of resources. Unlike traditional distributed processing jobs, model training involves unique execution and synchronization behaviors, along with numerous tunable parameters that directly affect model quality, performance and efficiency. Scaling efficiency is affected by network topology, latency, bandwidth, cluster-size, batch-size, model-size, and communication protocol [2], while statistical performance is influenced by communication frequency, data distribution [26] and training parameters such as learning-rate (LR) and batch-size. Large batch-sizes specially suffer from worse test performance due to "generalization gap" [8], [9].

In this work, we present Tula, a black-box, online service that develops practical and principled techniques for performance and cost-aware training while mitigating the generalization gap associated with large-batches. To enable systematic optimization, we construct an empirical performance model for large-batch distributed training: profiling-based compute analysis to estimate memory demands, coupled with a parallel model to predict compute and synchronization cost across different cluster and batch-sizes. Beyond classical parallel scaling considerations, Tula incorporates fundamental characteristics of Stochastic Gradient Descent (SGD) based optimization to address the generalization deficit of large-batches by scaling gradient updates. Tula is tested with convolutional models across various vision tasks and compared with multiple baselines. In general, we make the following contributions:

- Perform a comprehensive analysis of the trade-offs between parallel and statistical efficiency in large-batch distributed training.
- Develop a practical performance model that estimates resource requirements, execution overhead, and end-to-end runtime cost for a given model, dataset, training hyperparameters, and compute resources.
- Theoretically analyze the proposed gradient-scaling technique that projects large-batch updates to small-batch space, significantly improving model accuracy over vanilla SGD and achieving similar or *better* performance as other large-batch optimizers.
- Evaluate multiple models to demonstrate Tula's effectiveness in quantifying time and cost improvements over naïve configurations under diverse user objectives (e.g., minimizing time, cost, or a more balanced combination), and exploring multiple cost models for selecting the optimal batch-size in distributed training.

II. Background and Challenges

Performance trade-offs in distributed training: In this section, we characterize the performance trade-offs inherent in

distributed training. These observations inform the design of the parallel and statistical performance models in §III to improve large-batch training efficiency.

A. Parallel Efficiency in Distributed Training

Neural networks are trained by iteratively optimizing model parameters by minimizing a loss function using stochastic methods like SGD. This compute-intensive process is commonly accelerated via data parallelism where multiple nodes maintain a full model replica and compute gradients over a disjoint subset of the data, followed by synchronization and aggregation of updates. For an L-smooth objective function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ satisfying Equation (1a), SGD on each worker produces an unbiased estimator of the true gradient over random samples b with bounded variance σ^2 (Equation (1b)). In synchronous SGD, updates are averaged over N nodes and applied as per Equation (1c) with convergence rate $\mathcal{O}(1/\sqrt{I})$ for non-convex stochastic optimization over I iterations [17]. Updates are aggregated through a central parameter server or decentralized collective communication like all-reduce, each exhibiting distinct latency-bandwidth costs [2].

$$f(x) \leq f(y) + \langle \nabla f(y), x - y \rangle + \frac{L}{2}||x - y||^2 \ \ \forall \ \ x, y \quad \text{(1a)}$$

$$\mathbb{E}[g^{(i)}|w^{(i)}] = \nabla f(w^{(i)}) \Rightarrow \mathbb{E}||g^{(i)} - \nabla f(w^{(i)})||^2 \leq \sigma^2 \ \forall \ i \quad \text{(1b)}$$

$$w_{(n)}^{(i+1)} = w_{(n)}^{(i)} - \eta \cdot \frac{1}{N} \sum_{n=1}^{N} (\frac{1}{|b|} \sum_{b_{(i)} \in \mathcal{B}_{(i)}^{(n)}} \nabla f(w_{(n)}^{(i)}, b_{(i)})) \quad \text{(1c)}$$

Distributed training can be scaled either *vertically* or *horizontally*. We raise each worker's batch-size in vertical scaling, increasing the per-step compute time along with memory consumption due to additional forward passes and larger activation maps. Horizontal scaling adds more nodes to scale-out training that may result in extra synchronization overhead [11]. With a fixed local batch-size b, the global batch ($B = Nb$) increases proportional to cluster-size in the latter, while vertical scaling over a fixed N can scale to larger B by increasing the local batch-size. These competing effects result in complex trade-offs between computation and communication efficiency.

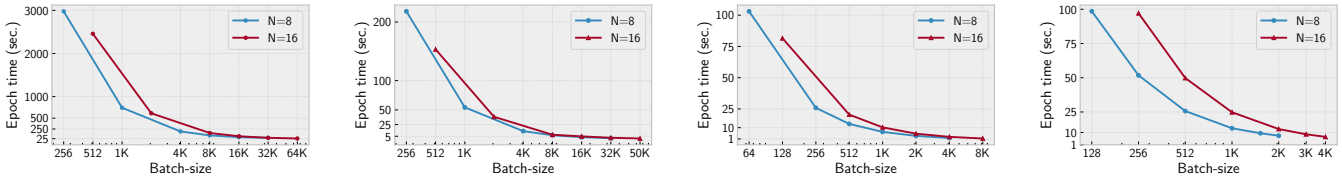
Figure (1) reports epoch completion time for different convolutional networks as the global batch-size is scaled over 8 and 16 V100 GPUs. An epoch requires $|D/B|$ steps for dataset D and global batch B, so increasing the batch-size reduces the total steps needed to complete each epoch. However, this increases the per-iteration cost (compute in vertical and communication in horizontal scaling), resulting in diminishing performance gains. Across all models, epoch time decreases sharply initially but gradually plateaus beyond a model-dependent threshold (e.g., 8192 for ResNet50 [12], 4096 in VGG11 [13], 2048 for AlexNet [14] and MobileNetv3 [15]). For a fixed cluster size N, the epoch time decreases as batch size increases, since fewer iterations are required to process the entire dataset, despite the higher per-iteration computational cost of larger local batches. However, as the global batch size continues to grow (toward the right end of the x-axis), intra-device parallelism reaches its limit, causing the epoch completion time to plateau. For the same batch-size, fewer workers at N=8 incur lower communication cost and thus achieve faster epoch times. For e.g., MobileNetv3 at 1K completed an epoch in *more than twice the time* with 16 nodes compared to a cluster of 8, so reduced synchronization cost outweighed the increased per-device compute cost here. However, as batch-size rises, memory constraints and reduced intra-worker parallel efficiency limit scalability, causing epoch times to converge across clusters. Under weak scaling, local batch-size is fixed and global batch rises with cluster-size, resulting in fewer steps per-epoch and improving performance despite extra communication. For e.g., ResNet50 with local batch-size 32 completes an epoch $\approx 3 \times 10^3$ seconds with 8 nodes and about 2.4×10^3 seconds with 16, about 25% faster over the larger cluster (both thus using the same global batch-size of 256).

Takeaway: *Distributed training exhibits non-linear scalability and diminishing returns from the interplay between computation, communication, and memory constraints.*

B. Statistical Efficiency in Distributed Training

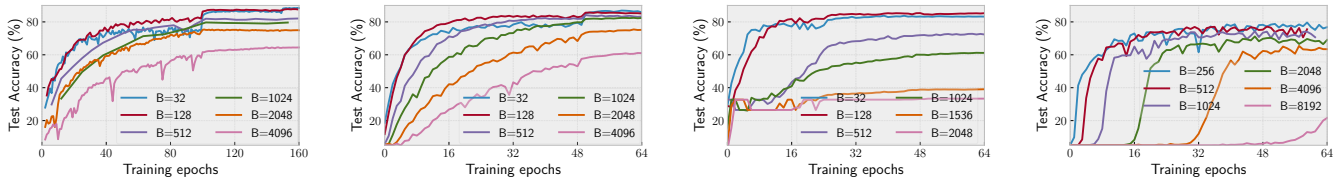
Although large-batch training can significantly improve throughput and achieve high training accuracy, it often suffers from degraded statistical performance on held-out data, commonly referred to as generalization gap [9], [18], [19]. Prior studies attribute this phenomenon to tendency of large-batches to converge to sharper minima in the optimization landscape [8], whereas smaller batches introduce stochasticity that biases optimization toward flatter minima with a narrower Hessian spectrum [20], empirically associated with improved generalization. We see this from the test accuracy of multiple models in Figure (2) by varying the global batch B for a fixed cluster-size. Across all models, generalization performance consistently declines as the training batch-size increases. The best accuracy of ResNet50 was attained at batch-size 32 and 128, albeit with lower training throughput due to limited parallelism. Increasing B to 4096 results in progressively worse accuracy. VGG11 observed a similar trend in test performance for the same batches. AlexNet exhibited even greater sensitivity to large-batch training, where accuracy dropped from 82% to 37% as batch-size was scaled from 128 to 2048. Similarly, MobileNetv3 degraded from 80% to 20% accuracy as we increased the batch-size from 256 to 8192. Beyond the batches shown in the figure, all evaluated models experienced severe generalization loss, rendering further batch scaling impractical.

