GraVAC: Adaptive Compression for Communication-Efficient Distributed DL Training

Sahil Tyagi
Indiana University Bloomington, USA
styagi@iu.edu

Martin Swany
Indiana University Bloomington, USA
swany@iu.edu

Abstract—**Distributed data-parallel (DDP) training improves overall application throughput as multiple devices train on a subset of data and aggregate updates to produce a globally shared model. The periodic synchronization at each iteration incurs considerable overhead, exacerbated by the increasing size and complexity of state-of-the-art neural networks. Although many gradient compression techniques propose to reduce communication cost, the ideal compression factor that leads to maximum speedup or minimum data exchange remains an open-ended problem since it varies with the quality of compression, model size and structure, hardware, network topology and bandwidth. We propose *GraVAC*, a framework to dynamically adjust compression factor throughout training by evaluating model progress and assessing gradient information loss associated with compression. *GraVAC* works in an online, black-box manner without any prior assumptions about a model or its hyperparameters, while achieving the same or better accuracy than dense SGD (i.e., no compression) in the same number of iterations/epochs. As opposed to using a static compression factor, *GraVAC* reduces end-to-end training time for ResNet101, VGG16 and LSTM by 4.32×, 1.95× and 6.67× respectively. Compared to other adaptive schemes, our framework provides 1.94× to 5.63× overall speedup.**

Index Terms—**deep learning, data-parallel training, gradient compression, sparsification, adaptive systems**

I. Introduction

Deep Learning (DL) is a supervised machine learning approach that optimizes a loss function over a non-convex surface by comparing model predictions with ground truth. Each training iteration in DL involves forward and backward pass, i.e., generate predictions from input data, assess loss, compute gradients and update model parameters via optimization method like gradient descent. Training is an iterative process, typically involving multiple passes over the entire dataset where each pass is called an *epoch*. DL is also heavily influenced by certain *hyperparameters* that affect training speed, quality, or both. Commonly used hyperparameters are learning rate, momentum, batch size, weight decay, epochs, activation function, etc.

Distributed data-parallel (DDP) methods further scale training across multiple nodes that train a globally shared model with I.I.D. data (independent and identically distributed) by periodically aggregating locally computed gradients at the end of each iteration. The compute requirements to train DL models doubles every 3.5 months [1], while the compute gains in chip design for ML accelerators and bandwidth gains in telecommunications networks double every 24 and 18 months [2], [3]. Thus, the infrastructure required to train state-of-the-art models tends to fall behind their compute and networking demands. Since upgrading network stack in the cloud, datacenter and HPC clusters can be infrequent as compared to appending new accelerators in pre-existing systems, gradient communication tends to be the major bottleneck in distributed training [4].

Different compression techniques have been proposed in recent years to mitigate this synchronization overhead. However, the optimal compression factor (CF) that minimizes data exchange or end-to-end training time depends on the model itself (i.e., its size, structure and depth), available network bandwidth and the compression overhead itself. Unlike traditional HPC and distributed computing applications that only measure parallel efficiency, DDP training has an additional statistical efficiency associated with it. Although the amount of computation performed on each iteration is the same, some iterations tend to be more crucial than others towards the overall learning of the model. Updates are especially sensitive in early stages and to hyperparameters like learning rate schedule, momentum and weight decay [5]. It would thus be intuitive to compare information loss in gradients on account of compression, and use a lower CF when considerably more information is lost and a higher CF when most information is preserved under compression. We can subsequently increase compression as training continues and gradients saturate, and decrease it back during the aforementioned critical stages.

We take into account the parallel and statistical efficiency aspect of gradient compression in this work: a high CF improves overall throughput (i.e., number of samples processed per second) by reducing communication cost, but increases information loss in the gradients resulting in either slower or insignificant updates. The two metrics in DDP compression are pareto-related as one improves at the detriment of the other. We propose ***GraVAC***: $\{\textbf{Gra}\}$dient $\{\textbf{V}\}$ariance-based $\{\textbf{A}\}$daptive $\{\textbf{C}\}$ompression [1] to dynamically adjust CF by comparing information loss from compression with that of the original gradients computed in backpropagation. *GraVAC* evaluates different CFs in a given search space and determines the CF that best balances parallel and statistical efficiency in DDP training with compression. We validate our approach over a variety of DL models and directly compare with static CF on compressors like Top-k [6], Deep Gradient Compression or DGC [7], Redsync [9] and Random-k [6].

[1] Code available at https://github.com/sahiltyagi4/GraVAC

TABLE 1
DL MODEL DESCRIPTION

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

II. Background and Related Work

DDP training can be implemented either via MPI-based collectives (AllReduce) [10]–[12] or using one or more centralized parameter servers (PS) [13] to accumulate and distribute model updates among workers.

A. Scaling Efficiency of DDP Training

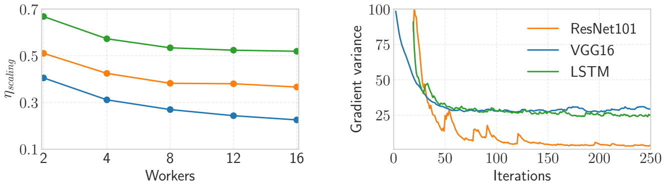
DL training is an iterative process that involves parameter updates at each step via gradient descent (GD) [14]. Full GD uses entire training data at every step, making the whole process slow and compute-intensive, while Stochastic GD processes a single sample and does not vectorize multiple samples on fast accelerators. Mini-batch GD is the optimal middle ground between Full and Stochastic GD where b samples are randomly sampled from I.I.D. data. Eqn. (1) describes the update rule in mini-batch GD where parameters w at $(i+1)$-*th* iteration on N workers minimize loss function $\mathcal{L}(\cdot)$ on input samples x_j of size b from distribution $\mathcal{X}_j$ with learning rate η. With weak scaling, we can increase the amount of per-iteration work by adding more workers and keeping per-worker batch-size b the same.

$$w_{i+1} = w_i - \eta \frac{1}{N} \sum_{n=1}^{n=N} \frac{1}{|b|} \sum_{j \in b} \frac{\partial}{\partial w_i} \mathcal{L}(x_{(j,n)}, w_i) \qquad (1)$$

The *ideal* throughput of a distributed application T_N executed across N workers is N times the throughput of a single worker T_1. The deviation is measured via "scaling efficiency" in Eqn. 2a. Assuming negligible IO overhead, iteration time in dense SGD is bounded by computation and communication time (Eqn. (2b)). It may be possible to overlap communication with computation, but only partially since the latter is comparatively much lower on modern GPUs and TPUs. Model communication has been shown to be an order of hundreds or even thousands of magnitudes higher than gradient computation. Thus, frequent synchronization (t_{sync}) is the bottleneck that halts linear scaling in DDP. Table 1 describes the size, density and convergence target of ResNet101 [15], VGG16 [16] and LSTM [17] with dense SGD communication. Latency is further exacerbated on constrained networks with limited bandwidth as large volumes of data is exchanged by multiple workers simultaneously.

$$\eta_{scaling} = T_N / N \cdot T_1 \qquad (2a)$$

$$t_{iter} \approx t_{compute} + t_{sync} \qquad (2b)$$



(a) Scaling efficiency in DDP

(b) Initial gradient sensitivity

Fig. 1. Communication overhead and early critical period in DDP training.

For a DL model with a total of M parameters, the time cost based on the α-β communication model (where α is the latency and β is the inverse of bandwidth) for tree-based allreduce is $(2\alpha logN + 2MlogN\beta)$ [18]. For ring-based allreduce, this becomes $2(N-1)\alpha + 2M\beta(N-1)/N$. Hence, communication cost increases as more workers are added to the mix in distributed training. Fig. 1a shows how overall throughput deviates from the ideal as cluster-size increases. The scaling efficiency is also influenced by the message size, i.e., total gradients/parameters to be communicated. In dense SGD, we observed scaling to be affected by the tensor-size distributions across the layers of a model as well. For e.g., LSTM has a better $\eta_{scaling}$ than ResNet101 despite being a larger model. This is because parameters in LSTM are spread across just 2 layers, compared to 101 in ResNet101.

