GRACE: A Compressed Communication Framework for Distributed Machine Learning

Hang Xu, Chen-Yu Ho, Ahmed M. Abdelmoniem, Aritra Dutta,
El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini and Panos Kalnis
KAUST

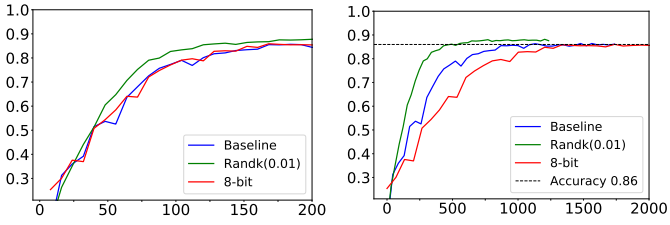
***Abstract*—Powerful computer clusters are used nowadays to train complex deep neural networks (DNN) on large datasets. Distributed training increasingly becomes communication bound. For this reason, many lossy compression techniques have been proposed to reduce the volume of transferred data. Unfortunately, it is difficult to argue about the behavior of compression methods, because existing work relies on inconsistent evaluation testbeds and largely ignores the performance impact of practical system configurations. In this paper, we present a comprehensive survey of the most influential compressed communication methods for DNN training, together with an intuitive classification (i.e., quantization, sparsification, hybrid and low-rank). Next, we propose GRACE, a unified framework and API that allows for consistent and easy implementation of compressed communication on popular machine learning toolkits. We instantiate GRACE on TensorFlow and PyTorch, and implement 16 such methods. Finally, we present a thorough quantitative evaluation with a variety of DNNs (convolutional and recurrent), datasets and system configurations. We show that the DNN architecture affects the relative performance among methods. Interestingly, depending on the underlying communication library and computational cost of compression / decompression, we demonstrate that some methods may be impractical. GRACE and the entire benchmarking suite are available as open-source.**

***Index Terms*—Survey, Distributed Machine Learning, Deep Learning, Gradient Compression, Benchmark.**

I. Introduction

Deep Neural Networks (DNNs) are becoming more complex. For example, ResNet-152 has 152 layers and 60.2M parameters [1], VGG-19 has 19 layers and 143M parameters [2], while BERT-Large has 24 layers and 340M parameters [3]. Combined with the large sizes of the training sets, parallelism during training becomes a necessity. Consequently, popular deep learning toolkits, including TensorFlow, PyTorch and MXNet, support *data parallelism*[1]: The DNN model under training is replicated in several compute nodes, a.k.a. *workers*, typically equipped with powerful GPUs. Each worker independently processes a partition of the data called mini-batch. Then, local intermediate results (typically, the local gradients) are exchanged through the network, and the aggregated values are sent back to the workers; the process is repeated over many epochs (i.e., full iterations of the training data).

Since the above-mentioned communication involves large amounts of data, the network becomes the bottleneck [5]–[7]. Luo et al. [5] argue that computation speed improves faster than network bandwidth; therefore, modern GPUs (e.g., NVIDIA V100) experience long idle times while waiting for communication. This causes inefficient utilization of the computational resources, longer training times and/or higher financial cost for cloud-based operations.



(a) x-axis: Epochs **(b) x-axis: Wall-time (s)**

Fig. 1: Top-1 accuracy for VGG16 on CIFAR-10 with TensorFlow on 8 workers via 25 Gbps network links. In (b) Randk converges in 450s, but 8-bit quantization needs 1200s.

To alleviate this problem, many works propose *lossy* compression during communication, to reduce the volume of transferred data. Typically, the so-called back-propagation phase of DNN training employs variants of the Stochastic Gradient Descent (SGD) [8] optimizer. Since training is stochastic in nature, intuitively, it should manage to converge despite the small errors introduced by the lossy compression. We identify four main classes of compressors in the literature: (*i*) *Quantization* [9]–[14], which reduces the number of bits of each element in the gradient tensor (e.g., cast float32 to float8); (*ii*) *Sparsification* [15]–[19], which transmits only a few elements per tensor (e.g., only the top-k largest elements); (*iii*) *Hybrid* methods [20]–[24], which combine quantization with sparsification; and (*iv*) *Low-rank* methods [25]–[28], which decompose the gradient into low-rank matrices.

Despite the abundance of compressed communication methods, it is unclear which one is more suitable and under what circumstances, or what the relative trade-offs are. Figure 1 demonstrates the problem on a standard TensorFlow benchmark (see §V for details) running on 8 workers with NVIDIA V100 GPUs and 25 Gbps network. Two common compression methods, Random-k [17] and 8-bit quantization [11], are compared against a baseline without compression. Most existing papers present an accuracy versus training epochs analysis, similar to Figure 1a, which shows almost equivalent effectiveness for all methods. However, in practice, users care about the actual elapsed time of the training process, shown in Figure 1b. Random-k converges in roughly 450 s

[1]Model and pipeline parallelism [4], which partition one replica of the model to multiple compute nodes, is orthogonal to data parallelism, but outside the scope of this paper.

TABLE I: Classification of surveyed gradient compression methods. Note that $\|\tilde{g}\|_0$ and $\|g\|_0$ are the number of elements in the compressed and uncompressed gradient, respectively; nature of operator Q is random or deterministic; EF-On indicates if error feedback is used in our experiments. We implement 16 methods on TensorFlow and PyTorch.

	Compression	Ref.	Similar Methods	$\|\tilde{g}\|_0$	Nature of Q	EF-On	Implementation
Quantization	8-bit quantization	[11]		$\|g\|_0$	Det	✓	TFlow
	1-bit SGD	[13]	[10], [21], [24]	$\|g\|_0$	Det	✓	TFlow, PyTorch
	SignSGD	[10]	[13], [29]	$\|g\|_0$	Det	×	TFlow, PyTorch
	SIGNUM	[30]	[10], [29]	$\|g\|_0$	Det	×	TFlow, PyTorch
	QSGD	[9]	[14], [27], [31] [32]–[34]	$\|g\|_0$	Rand	×	TFlow, PyTorch
	LPC-SVRG	[33]	[9], [31], [34]	$\|g\|_0$	Rand		
	Natural	[31]	[9], [33], [34]	$\|g\|_0$	Rand	✓	TFlow, PyTorch
	TernGrad	[14]	[9], [27], [33]	$\|g\|_0$	Rand	×	TFlow, PyTorch
	EFsignSGD	[12]	[29]	$\|g\|_0$	–NA–	✓	TFlow, PyTorch
	INCEPTIONN	[35]		$\|g\|_0$	Det	×	TFlow
Sparsification	Random-k	[17]		k	Rand	✓	TFlow, PyTorch
	Top-k	[15]	[17]	k	Det	✓	TFlow, PyTorch
	Threshold-v	[36]	[15]	Adaptive	Det	✓	TFlow, PyTorch
	Deep Gradient (DGC)	[16]		Adaptive	Det	✓	TFlow, PyTorch
	Adaptive sparsification	[19]	[27]	Adaptive	Rand		
	Variance-based sparsification	[18]		Adaptive	Det		
	Sketched-SGD	[37]	[15], [17]	k	Det		
Hybrid	Adaptive threshold SGD	[21]	[10], [13], [24]	Adaptive	Det	✓	TFlow
	SketchML	[22]	[21], [24]	Adaptive	Rand	✓	TFlow
	3LC	[23]	[14]	Adaptive	Det		
	Qsparse-local-SGD	[20]		Adaptive	Rand		
LowRank	ATOMO	[27]	[19]	sparsity budget	Rand		
	GradiVeQ	[28]	[27]	$(m+L)r$	Det		
	PowerSGD	[26]	[25]	$(m+L)r$	Det	✓	TFlow, PyTorch
	GradZip	[25]	[26]	$(m+L)r$	Det		

and is obviously preferable than the baseline that requires 850 s. Interestingly, 8-bit quantization converges after 1200 s, rendering it worse than using no compression at all.