Takeaway: *Large batches improve parallel efficiency, they can substantially degrade convergence quality. Consequently, any system that optimizes distributed training performance must jointly account for both parallel and statistical efficiency.*



(a) ResNet50 (b) VGG11 (c) AlexNet (d) MobileNetv3

Fig. 1. Epoch time of ResNet50 on ImageNet, VGG11 on CIFAR100, AlexNet on CalTech101, and MobileNetv3 on CalTech256 across two cluster-sizes.



(a) ResNet50 (b) VGG11 (c) AlexNet (d) MobileNetv3

Fig. 2. Test accuracy vs. batch-size. Smaller batches achieve better test accuracy than large-batch training, illustrating generalization gap in the latter.

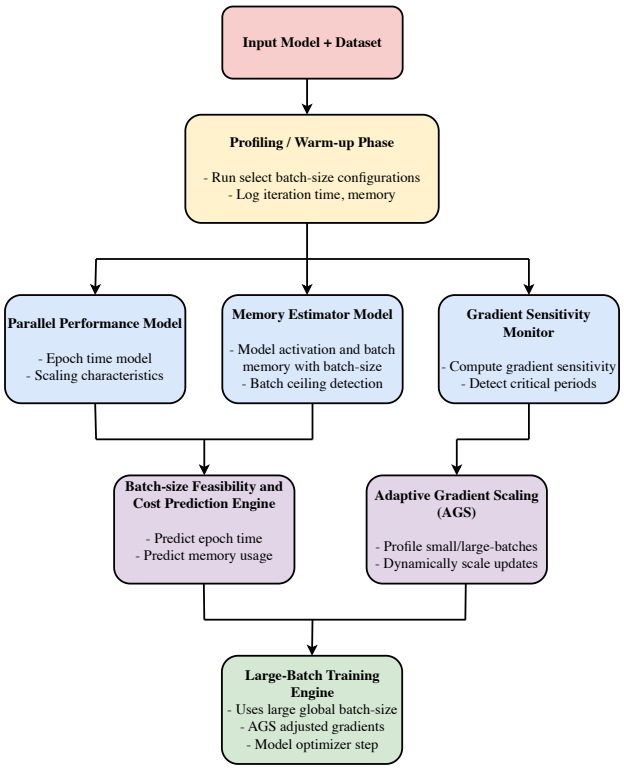


Fig. 3. A schematic overview of Tula's workflow.

III. Design

Parallel training across distributed nodes yields substantial speedups, albeit with diminishing returns as we scale further. The optimal configuration depends on multiple system and workload characteristics, including network topology, latency and bandwidth, model-size, cluster-size, batch-size, and hardware resources. Even with a systems-optimal large-batch configuration, distributed training still suffers from statistical inefficiency due to generalization gap. Tula jointly optimizes parallel and statistical efficiency by first identifying the optimal batch and cluster-size configuration that minimizes training time, cost or the knee-point of the trade-off curves, while simultaneously improving large-batch generalization. The overall logical flow of Tula is illustrated in Figure (3), with each block described in the following sections.

A. Parallel Performance Model

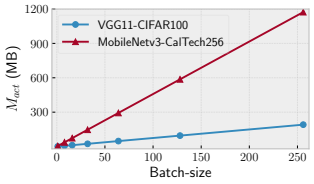
From a computational standpoint, SGD-based optimizations are iterative and repetitive, with each iteration i loading, processing and moving b mini-batch of data, compute loss and gradients, then aggregate and apply updates. Consequently, the per-step time t_{step} can be decomposed into computation [10] and communication time in synchronous training:

$$t_{step}^{(i)} = t_{compute}^{(i)} + t_{sync}^{(i)} \tag{2a}$$

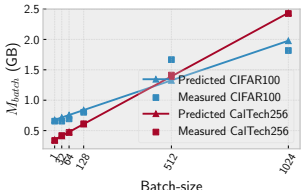
$$t_{comp} \propto b \tag{2b}$$

$$t_{sync} \propto \begin{cases} N, & \text{centralized parameter server} \\ (1 - \frac{1}{N}), & \text{ring-allreduce} \end{cases} \tag{2c}$$

From §II-A, scaling vertically incurs extra $t_{compute}$ due to larger local batches, while horizontal scaling affects t_{sync} due to additional nodes. For data-parallel training in modern cloud and data-centers, communication cost is dominated by bandwidth rather than latency [2], [11]. Centralized parameter servers incurs bandwidth cost that scales linearly with the

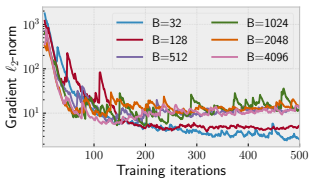


(a) Activation memory

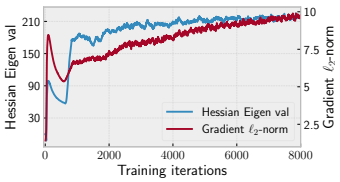


(b) Batch memory

Fig. 4. (a) $M_{act} \propto$ batch-size. (b) Linear model to predict M_{batch}.



(a) ResNet50 gradient ℓ_2-norm



(b) VGG11 second vs. first-order data

Fig. 5. (a) Gradient ℓ_2-norm in early iterations of ResNet50. (b) Largest Hessian eigenvalue and norm of the gradients over the iterations of VGG11.

number of nodes, while decentralized collectives like ring-allreduce are bandwidth-optimal. Although other collectives (e.g., tree-allreduce, reduce-scatter-allgather, etc.) have different latency–bandwidth trade-offs, Tula accommodates by incorporating its respective synchronization models into t_{sync}.

$$M_{total} = M_{param} + M_{grad} + M_{opt} + M_{act} + M_{batch} \quad (3)$$

1) Memory Estimation Model:

While a large batch-size takes fewer iterations to complete training epochs and improve throughput, the largest feasible batch-size is constrained by the available memory. The overall memory footprint of convolutional networks is approximated in Equation (3). Here, the parameter, gradient and optimizer-state memory, M_{param}, M_{grad} and M_{opt}, are largely static for a given model and precision format [27], [28]. In contrast, activation memory M_{act} and batch memory M_{batch} rise with training batch-size and often become the limiting factor. Although intermediate checkpointing lowers the activation memory cost, it incurs additional compute cost [28].

From Figure (4), M_{act} scales linearly with the batch-size for representative models, and M_{batch} can be accurately approximated using a simple linear model for different datasets. The scatter points in Figure (4b) represents the actual dataloader memory for CIFAR100 and CalTech256 datasets at various batches, while the line plot signifies the estimated memory predicted using a linear model with respect to batch-size. By leveraging Equation (3) and lightweight profiling, Tula can estimate the maximum batch-size supported by a given model, dataset, and hardware, thus bounding the feasible search space for optimization.

2) Performance and Cost Modeling:

The objective of parallel performance model is to infer time and cost of training as a function of cluster and batch-size to estimate trade-off curves like Figure (1). With a user-specified minimum batch-size and a memory model inferred upper bound, (b_{min}, b_{max}), Tula profiles a subset of candidate configurations briefly to fit computation and communication models of Equation (2).

$$T = I \cdot \widetilde{t}_{step} \quad \Rightarrow \quad T = \frac{ED}{B} \cdot \widetilde{t}_{step} \quad (4a)$$

$$C = T \cdot N \cdot \tilde{p} \quad (4b)$$

Total training time is estimated by Equation (4a), where E is the total epochs for I steps with dataset-size D and average iteration time $\widetilde{t}_{step}$. The total cost with a static pricing model (common in the cloud) is given by Equation (4b), where $\tilde{p}$ denotes the per-node price with N nodes in total. Tula offers two search modes: *full* or *partial*, where the former profiles all possible (N, B) while the latter minimizes exploration cost by profiling only the extremum configurations (min-max cluster and batch-size) and extrapolates via regression. While partial-search substantially lowers profiling cost, it incurs estimation errors due to limited samples, a trade-off we evaluate in §IV.