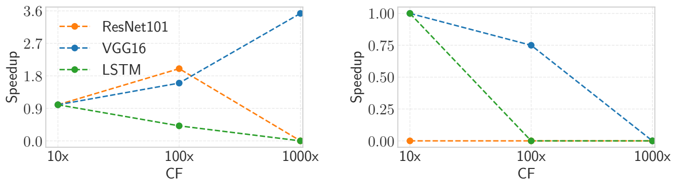
B. Gradient Variance in Deep Learning

Prior work has demonstrated that gradient information can help measure the statistical efficiency of distributed training [19], [20]. There is a strong correlation between changes in the eigen values of second-order hessian [21] and first-order gradients (i.e., variance). [22], [23] explores how gradients behave in early stages of DL training and during certain critical periods, influenced by hyperparameters like learning rate schedule, gradient clipping and type of SGD used (e.g., zero, first or second-order moments). Fig. 1b attests those findings where we plot variance over the starting iterations and notice how drastically the gradients change and saturate over training.

C. Gradient Compression

Many lossy compression techniques have been proposed for DDP and federated learning in recent years. Lossy compression incurs a fundamental trade-off between data-size and information loss; one can either reduce message size by losing more information, or preserve data quality by keeping majority of the original bits intact. In the context of DDP, higher CF reduces communication time at the cost of accuracy degradation or more steps/epochs required for the same convergence. CF measures the size of original gradients to the size of compressed tensors. E.g., compressing 10% gradients gives CF of 10x, while 1% gives 100x. Lossy compression can be broadly classified into *quantization*, *sparsification* or *low-rank approximations*.

The bit-width of single-precision (32-bit) floats is reduced in gradient quantization. Techniques like automatic mixed



(a) Top-*k*

(b) Random-*k*

Fig. 2. CF with maximal speedup (to reach Table 1 targets) varies for each model and compression technique used. The results are normalized by 10x CF while a speedup of 0.0 implies convergence failure.

precision (AMP) [24] reduces gradients to half-precision, resulting in 2x CF. QSGD [25] balances the trade-off between accuracy and quantization precision. 1-bit SGD [26] reduces 32-bit floats to 1-bit and propagates quantization error via error-feedback. Sparsification methods communicate only a fraction of the gradient values along with their indices and set everything else to 0. Top-*k* sparisifies by extracting the top *k*% values while Random-*k* does so randomly with negligible compression overhead. DGC discards gradients below a certain threshold along with using momentum correction and gradient clipping. Methods like Redsync [40] combine quantization and sparsification, but the estimation quality is not accurate [27]. Approaches like PowerSGD [28] and Pufferfish [29] achieve compression via low-rank updates. The former can be viewed as adding regularization in DL, while the latter performs low-rank factorization on fully connected, convolutional and LSTM layers.

What should be the ideal CF in Compression-based DDP?

The ideal CF is one that reduces communication time without trimming too much gradients which can be detrimental to final model. Compression has its own associated costs depending on the target CF and computational complexity of the mechanism itself. These factors affect both the parallel efficiency of distributed training as well as statistical inefficiency due to information loss from compression. Fig. 2 aptly demonstrates this where the CF that gives maximum speedup varies for each model and compression technique employed. The models are trained to Table 1 targets. ResNet101 on Top-*k* achieves most speedup at 100x, while VGG16 and LSTM peak at CFs 1000x and 10x respectively. On the other hand, ResNet101 fails to converge for any CF with Random-*k* compression. VGG16 and LSTM converged with 10x and failed with other CFs. Although a typical ML practitioner may not necessarily need to think about a plethora of compression methods, choosing the right CF with any compressor and DL model that minimizes training time, or even converges successfully, presents a non-trivial challenge.

Dynamic compression mechanisms like AdaQS [30] perform quantization using gradient mean to standard deviation ratio (MSDR). Systems like Accordion [31] and ScaDLES [32] switch between low and high compression based on critical regime identification. We tackle the ideal CF exploration problem in *GraVAC* in a gradient-driven manner by comparing variance of prior and post-compression gradients. For clarity, prior-compression gradients refer to the original tensors computed in backward pass. By measuring the information lost in compression, we dynamically adjust CF over each iteration. Starting with a low CF initially, we gradually increase compression as training progresses. On encountering senstive or critical regions, *GraVAC* switches to a lower CF that least degrades convergence.

III. Design and implementation

In this section, we first describe the trade-off between parallel and statistical efficiency of DDP training *with* compression. Then we describe the metrics “compression gain“ and “compression throughput“ to combine the two, and explain *GraVAC*'s adaptive compression algorithm.

A. Parallel Efficiency of Gradient Compression

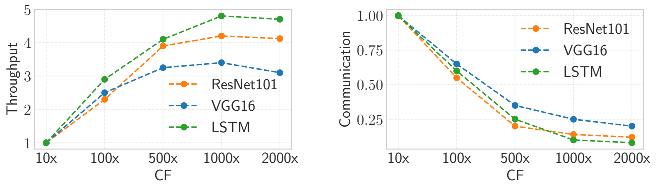
The end goal of gradient compression is to improve DDP scaling efficiency. Application scaling is governed by the DDP mechanism (ring-based, tree-based allreduce or parameter servers), communication library used (MPI, NCCL [11], Gloo [10] or RPC) and available bandwidth. Keeping the latter and network infrastructure aside, speedup in any DL model depends on the target CF, quality of estimation and compression overhead. The overall iteration time in Eqn. 2b is adjusted for compression as

$$t_{iter}^{(c)} \approx t_{compute} + t_{sync}^{(c)} + t_{compress}^{(c)} + t_{decompress}^{(c)}$$

where it takes $t_{compress}^{(c)}$ time to reduce gradients to CF c such that it reduces communication time to $t_{sync}^{(c)}$. $t_{decompress}^{(c)}$ is the time taken to reconstruct the compressed gradients to the same dimension as the original gradients. A viable compressor must have its compression time considerably lower than synchronization time.

The parallel efficiency of a distributed application suffers with more workers due to higher synchronization costs. Improving the network bandwidth alleviates this to only a certain extent. [4] investigates how DDP throughput improves marginally with higher bandwidth. They observed that ResNet50 peaks to 75% scale-out on a 25 Gbps network and remains the same even for 100 Gbps. Its because network transport implementation of current DL frameworks cannot fully utilize the available network bandwidth. Thus, even though cloud providers like GCP provide anywhere from 10-32 Gbps bandwidth depending on the machine type and VM size, they may not be utilized to their full potential.

Fig. 3 shows how the throughput increases and communication overhead reduces with compression. The results are relative to CF 10x for each model. We perform layerwise DGC compression over a 32 GPU cluster. System throughput is determined only by compression overhead and communication time as the compute time in backpropagation stays the same across all CFs. Based on the compressor used, compression latency may vary with target CF. For e.g., it decreases with larger CF as Top-*k* uses max-heap and sorts the top *k*% elements



(a) Relative throughput

(b) Relative communication

Fig. 3. Throughput and communication speedup for layerwise DGC compression, normalized by 10x CF.

in $O(N+k\log k)$ time. Throughput for ResNet101 and VGG16 saturates at 500x and does not improve thereafter, while LSTM saturates at 1000x (Fig. 3a). Communication savings also diminish at higher CFs due to small message size and network saturation (Fig. 3b). Thus, the highest CF may not necessarily correspond to the largest throughput.

B. Statistical Inefficiency of Gradient Compression

Gradient compression mechanisms rely on *error-feedback* [35], [36] which essentially acts as delayed updates, as commonly noted in asynchronous training. The gradients ineligible for compression in the current iteration are not discarded, but added to *residual gradients* which in turn are added to gradients computed in the next iteration. Residual gradients and error-feedback helps preserve important features and is critical to convergence [6]–[8]. Applying compression without error-feedback has been shown to achieve lower accuracy in deep learning models [35]. At the same time, residual gradients can sometimes degrade generalization performance due to stale updates.