In general, the majority of the existing work exhibits one or more of the following shortcomings: (*i*) Theoretical analysis is based on unrealistic assumptions, such as convexity; (*ii*) Implementation is stand-alone and does not reflect real-world scenarios that utilize one of the popular deep learning toolkits; (*iii*) Experimental evaluation ignores the computational cost of compression/decompression, which, in some cases, is larger than the savings by the reduced communication; (*iv*) Only convergence versus the number of epochs is reported, whereas actual execution time is ignored; (*v*) Experimental evaluation is performed on non-standard benchmarks; or, for a restricted set of models (e.g., only convolutional neural networks); or, even without considering DNNs at all.

Motivated by these shortcomings, in this paper, we follow a systematic approach to survey, categorize and evaluate quantitatively the existing work on compressed communication for Deep Learning under an extensive range of real-world models, datasets, and system configurations. We also propose the GRACE framework that allows (*i*) researchers to easily implement novel methods using our API and evaluate them on a standard testbed, and (*ii*) practitioners to investigate the trade-offs and select the method that suits the characteristics of their particular model and dataset. Our contributions are:

Survey. In §III, we present a comprehensive survey of the most influential works in compressed gradient communication; refer to Table I for a summary.

Framework and API. In §IV, we propose GRACE, a unified framework and programming API that exposes the necessary functions (e.g., `compress`, `decompress`, and `memory_compensate`) for implementing a variety of compressed communication methods. We embed GRACE in TensorFlow and PyTorch and implement 16 representative methods (see Table I). We release our code, execution scripts, evaluation metrics, and raw data; and provide the models and datasets.[2] Essentially, we develop a self-contained *testbed* that can be the standard of evaluating future compression methods.

Quantitative evaluation. In §V, we use a variety of models that include convolutional (CNN) as well as recurrent neural networks (RNN); and datasets from diverse domains that include image classification and segmentation, recommendation systems, and language modeling. We vary the number of workers as well as the network bandwidth; and report a rich set of metrics including throughput, data volume, accuracy, and computation overhead. While it is not possible to be exhaustive, we believe our results spanning a comprehensive set of 5 benchmarks, 7 model architectures and 4 ML tasks offer insights and draw lessons that are broadly applicable.

[2]Public release at https://github.com/sands-lab/grace.

Our results reveal that the speed and accuracy of each compression method depend on the particular DNN under training. Performance is also influenced by the underlying network communication libraries (e.g., OpenMPI [38] or NCCL [39]) and network bandwidth. Interestingly, many methods fail to match the no-compression baseline in terms of accuracy, as well as in terms of execution time, due to the computation overhead of compression/decompression; this issue is more pronounced for faster networks.

II. Background

We focus on *data-parallel* distributed training [9], [10], [40]–[43], where each worker possesses a local copy of the entire model; computes local updates; and communicates regularly with all other workers to synchronize with the aggregated global state. Global aggregation is commonly implemented through a collective communication library (e.g., Horovod [44]) in a peer-to-peer topology.[3]

Distributed data-parallel learning. A distributed optimization problem minimizes a function f:

$$\min_{x\in\mathbb{R}^d} f(x) \stackrel{\text{def}}{=} \frac{1}{n}\sum_{i=1}^{n} f_i(x), \tag{1}$$

where n is the number of workers. Each worker has a local copy of the model and a partition of the training data. The workers jointly update the model parameters $x \in \mathbb{R}^d$, where d is the number of parameters.

Consider a deep neural network (DNN), and let $x \stackrel{\text{def}}{=} \{W, b\}$ be the space that contains the model parameters (also known as weights W and biases b). Given a set of input data D with their corresponding *true* labels, the training phase learns x for each layer of the network. Let

$$f(x) \stackrel{\text{def}}{=} \frac{1}{n}\sum_{i=1}^{n} \underbrace{\left[\sum_{j=1}^{m} \mathcal{L}_j\left(\hat{y}_j\left(x, \hat{x}_{i,j}\right), y_j\right)\right]}_{:=f_i(x)} + R(x) \tag{2}$$

be the *loss* function such that, at each worker i, $\hat{x}_{i,j}$ is the input from its data partition D_i, y_j is the true label, $\mathcal{L}_j$ is the loss function (e.g., squared loss, cross-entropy loss, etc.) that calculates the discrepancy between the true label y_j and the predicted value $\hat{y}_j$, and R is a regularizer. Calculating the loss function for each training sample is called *forward* pass. During training, the parameter space x is updated by minimizing Equation (2) via a stochastic optimization algorithm that calculates the gradients of the loss function with respect to each layer of the DNN; a process known as *back-propagation*. In practice, each data partition D_i is further split into *mini-batches*, each with m data points. Each worker i performs the forward pass for all input data in a mini-batch; then performs back-propagation to calculate the stochastic gradients over the entire mini-batch; communicates with all other workers to aggregate all local gradients; and finally, uses the aggregated global state to update its parameters x.

[3]Our work is also applicable to master-worker architectures, where aggregation is performed in a central *parameter server*.

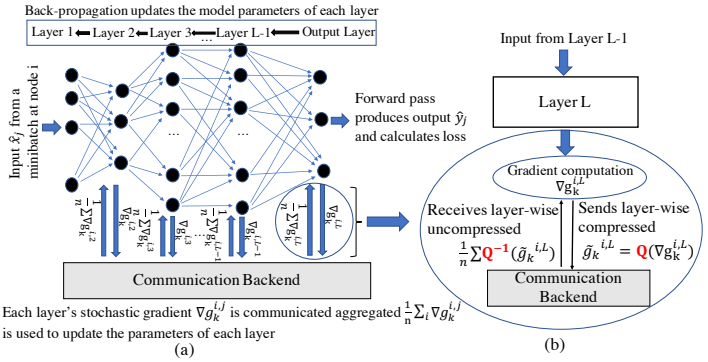


Fig. 2: (a) DNN architecture at node i. (b) Gradient compression mechanism for the L^{th} layer of a DNN.

Stochastic gradient descent (SGD). SGD [8] is a first-order iterative optimization algorithm. At iteration $k+1$, SGD updates the model parameters as:

$$x_{k+1} = x_k - \eta_k g_k \tag{3}$$

where $\eta_k > 0$ is the learning rate and g_k is the stochastic gradient at iteration k (i.e., an unbiased estimator of the gradient of f). To converge faster, SGD is often equipped with a short-term memory z, called *momentum*. For instance, Nesterov [45] computes the gradient g at a look-ahead point $x_k + \gamma z_k$ as: $z_{k+1} = \gamma z_k - \eta_k g(x_k + \gamma z_k)$, where $0 \leq \gamma \leq 1$. Then, Equation (3) of SGD becomes: $x_{k+1} = x_k + z_{k+1}$. In addition to SGD, several *accelerated* versions, such as ADAM [46], or ADAGrad [47], are used for DNN training.

III. Gradient Compression

We focus on *gradient* compression.[4] Let $g_k^{i,L}$ be the local gradient[5] in worker i at layer L of the DNN during training iteration k. Instead of transmitting $g_k^{i,L}$, the worker sends $Q(g_k^{i,L})$, where Q is a compression operator (see Figure 2). The receiver has a decompression operator Q^{-1} that reconstructs the gradient. Typically, this process is lossy; for this reason, several methods incorporate a *memory* (or *error feedback*) mechanism to compensate for the accumulated errors.