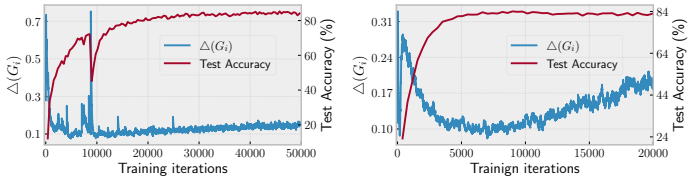
3) Configuration Selection Policy:

Given a job and available resources, Tula selects the batch and cluster-size configuration that best satisfies a user-specified objective like minimizing training time, cost, or selecting the trade-off curve's knee-point (identified using the Kneedle algorithm [32]). *In Tula, the feasible search space is specified by user-defined constraints on the minimum batch and cluster-size, largest available cluster-size, combined with memory-based upper batch-size bounds*. Extremely small batches or clusters fail to exploit available parallelism, while overly large batches exacerbate statistical inefficiency and memory pressure. By explicitly modeling these trade-offs, Tula enables a principled, efficient configuration selection mechanism for large-batch training.

B. Gradient Sensitivity in Distributed Training

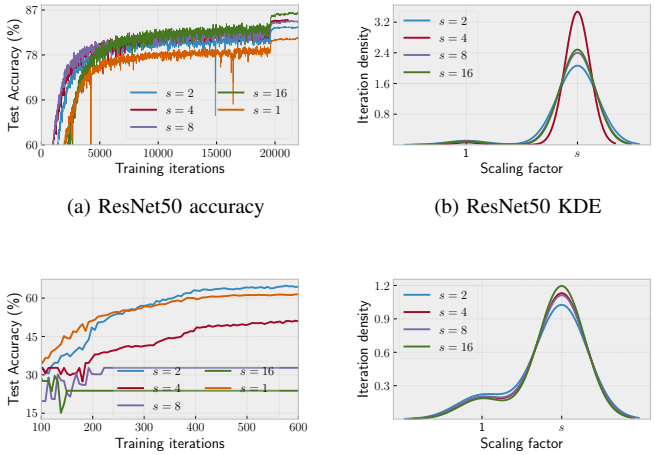
The trajectory of model updates varies significantly throughout training due to the stochastic nature of gradient-based optimization. Models are particularly sensitive during early training phases [21] and during specific critical periods of convergence [22]. The degree of sensitivity depends on the model, dataset characteristics and optimization hyperparameters [23]. Consequently, not all training iterations contribute equally to convergence, motivating the need to identify and treat sensitive updates differently. Figure (5a) illustrates the evolution of gradient ℓ_2-norm for various batches in ResNet50. Initially, large-batch updates are smaller and less noisy as they more closely approximate the full-batch gradient, while smaller-batches exhibit higher variance and larger magnitudes.

As training progresses, this behavior reverses: large-batch gradients grow in magnitude and converge toward sharper minima, whereas small-batch gradients stabilize around flatter regions of the loss landscape. Second-order information like



(a) ResNet50 (b) VGG11

Fig. 6. Gradient variability and test accuracy are correlated such that model convergence path is mirrored in the trajectory of the rate of gradient change.



(a) ResNet50 accuracy (b) ResNet50 KDE

(c) AlexNet accuracy (d) AlexNet KDE

Fig. 7. Convergence quality and Kernel Density Estimates (KDE) of iterations for batch-size 1024 under static gradient scaling $\mathcal{U}(s)$ with $\delta = 0.5$ and various scaling factors s. Static scaling improves accuracy over vanilla large-batch training ($s = 1$), but excessive scaling degrades convergence quality.

eigenvalues of the Hessian matrix is an effective indicator of sensitive training phases [24], but computing it at every iteration is prohibitively expensive. However, prior work shows that changes in Hessian eigenvalues can be approximated using first-order gradient statistics [25]. Figure (5b) compares the largest Hessian eigenvalue and gradient norm in VGG11, demonstrating that abrupt changes in curvature are well reflected in first-order gradients, albeit at a significantly lower computational cost.

To quantify temporal sensitivity, we track changes in the gradient norm for batch-size b over iteration i as:

$$\triangle(G_{(b)}^{(i)}) = \left| \frac{|G_{(b)}^{(i)}|^2 - |G_{(b)}^{(i-1)}|^2}{|G_{(b)}^{(i-1)}|^2} \right| \tag{5}$$

The gradient variability metric $\triangle(G_{(b)}^{(i)})$ serves as an effective, low-cost solution to track sensitive and critical updates in an online manner during training [23], [25]. Figure (6) plots this metric alongside test accuracy for ResNet50 and VGG11. In ResNet50, the rapid increase in accuracy during early training coincides with high variability in $\Delta(G^{(i)})$. LR decay around 9K steps induces abrupt accuracy changes that are simultaneously detected as sharp spikes in gradient change. As training converges, both accuracy and $\Delta(G^{(i)})$ stabilize. VGG11 exhibits a similar correlation, with steady improvements in accuracy mirrored by a gradual decline in gradient variability. The degree of sensitivity may differ across Deep Neural Networks (DNN)s due to variations in architecture, depth, and dataset characteristics. Nevertheless, $\Delta(G^{(i)})$ is able to identify critical phases within each training run by tracking relative changes in gradient magnitudes.

Takeaway: *First-order gradient statistics give a lightweight and effective mechanism for detecting sensitive regions of the training trajectory. Leveraging this insight, we design an adaptive gradient scaling mechanism that modulates large-batch updates based on training sensitivity, thereby mitigating the generalization gap without incurring significant computational overhead (described in §III-C).*

C. Statistical Performance Model

At equivalent training stages, large-batch updates are generally smoother than small-batch gradients, especially during early training when optimization is most sensitive [21]. Gradients computed over large batches better approximate the true full-batch gradient [5], while small-batch updates exhibit higher variance. While this noise can slow early convergence, it provides implicit regularization in later stages, guides models towards flatter minima and improves generalization. In contrast, large-batch optimization often converges to sharp minima [8], resulting in overfitting and degraded test performance. We model the discrepancy between small and large-batch updates at iteration i as an adaptive noise term:

$$G_{b_{small}}^{(i)} = G_{b_{large}}^{(i)} + \gamma^{(i)}(b_{small}, b_{large}) \tag{6}$$

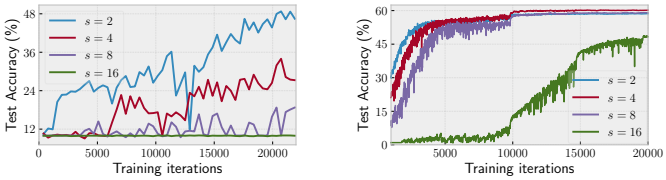
where $\gamma^{(i)}$ captures the variance induced by batch-size differences. Rather than injecting fixed noise [29], we transform large-batch gradients through a mapping mechanism $\mathcal{M}(\cdot)$ that adapts to training sensitivity, as shown in Equation (7). $\mathcal{M}(\cdot)$ captures the spatio-temporal relationship between large and small-batch updates, effectively scaling large-batch gradients to approximate the regularization effect of smaller batches. If applied judiciously, this can potentially improve convergence while retaining the efficiency of large-batch parallelization.

$$G_{b_{large}}^{(i)} = \frac{1}{|b_{large}|} \nabla f(w^{(i)}, b_{large}) \tag{7a}$$

$$\widetilde{G}^{(i)} = \mathcal{M}(G_{b_{large}}^{(i)}), \quad w^{(i+1)} = w^{(i)} - \eta \cdot \widetilde{G}^{(i)} \tag{7b}$$

1) Static Gradient Scaling:

We first consider a static mapping that applies a stepwise scaling function based on gradient sensitivity, shown in Equation (8). Sensitive phases are detected using $\Delta(G^{(i)})$ from Equation (5) and setting a threshold δ such that updates are unscaled when $\Delta(G^{(i)}) \geq \delta$ (i.e., in sensitive phases) and scaled otherwise.