DDP training with very high CFs can negatively impact training time, convergence quality, or both if the compressed gradients are too sparse or quantized to update the model in any significant way. *It is thus crucial to have an indicator that quantifies information loss between compressed and the original gradients.* We do so by comparing variance between the original and compressed tensors on every iteration and see how it relates to actual model convergence. Denoting the original gradients as *BC* (*Before-Compression*) and compressed tensors as *AC* (*After-Compression*), we compare BC and AC tensors in two separate configurations with CFs 10x and 1000x in Fig. 4, 5 and 6. We compare the convergence curves for the two CFs with *Dense SGD* (i.e., no compression) to see how much accuracy degrades with compression.

AC 10x is nearly identical to its *BC* counterpart in ResNet101 (Fig. 4a) while there is considerably more information loss in between *BC* and *AC* 1000x (Fig. 4b). This translates to their convergence curves in Fig. 4c as well where 10x and dense SGD runs follow a similar convergence trajectory while 1000x achieves considerably lower accuracy for the same iterations.

VGG16 follows a similar trend with 10x CF. The *BC* and *AC* gradient variance (Fig. 5a) is nearly identical and so are the convergence curves for 10x and Dense SGD (Fig. 5c). We notice a slight deviation between *BC* and *AC* at 1000x initially in Fig. 5b, which correlates to slow convergence in the early iterations for 1000x in Fig. 5c. As the deviation *BC* and *AC* decreases, we see both CFs converge to the same accuracy as Dense SGD in the same iterations.

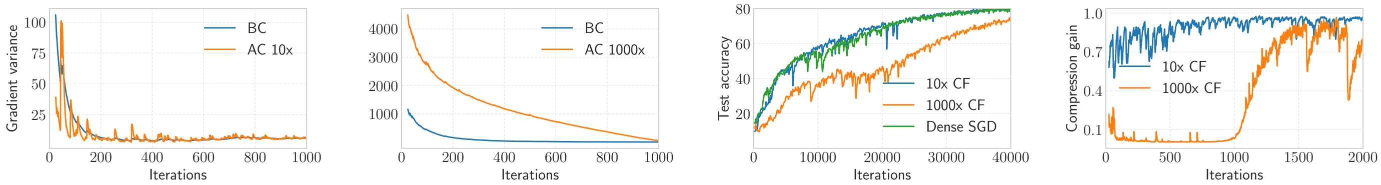
The *AC* 10x and 1000x gradients lie on similar scales as *BC* in LSTM, although the higher CF has slightly higher variance (Fig. 6a and 5b). As seen from Fig. 5c, Dense SGD has the least perplexity (thus, better model quality), followed by 10x and 1000x CFs.

To compare the information loss between the original and gradients compressed to CF c, we define a simplistic metric called *Compression gain*. As part of error feedback, we update the gradients such that $g_{ef}^{(i)} = g_0^{(i)} + \mathsf{residual\_gradients}^{(i-1)}$ for $i \geq 1$. Here, $g_0^{(i)}$ are the original gradients calculated via backpropagation at iteration i, while $\mathsf{residual\_gradients}^{(i-1)}$ are left-overs from the last iteration $(i-1)$ and before, which are added back as part of error-feedback to produce $g_{ef}^{(i)}$ for the current iteration. With compression operator $\mathcal{C}$, gradients are compressed as $g_c^{(i)} = \mathcal{C}[g_{ef}^{(i)}]$. Compression gain is then measured as the ratio of expected variance of compressed gradients $g_c^{(i)}$ and the original gradients modified with error-feedback, i.e., $g_{ef}^{(i)}$:

$$\mathsf{Compression\ gain} = \frac{\mathbb{E}[||g_c^{(i)}||^2]}{\mathbb{E}[||g_{ef}^{(i)}||^2]}$$

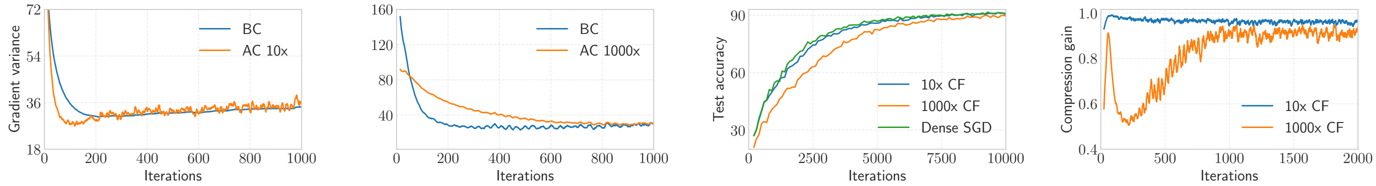
In prior work, gradient noise has been well studied in deep learning literature pertaining to divergence between locally-computed and aggregated gradients in DDP [20], [37], [38]. These works use gradient information to tweak the global batch-size in DDP to optimize job completion time or allocate optimal resources for a job. Instead of looking at local and global gradients, *GraVAC*'s novelty comes from evaluating the noise between the original and compressed tensors. The gradients computed over each iteration can be noisy. Thus, we keep a moving average of the respective variances of the original and compressed gradients. The computation and memory footprint of this approach is low since the window-size in moving average is finite and only a single-precision floating point is stored for every iteration. Compression gain is bounded between $\{0, 1]$ such that it is low when $\mathcal{C}$ trims too much information. As models keep training, gradients saturate and higher compression becomes more viable in later stages of training. Hence, compression gain increases over training as compressed tensors become more aligned with the original gradients.

We plot compression gains for the three models when training with fixed CF 10x and 1000x respectively, shown in Fig. 4d, 5d and 6d. In each model, 10x has higher compression gain than 1000x since more information is preserved in the smaller CF. *It should also be apparent that Dense SGD training has a constant gain of 1.0.* For all models, convergence curve of 10x follows a similar trajectory as Dense SGD. Correspondingly, the compression gain of 10x stays close to 1.0 throughout. In ResNet101, gain of 1000x is low initially and grows in an oscillating manner, although still lower than gains of 10x and



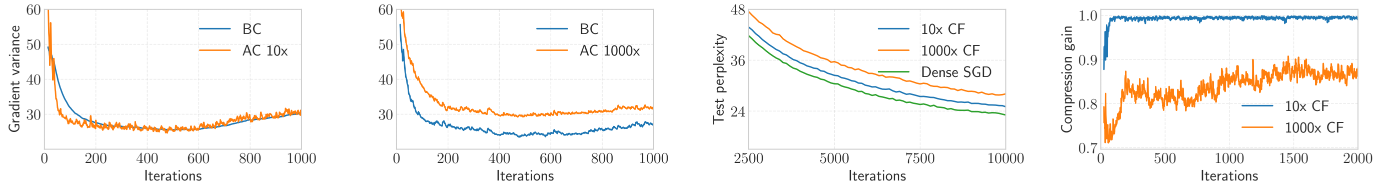
(a) 10x CF (b) 1000x CF (c) Convergence curve (d) Compression gain

Fig. 4. ResNet101: Prior and Post-Compression gradients, test accuracy and compression gain for CFs 10x and 1000x.



(a) 10x CF (b) 1000x CF (c) Convergence curve (d) Compression gain

Fig. 5. VGG16: Prior and Post-Compression gradients, test accuracy and compression gain for CFs 10x and 1000x.



(a) 10x CF (b) 1000x CF (c) Convergence curve (d) Compression gain

Fig. 6. LSTM: Prior and Post-Compression gradients, test perplexity (lower is better) and compression gain for CFs 10x and 1000x.

Dense SGD. The low gains in the first 1000 iterations of CF 1000x correlates to the considerable gap between *BC* and *AC* gradients in Fig. 4b and lower accuracy in Fig. 4c. VGG16 is more robust to higher CFs (Fig. 5c), as also seen from the high compression gains of CF 1000x in Fig. 5d. For LSTM, compression gain for 10x stays close to 1.0 and between 0.8-0.9 for 1000x. The proximity of the two CFs to Dense SGD's gain of 1.0 is equivalent to their perplexity curves in Fig. 6c. From these results we see how compression gain serves as a viable indicator of the statistical efficiency of DDP with compression.