Formally, a *Compressor* is a random operator $Q(\cdot) : \mathbb{R}^d \to \mathbb{R}^d$, that satisfies $\mathbb{E}_Q\|x - Q(x)\|^2 \leq \Omega\|x\|^2$, where $\Omega > 0$ is the compression factor and the expectation is taken over the randomness of Q. If $\Omega = 1 - \delta$ and $\delta \in (0, 1]$, then Q is a *δ-compressor*; many sparsifiers belong to this category. If $\mathbb{E}(Q(x)) = x$, then Q is unbiased, otherwise it is biased. We classify gradient compression techniques into four categories, shown in Table I: quantization, sparsification, hybrid and low-rank. The most influential methods are presented below. For more details, refer to our companion technical report [48].

A. Quantization

Quantization reduces the number of bits of each element of the gradient, either by truncation or by mapping to a predefined set of code-words.

[4]For the orthogonal topic of *parameter* compression, see §VI.

[5]For simplicity, we will omit i, L from $g_k^{i,L}$ when possible.

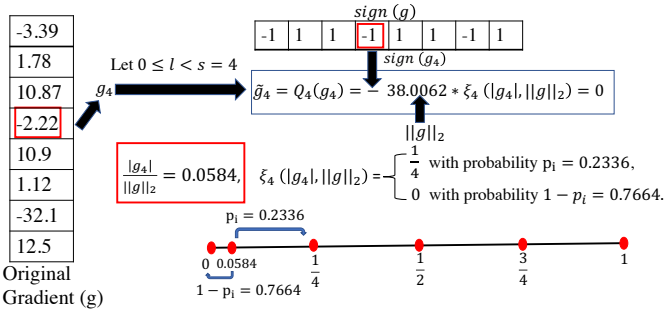


Fig. 3: QSGD example with $s = 4$, $l = 3$. The possible code-words are $0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1$ and are represented by 3-bits.

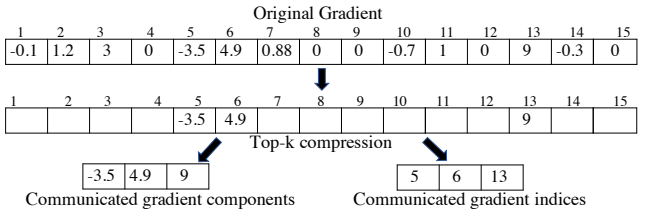


Fig. 4: Example Top-k compression: 20% of the gradient components and corresponding indices are sent.

8-bit quantization. Dettmers [11] maps each float32 element of the gradient to 8 bits: 1 sign, 3 exponent and 4 mantissa bits. To minimize precision loss, Dettmers also proposed a dynamic scheme, where exponent bits range from 0 to 6.

1-bit SGD. Seide et al. [13] propose an extreme form of quantization: all gradient elements that are less than a user-defined threshold τ (0 by default) are quantized to '0'; all other elements are quantized to '1'. Q^{-1} decodes '0's and '1' to the mean of the negative and non-negative values of the local gradient, respectively. This work also introduces a memory mechanism, $m_k = g_k - Q^{-1}(\tilde{g}_k)$ to compensate for the accumulated error. Let $\tilde{g}_k$ be the compressed gradient at iteration k; then $\tilde{g}_k = Q(g_k + m_k)$.

SignSGD, SIGNUM and EFsignSGD. SignSGD [10] transmits the sign of gradient elements by quantizing the negative components to -1 and the others to 1. SIGNUM [30] is a momentum version (see §II) of SignSGD. EFsignSGD [12] improves SignSGD's convergence via a memory mechanism. Zheng et al. [29] extended the error feedback approach to a bidirectional blockwise scheme with Nesterov momentum.

Ternary gradient. TernGrad [14] uses three values $\{-1, 0, 1\}$ scaled by the infinity norm of the stochastic gradient g. First, the elements of a bit-mask b are selected with probability $P(b_i = 1|g[i]) = g[i]/\|g\|_\infty$. Then, g is quantized to $\tilde{g} = \|g\|_\infty \text{sign}(g) \odot b$, where $\odot$ denotes element-wise product. TernGrad tends to achieve better convergence rate if the gradient components are evenly distributed.

Quantized SGD. QSGD is a codebook-based scheme by Alistarh et al. [9]. Wu et al. [32] extend QSGD with error feedback. QSGD quantizes each component $g[i]$ of the stochastic gradient via randomized rounding to a discrete set of values (i.e., code-words):

$$\tilde{g}[i] = \begin{cases} \|g\|_2 \text{sign}(g[i]).(\frac{l}{s}) & \text{with probability } p_i = \frac{s|g[i]|}{\|g\|_2} - l \\ \|g\|_2 \text{sign}(g[i]).(\frac{l+1}{s}) & \text{with probability } 1 - p_i \end{cases}$$

where $\|\cdot\|_2$ is the Euclidean norm, $s \geq 1$ and $l \in \mathbb{N}$ are user-defined parameters, such that $0 \leq l < s$ and $\frac{|g[i]|}{\|g\|_2} \in [l/s, (l+1)/s]$. An example is shown in Figure 3; there are 5 code-words, therefore, each element $g[i]$ of the original stochastic gradient is quantized to 3 bits.

LPC-SVRG and Natural Compression. LPC-SVRG [33] is a codebook-based approach that combines gradient clipping with quantization. For bit-width w and scaling factor $\delta > 0$, gradient component $g[i] \in [\varepsilon, \varepsilon + \delta]$ is quantized to ε with probability $p_i = \frac{\varepsilon + \delta - g[i]}{\delta}$; or to $\varepsilon + \delta$, otherwise, where $\varepsilon \in \{-2^{w-1}\delta, \ldots, -\delta, 0, \delta, \ldots, (2^{w-1} - 1)\delta\}$. Quantized-SVRG [9] is a related method with a variance reduction mechanism. Horvath et al. [31] proposed a similar scheme, called natural compression, that rounds the input to one of the two closest integer powers of 2.

INCEPTIONN [35] quantizes each 32-bit floating-point gradient element into four different levels (i.e., 32, 16, 8 and 0-bit) and a 2-bit tag indicating the compression level. This work is implemented on FPGA-based network cards (NICs) to reduce the computational overhead of compression / decompression.

B. Sparsification

Sparsification methods select only a subset of the elements of the original stochastic gradient g, resulting in a sparse vector. Let b be a bitmask vector with the same number of elements as g. An '1' bit in $b[i]$ indicates that the corresponding gradient element $g[i]$ is selected. The element-wise product $g \odot b$ generates a sparse vector of the original stochastic gradient. The sparse vector can be represented as two vectors: one contains the values of the selected elements of g, whereas the other contains the indices of the corresponding '1' in b.

Random-k [17]. Let d be the size of the bitmap b. A set of k indices are randomly selected out of d possible ones, and the k corresponding bits of b are set to '1'. By design, Random-k is biased, but can be made unbiased by multiplying g with a factor $\frac{d}{k}$. There is also a version with error feedback.

Top-k, Sketched-SGD and Threshold-v. Top-k [15] selects bitmask b such that $b[i] = 1$ if $|g[i]|$ belongs to the k largest values of g (in absolute value); otherwise, $b[i] = 0$; Figure 4 shows an example. Stich et al. [17] propose a similar scheme with memory. Ivkin et al. [37] propose Sketched-SGD, which uses count-sketch to select the "heavy hitters" that approximate the Top-k components of the gradient. In contrast to Top-k, Threshold-v [36] selects the elements whose absolute values are larger than a fixed threshold. However, an appropriate threshold is hard to determine as it depends on the model.