(a) ResNet50 with δ = 0.8 (b) VGG11 with δ = 0.2

Fig. 8. Impact of gradient variability threshold δ. Larger δ biases training toward scaled updates, while a smaller δ favors unscaled (sensitive) updates.

$$\mathcal{U}(s) = \begin{cases} 1, & \text{sensitive phase} \\ s, & \text{otherwise} \end{cases} \tag{8a}$$

$$\widetilde{G}^{(i)} = \mathcal{U}(s) \odot G_b^{(i)} \tag{8b}$$

Figure (7) illustrates the statistical impact of static scaling when model updates are blindly scaled up by various $s \in [2, 4, 8, 16]$ with δ = 0.5 and a base batch-size of 1024. For ResNet50 in Figure (7a), convergence quality improved across all scaling factors (for any $s > 1$) over vanilla large-batch training (s = 1). On the other hand, AlexNet saw improved accuracy over moderate scaling (s = 2) but degraded performance from excess scaling ($s > 2$). Correspondingly, we plot the kernel density estimates (KDEs) of ResNet50 and AlexNet in Figures (7b) and (7d), which quantifies the fraction of iterations that were subjected to scaling $s > 1$ versus the portion of training iterations that used the original updates as is (i.e., s = 1) based on the sensitivity of model updates. The KDEs show how sensitivity-aware scaling preserves critical updates by switching the scaling factor between 1.0 and s on a per-iteration basis. Compared to ResNet50, AlexNet was more vulnerable to gradient sensitivity for every s, as seen from its KDE estimates where a considerable portion of updates default to the base scaling of 1.0. This high degree of sensitivity translates to its convergence quality as well, where higher s failed to outperform vanilla large-batch training (s = 1). In contrast, ResNet50 was more robust to gradient sensitivity across all s, and most iterations trained with the scaled factor s. Due to its low sensitivity, ResNet50 converged successfully for every $s > 1$ that we evaluated.

Impact of gradient variability threshold: Figure (8) plots the convergence curves for ResNet and VGG as we vary the threshold metric over multiple scaling factors. The results show that δ significantly affects convergence, where a small δ is conservative and favors unscaled updates while a larger δ applies scaling more aggressively. For ResNet50, increasing δ from 0.5 to 0.8 (Figure (7a) and (8a)) substantially degrades test accuracy. On the other hand, VGG11 benefits significantly with a smaller threshold of 0.2, where scaling factors 2, 4 and 8× achieved nearly the same convergence, while s = 16 attained nearly 12% lesser accuracy (albeit still better than ResNet50 at various s). Empirically, δ = 0.5 gives a robust balance across all models.

Algorithm 1: Adaptive Gradient Scaling (AGS)

Input: Batches (b_{small}, b_{large}), threshold δ, LR η, stability term $\epsilon = 10^{-8}$, *gradscale* ← **1** (per-parameter)

1 **for** *epoch* $e = 1, 2, \ldots$ **do**
2 **for** *iteration* $i = 1, 2, \ldots$ **do**
3 $G^{(i)}_{b_{large}} \leftarrow \nabla f(w^{(i)}, b^{(i)}_{large})$
4 Compute $\Delta(G^{(i)})$ using Eq. (5)
5 **if** $\Delta(G^{(i)}) < \delta$ **then**
6 $\widetilde{G}^{(i)} \leftarrow$ *gradscale* $\odot\, G^{(i)}_{b_{large}}$
7 **else**
8 $\widetilde{G}^{(i)} \leftarrow G^{(i)}_{b_{large}}$
9 $w^{(i+1)} \leftarrow w^{(i)} - \eta \widetilde{G}^{(i)}$
10 *gradscale* ← UPDATEGRADSCALE()
11 **function** UPDATEGRADSCALE()
12 $s_{max} \leftarrow \sqrt{b_{large}/b_{small}}$
13 Sample $b^{(i)}_{small} \subset b^{(i)}_{large}$, $|b_{small}| < |b_{large}|$
14 $G^{(i)}_{b_{small}} \leftarrow \nabla f(w^{(i)}, b^{(i)}_{small})$
15 **foreach** *parameter* (g_{small}, g_{large}) **do**
16 $s \leftarrow \min\left(\left|\frac{g_{small}}{g_{large} + \epsilon}\right|, s_{max}\right)$
17 *gradscale*.$key(p) \leftarrow s$
18 **return** *gradscale*

2) Adaptive Gradient Scaling (AGS):

While static gradient scaling improves accuracy in many cases, it remains sensitive to the choice of scaling factor and can degrade convergence when scaling is overly aggressive. To address this limitation, we propose a profiling-based Adaptive Gradient Scaling (AGS), which dynamically transforms large-batch gradients to approximate small-batch behavior. Unlike static scaling, AGS applies per-parameter scaling and adapts over the course of training. As shown in Algorithm (1), updates from large-batches are computed on every step, while training sensitivity is monitored using the gradient variability metric $\Delta(G^{(i)})$. When training enters a sensitive phase ($\Delta(G^{(i)}) \geq \delta$), AGS preserves the original gradients. Otherwise, updates are scaled using a learned per-parameter factor. The scaling factors are updated periodically via lightweight profiling, illustrated in the function UPDATEGRADSCALE. A small sub-batch is sampled from the large-batch, and its gradients are compared against large-batch gradients (while adding numerical stability term ϵ) to estimate the relative magnitude per parameter. This overhead is tolerable if it is executed infrequently; this cost can rise considerably when small and large-batch updates are compared more frequently.

To prevent instability due to stochastic noise, AGS bounds each scaling factor by s_{max} (line 12) following prior heuristics [30], [31]. This constraint mitigates gradient explosion when the ratio of small to large-batch gradients becomes large. Conversely, AGS can also reduce update magnitudes ($s < 1$) when large-batch gradients exceeds small-batches—

a behavior observed in the later training stages (seen in Figure (5a)).

Convergence properties: Under an L-smooth loss function and bounded per-parameter scaling factors $s \in [s_{min}, s_{max}]$, AGS converges at a rate of $\mathcal{O}(1/I)$ for a fixed learning-rate $\eta > 2s_{min}/(Ls_{max}^2)$, where I is the total training iterations. When the learning rate decays as $\eta \propto 1/\sqrt{I}$, AGS achieves the same $\mathcal{O}(1/\sqrt{I})$ convergence-rate as standard SGD:

Lemma 1 (Convergence in Adaptive Gradient Scaling)**.** *Assume that the objective function f is L-smooth and that stochastic gradients computed using large batches are unbiased with bounded variance, i.e.,*

$$\mathbb{E}[g^{(i)} \mid w^{(i)}] = \nabla f(w^{(i)}), \qquad \mathbb{E}\|g^{(i)} - \nabla f(w^{(i)})\|^2 \le \sigma^2. \tag{9}$$

Let the learning-rate satisfy $\eta > 0$, and let AGS enforce bounded per-parameter scaling factors such that for every iteration i and model parameter p,

$$0 < s_{min} < s_p^{(i)} \le s_{max} \tag{10}$$

If the learning-rate satisfies

$$\eta < \frac{2s_{min}}{Ls_{max}^2}, \tag{11}$$

then the iterates generated by AGS satisfy

$$\frac{1}{I}\sum_{i=1}^{I}\mathbb{E}\|\nabla f(w^{(i)})\|^2 \le \frac{f(w^{(1)}) - f^*}{\epsilon I} + \frac{L\eta^2 s_{max}^2\sigma^2}{2\epsilon}, \tag{12}$$

where $f^* = \inf f$ and

$$\epsilon := \eta s_{min} - \frac{L\eta^2 s_{max}^2}{2} > 0. \tag{13}$$

Consequently, AGS converges to a stationary point at rate $\mathcal{O}(1/I)$ for constant η, and at rate $\mathcal{O}(1/\sqrt{I})$ when $\eta \propto 1/\sqrt{I}$.