C. Combining System Throughput and Compression Gain

As described earlier in II-C as well as Fig. 2, choosing a high CF unintuitively does not necessarily improve training time and may even degrade final model quality. Thus, to account for both the parallel and statistical efficiency DDP training *with* gradient compression, we combine *system throughput* (T_{system}) and *compression gain* into a single metric called *Compression Throughput*:

$$\text{T}_{compression} = \text{T}_{system} \times \text{Compression gain}$$

If CF is high, system throughput would be high as well but compression gain would relatively be lower, decreasing the resulting $\text{T}_{compression}$. On the other hand, compression gain will be high for a low CF, but system throughput will be lower due to relatively higher communication overhead. *With Compression Throughput, we capture this pareto-relationship between the parallel (system throughput) and statistical efficiency (compression gain) of gradient compression in DDP.*

We build *GraVAC* as a modular extension on top of PyTorch's [33] DDP module [34] using Python in about 3000 lines of code. A base GravacOptimizer wraps common SGD optimizers implemented in PyTorch by extending the base torch.optim.Optimizer class. The optimizer takes an additional Compressor object that specifies the type of compression technique used. We implement four pre-existing techniques as compressor classes in this paper: Top-*k*, DGC, Redsync and Random-*k*. Compression for the appropriate CF and its gain is computed before the optimizer step function which applies the aggregated gradient updates on model parameters.

GraVAC Algorithm: Alg. 1 describes *GraVAC*'s approach of using compressor $\mathcal{C}$ to scale CFs in the exploration space $[\theta_{min}, \theta_{max}]$, where each candidate CF is evaluated for window steps and incremented in step-size of θ_s w.r.t. θ_{min}. For e.g., scaling from CF 10x to CF 20x means $\theta_s = 20/10 = 2$x. The threshold ϵ denotes the minimum compression gain required

Algorithm 1: *GraVAC*'s Adaptive Compression

```
1  Input: $\theta_{min}$, $\theta_{max}$, $\epsilon$, $\theta_s$, $\omega$, window, compressor $\mathcal{C}$
2  $w_o$ : initial model state, N: total nodes, b: per-worker
     batch-size, residual = 0; $\mathrm{T}_{sys}, \mathrm{T}_{compress}$ = empty()
3  Train for $i$ = 1,2,3...                                  ▷ training iterations
4      $g_o^{(i)}, t_o = \nabla f(x^{(i)}, w_i)$                         ▷ backpropagation
5      $g_o^{(i)} = g_o^{(i)} +$ residual                           ▷ error-feedback
6      $g_{min}^{(i)}, t_{min} = \mathcal{C}(g_o^{(i)}, \theta_{min})$   ▷ compress to CF $\theta_{min}$
7      $\delta_{min} = \mathrm{EWMA}(\frac{||g_{min}^{(i)}||^2}{||g_o^{(i)}||^2})$   ▷ $\theta_{min}$ compression gain
8      $g_c^{(i)}, t_c^{(i)} = \mathcal{C}(g_{min}^{(i)}, \theta_s)$   ▷ compress to CF $(\theta_s \cdot \theta_{min})$
9      $\delta_c = \mathrm{EWMA}(\frac{||g_c^{(i)}||^2}{||g_o^{(i)}||^2})$        ▷ gain for CF $(\theta_s \cdot \theta_{min})$
10     $t_{compress} = t_{min} + t_c$               ▷ total compression time
11     if $\delta_c \geq \epsilon$ :
12         $\tilde{g}^{(i)}$, $t_s$ = Aggregate($g_c^{(i)}$)       ▷ synchronize $g_c^{(i)}$
13         residual = $g_o^{(i)} - g_c^{(i)}$                   ▷ update residual
14         $t_{iter}$ = $t_o$ + $t_{compress}$ + $t_s$              ▷ iteration time
15         UpdateStep($\theta_s \cdot \theta_{min}, \delta_c, t_{iter}$)
16     else if $\delta_c < \epsilon$ and $\delta_{min} \geq \epsilon$ :
17         $\tilde{g}^{(i)}$, $t_s$ = Aggregate($g_{min}^{(i)}$)   ▷ synchronize $g_{min}^{(i)}$
18         residual = $g_o^{(i)} - g_{min}^{(i)}$               ▷ update residuals
19         $t_{iter}$ = $t_o$ + $t_{compress}$ + $t_s$              ▷ iteration time
20         UpdateStep($\theta_{min}, \delta_{min}, t_{iter}$)
21     else
22         $\tilde{g}^{(i)}$, $t_s$ = Aggregate($g_o^{(i)}$)       ▷ synchronize $g_o^{(i)}$
23         residual = 0                        ▷ no residual gradients
24         $t_{iter}$ = $t_o$ + $t_s$                             ▷ iteration time
25         UpdateStep($1, 1, t_{iter}$)
26     $w_{i+1} = w_i - \eta \cdot \tilde{g}^{(i)}$                   ▷ apply SGD update
27     $\theta_s$ = CheckGraVAC($i$, $\theta_s, \delta_{min}, \delta_c$)

28 procedure UpdateStep($\theta, \delta, t_{iter}$):
29     $\mathrm{T}_{sys}$ = N $\cdot$ b/$t_{iter}$                    ▷ system throughput
30     $\mathrm{T}_{compress}[\theta] = \mathrm{T}_{sys} \cdot \delta$   ▷ compression throughput

31 procedure CheckGraVAC($i$, $\theta_s, \delta_{min}, \delta_c$):
32     if $i$ % window == 0 :
33         $\theta_s$ = ScalingPolicy($\theta_s$)  ▷ compression scale-up
34         if $\omega \geq \frac{|\delta_{min} - \delta_c|}{\delta_{min}}$ :
35             $\theta_{min} = \theta_s \cdot \theta_{min}$      ▷ scale-up minimum CF
36         ct = sort($\mathrm{T}_{compress}$.values())   ▷ $\mathrm{T}_{compress}$ vals
37         if $|\frac{ct[-1] - ct[-2]}{ct[-2]}| \leq \omega$ :
38             $\theta_{ideal} = \mathrm{T}_{compress}.get(\mathrm{ct}[-2])$   ▷ ideal CF
39             return $\theta_{ideal}/\theta_{min}$                 ▷ gives optimal $\theta_s$
40         else
41             return $\theta_s$            ▷ else use old scaling factor
```

for any CF to be eligible for communication in *GraVAC*, while threshold ω is used to measure saturation in compression throughputs and for scaling up θ_{min}. We explain this in the following sections in more detail. For every iteration, we compute gradients $g_o^{(i)}$ with model parameters w_i on training sample $x^{(i)}$ in time t_o (line 4). To incorporate error-feedback,residual holds the leftover gradients not communicated from previous iterations. The shape and memory size of tensors in residual is the same as gradients itself. As shown in line 5, we add residual gradients to the gradients computed in the current iteration. In the first stage, we compress original gradients using $\mathcal{C}$ to compressed gradients $g_{min}^{(i)}$ corresponding to minimum CF θ_{min} (line 6). We then compute the compression gain corresponding to θ_{min} (line 7), and smoothen out the inter-iteration gain through exponential weighted moving average (EWMA) smoothing. In our evaluation, we set the EWMA smoothing factor to $N/100$, where N is the number of participating workers. We evaluate the next candidate CF by stepping up the previous θ_{min} and further compressing the already compressed gradients $g_{min}^{(i)}$ by stepsize θ_s (line 8). Thus, candidate CF evaluated in this case is $\theta_s \cdot \theta_{min}$. This is done as part of our multi-level compression strategy to avoid compressing the large, original tensors $g_o^{(i)}$ twice. We measure the time savings of our multi-level approach in section IV-C.