Deep gradient compression (DGC) [16]. Each worker i calculates the local gradient g_k^i and updates it as: $u_k^i = \beta u_{k-1}^i + g_k^i$. One can think of the above step as *momentum* added to the local gradient, a form of error feedback. Then, the gradient is accumulated via: $v_k^i = v_{k-1}^i + u_k^i$. Only gradient elements $g[i] < -\tau$ and $g[i] > \tau$ are transmitted, where τ is a user-defined threshold. To identify the threshold while incurring

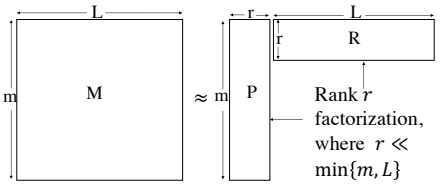


Fig. 5: Low-rank compression: Matrix M is decomposed into two low-rank matrices P, R each of rank r.

low overhead, Abdelmoniem et al. [49] propose an estimation technique based on modeling the gradient according to sparsity-inducing distributions.

Variance-based sparsification. Wangni et al. [19] observe that the variance of the gradient affects the convergence rate; they propose an unbiased sparse coding to maximize sparsity and control the variance. They assign a probability p_i and generate bitmask vector b such that $P\{b[i] = 1\} = p_i$, to obtain the compressed gradient element $\tilde{g}[i] = Z_i \frac{g[i]}{p_i}$. A similar method is proposed by Tsuzuku et al. [18].

C. Hybrid compressors

Hybrid methods combine quantization with sparsification.

Qsparse local SGD. Basu et al. [20] combine quantization with Top-k or Random-k sparsification. They implement synchronous and asynchronous versions with error feedback.

Hard and adaptive threshold SGD. Strom et al. [24] employ a user-defined threshold τ. Gradient elements $g[i] \in [-\tau, \tau]$ are omitted; therefore, the gradient is sparsified. For the remaining gradient elements, if $g[i] < -\tau$, it is quantized to '0'; else, if $g[i] > \tau$, it is quantized to '1'. Those elements are then packed into 32-bit words with one bit for the quantized value ('0' or '1') and 31 bits for the element index. During decompression, '0's and '1's are decoded to $-\tau$ and τ, respectively. Note that the appropriate value of τ is model-specific and hard to determine in practice. Instead of a fixed τ, Adaptive [21] uses a ratio $\alpha < 1$ of the proportion of negative and non-negative gradient elements. Adaptive samples the gradient to determine dynamically for each mini-batch two thresholds τ^- and τ^+ that satisfy the α-ratio.

SketchML. Jiang et al. [22] propose sketch-based compression. The algorithm selects only the non-zero elements of the gradient (i.e., sparsification) and builds a non-uniform quantile sketch [50]. Gradient values of each bucket are encoded with the bucket's index (i.e., quantization). The algorithm further compresses the bucket indices through hashing.

3LC [23] first calculates $\mathcal{M} = s\|g\|_\infty$, the highest magnitude gradient element scaled by a sparsity-multiplier parameter $s \in [1, 2)$. Then the quantized gradient is obtained by rounding the scaled gradient $(1/\mathcal{M})g$. The output is further compressed by aggressive lossless encoding. 3LC also implements error compensation.

D. Low-rank decomposition

DNNs are over-parameterized and exhibit low-rank structure [51], [52]. Based on this observation [53], [54], low-rank methods represent the gradient as a matrix $M \in \mathbb{R}^{m\times L}$ and factorize it into two low-rank matrices $P \in \mathbb{R}^{m\times r}$ and $R \in \mathbb{R}^{r\times L}$ that are smaller than M (see Figure 5). Typically, the factorization is approximate.

ATOMO and GradiVeQ. ATOMO [27] factorizes the gradient matrix M in a way that minimizes the variance of the quantized stochastic gradient. Let $\tilde{g}$ be an unbiased estimator of stochastic gradient g that has atomic decomposition $g = \sum_i \lambda_i a_i$, where $\mathcal{A} = \{a_i\} \subset V$ are atoms in an inner product space V with $\|a_i\| = 1$. If for each i and $0 \le p_i \le 1$, $t_i \sim \text{Bernoulli}(p_i)$, then ATOMO uses the estimator $\tilde{g} = \sum_i \frac{\lambda_i t_i}{p_i} a_i$ and by using a sparsity budget $\|p\|_1 = s$ solves a meta-optimization problem. This controls the gradient variance and represents g with a set of fewer basis elements that yield a low-rank approximation of g. The same authors proposed spectral-ATOMO, based on the singular value decomposition (SVD) of the gradient. GradiVeQ (gradient vector quantizer) [28] is also based on SVD.

Remark 1. With respect to the standard basis (atom), set $q = 2$ and ∞, respectively, in $s = \|g\|_1/\|g\|_q$ and probability $p_i = |g[i]|/\|g\|_q$. Then one can recover QSGD and TernGrad, respectively, from ATOMO.

PowerSGD and GradZIP. PowerSGD [26] uses power iteration to decompose the original gradient matrix M into two r-rank matrices P and R. The scheme is biased and the authors proposed to use a post-compression momentum. A similar method, GradZIP [25], employs an extra regularizer $\|P\|_F^2 + \|R\|_F^2$ and uses an alternating direction method to obtain factors P and R.

E. General comment on convergence

While some compressed distributed SGD algorithms are analyzed in the non-convex setup, some papers only provide the convergence guarantee when f is convex, under standard assumptions such as L-smoothness of f (e.g., see [17], [32]). Under these assumptions, for convex f, the convergence of compressed distributed SGD is $O(1/K)$, the same as the no-compression vanilla SGD, where K is the iteration count. For a non-convex function f (as it is the case with DNNs), it is typical to show that the quantity $\min_{k\in[K]} \mathbb{E}(\|\nabla f_k\|^2) \to 0$ as $K \to \infty$. With compressed and distributed SGD, the majority of the work shows the classical convergence rate $O(1/\sqrt{K})$ for non-convex functions. We refer to [36] for a general non-convex convergence analysis of distributed SGD *without error feedback* for both biased and unbiased compressors. However, *with error feedback*, the convergence analyses of compressed distributed SGD algorithms are more mathematically involved. Stich et al. [17] show, with an error feedback, sparsified SGD maintains the same convergence rate as no-compression vanilla SGD for the single node case and strongly convex f. Additionally, for the single node case, Karimireddy et al. [12] show that error feedback can alleviate the convergence issues of any arbitrary compression operator. Many works generalize the aforementioned scenarios to the distributed setting [55].

IV. GRACE - A UNIFIED FRAMEWORK

We develop GRACE, a unified framework for compressed communication for distributed deep learning. We instantiate GRACE within two popular ML toolkits, TensorFlow and PyTorch. GRACE encompasses a wide range of compression methods, capturing all the methods discussed in §III, and yet it exposes a simple programming API with which one can implement compression methods succinctly. GRACE provides a reference for fair quantitative evaluation across diverse methods and serves as a platform for rapid prototyping of new ones.

A. Distributed training loop

Our framework builds upon the distributed training loop with compressed communication depicted as pseudo-code in Algorithm 1. Each node executes the training loop in parallel and periodically synchronizes with other nodes.

Customizable components. Algorithm 1 references the following components that are customized for different compressors:

- $Q(\cdot)$ and $Q^{-1}(\cdot)$: denote the compression and decompression operators, respectively.
- $\phi(\cdot)$: is the memory compensation function, which compensates at each iteration the node-local gradient with the previous iteration's error feedback.
- $\psi(\cdot)$: is the memory update function that combines at each iteration the memory with the node-local gradient and error feedback.
- communication strategy: two types of collective communication strategies are explicitly supported, with support for custom gradient aggregation functions (Agg).