Proof. Since f is L-smooth, the update rule of AGS,

$$w^{(i+1)} = w^{(i)} - \eta\, s^{(i)} \odot g^{(i)}, \tag{14}$$

satisfies

$$f(w^{(i+1)}) \le f(w^{(i)}) - \eta\langle \nabla f(w^{(i)}), s^{(i)} \odot g^{(i)}\rangle + \frac{L\eta^2}{2}\|s^{(i)} \odot g^{(i)}\|^2 \tag{15}$$

Taking conditional expectation with respect to $w^{(i)}$ and noting that $s^{(i)}$ is deterministic relative to the batch randomness yields

$$\mathbb{E}[f(w^{(i+1)}) \mid w^{(i)}] \le f(w^{(i)}) - \eta\,\langle \nabla f(w^{(i)}), s^{(i)} \odot \nabla f(w^{(i)})\rangle + \frac{L\eta^2}{2}\,\mathbb{E}\left\|s^{(i)} \odot g^{(i)}\right\|^2. \tag{16}$$

Using the boundedness of the scaling factors, we have

$$\langle \nabla f, s^{(i)} \odot \nabla f\rangle \ge s_{min}\|\nabla f\|^2, \;\; \|s^{(i)} \odot g^{(i)}\|^2 \le s_{max}^2\|g^{(i)}\|^2. \tag{17}$$

Moreover, by the variance assumption,

$$\begin{aligned}\mathbb{E}\|g^{(i)}\|^2 &= \|\nabla f(w^{(i)})\|^2 + \mathbb{E}\left\|g^{(i)} - \nabla f(w^{(i)})\right\|^2 \\ &\le \|\nabla f(w^{(i)})\|^2 + \sigma^2.\end{aligned} \tag{18}$$

Substituting these bounds yields

$$\mathbb{E}[f(w^{(i+1)}) \mid w^{(i)}] \le f(w^{(i)}) - \eta s_{min}\|\nabla f(w^{(i)})\|^2 + \frac{L\eta^2 s_{max}^2}{2}\Big(\|\nabla f(w^{(i)})\|^2 + \sigma^2\Big). \tag{19}$$

Rearranging terms gives us

$$\Big(\eta s_{min} - \frac{L\eta^2 s_{max}^2}{2}\Big)\|\nabla f(w^{(i)})\|^2 \le f(w^{(i)}) - \mathbb{E}[f(w^{(i+1)}) \mid w^{(i)}] + \frac{L\eta^2 s_{max}^2\sigma^2}{2}. \tag{20}$$

Choosing $\eta < 2s_{min}/(Ls_{max}^2)$ ensures the coefficient on the left-hand side is positive. Setting

$$\epsilon := \eta s_{min} - \frac{L\eta^2 s_{max}^2}{2}, \tag{21}$$

and taking full expectation and summing over I iterations yields

$$\epsilon\sum_{i=1}^{I}\mathbb{E}\|\nabla f(w^{(i)})\|^2 \le f(w^{(1)}) - \mathbb{E}[f(w^{(I)})] + \frac{IL\eta^2 s_{max}^2\sigma^2}{2}. \tag{22}$$

Setting $f^* = \inf f$ and dividing by $I\epsilon$ completes the proof. □

IV. Evaluation

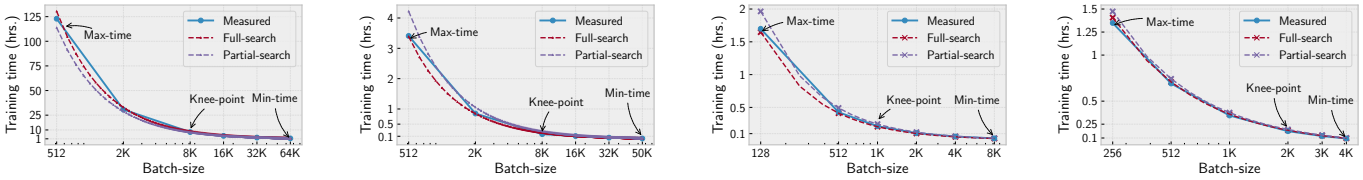
A. Implementation

We implement Tula over PyTorch v2.6.0, with its scaling indicators realized by extending the PyTorch Distributed Data-Parallel (DDP) module. The *TulaOptimizer* extends the base *torch.optim.Optimizer* class to wrap any standard optimizer, and tracks gradient norms across steps and computes gradient variability metric in an online manner.

During the initial exploratory phase, an external model service profiles training runs and collects scaling indicators to construct a performance model. By default, Tula employs a partial-search strategy due to its lower overhead and comparable accuracy to full-search strategy (detailed in §IV-C). Once an optimal configuration is chosen, training proceeds using large batches with adaptive gradient scaling (AGS). Scaling factors are updated after every epoch by default, although this frequency is configurable. AGS in Tula is parameterized by a gradient variability threshold δ and a small reference batch-size used to approximate small-batch updates.

B. Experimental setup

We run experiments with varying configurations on up to 16 nodes, with each node having a 48-core Intel Xeon E5-2560 CPU, 128 GB of memory, and 1 V100S GPU with 32 GB memory running Ubuntu 22.04. The nodes are connected via 10Gbps Ethernet and communicate using NCCL v2.21.5 with setting *NCCL_ALGO = ring*. Although the GPUs used in our experiments are relatively dated given the rapid evolution of AI/ML hardware, with modern accelerators



(a) ResNet50 (b) VGG11 (c) AlexNet (d) MobileNetv3

Fig. 9. Training time vs. batch-size on a 16-node cluster. Dashed lines denote predictions from performance models built using full and partial-search. Both strategies accurately capture runtime trends and identify near-optimal batch-size. Larger batches offer diminishing returns; training time does not significantly reduce from the knee-point to the minimum-time configuration at the largest batch-size. However, the latter suffers considerably more generalization loss.

featuring tensor-optimized execution units, high-bandwidth memory, and tightly integrated interconnect fabrics. While raw compute and memory bandwidth have increased by orders of magnitude, the rapid growth in model scale has preserved communication and memory movement as dominant system bottlenecks in distributed training.

We evaluate four models: ResNet50, VGG11, AlexNet and MobileNetv3 over ImageNet-1K, CIFAR100, CalTech101 and CalTech256. These datasets contain 1.2M, 50K, 9K and 30K images respectively. ResNet50 on ImageNet reports top-5 while all other DNNs report top-1 test accuracy. The models train with cross-entropy loss, SGD with momentum factor 0.9, initial LR of 0.05 in ResNet50 and 0.01 across others. MobileNetv3 applied weight decay 0.0001, while others used decay factor 0.0005. The training schedule decays LR by $10\times$ in ResNet50 after $[100, 150, 200]$ epochs, AlexNet after $[25, 50, 75]$ epochs, and MobileNetv3 after $[40, 80]$ epochs. The LR in VGG11 is decayed by $5\times$ after $[50, 75, 100]$ epochs respectively.

Across all models, the search space for the performance model sets the minimum and maximum cluster-size to 2 and 16, with candidate configurations increasing exponentially, i.e., $[2, 4, 8, 16]$. Batch-sizes are bounded with a minimum of 32, while the upper bound is determined by the memory estimation model. Candidate batches also grow exponentially within this range. Under full-search, each candidate configuration is briefly profiled, whereas partial-search evaluates only two points (the extremums of batch and cluster-size).

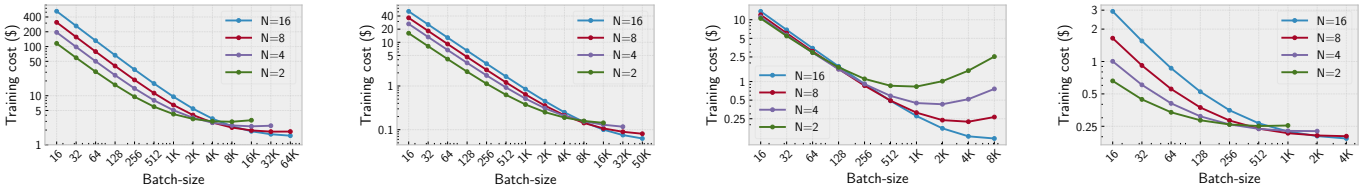
Since memory estimation can be conservative, Tula may overestimate the largest feasible batch-size and encounter out-of-memory (OOM) errors. Tula handles this by frequent job checkpointing and relaunching progressively smaller batches until a feasible configuration is found (akin to PyTorch Lightning's *auto_scale_batch_size* feature). Thus, Tula first relies on its memory model and, if necessary, refines the estimate through runtime probing. Once the performance model is built and an optimal batch-size chosen based on user objective (minimizing time, cost, or identifying the knee-point), Tula deploys training with AGS (setting $\delta = 0.5$ in current experiments) to improve its large-batch generalization quality.