Next, we compute the gradients and compression gain of candidate CF $\theta_s \cdot \theta_{min}$ (line 8-9), and denote the total compression time $t_{compress}$ as the sum of time to compress original gradients to $g_{min}^{(i)}$ (line 6) and the time to further compress $g_{min}^{(i)}$ to $g_c^{(i)}$ (line 8). Based on the compression gains obtained and threshold ϵ, we choose the appropriate gradients to call the collective operation on. If the gain of our candidate CF meets ϵ (line 11), we go ahead and communicate compressed gradients $g_c^{(i)}$ among workers. We update the residual gradients in accord with $g_c^{(i)}$ as well (line 13), calculate the total iteration time (line 14) and update the system as well as compression throughput for CF $\theta_s \cdot \theta_{min}$ via UpdateStep function. $\mathrm{T}_{compress}$ is a dictionary or a hashmap that stores compression throughput of each candidate CF, min-max CF as well as dense SGD setting (i.e., CF 1x).

If the gain of $g_c^{(i)}$ does not meet the threshold, but gain δ_{min} of θ_{min} does (line 16), we instead synchronize compressed gradients $g_{min}^{(i)}$ corresponding to θ_{min}. In a similar fashion as before, we update the residuals, this time with $g_{min}^{(i)}$ instead of $g_c^{(i)}$ (line 18), compute iteration time and assess compression throughput. *It is important to remember that synchronization overhead to communicate $g_{min}^{(i)}$ is more than $g_c^{(i)}$ due to the former's lower CF. The trade-off we make in GraVAC is to incur higher communication latency for more accurate representation of the original gradients (measured by compression gain) and vice-versa.*

If both θ_{min} and currently evaluated CF do not meet the set threshold, we incur maximum communication latency by transmitting the original gradients via dense SGD (line 22). In this case, residual gradients are set to 0 and no compression overhead is included as part of iteration time and computing system/compression throughput. The CF and compression gain are both 1, as set in the UpdateStep function at line 25.

Following SGD update (line 26), we evaluate *GraVAC* to assess the performance of CFs evaluated so far. This happens at a frequency determined by window. Here, we adjust θ_s by a certain factor to scale up compression, determined by the chosen ScalingPolicy. The scaling policy tunes compression only until the upper bound θ_{max}. We explore two scaling policies in this paper that we describe in detail under section IV-B. After scaling θ_s, we also assess if the minimum CF, i.e., θ_{min} can be scaled up as well. The intuition is that as training progresses, model gradually starts converging as well and we can use higher compression even for the minimum CF later on. In addition to candidate CFs, we thus scale up the minimum CF as well. The transition is made if the current gain δ_c is within $\omega\%$ of the gain of previous θ_{min} (line 34). Once enough CFs are evaluated, we look at the two largest compression throughputs (line 36) and fetch the corresponding CF if they are within the bounds of ω. We do this as it means the compression throughput has saturated and thus, we pick the lower CF as θ_{ideal} (line 38) and send the appropriate step-size (line 39). If the threshold ω is not met, we use θ_s as is.

When does compression scale-up? As seen from Alg. 1, the compression scale-up happens during *GraVAC*'s evaluation phase where we scale the step-size θ_s in accordance with a specific scaling policy. At the same time, we escalate the minimum CF θ_{min} to currently evaluated CF if the two compression gains are within $\omega\%$ of each other.

When does compression scale-down? Compression scale-down is determined by ϵ (shown via conditional statements lines 11-25). If current CF loses considerably more information in compressed gradients $g_c^{(i)}$, we use the lower CF θ_{min}. If the latter fails to meet ϵ as well, we send uncompressed gradients $g_o^{(i)}$ as a last resort.

IV. Evaluation

A. Cluster Setup and Training Hyperparameters

We evaluate *GraVAC* on a 32 GPU setup on the Google Cloud Platform (GCP) across 8 VMs. Each VM is a n1-standard-8 machine type with 8 vCPUs, 30 GB system memory and 4 NVIDIA V100 GPUs with 16 GB VRAM each. The machines are configured with PyTorch 1.10.1, CUDA 11.3, CUDA driver 465.19.01 and NCCL 2.10.3.

We evaluate the three models described in Table 1. ResNet101 is trained with per-worker batch size 32, momentum 0.9, weight decay 0.0001 and SGD optimizer with initial learning rate (lr) 0.1 decayed by a factor of 10 at 9K and 14K iterations respectively. VGG16 is also trained with per-worker batch-size 32, weight decay 0.0005, momentum 0.9 and SGD with fixed lr 0.1. Lastly, LSTM is measured with test perplexity (i.e., exponential of test loss) with per-worker batch-size 20, momentum 0.9, weight decay 0.0001 and SGD with fixed lr 0.1. The model is initialized with 1500 embedding dimensions and 2 hidden layers with 35 bptt steps.

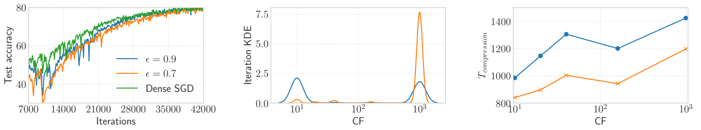
We evaluate *GraVAC* with different scaling policies and look at their convergence curves (i.e. test accuracy/perplexity vs. iterations), average compression throughput of candidate CFs and kernel density estimates (KDE) of training iterations using different CFs over the course of training. KDE gives the distribution over the iterations for all CFs and plotted on the log-scale with smoothing bandwidth of 0.1 passed to the gaussian KDE.

B. GraVAC*'s Adaptive Compression Policies*

In this section, we look at how *GraVAC* achieves optimal CF for a given θ_{min}, θ_{max}, ϵ, window, ω and stepsize. To see how a model converges and communication costs vary by evaluating different candidate CFs in the search space, we employ an *Exponential* policy that upscales CFs aggressively, and a relatively smoother *Geometric* scaling policy that scales CFs as a geometric progression.

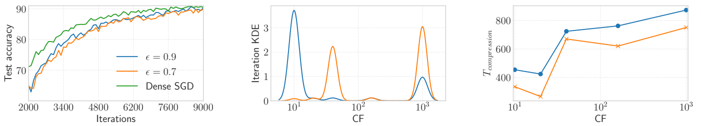
1) Exponential scaling policy: In this policy, we implement the ScalingPolicy function from Alg. 1 such that CFs are scaled up in exponents of 2 w.r.t the first initialized θ_{min}. On top of DGC, we set θ_{min} and θ_{max} to 10x and 1000x, window=500 and ω=1%. So we scale up by factors of 2^1, 2^2, 2^4, 2^8 w.r.t 10x up until 1000x. The candidate CFs thus evaluated in this policy are 10x, 20x, 40x, 160x and 1000x. We run *GraVAC* on two configuration with different thresholds on compression gain, ϵ = 0.7 and 0.9. The lower ϵ relaxes the constraint on the gain for higher CFs to be eligible for communication, thus achieving higher compression. A large ϵ (i.e., close to 1) allows for compression only if the compressed tensors are highly representative of the original gradients. First, we compare these two thresholds with Dense SGD as the latter demonstrates the ideal convergence scenario. Then, we compare *GraVAC* with different compression techniques on static CFs and look at final model accuracy, communication savings and overall speedup.

ResNet101: Fig. 7 shows how *GraVAC* achieves the same convergence as dense SGD in the same number of iterations. The low and high ϵ reduce overall communication volume by 163× and 19× over dense SGD. *We measure communication volume as the ratio of cumulative single-precision floats exchanged among workers in GraVAC relative to dense SGD.* Training cycle is slightly more volatile with compression, as seen from the accuracy drop due to lr decay at around 9000-th iteration. The drop is more apparent for ϵ = 0.7 as we continue to train with higher CFs on account of the lower threshold. Comparatively, ϵ = 0.9 is more robust to hyperparameter tuning like lr decay as we tend to train with a lower CF due to higher threshold. This is corroborated from Fig. 7b which shows distribution of training iterations over the CFs. We equally train with 10x and 1000x for ϵ = 0.9, while we mostly train with 1000x for ϵ of 0.7. For the compression throughputs of ϵ = 0.9 in Fig. 7c, it might seem counterintuitive at first that although $T_{compression}$ is maximum for 1000x and minimum for 10x, we still evenly train with the two CFs. This is on account of the high threshold and because θ_{min} did not scale up and remained at 10x for ResNet101. Thus, whenever



(a) Test accuracy (b) Iteration density (c) $T_{compression}$

Fig. 7. ResNet101:*GraVAC* with ϵ = [0.7, 0.9] and Dense SGD.