Training loop process. Each node locally computes a stochastic gradient g_k^i based on a mini-batch of training samples (Line 4). Then, it combines g_k^i with its memory m_k^i via $\phi(\cdot)$.[6] Next, the node applies compression operator Q on $\phi(g_k^i, m_k^i)$ to produce $\tilde{g}_k^i$ (Line 5). Memory m_k^i is updated using $\psi(\cdot)$ (Line 6). Now, each node communicates its $\tilde{g}_k^i$ using a collective communication primitive (Lines 8 and 11). Subsequently, every node obtains an aggregate of decompressed gradient g_k, typically $g_k = \frac{1}{n}\sum_i Q^{-1}(\tilde{g}_k^i)$. At this point, we distinguish the case of `Allreduce` and `Broadcast` or `Allgather` because the former results in the aggregate of the compressed gradients, whereas the latter involves a one-to-all or all-to-all communication, followed by a local aggregation step (the Agg function), which is customized for different methods. Finally, with g_k, each node updates its model parameters x by Equation (3) (Line 15). The loop repeats until convergence.

Layer-wise gradient as tensors. We denote the stochastic gradient g_k^i of a model as a single vector (at node i). This is merely for ease of presentation. Our framework equally applies to modern ML toolkits, where it is common during back-propagation to compute g_k^i incrementally for each DNN layer as some sequence $\hat{g}_k^{i,j}$ for decreasing j.

Different optimizers. Although we cast our training loop as a distributed SGD process, we note that the customizable components (Q, Q^{-1}, ϕ, ψ) are *optimizer independent*. Instead of SGD, any stochastic algorithm, such as AdaGrad [47], ADAM [46], can be used as optimizer in Algorithm 1. Our experiments use different optimizers, including SGD, RMSProp and SGD with momentum.

[6]The case with no memory compensation (hence, no memory) is a special case, where $\phi(g_k^i, m_k^i) = g_k^i$ and $\psi(m_k^i, g_k^i, \tilde{g}_k^i) = 0$.

Algorithm 1 Distributed Training Loop

Input: Number of nodes n, learning rate η_k, compressor Q, decompressor Q^{-1}, memory compensation function $\phi(\cdot)$, and memory update function $\psi(\cdot)$
Output: Trained model x

1: **On** each node i:
2: **Initialize:** $m_0^i = \mathbf{0}$ {vector of zeros}
3: **for** $k = 0, 1, \ldots,$ **do**
4: **Calculate** stochastic gradient g_k^i
5: $\tilde{g}_k^i = Q(\phi(m_k^i, g_k^i))$
6: $m_{k+1}^i = \psi(m_k^i, g_k^i, \tilde{g}_k^i)$
7: **if** compressor uses `Allreduce` **then**
8: $\tilde{g}_k = \texttt{Allreduce}(\tilde{g}_k^i)$
9: $g_k = Q^{-1}(\tilde{g}_k)\ /\ n$
10: **else if** compressor uses `Broadcast|Allgather` **then**
11: $[\tilde{g}_k^1, \tilde{g}_k^2, \cdots, \tilde{g}_k^n] = \texttt{Broadcast}(\tilde{g}_k^i)\ |\ \texttt{Allgather}(\tilde{g}_k^i)$
12: $[g_k^1, g_k^2, \cdots, g_k^n] = Q^{-1}([\tilde{g}_k^1, \tilde{g}_k^2, \cdots, \tilde{g}_k^n])$
13: $g_k = Agg([g_k^1, g_k^2, \cdots, g_k^n])$
14: **end if**
15: $x_{k+1}^i = x_k^i - \eta_k g_k$
16: **end for**
17: **return** x {each node has the same view of the model}

Memory compensation functions. We use the following form of functions $\phi(\cdot)$ and $\psi(\cdot)$ in this paper:

$$\begin{aligned} \phi(m_k^i, g_k^i) &= \beta m_k^i + \gamma g_k^i \\ \psi(m_k^i, g_k^i, \tilde{g}_k^i) &= \phi(m_k^i, g_k^i) - \tilde{g}_k^i \end{aligned} \tag{4}$$

where $\beta > 0$ is the memory decay factor and $\gamma > 0$ weighs the relevance of the latest stochastic gradient. We use $\beta = \gamma = 1$ unless otherwise noted. Users may customize these functions.

Communication with parameter server. Our framework is compatible with parameter server-based communication. Conceptually, a parameter server provides a gradient aggregation function equivalent to `Allreduce`. However, the Horovod toolkit we base our implementation on, exclusively supports collective communication libraries.

B. Programming interface

We provide an API for `compress` Q, `decompress` Q^{-1}, `memory_compensate` ϕ, `memory_update` ψ and `aggregate` Agg functions that are mentioned in the pseudo-code. The framework considers context `ctx` as an opaque object that carries any necessary metadata to allow for decompression, which should return a tensor with same data type and shape as the original tensor. For instance, in sparsification methods, `ctx` contains the shape and size of the original tensor. Below is an example function definition that takes a tensor with unique name and returns a list of compressed objects with the context needed to decompress them:

`compress : tensor, name` $\rightarrow$ `[comp], ctx`

Our framework provides defaults for `aggregate`, as well as `memory_compensate` and `memory_update` (Equation 4). The user needs to implement `compress` and `decompress` for each compression method, and indicate to the framework the communication strategy to use.

Compression typically produces tensors of different dimensions or data types than the original ones. For instance, sparsification results in smaller tensors while quantization results in either different data types or bit-packed elements. As these manipulations are common across several methods, our API implements the following helper functions:

API	Description
`quantize`	Quantizes tensor values and returns values in lower bits
`dequantize`	Dequantizes a tensor and restores original bits
`sparsify`	Sparsifies a tensor in a certain selection algorithm
`desparsify`	De-sparsifies and restores original shape by filling zeros
`pack`	Encodes several lower-bit values into one higher-bit value
`unpack`	Unpacks and restores the original decoded form

We support both TensorFlow and PyTorch; however, they have different APIs. Following Horovod's strategy, we create two similar yet distinct implementations.

Tensor manipulation operations. Both TensorFlow and PyTorch provide high-level tensor-manipulation APIs in Python as well as a C++ library to define custom tensor operations. We adopt the Python API since it is typically used by model creators and is backed by efficient low-level kernels for GPUs or other hardware accelerators; however, this does not prevent the user from integrating custom C++ operators.

Communication primitives. We leverage Horovod [44] for communication that exposes three collective communication primitives: `Allreduce`, `Allgather`, and `Broadcast`. On the backend, these are implemented by several alternate libraries: OpenMPI, NVIDIA NCCL and Facebook Gloo [56].

Communication strategies. We support two types of communication strategies: (*i*) `Allreduce` is the most efficient operation. However, it is not readily suitable for several scenarios. The main limitations in the underlying implementation are that it does not support sparse tensors and requires that input tensors be of the same data type and dimension. Moreover, it can only aggregate tensors by summing. In contrast, (*ii*) `Allgather` and `Broadcast` do not perform any aggregation and support input tensors of different forms. This is well suited for quantization when aggregation needs to be performed on dequantized values as well as for sparsification when different nodes select gradient elements at non-overlapping indices.

C. Implemented compressors

We implement within GRACE 16 representative compressors (see Table I). Where available, we draw from publicly available implementations; however, in many cases these are not released, although we reached out to the original authors to acquire them. We follow faithfully the algorithm descriptions presented in their corresponding papers, and we try to reproduce the original accuracy results. Our implementation is a best-effort approach that reflects the intention of the respective methods, although it may not be as efficient as the original ones. We avoid creating custom C++ tensor operations because the Python API is functionally sufficient and because it would have required an excessive effort given the large number of methods. Below, we highlight noteworthy implementation details.