C. Performance Model and Optimal Batch-Size

1) Estimating training time and cost:

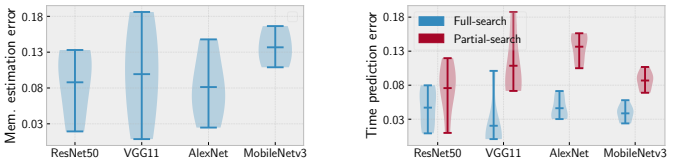
Using the performance model described in §III-A, we estimate the time and cost of distributed training across a range of global batch-sizes. Figure (9) reports the measured training time along with predictions obtained from performance models constructed via full- and partial-search. For all DNNs, the predicted curves closely track the observed runtime trends as batch-size increases. The full-search model is constructed by exhaustively sampling all configurations within the search space, collecting performance metrics for each, and fitting the model using the multiple data points recorded during the profiling phase. In contrast, the partial-search model is trained using only the boundary configurations—specifically, the minimum and maximum batch sizes selected a priori—and fitting over these extremal points. Owing to the richer set of profiling data available to the full-search strategy, its predictions (red) more closely aligns with the ground truth (blue) in Figure (9). While full-search yields slightly more accurate estimates, partial-search achieves comparable accuracy with substantially lower profiling overhead, as it evaluates only the boundary configurations. As the batch-size approaches its upper bound (i.e., the rightmost region of the curve), the estimated time or cost predicted by both strategies converges and saturates at nearly the same level.

Assuming a fixed pricing model (see Equation (4b)), training cost follows the same trajectory as execution time. This reflects common cloud pricing practices where compute resources are charged at a constant hourly rate. For example, a Google Cloud virtual machine with a single NVIDIA V100 GPU costs $2.48 per hour in the US region in 2025. Under such a pricing model, Tula enables users to select the batch-size that minimizes training cost or identify the knee-point beyond which increasing the batch-size yields diminishing returns in terms of time savings. For the 16-node configuration, this knee-point corresponded to a batch-size of 8192 for ResNet50 and VGG11, 1024 for AlexNet and 2048 for MobileNetv3. Beyond these points, further increases in batch-size does not significantly reduce training time/cost, but may further exacerbate the generalization gap if the chosen batch-size is too large.



(a) ResNet50 (b) VGG11 (c) AlexNet (d) MobileNetv3

Fig. 10. Training cost under a memory-based pricing model that charges \$0.15/hour/GB of GPU memory usage. The optimal batch-size shifts under this cost model due to increased memory footprint on larger batches.



(a) Memory prediction error (b) Time prediction error

Fig. 11. Relative error of Tula in estimating GPU memory and training time.

2) Memory-based pricing model:

Training cost is ultimately dictated by the underlying pricing model, which may depend on resource utilization rather than the available node count. To evaluate such scenarios, we consider an alternative pricing strategy in which GPUs are charged based on their memory usage. Under this cost model, a V100 GPU instance incurs cost $\$0.15/\text{hour}/\text{GB}$ of memory.

We construct the performance model using partial-search and estimate the cost of training as a function of batch-size under memory-based pricing for cluster-size configurations $[2, 4, 8, 16]$. Figure (10) illustrates the resulting cost surfaces. In contrast to fixed-rate pricing, the optimal batch-size shifts due to the increased memory footprint associated with larger batches. At 16 nodes, the cost-optimal batch size is 16384 for ResNet50 and VGG11, 2048 for AlexNet, and 1024 for MobileNetv3. In some cases like AlexNet (Figure (10c)), cost may increase at larger batches when additional activation memory cost outweighs the savings from allocating fewer nodes. Thus, training on a 16 node cluster becomes cheaper than training on 2, 4 or 8 for batches 1024 or greater. These results highlight the importance of jointly considering system performance, memory usage, and pricing models when selecting the batch-size in distributed training.

3) Estimation accuracy:

The accuracy of Tula's performance models is evaluated by comparing predicted memory usage and training time against observed values. Figure (11a) reports the relative error of the memory estimation model across all batch-sizes explored during search phase (shown here for all local batches corresponding to $N = 16$ from Figures (9) and (10)). *Across all the evaluated neural networks, the median memory prediction error ranges only between 8–14%*. When estimation error does indeed result in out-of-memory failures, Tula automatically falls back to the next feasible configuration and redeploys the job, ensuring robustness in practice.

Table I. Top-1 test accuracy with vanilla large-batch training (LBT), LARS, Post-local SGD (PoLo), LR scaling (LRS), and Tula's AGS. For every batch-size, **green** → high, **red** → low, **yellow** → intermediate accuracy.

Model	Batch-size	Test accuracy (%)				
		LBT	LARS	PoLo	LRS	*Tula*
ResNet50	1024	79.67	83.26	74.03	83.33	*85.28*
	2048	75.55	85.02	65.58	74.44	*84.58*
	4096	64.72	86.77	62.46	70.11	*82.16*
VGG11	1024	83.30	52.93	82.90	7.50	*84.42*
	2048	75.92	50.91	81.13	5.00	*82.58*
	4096	70.24	48.64	79.50	5.00	*81.14*
AlexNet	256	80.62	55.40	85.11	83.38	*84.44*
	512	72.80	56.00	85.95	72.30	*83.00*
	1024	62.60	57.00	85.67	32.74	*81.20*
MobileNetv3	1024	76.80	67.64	81.14	78.66	*81.56*
	2048	71.15	72.82	75.54	10.00	*79.68*
	4096	66.48	74.29	73.91	5.00	*75.70*

Figure (11b) shows the relative error in training-time prediction for models constructed using full and partial-search strategies. Full-search achieves higher accuracy (and lower error) due to more extensive profiling across configurations, *yielding median errors below 2–5%*. Partial-search, which profiles only the boundary configurations, *incurs higher median error but remains within 7.5–14% across all models*. Despite its slightly lower accuracy, partial-search reliably captures relative performance trends across batch and cluster configurations.

It is important to note that Tula is designed to identify configurations that optimize training time, cost, or knee-point behavior rather than to minimize absolute prediction error. Our results show that, even with partial-search, Tula consistently identifies the ideal configuration that optimizes for parallel efficiency under different scaling strategies and objectives.

D. Statistical Performance with Large-Batches

1) Evaluation methodology:

From the performance models in Figure (9), Tula identifies knee-points on the time–batch-size trade-off curves as the optimal configuration. *In this section, we evaluate convergence quality of models over batches around these knee-points. Beyond these points, further batch scaling provides diminishing returns in time or cost reduction.*

We compare Tula's AGS against popular techniques: vanilla large-batch training (LBT), learning-rate scaling (LRS) [7],

Post-local SGD (PoLo) [33], and LARS [9]. LRS and LARS apply linear learning-rate scaling relative to a base batch-size of 128, while PoLo switches from synchronous to local-SGD after 5 epochs, aggregating updates 4 times every epoch. All these methods apply LR warmup for 5 epochs.