(a) Test accuracy (b) Iteration density (c) $T_{compression}$

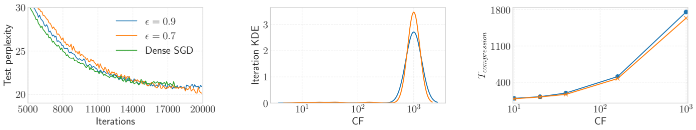
Fig. 8. VGG16: *GraVAC* with ϵ = [0.7, 0.9] and Dense SGD.

the compression gain of any candidate CF did not meet the threshold, we synchronized gradients compressed at 10x. For ϵ of 0.7, compression throughput was maximum for 1000x and we trained at this CF for most iterations as the corresponding gain easily met that threshold.

VGG16: Like ResNet101, VGG16 also converges to the same accuracy as dense SGD within the same iterations, where ϵ = 0.7 and 0.9 reduce communication volume by 80× and 13.5× over dense SGD (Fig. 9). Although $T_{compression}$ is maximum at 1000x for ϵ = 0.9, the corresponding gain was *not* as high to meet the threshold. Because of this, we switch back to θ_{min} and thus train with 10x for majority iterations as seen from the kernel density estimates in Fig. 8b. However, when ϵ was lower, we were able to find 40x CF to meet that threshold. $T_{compression}$ corresponding to this CF was second largest in our exploration space. As candidate CFs are evaluated over the iterations, the model gradually converges and as a result, compression gain improves even further on larger CFs as training progresses. Ultimately, we arrive on θ_{ideal} = 1000x corresponding to the maximum compression throughput (Fig. 8c).

LSTM: Like the models before, *GraVAC* with either ϵ converged in the same iterations as dense SGD training, while reducing the communication volume by 279× and 289× for ϵ of 0.9 and 0.7 respectively. Given the dataset, model and training hyperparameters, we already saw from Fig. 6d that compression gain for LSTM was high for both 10x and 1000x. We observed a similar trend here as compression gain corresponding to 1000x easily satisfied both thresholds and thus, we train with the largest available CF for most iterations (Fig. 9b). Correspondingly, the compression throughput is maximum at this CF as well.

Further, we compare *GraVAC* with static CFs running on different compression techniques. In particular, we train our models with Top-k, DGC, Redsync and Random-k at CFs 10x and 1000x. We run each compression technique to report the final accuracy/perplexity until it does not improve any further, difference in convergence compared to dense SGD baseline from Table 1, and relative training speedup over Top-k 10x for each model. The results are tabulated in Table 2. We do not consider dense SGD training in this comparison since we already established previously how *GraVAC* is able to achieve the same convergence in the same iterations, and other compression techniques have already been compared to dense SGD in prior works. For ResNet101, 1000x CF on Redsync, DGC and Top-k have considerably high speedups than 10x Top-k. However, these methods at 1000x CF achieve considerably less accuracy than Top-k at 10x. At 1000x, Top-k, DGC and Redsync do not improve beyond 76.4%, 78.6% and 77.4% top-1 test accuracy. Random-k faild to converge at either CF and accuracy did not improve beyond 20% . Because of *GraVAC*'s adaptive scheme, we converge to 80.2% accuracy while still reducing training time by 4.32×.



(a) Test accuracy (b) Iteration density (c) $T_{compression}$

Fig. 9. LSTM: *GraVAC* with ϵ = [0.7, 0.9] and Dense SGD.

TABLE 2
GraVAC'S MODEL QUALITY AND SPEEDUP OVER STATIC CFS

Model	Compression	Acc./Ppl	Diff.	Speedup
ResNet101	Top-k 10x	80.14%	+0.14%	1×
	Top-k 1000x	76.4%	−3.6%	3.02×
	DGC 10x	80.4%	+0.4%	1.23×
	DGC 1000x	78.6%	−1.4%	5.19×
	Redsync 10x	79.4%	−0.6%	1.2×
	Redsync 1000x	77.4%	−2.6%	6.94×
	Random-k 10x	-	-	-
	Random-k 1000x	-	-	-
	GraVAC	**80.2%**	**+0.2%**	**4.32×**
VGG16	Top-k 10x	91.2%	+1.2%	1×
	Top-k 1000x	90.68%	+0.68%	3.22×
	DGC 10x	90.8%	+0.8%	0.935×
	DGC 1000x	90.4%	+0.4%	3.35×
	Redsync 10x	90.45%	+0.45%	0.99×
	Redsync 1000x	90.3%	+0.3%	3.6×
	Random-k 10x	87.8%	−2.2%	0.7×
	Random-k 1000x	-	-	-
	GraVAC	**90.48%**	**+0.48%**	**1.95×**
LSTM	Top-k 10x	22.0	+0.0	1×
	Top-k 1000x	26.78	−4.78	3.36×
	DGC 10x	21.67	+0.33	1.23×
	DGC 1000x	25.14	−3.14	6.25×
	Redsync 10x	21.65	+0.35	1.17×
	Redsync 1000x	24.24	−2.24	6.9×
	Random-k 10x	24.15	−2.15	1.3×
	Random-k 1000x	-	-	-
	GraVAC	**21.25**	**+0.75**	**6.67×**

For VGG16, we previously observed that the model is already quite robust to high compression (Fig. 5). We see that again here for Top-k, DGC and Redsync at 1000x cross 90% accuracy with 3.22, 3.35 and 3.6× speedup over Top-k 10x. Random-k at 10x also converged, albeit to a lower 87.8% accuracy and slower convergence. Since *GraVAC* attains 90.48% test accuracy with 1.95× training speedup, other compression schemes were more optimal in this case simply because they used high CFs.

In LSTM, *GraVAC* obtains the least perplexity of 21.25 while still providing maximum speedup of 6.67× over Top-k 10x. Random-k 10x converged to 24.15 perplexity and did not improve further, while Random-k 1000x failed here again. Of all the configurations, only Top-k, DGC and Redsync at 10x CF and *GraVAC* achieved better perplexity than dense SGD.

Thus, we see how *GraVAC* is able to train models like ResNet101 and LSTM to high accuracy/perplexity and still reduce training time significantly. Static compression schemes achieve high accuracy at low CF at the cost of high communication overhead, thus providing lower speedup. Large CFs considerably reduce communication, but the final model quality is not at par with *GraVAC*. On the flip side, some over-parameterized models like VGG16 can be robust to compression and still converge successfully at high static CFs.

2) Geometric scaling policy: We also propose a relatively smoother compression policy where ScalingPolicy increments CFs as a geometric progression with common ratio 2. We deploy *GraVAC* with Redsync on ResNet101 and set θ_{min} = 10x, θ_{max} = 2000x, ϵ = 0.7, window = 2000 steps and ω = 1%. Thus, candidate CFs are 10x, 20x, 40x, 80x, 160x, 320x, 640x, 1280x and 2000x. Fig. 10a shows the accuracy curve over the iterations. Compared to dense SGD (Fig. 7a), *GraVAC* with geometric scaling converged *while reducing communication volume by 76×*. In contrast to exponential scaling, convergence is relatively slower because we evaluate each candidate CF for a larger window size. As a result, gradients get even smaller as *GraVAC* gradually arrives at larger CFs and compression gain increases beyond ϵ. Thus, we see similar iteration densities from CF 10x to 640x (Fig. 10b). After the first 7 CFs are evaluated over 2000 steps each, we mostly train with CF 1280x from 16K iterations onward (because 8 × 2000 = 16000). We did not scale to 2000x in our evaluation since compression throughput for 1280x and 2000x was 1029.9 and 1035.4, which falls within ω's bound of 1%. *This case highlights the effectiveness of GraVAC such that it does not scale the CF beyond a point when it stop improving the parallel or statistical efficiency of gradient compression.* In this case, *GraVAC* does not compress beyond 1280x as it corresponds to the maximum compression throughput (and at a lower CF of 1280x compared to 2000x).