Quantization. `quantize` converts the original 32-bit floating-point values into a lower-bit representation. `ctx` stores additional information (such as mean and different norms) needed to dequantize. In some scenarios, `pack` can further compress the data by encoding several lower-bit values into a single 32-bit value. `dequantize` transforms quantized values to an approximation of the original values. `unpack` decodes the packed data into their original representation.

Sparsification. `sparsify` flattens the original gradient into a rank-1 tensor. Then, it selects m out of d elements, and creates two $1 \times m$ rank-1 tensors to represent the selected values and indices. `ctx` stores the shape of the original tensor. `desparsify` restores the values into a rank-1 tensor of size d, fills missing values with zeros, and reshapes the tensor to the original gradient..

Adaptive [21]. Adaptive splits the gradient into a positive- and a negative-value part. We apply `quantize` to encode values in a ternary format and separate the +1 and -1 values. We use `sparsify` to select elements according to a sparsification ratio α. As the values are all ones, we only send the mean and the selected indices of each part.

DGC [16]. The momentum correction used in DGC is similar to memory compensation. We implement it by customizing the memory functions. `memory_compensate` adjusts the values by both memory and momentum. `memory_update` uses the minimum absolute value in the compressed gradient as the threshold to get the mask and to update the memory.

V. Experimental Evaluation

We perform a comprehensive quantitative evaluation of the 16 implemented compressors mentioned in Table I. Our results cover 5 benchmarks, 7 model architectures and 4 deep learning tasks (i.e., image classification, segmentation, recommendation, and language modeling). Due to space limit, below, we present the most representative results; for the complete set, refer to our technical report [48].

We break down the efficiency gains of compression along two metrics: (i) the data volume that each worker generates, and (ii) the training throughput (in terms of training samples per second). Measuring data volumes characterizes the intrinsic communication-level algorithmic efficiency of a method; whereas throughput offers the extrinsic measure of performance gains while other practical system artifacts are at play (e.g., computational overheads of compression).

A. Experimental setup and methodology

Environment. We run most of the experiments on 8 dedicated machines with Ubuntu 18.04.2 LTS and Linux v.4.15.0-74, 16-Core Intel Xeon Silver 4112 at 2.6 GHz, 512 GB RAM and one NVIDIA Tesla V100 GPU card with 16 GB on-board memory. They are interconnected via network links at 1, 10 and 25 Gbps. For time-insensitive metrics (e.g., accuracy, data volume) we also use a shared cluster with a heterogeneous group of nodes each equipped with at least one NVIDIA Tesla V100 GPU card. We deploy CUDA 10.1, PyTorch 1.3, TensorFlow 1.14, Horovod 0.18.2, OpenMPI 4.0 and NCCL 2.4.8.

TABLE II: Summary of the benchmarks and quality metrics used in this work.

Task	Model	Dataset	Training parameters	Gradient vectors	Epochs	Quality metric	Baseline quality
Image Classification	ResNet-20 [1]	CIFAR-10 [57]	269,467	51	328	Top-1 Accuracy	90.86%
	DenseNet40-K12 [58]	CIFAR-10 [57]	357,491	158	328		92.07%
	Custom ResNet-9 [59]	CIFAR-10 [57]	6,573,120	25	24		91.67%
	VGG16 [2]	CIFAR-10 [57]	14,982,987	30	328		86.32%
	ResNet-50 [1]	ImageNet [60]	25,559,081	161	90		75.37%
	VGG19 [2]	ImageNet [60]	143,671,337	38	90		68.90%
Recommendation	NCF [61]	Movielens-20M [62]	31,832,577	10	30	Best Hit Rate	95.98%
Language Modeling	LSTM [63]	PTB [64]	19,775,200	7	25	Test Perplexity	100.168
Image Segmentation	U-Net [65]	DAGM2007 [66]	1,850,305	46	2,500	Intersection over Union (IoU)	96.4%

Benchmarks. We use industry-standard benchmarks from TensorFlow [67], [68] and NVIDIA [69]. These benchmarks span 4 common deep learning tasks from different domains and involve a mix of convolutional and recurrent neural networks, and ones with large embedding layers. The trainable parameters span 3 orders of magnitude. The number of communicated gradient vectors range from 7 to 161. The *quality of the model* is reported under diverse nomenclatures according to benchmark-specific metrics as shown in Table II.

Methodology. We run each experiment for a fixed number of training epochs (complete iterations over the training set) according to every benchmark's specification. The reported quality of the model (e.g., accuracy), which is based on a held-out test set, is the best one witnessed throughout training.

We use *no compression* as the *baseline* for comparison. We ensure our baselines converge to state-of-the-art results (Table II). The default optimizers are: SGD with momentum for image classification, RMSProp for segmentation, ADAM for recommendation and SGD for language modeling. Compressors use the same optimizer as the baseline, except for image classification whereby PowerSGD, Random-k, DGC, SignSGD and SIGNUM use vanilla SGD as it achieves better quality.

When reporting relative results, they are normalized to the relevant metrics measured for the baseline case. We took care to make repetitions to validate statistically the model quality, except when it is too time-consuming to do so (as in training with ImageNet for instance). We focus mainly on TensorFlow results and comment on the differences with PyTorch where relevant; refer to our technical report [48] for PyTorch results.

Unless otherwise noted, we use the default configuration with each benchmark. We keep all hyperparameters the same as the baseline, except for the cases where specific settings are stated in a compressor's original paper. Specifically, for EFsignSGD, we set $\beta = 1$ and γ equal to the initial learning rate. The performance of compressors is sensitive to a range of factors such as the optimizer (e.g., SGD or ADAM), standard hyperparameters (e.g, learning rate) and varying degree of compression. Where practical, we experiment with multiple values of these factors and report on their effects; however, a complete sensitivity analysis is out of scope.

We report throughput as the average measured at steady state at the last 100 iterations during training. We measure the transmitted data volume in bytes based on the input size and a standard representation of data types (e.g., 4 bytes for float32, or 1 byte for 256-level quantized data).

The results shown in this section refer to experiments with 10 Gbps network links and OpenMPI over TCP as collective library.[7] We also run experiments with 25 Gbps links and observe mild improvements in throughput (on average, 1.3%).