2) Results:

Table (I) shows that Tula consistently mitigates the generalization degradation observed with vanilla large-batch training (LBT) across all models and batch-sizes. In LBT, test accuracy deteriorates in correspondence to larger global batches on account of the generalization gap in large-batch training. In VGG11 and MobileNet, AGS saw the best accuracy across all evaluated batches. For ResNet50 and AlexNet, AGS remains competitive and avoids severe accuracy degradation at larger batches, although LARS and PoLo observed slightly higher accuracy. In contrast, LRS demonstrates unstable behavior at large batches, with accuracy deteriorating markedly as the scaled LR grows excessively large. The scaling factor in LRS is directly proportional to the ratio between the target large batch and the reference small batch-size; consequently, scaling to very large batches from a small base batch-size can produce an excessively high multiplier. Moreover, unlike AGS, LRS applies this scaling factor uniformly at every iteration, regardless of the optimization landscape, which can further destabilize the training process (as specially seen in models like VGG11 and MobileNetv3). Although LARS performed well on ResNet50, it did not converge well on other models. This is because LARS (which scales the LR by $||w||/||\nabla w||$) was originally developed for residual networks [9], while models like VGG11, AlexNet and MobileNetv3 have different characteristics. Without residual paths, large per-layer scaling can significantly cause gradient instability, while depthwise layers (such as those in MobileNetv3) have very small parameter norms that can produce extreme scaling ratios in LARS, resulting in unstable LR that destabilizes training. In case of PoLo, synchronization frequency plays a crucial role in model convergence. Reduced synchronization can lead to model drift, and converging local models to sharp minima even before model aggregation. We thus see low to intermediate accuracy in most cases, with the exception of AlexNet. We suspect that PoLo performed well here due to the model's small size with fewer deep non-linearity over a modest dataset like CalTech101. In contrast, ResNet50 training over ImageNet saw least accuracy with PoLo as reduced synchronization deprived the model of beneficial stochasticity.

Overall, while existing techniques perform well only for specific models or batch regimes, Tula's AGS provides robust and consistently high accuracy, making it well-suited for system-driven batch-size optimization in large-scale distributed training.

V. Related Work

Tula lies at the intersection of distributed training systems and large-batch optimization. Its design addresses two complementary challenges: (i) selecting the efficient batch-size and cluster configurations for distributed training on shared or cloud infrastructure [34], and (ii) mitigating the statistical degradation that often accompanies large-batch training. The optimal batch-size depends jointly on the model, dataset, hardware configuration, and user objective (e.g., minimizing time, cost, or identifying a knee-point), thus motivating system-level approaches that reason across these dimensions.

Several prior systems jointly consider parallel efficiency and statistical behavior when training deep models at scale. Pollux [4] dynamically schedules training jobs on heterogeneous clusters based on performance modeling, while KungFu [6] adapts resource allocation in response to observed training dynamics. AdaScale SGD [35] adjusts the effective LR based on gradient variance to improve scaling efficiency. In contrast to these approaches, Tula explicitly models batch-size effects on both performance and generalization, enabling principled batch selection prior to full-scale training.

Memory-aware training has also received significant attention. ZeRO [28] analyzes optimizer and model states to reduce memory footprint and enable larger batches, particularly for transformer models. PyTorch Lightning's *auto_scale_batch_size* feature uses iterative probing to identify the largest batch-size that fits in GPU memory. Tula's memory estimation and checkpoint-and-redeploy mechanism is complementary to these efforts, allowing it to recover gracefully from estimation errors while minimizing profiling overhead. Prior work has also explored training sensitivity to reduce communication costs in distributed training [23], [25], [39].

A large body of work focuses on mitigating the generalization gap observed in large-batch training. Common techniques include learning-rate scaling and warmup [7], batch and layer normalization, adaptive learning-rates, and gradient clipping. LR scaling (LRS) increases the learning-rate proportional to the batch-size, while other related methods co-tune batch-size and LR [30], [36]. Other approaches introduce noise or extrapolation to avoid convergence to sharp minima [29], [37]. Post-local SGD (PoLo) [33] begins with synchronous training and later switches to local updates to improve generalization at scale. Layer-wise optimizers such as LARS [9] and LAMB [38] further adapt LR based on layer-wise gradient and weight magnitudes.

These optimizations are orthogonal and complementary to Tula's adaptive gradient scaling. Tula provides a system-level framework for selecting optimal configurations and stabilizing training at scale, into which alternative large-batch optimization strategies can also be readily integrated.

VI. Current Limitations and Discussion

While Tula demonstrates strong improvements in both parallel and statistical efficiency, it has certain limitations that motivate future extensions. Tula's performance model assumes relatively stable communication characteristics. In shared or congested networks, communication costs can vary significantly over time, potentially reducing prediction accuracy. We plan to explore techniques to adapt the performance model to dynamic network conditions in the future.

While partial-search significantly reduces profiling cost over full-search, this overhead may still be non-trivial in environments with long job queue times or strict scheduling constraints, like shared HPC systems. Our evaluation considers only fixed-rate and memory-based pricing schemes. In practice, cloud pricing can involve additional factors such as spot instances and heterogeneous hardware. Tula does not currently incorporate such dynamic or multi-dimensional cost models.

The convergence behavior of Tula depends on hyperparameters of AGS, including the gradient variability threshold δ, the choice of reference small batch-size, and the frequency of per-parameter scaling update that compares small and large-batch gradients. Setting extreme values on AGS, such as a very small reference batch-size and a very high large-batch could destabilize updates if the scaling factor is high (by making s_{max} and g_{small} in Algorithm (1) very high). The threshold δ in AGS governs whether updates are rescaled or applied without modification; setting it too high may unnecessarily scale large-batch updates, even when the underlying optimization landscape does not exhibit steep curvature or sharp gradients. On the other hand, setting a very small δ may not scale updates at all, effectively acting as vanilla large-batch training. Through empirical evaluation, we found $\delta = 0.5$ to provide the best trade-off, and therefore adopt it as the default setting in our experiments.

While we observed that fixed parameter values are effective across all the evaluated models and datasets, these settings may require adjustment for different architectures, datasets, or optimization regimes. In future, we plan to incorporate a checkpoint-based auto-tuning mechanism that monitors validation accuracy during training and prunes configurations that negatively impact convergence quality.

Our current evaluation focuses on convolutional models for vision workloads. Although Tula is architecturally model-agnostic, extending it to other domains like large language models (LLMs) and graph neural networks (GNNs) introduces additional challenges, including different scaling and generalization characteristics. In particular, memory estimation for LLMs needs to account for factors beyond batch-size, such as sequence length, transformer depth, and hidden dimensions. Additionally, layer heterogeneity in LLMs can exhibit varying gradient statistics. For e.g., attention layers can have sharp curvature, layernorm parameters have smaller but sensitive gradients, while embedding layer can have highly sparse gradients [42]. Thus, it may not be intuitive to apply gradient scaling uniformly across all model layers in this case, as it may disturb layernorm statistics or over-amplify noisy embedding gradients. Further, AdamW optimizer (popularly used in LLM training) combined with gradient scaling, would effectively modify the diagonal preconditioner and may lead to unexpected behavior.

On the other hand, gradient noise in GNN like architectures is graph structure dependent, so performing gradient scaling in a static way may amplify structural bias and degrade convergence quality. To address these challenges, we are exploring more expressive gradient-scaling strategies by leveraging auxiliary deep models (e.g., transformers or autoencoders), effectively acting as a knowledge distillation process (mapping large-to-small batch gradient space), similar to recent works on learning-based optimization [40], [41].

VII. Conclusion

Training neural networks efficiently on distributed resources requires carefully balancing parallel efficiency, resource cost, and statistical performance. These factors interact in complex and often non-obvious ways, making naïvely choosing batch-size and cluster-size configuration sub-optimal. Tula addresses this challenge by providing an online, profiling-driven service that models training performance across different configurations, identifies optimal operating points, and improves convergence quality in large-batch regimes.

Our evaluation shows that Tula's parallel performance model accurately captures training time and cost trends, enabling it to identify the batch-size that best satisfies user-specified objective such as minimal training time or cost. A more balanced and practical approach is to identify the batch-size corresponding to the knee-point of the pareto frontier. Beyond this point, increasing the batch-size yields diminishing returns in efficiency while exacerbating the generalization gap. By selecting a batch-size at or near this knee-point, Tula avoids unnecessary resource usage and degraded model quality. Since the chosen batch-size can be prone to generalization gap relative to smaller training batches, the proposed AGS technique improves final model quality over vanilla large-batch training.

Across multiple vision workloads and convolutional models, batch configurations selected by Tula achieves substantial performance gains over naïvely chosen settings, with up to 16× speedup observed in ResNet50, 20× in VGG11, 8× in AlexNet and 7.7× in MobileNetv3. In addition, Tula's adaptive gradient scaling (AGS) consistently mitigates the generalization gap associated with large-batch training, improving final test accuracy by an average of 8.8% compared to vanilla large-batch baselines for the same effective batch-size.