C. Gains of Multi-level Compression in GraVAC

Alg. 1 explains how at each iteration, *GraVAC* scales compression from initial θ_{min} to current CF being evaluated (i.e., θ_c), up to the maximum allowed θ_{max}. Thus, compressing the original gradients (computed over backward pass) twice; i.e., once over θ_{min} and then again on θ_c can incur significant overhead, especially on larger models. The latency of a compressor may vary with the size of the tensor to compress as well as the target CF. To reduce the cumulative overhead of compressing original tensors multiple times, we apply a multi-level compression scheme as follows: given a compressor $\mathcal{C}$ and tensor $\mathcal{X}$ to be compressed to CFs θ_1 and θ_2 such that $\theta_2 > \theta_1$, rather than compressing each CF on $\mathcal{X}$ as:

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

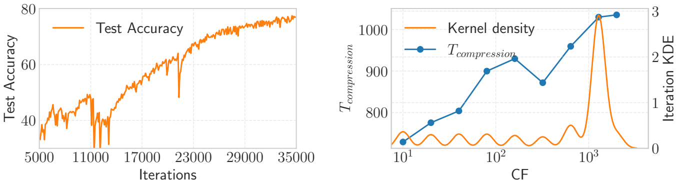
to produce compressed tensors where $|\mathcal{X}_2| < |\mathcal{X}_1| < |\mathcal{X}|$. In *GraVAC*, we first compute $\mathcal{X}_1$ and then compress this tensor to $\theta_2^{'}$ to produce $\mathcal{X}_2^{'}$:

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

The resulting tensor $\mathcal{X}_2^{'}$ is such that $\mathcal{X}_2^{'} = \mathcal{X}_2$ for $\theta_2^{'} = \theta_2/\theta_1$. The appeal of doing so is that the second compression operation is applied on a smaller tensor $\mathcal{X}_1$ instead of $\mathcal{X}$ again. We tabulate the savings of multi-level compression in Table 3. Let's consider a scaling case of *GraVAC* where $\theta_{min} = 10$x and current CF evaluated is 1000x. Then multilevel *GraVAC* first compresses to 10x and then further compresses the reduced tensors to 100x, i.e., $\theta_1 = 10$x and $\theta_2^{'} = 100$x so that $\theta_2 = 1000$x. In direct approach, we first compress original gradients to 10x, then compress the original gradients again to 1000x. From our results, we see that multi-level compression is at least 1.1× and up to 1.83× faster than directly compressing the original tensors twice.

D. Comparing GraVAC *with Prior Art*

In this section, we compare *GraVAC* with another adaptive scheme called Accordion [31]. For the three models, we use bounds of Rank-1 and Rank-4 for compression in Accordion, as described in [31] and compare with *GraVAC* in terms of communication and time savings (i.e., training speedup) to achieve the same test accuracy/perplexity. The savings are normalized by Accordion's performance for each respective model, shown in Table 4. For ResNet101, *GraVAC* reduces total communication volume by 44.5× and reduces training time by 1.94× over Accordion. *GraVAC* speeds up training by 5.63× over Accordion for communication-heavy models like VGG16. In LSTM training, *GraVAC* converges twice as fast by reducing communication volume up to 104.2×.



(a) Convergence

(b) $T_{compression}$ and KDE

Fig. 10. ResNet101: *GraVAC* with Geometric scaling policy.

TABLE 3
GraVAC'S MULIT-LEVEL (MTL) COMPRESSION SPEEDUP

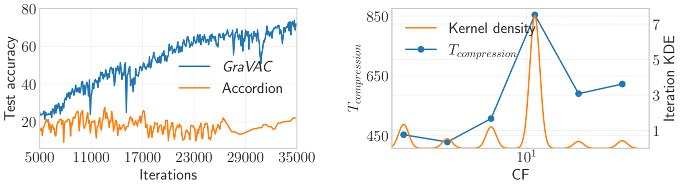
Model	Method	Direct (ms)	MTL (ms)	Speedup
ResNet101	Top-*k*	606	332	1.83×
	DGC	90	59	1.52×
	Redsync	33	29.8	1.1×
	Random-*k*	23	14	1.64×
VGG16	Top-*k*	181	121	1.49×
	DGC	122	95.5	1.27×
	Redsync	101.4	87.7	1.16×
	Random-*k*	41.6	31	1.34×
LSTM	Top-*k*	200	126	1.59×
	DGC	88	63	1.4×
	Redsync	69.4	46.4	1.5×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

TABLE 4
GraVAC VS. ACCORDION: COMMUNICATION AND TIME SAVINGS

Model	Method	Floats sent	Comm. sav.	Time sav.
ResNet101	Accordion	$4.17\ \times 10^{11}$	1×	1×
	GraVAC	$\mathbf{9.38 \times 10^{9}}$	**44.5×**	**1.94×**
VGG16	Accordion	$3.83\ \times 10^{11}$	1×	1×
	GraVAC	$\mathbf{1.7 \times 10^{10}}$	**22.4×**	**5.63×**
LSTM	Accordion	$4.2\ \times 10^{11}$	1×	1×
	GraVAC	$\mathbf{4 \times 10^{9}}$	**104.2×**	**2.06×**

Accordion is based on detecting critical regions during training, i.e., when inter-iteration gradients computed in backward pass change significantly and cross a certain user-defined threshold. Accordion switches between 2 compression factors such that it uses the low CF in critical regions and the higher CF otherwise. On the other hand, *GraVAC* looks at information loss on account of compression (i.e., statistical efficiency) and not just relative gradient change in sensitive regions of training. That is, *GraVAC* looks at intra-iterations gradients as well (between original and gradients compressed at different CFs). Additionally, *GraVAC* scales compression across a wider range and carefully inspects intermediary CFs as potential compression candidates. Thus, we obtain higher speedups when training with *GraVAC*.

1) GraVAC *vs. Accordion on Random-k Compression:* We previously saw in Fig. 2b and Table 2 that ResNet101 failed to converge at any CF with Random-*k* compression. In this section, we present a special case of using Random-*k* under the hood with both *GraVAC* and Accordion. Although the compression quality of Random-*k* is lower compared to other compressors, we present this as a special case to demonstrate how *GraVAC* is more dynamic and operates at a finer granularity. We launch *GraVAC* with Random-*k* on θ_{min} = 1.5x, θ_{max} = 1000x, window = 2000 and ϵ = 0.7. The CFs are scaled up via *geometric scaling policy*. Accordion was also deployed with the same min-max bounds on CF as *GraVAC*, i.e., low CF = 1.5x and high CF = 1000x. The convergence curves comparing *GraVAC* and Accordion are shown in Fig. 11a. Unlike static 10x Random-*k* compression (Fig. 2b) that failed to converge, we were able to achieve to 78% top-1 test accuracy for ResNet101 with *GraVAC*. The CFs used for training by *GraVAC* were 1.5x, 3x, 6x, 12x, 24x and 48x. All candidate CFs beyond this were ignored as they did not meet the required threshold of ϵ. CF 12x has the highest density, implying most iterations used this CF for training (Fig. 11b). Correspondingly, compression throughput is maximum for this CF as well. Compared to dense SGD, we reduced overall communication volume by 18×.



(a) Model convergence

(b) *GraVAC* $T_{comp.}$ and KDE

Fig. 11. *GraVAC* and Accordion on Random-*k* compression.

As for Accordion on Random-*k*, we see in Fig. 11a that training saturates at 20% accuracy. This is because Accordion does *not* consider the efficacy of the compression technique itself, and only switches between a low and high CF if the uncompressed, inter-iteration gradients change beyond a certain measure. With a low CF 1.5x, information loss in Random-*k* was too high to update ResNet101 in a meaningful way.