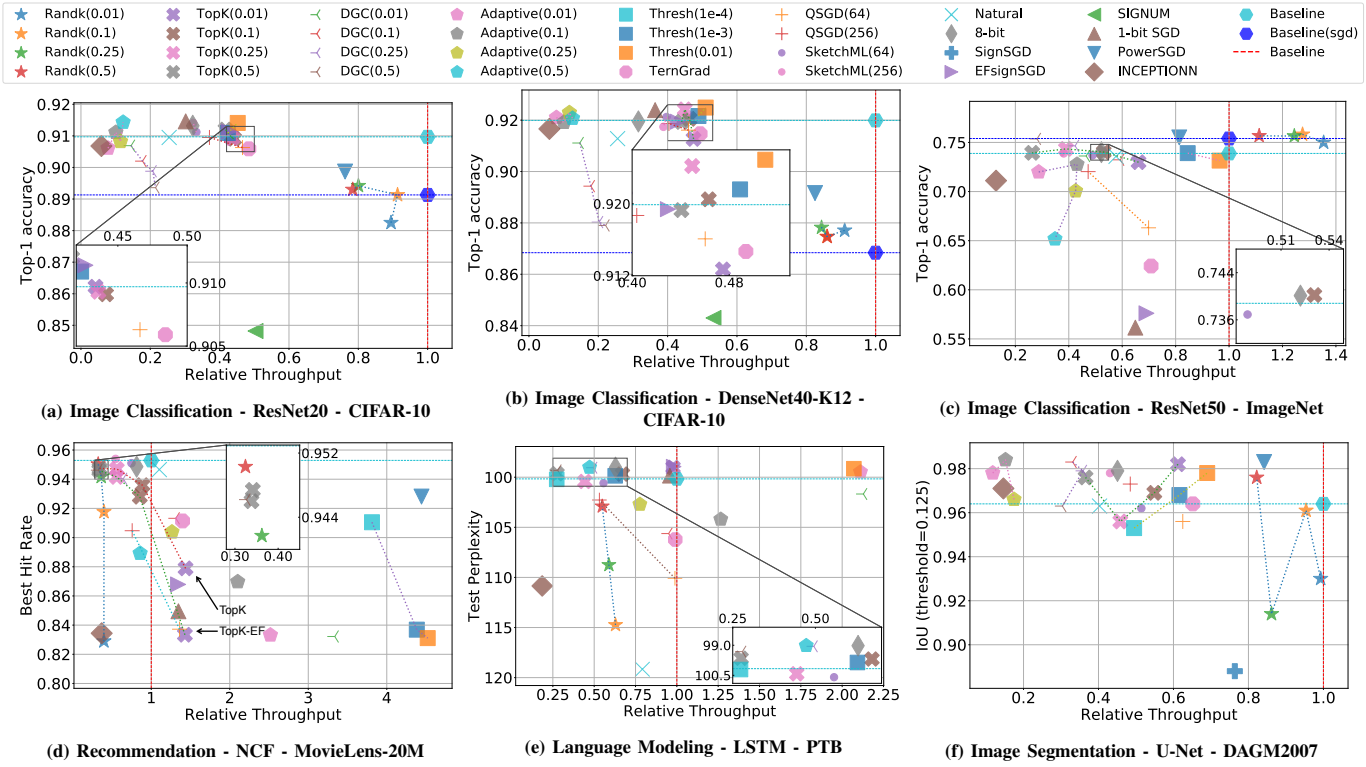
B. Model quality vs. training throughput

Figure 6 shows the effects of compression on model quality as a function of throughput, normalized to the baseline (highlighted with a vertical line in red). We present results across different benchmarks. Compressors that achieve poor quality (below the y-axis cut-off), are omitted. In general, we observe that training converges to solutions with quality metrics comparable to the respective baselines in most cases. In some cases, the model quality is slightly higher than the baseline. This can be attributed to the stochastic nature of the process, allowing compression to cancel out bad gradient directions; [16] also reported this phenomenon.

With respect to throughput, several compressors perform worse than the baseline. This happens for any benchmark where the trained model is primarily computation-bound (e.g., ResNet, DenseNet, U-Net). In contrast, for communication-bound models (e.g., NCF, VGG), there are several combinations of compressors that mark a significant throughput improvement. Note that, computation-bound models can also become communication-bound due to lower speed network (e.g., in the case of federated learning).

The recommendation benchmark (Figure 6d) is particularly interesting. First, this is a previously unexplored benchmark in the literature on compressed communication (which primarily has focused on convolutional NNs). Second, it highlights that there exists, in this particular case, a definite trade-off between model quality and throughput: while many compressors achieve 1.5× to 4.5× speedup, quality lowers by up to 10%. Third,

[7] NCCL is faster than OpenMPI, but it constrains input sizes, preventing a fair comparison.



(a) Image Classification - ResNet20 - CIFAR-10

(b) Image Classification - DenseNet40-K12 - CIFAR-10

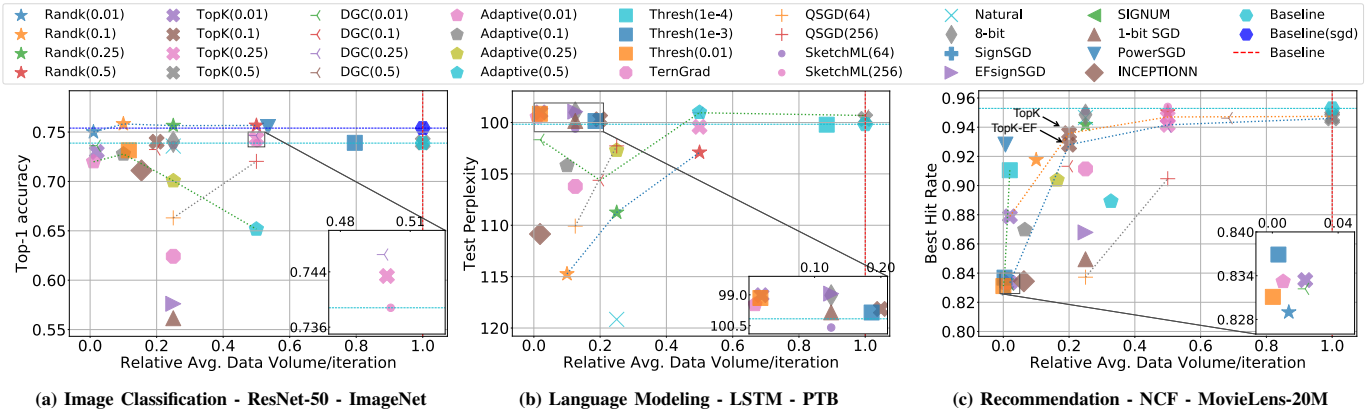
(c) Image Classification - ResNet50 - ImageNet

(d) Recommendation - NCF - MovieLens-20M

(e) Language Modeling - LSTM - PTB

(f) Image Segmentation - U-Net - DAGM2007

Fig. 6: Performance of compressors in terms of model quality vs training throughput.



(a) Image Classification - ResNet-50 - ImageNet

(b) Language Modeling - LSTM - PTB

(c) Recommendation - NCF - MovieLens-20M

Fig. 7: Performance of compressors in terms of model quality vs data volume.

it illustrates that for compressors with tunable degree of compression, quality lowers as compression is more aggressive. Interestingly, these observations are not common in other benchmarks. For instance, QSGD and Top-k in CIFAR-10 experiments score a ballpark model quality across varying degree of compression.

Table I indicates with a ✓ where error feedback (*EF*, a.k.a. memory) is applied; we find that EF improves accuracy for those compressors, in particular with sparsification. However, our results empirically establish that EF harms the convergence of several quantization methods (SignSGD, SIGNUM, QSGD and Terngrad). In the case of SignSGD and SIGNUM, the issue is known and is fixed by design by EFsignSGD. Interestingly, and exclusively for the recommendation task, applying EF with Top-k, 8-bit, and Natural Compression leads to worsened model quality. We highlight the difference for Top-k in Figure 6d.

Takeaways. No method consistently performs well across all benchmarks and there is no strong correlation between throughput and model quality.

C. Model quality vs. transmitted data volume

We now consider the model quality versus the transmitted data volume[8] trade-off. Figure 7 shows for each compressor

[8]Because we do not implement packing, the data volumes are inflated for quantization methods. However, in a relative sense our results still hold.

its best model quality and the average communicated data volume per iteration to achieve that quality (refer to [48] for the full set of results). In general, we observe that a compressor that sends more data leads to higher model quality. This is true in most cases especially in the language modeling, image classification for ImageNet, and recommendation tasks as shown in Figures 7a, 7b, 7c, respectively. However, we observe that for some compressors (e.g., Adaptive), a higher data volume results in lower model quality. This is consistent with previously published results [16], [21], [26].

Takeaways. The quality vs. data volume trade-off is non trivial; therefore, compression should be tuned carefully to deliver the best benefits for a given scenario.