Acknowledgments

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] Kaplan, Jared, Sam McCandlish, T. J. Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeff Wu and Dario Amodei. "Scaling Laws for Neural Language Models." ArXiv abs/2001.08361 (2020).

[2] Tyagi, Sahil and Martin Swany. "Flexible Communication for Optimal Distributed Learning over Unpredictable Networks." 2023 IEEE International Conference on Big Data (BigData) (2023): 925-935.

[3] Süzen, Ahmet Ali, Burhan Duman and Betül Şen. "Benchmark Analysis of Jetson TX2, Jetson Nano and Raspberry PI using Deep-CNN." 2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA) (2020): 1-5.

[4] Qiao, Aurick, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger and Eric P. Xing. "Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning." 2021 USENIX Operating Systems Design and Principles (OSDI) (2021).

[5] McCandlish, Sam, Jared Kaplan, Dario Amodei and OpenAI Dota Team. "An Empirical Model of Large-Batch Training." ArXiv abs/1812.06162 (2018).

[6] Mai, Luo, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete and Peter R. Pietzuch. "KungFu: Making Training in Distributed Machine Learning Adaptive." USENIX Symposium on Operating Systems Design and Implementation (2020).

[7] Goyal, Priya, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia and Kaiming He. "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour." ArXiv abs/1706.02677 (2017).

[8] Keskar, Nitish Shirish, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy and Ping Tak Peter Tang. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima." 2017 International Conference on Learning Representations (ICLR) (2017).

[9] You, Yang, Igor Gitman and Boris Ginsburg. "Large Batch Training of Convolutional Networks." arXiv: Computer Vision and Pattern Recognition (2017).

[10] Tyagi, S., and 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.

[11] Zhang, Zhen, Chaokun Chang, Haibin Lin, Yida Wang, R. Arora and Xin Jin. "Is Network the Bottleneck of Distributed Training?" Proceedings of the Workshop on Network Meets AI & ML (2020).

[12] He, Kaiming, X. Zhang, Shaoqing Ren and Jian Sun. "Deep Residual Learning for Image Recognition." 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2015): 770-778.

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

[14] Krizhevsky, Alex, et al. "ImageNet classification with deep convolutional neural networks." Communications of the ACM 60 (2012): 84 - 90.

[15] Howard, Andrew G. et al. "Searching for MobileNetV3." 2019 IEEE/CVF International Conference on Computer Vision (ICCV) (2019): 1314-1324.

[16] Tyagi, S., and 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.

[17] Tang, Zhenheng, Shaohuai Shi, Xiaowen Chu, Wei Wang and Bo Li. "Communication-Efficient Distributed Deep Learning: A Comprehensive Survey." ArXiv abs/2003.06307 (2020).

[18] Hoffer, Elad, Itay Hubara and Daniel Soudry. "Train longer, generalize better: closing the generalization gap in large batch training of neural networks." Neural Information Processing Systems (NeurIPS) (2017).

[19] Golmant, Noah, Nikita Vemuri, Zhewei Yao, Vladimir Feinberg, Amir Gholami, Kai Rothauge, Michael W. Mahoney and Joseph E. Gonzalez. "On the Computational Inefficiency of Large Batch Sizes for Stochastic Gradient Descent." ArXiv abs/1811.12941 (2018).

[20] Yao, Zhewei, Amir Gholami, Qi Lei, Kurt Keutzer and Michael W. Mahoney. "Hessian-based Analysis of Large Batch Training and Robustness to Adversaries." 2018 Neural Information Processing Systems (NeurIPS) (2018).

[21] Frankle, Jonathan, David J. Schwab and Ari S. Morcos. "The Early Phase of Neural Network Training." 2020 International Conference on Learning Representations (ICLR) (2020).

[22] Achille, Alessandro, Matteo Rovere and Stefano Soatto. "Critical Learning Periods in Deep Neural Networks." ArXiv abs/1711.08856 (2017).

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

[24] Jastrzebski, Stanislaw, Zachary Kenton, Nicolas Ballas, Asja Fischer, Yoshua Bengio and Amos J. Storkey. "On the Relation Between the Sharpest Directions of DNN Loss and the SGD Step Length." arXiv: Machine Learning (2018).

[25] Agarwal, Saurabh, Hongyi Wang, Kangwook Lee, Shivaram Venkataraman and Dimitris Papailiopoulos. "Adaptive Gradient Communication via Critical Learning Regime Identification." Conference on Machine Learning and Systems (2021).

[26] Tyagi, S., Cozma, A., Kotevska, O., and Wang, F. (2025). OmniFed: A Modular Framework for Configurable Federated Learning from Edge to HPC. SC25-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, 516-523.

[27] Tyagi, Sahil and Prateek Sharma. "OmniLearn: A Framework for Distributed Deep Learning Over Heterogeneous Clusters." IEEE Transactions on Parallel and Distributed Systems 36 (2025): 1253-1267.

[28] Rajbhandari, Samyam, Jeff Rasley, Olatunji Ruwase and Yuxiong He. "ZeRO: Memory optimizations Toward Training Trillion Parameter Models." SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (2019): 1-16.

[29] Wen, Yeming, Kevin Luk, Maxime Gazeau, Guodong Zhang, Harris Chan and Jimmy Ba. "An Empirical Study of Stochastic Gradient Descent with Structured Covariance Noise." 2020 International Conference on Artificial Intelligence and Statistics (AISTATS) (2020).

[30] Smith, Samuel L., Pieter-Jan Kindermans and Quoc V. Le. "Don't Decay the Learning Rate, Increase the Batch Size." International Conference on Learning Representations (ICLR) (2017).

[31] You, Yang, Jonathan Hseu, Chris Ying, James Demmel, Kurt Keutzer and Cho-Jui Hsieh. "Large-Batch Training for LSTM and Beyond." SC19: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (2019): 1-16.

[32] Satopaa, Ville A., Jeannie R. Albrecht, David E. Irwin and Barath Raghavan. "Finding a "Kneedle" in a Haystack: Detecting Knee Points in System Behavior." 2011 31st International Conference on Distributed Computing Systems Workshops (2011): 166-171.

[33] Lin, Tao, Sebastian U. Stich and Martin Jaggi. "Don't Use Large Mini-Batches, Use Local SGD." 2018 International Conference on Learning Representations (ICLR) (2018).

[34] Mayer, Ruben and Hans-Arno Jacobsen. "Scalable Deep Learning on Distributed Infrastructures." ACM Computing Surveys (CSUR) 53 (2019): 1-37.

[35] Johnson, Tyler B., et al. "AdaScale SGD: A User-Friendly Algorithm for Distributed Training." 2020 International Conference on Machine Learning (ICML) (2020).

[36] Smith, Samuel L. and Quoc V. Le. "A Bayesian Perspective on Generalization and Stochastic Gradient Descent." 2017 International Conference on Learning Representations (ICLR) (2017).

[37] Lin, Tao, Lingjing Kong, Sebastian U. Stich and Martin Jaggi. "Extrapolation for Large-batch Training in Deep Learning." 2020 International Conference on Machine Learning (ICML) (2020).

[38] You, Yang et al. "Large Batch Optimization for Deep Learning: Training BERT in 76 minutes." 2020 International Conference on Learning Representations (ICLR).

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

[40] Abrahamyan, Lusine et al. "Learned Gradient Compression for Distributed Deep Learning." IEEE Transactions on Neural Networks and Learning Systems 33 (2021): 7330-7344.

[41] Calian, Dan Andrei et al. "DataRater: Meta-Learned Dataset Curation." ArXiv abs/2505.17895 (2025).

[42] Spring, R., Kyrillidis, A., Mohan, V., and Shrivastava, A. (2019). Compressing Gradient Optimizers via Count-Sketches. 2019 International Conference on Machine Learning (ICML).

[43] Tyagi, S., and Swany, M. (2022). ScaDLES: Scalable Deep Learning over Streaming data at the Edge. 2022 IEEE International Conference on Big Data (Big Data), 2113-2122.