V. CONCLUSION

Gradient noise has previously been used as a scalability indicator for batch and cluster-size scaling in deep learning [19], [20], [37]–[39]. Adaptive compression schemes like Accordion [31] switch between two compression levels based on when the inter-iteration gradients change by some margin. *GraVAC*'s key insight is to tweak compression factor over the course of training while balancing the pareto-relationship between parallel and statistical efficiency in gradient compression. We use "compression gain" to measure information loss on account of compression and choose a CF appropriately. In our evaluation, we see that *GraVAC* converges 1.95 to 6.67× faster than choosing a static CF, while converging in the same number of iterations as dense SGD. Compared to Accordion, we observed up to 5.63× reduction in end-to-end training time.

One should be mindful when training models with *GraVAC* as it introduces parameters like compression threshold (ϵ) and window size that may affect overall training performance. Setting too small a window size may result in poor convergence as all the candidate CFs may be exhausted while the model is still in early training stages and gradients are still volatile. As for ϵ, choosing a very small threshold may enable high compression but may lead to model degradation by allowing high CF gradients from the beginning that will not update the model in a significant way.

REFERENCES

[1] AI and Compute, https://openai.com/blog/ai-and-compute/, 2018-05-16.

[2] Cherry, S.IEEE Spectrum, “Edholm’s law of bandwidth“, 2004. DOI 10.1109/MSPEC.2004.1309810.

[3] Schaller, Robert R.“Moore’s Law: Past, Present, and Future“, 1997.IEEE Press, DOI 10.1109/6.591665.

[4] Zhang Z., Chang C. , Lin H. Wang Y. , Arora R. and Jin X., “Is Network the Bottleneck of Distributed Training¿‘, NetAI 2020. DOI 10.1145/3405671.3405810.

[5] Alessandro A., Matteo R., Stefano S. “Critical Learning Periods in Deep Neural Networks“, 2019. arXiv 1711.08856.

[6] Dan A., Torsten H., Mikael J., Sarit K., Nikola K. and Cédric R. “The Convergence of Sparsified Gradient Methods“, 2018. arXiv 1809.10505.

[7] Yujun L., Song H., Huizi M., Yu W and Bill D. “Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training“, ICLR 2018.

[8] Shi, S., Chu, X., Cheung, K.C., and See, S. (2019). Understanding Top-k Sparsification in Distributed Deep Learning. ArXiv, abs/1911.08772.

[9] Fang J., Fu H., Yang G and Hsieh, CJ. “RedSync: Reducing synchronization bandwidth for distributed deep learning training system“, 2019. Journal of Parallel and Distributed Computing, DOI 10.1016/j.jpdc.2019.05.016.

[10] Facebook Gloo, https://github.com/facebookincubator/gloo.

[11] https://developer.nvidia.com/nccl, NVIDIA Collective Communication Library.

[12] “MPI: A message passing interface“, Supercomputing ’93:Proceedings of the 1993 ACM/IEEE Conference on Supercomputing. DOI 10.1145/169627.169855.

[13] Mu L., David G. A., Jun W.P., Alexander J. S., Amr A., Vanja J., James L., Eugene J. S and Bor-Yiing S. “Scaling Distributed Machine Learning with the Parameter Server“, 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14).

[14] Ruder S., “An overview of gradient descent optimization algorithms“. DOI 10.48550/ARXIV.1609.04747.

[15] Kaiming H., and Xiangyu Z., Shaoqing R. and Jian S. “Deep Residual Learning for Image Recognition“, 2015, arXiv 1512.03385.

[16] Karen S. and Andrew Z., “Very Deep Convolutional Networks for Large-Scale Image Recognition“, 2015.

[17] Hochreiter S. and Schmidhuber J., “Long Short-Term Memory, Neural Computation, 1997. DOI 10.1162/neco.1997.9.8.1735.

[18] Agarwal S., Wang H., Venkataraman S. and Papailiopoulos D., “On the Utility of Gradient Compression in Distributed Training Systems“, MLSys 2022.

[19] Johnson T. B., Agrawal P., Gu H. and Guestrin C.,“AdaScale SGD: A User-Friendly Algorithm for Distributed Training“, https://arxiv.org/abs/2007.05105.

[20] Luo M., Guo L., Marcel W., Konstantinos F., Andrei-Octavian B. and Peter P.,“Kungfu: Making Training in Distributed Machine Learning Adaptive“, 14th USENIX OSDI 2020.

[21] Sagun L., Bottou L. and LeCun Y.,“Eigenvalues of the Hessian in Deep Learning: Singularity and Beyond“, 2016. DOI 10.48550/ARXIV.1611.07476.

[22] Jonathan F., David J. S. and Ari S. M., “The Early Phase of Neural Network Training“, International Conference on Learning Representations, 2020.

[23] Alessandro A., Matteo R. and Stefano S.,“Critical Learning Periods in Deep Neural Networks“, 2019, arXiv 1711.08856.

[24] Paulius M., Sharan N., Jonah A., Gregory D., Erich E., David G., Boris G., Michael H., Oleksii K., Ganesh V. and Hao W.,“Mixed Precision Training“, 2018. arXiv 1710.03740.

[25] Dan A., Demjan G., Jerry L., Ryota T. and Milan V.,“QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding“, 2017. arXiv 1610.02132.

[26] Seide F., Fu H., Droppo J., Li G. and Yu D.,“1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs“, Interspeech 2014.

[27] Ahmed M. A., Ahmed E., Mohamed-Slim A. and Marco C.,“An Efficient Statistical-based Gradient Compression Technique for Distributed Training Systems“, MLSys 2021.

[28] Thijs V., Sai P. K. and Martin J.,“PowerSGD: Practical Low-Rank Gradient Compression for Distributed Optimization“, NeurIPS, 2019.

[29] Hongyi W., Saurabh A. and Dimitris P., “Pufferfish: Communication-efficient Models At No Extra Cost“, MLSys 2021.

[30] Guo J., Liu W., Wang W., Han J., Li R., Lu Y. and Hu S.,“Accelerating Distributed Deep Learning By Adaptive Gradient Quantization“, ICASSP 2020.

[31] Saurabh A., Hongyi W., Kangwook L., Shivaram V. and Dimitris P.,“Accordion: Adaptive Gradient Communication via Critical Learning Regime Identification“, arXiv 2010.16248.

[32] S. Tyagi and M. Swany, “ScaDLES: Scalable Deep Learning over Streaming data at the Edge“, IEEE Big Data 2022.

[33] Adam P., Sam G., Francisco M., Adam L., James B., Gregory C., Trevor K., Zeming L., Natalia G., Luca A., Alban D., Andreas K., Edward Y., Zach D., Martin R., Alykhan T., Sasank C., Benoit S., Lu F., Junjie B. and Soumith C.,“PyTorch: An Imperative Style, High-Performance Deep Learning Library“, NeurIPS 2019.

[34] Li S., Zhao Y., Varma R., Salpekar O., Noordhuis P., Li T., Paszke A., Smith J., Vaughan B., Damania P. andChintala S.,“PyTorch Distributed: Experiences on Accelerating Data Parallel Training“, VLDB Endowment 2020.

[35] Karimireddy S. P., Rebjock Q., Stich S. U. and Jaggi M.,“Error Feedback Fixes SignSGD and other Gradient Compression Schemes“,ICML2019.

[36] Zheng S., Huang Z. and Kwok J. T.,“Communication-Efficient Distributed Blockwise Momentum SGD with Error-Feedback“, NeurIPS 2019.

[37] Sam M., Jared K., Dario A and OpenAI Dota Team,“An Empirical Model of Large-Batch Training“, 2018. arXiv abs/1812.06162.

[38] Aurick Q., Sang K. C., Suhas J. S., Willie N., Qirong H., Hao Z., Gregory R. G. and Eric P. X.,“Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning“, 15th USENIX OSDI. 2021.

[39] Tyagi S., Sharma P.,“Scavenger: A Cloud Service for Optimizing Cost and Performance of ML Training“, IEEE/ACM Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2023.

[40] Fang J., Fu H., Yang G., Hsieh C.J., “Accelerating Distributed Deep Learning Training with Gradient Compression“, https://arxiv.org/pdf/1808.04357.pdf, 2018.