D. Computational overheads of compression

We run a micro-benchmark experiment that measures the combined latency of `compress` and `decompress` in isolation; Figure 8 shows the results as a violin plot for 30 repetitions; the operators run on the GPU. Results show that compressors induce non-negligible overheads. We profile the code and observe the following: (*i*) Both Adaptive and DGC involve a loop to adjust the threshold to best match the target ratio. This is expensive; throughput improved by $\approx 2\times$ by executing only one iteration. (*ii*) As shown in Figure 8, Random-k shows high overhead as the `tf.random.shuffle` operation executes on the CPU due to lack of a GPU kernel. However, during real training, TensorFlow can schedule device-host data transfer so that it overlaps with GPU computation, so this overhead is at times mitigated. (*iii*) In Random-k, `tf.random.shuffle` takes excessively long time on CPU for both the large embedding and fully-connected layers in recommendation and language modeling. The execution time far exceeds the execution of the forward pass and hence communication phase stalls by waiting for this operation. (*iv*) 8-bit invokes a `find_bins` operation for each quantized value which, due to lack of a GPU implementation, is executed on the CPU. (*v*) We also observe that some methods rely on expensive operations (i.e., `tf.where`). These are sparsification methods that rely on a threshold (e.g., Threshold-v, DGC) and quantization methods that choose target elements meeting a criteria (e.g., 1-bit SGD, Terngrad, 8-bit, Natural Compression). Sketch-ML also imposes high overhead due to sketch operations.

Takeaways. Implementing compressors requires careful engineering, with custom GPU or well-optimized CPU code, to account for their intrinsic computational overheads.

E. Machine learning toolkit, transport and links

Figure 9 shows the throughput of different compressors in CIFAR-10 image classification task using PyTorch with different communication protocols: TCP and remote direct memory access (RDMA). Throughput is mostly consistent yet higher than what we observe for most compressors in the TensorFlow image classification tasks. The RDMA transport protocol is consistently better than TCP.

Figure 10 shows the relative throughput for the same experimental setup as Figure 6c, except it uses 1 Gbps network links. As this setup emphasizes the network bottleneck, there is now a large number of compressors that obtain a throughput speedup over the baseline.

Takeaways. The machine learning toolkit as well as the transport protocol and network speed, affect compressor throughput.

F. Summary of observations

- No particular compression method outperforms every other across all experimental scenarios.
- The computational overheads of compression are not negligible. At higher communication bandwidths (10 Gbps or more), avoiding compression typically results in faster training, which agrees with the results of [35], [70].
- With some exceptions, error feedback (EF) is widely applicable to sparsification and improves accuracy significantly. However, its side-effect is memory overhead, which may lead to smaller mini-batch sizes.
- Higher data volume does not imply higher accuracy; however, we observe that when compression is heavy, a low data volume tends to decrease accuracy.
- The hosting ML framework influences performance only to a minor extent; the major performance variations are due to the underlying collective communication libraries.

VI. Related work

Yang [71] was one of the first to study the trade-off between computation and communication for distributed stochastic optimization. Since then, numerous approaches have been proposed; refer to the survey by Ben-Nun and Hoefler [72] for details. Below we cite those that are relevant to our work.

Compression for ad-hoc P2P overlays. Unlike our work, which assumes all-to-all aggregation semantics (e.g., Allreduce), others [43], [73] consider an ad-hoc peer-to-peer network overlay, where nodes communicate only with neighbours. Althought some of these methods use familiar techniques, like sparsification and quantization, their main characteristic is that they redefine the aggregation semantics to involve only a subset of workers at a time. We leave it as future work to integrate in our framework's communication primitives that accommodate the P2P overlay setting.

Fewer communication rounds. Some methods reduce the volume of transmitted data by communicating less often. CoCoA [74] is dual coordinate ascent algorithm that performs several local steps before communicating with other workers. Wang and Joshi [75] propose periodic averaging SGD, to update the local model at each worker node and then use periodic average to update the final parameters.

Asynchronous communication. Hogwild! [76] propose *asynchronous* parallel SGD, where the computing nodes access shared memory and can modify the parameters at any time without locking. De Sa et al. [77] develop a low-precision asynchronous SGD method and provide an FPGA implementation. Asynchronous communication is outside the scope of our paper.

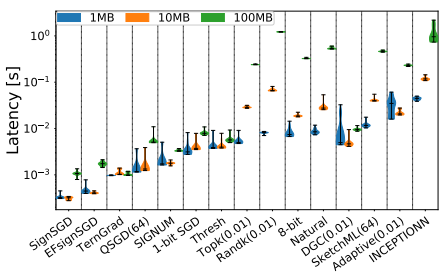


Fig. 8: Latency of `compress` and `decompress` for different compressors with a range of input sizes.

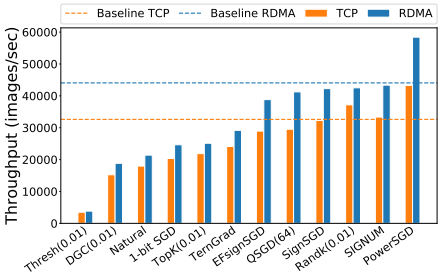


Fig. 9: Throughput for ResNet-9 on CIFAR-10 contrasting TCP vs. RDMA performance in PyTorch.

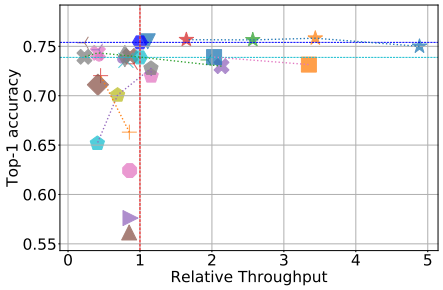


Fig. 10: Performance of compressors for ResNet-50 on ImageNet via 1 Gbps network. Legend in Figure 6.

Communication primitives. SwitchML [7] uses a programmable network switch to implement in-network aggregation. Instead of compression, SwitchML reduces the transmitted data by computing on the network switch. A similar idea is explored in DAIET [78]. SparcML [79], on the other hand, implements a stream structure to support sparse tensors.

Other communication strategies. OmniReduce [80] implements sparse Allreduce and sends the non-zero gradient blocks to the workers. Gajjala et al. [81] use Huffman encoding for efficiently packing and transmitting the quantized vectors. DeepReduce [82] is a compressed communication framework that allows both independent and combined compression of values and indices of sparse tensors.

Model compression. Instead of compressing the communicated gradient, many papers propose to compress the model parameters. ZipML [34], in particular, applies compression similar to that of QSGD to the model, data, and gradient. Model compression is orthogonal to our work and out of scope; we refer to a survey by Guo [83].

VII. Conclusion

We survey the most influential methods on gradient compression for distributed, data-parallel DNN training. We propose GRACE, a unified framework with the corresponding TensorFlow and PyTorch API, and implement 16 representative compression methods. We use convolutional and recurrent DNNs, as well as a variety of datasets and system configurations, to perform thorough quantitative evaluation and report metrics that include accuracy, throughput and communication volume. We observe that the computational overhead of compression / decompression is non-trivial and may render several methods inapplicable in practice. We release our API, code and experimental results, as well as the DNN models and datasets. We envision that our work will benefit: (*i*) researchers, who will use it as the basis for consistent implementation and evaluation of new methods; and (*ii*) practitioners, who need an appropriate compression method for their training setup.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *CVPR*, 2015.

[2] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *ICLR*, 2015.

[3] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *NAACL*, 2018.

[4] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: Generalized Pipeline Parallelism for DNN Training," in *SOSP*, 2019.

[5] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy, "PHub: Rack-Scale Parameter Server for Distributed Deep Neural Network Training," in *SoCC*, 2018.

[6] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A Generic Communication Scheduler for Distributed DNN Training Acceleration," in *SOSP*, 2019.

[7] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy *et al.*, "Scaling Distributed Machine Learning with In-Network Aggregation," in *NSDI*, 2021.

[8] H. Robbins and S. Monro, "A Stochastic Approximation Method," *Annals of Mathematical Statistics*, vol. 22, 1951.

[9] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding," in *NeurIPS*, 2017.

[10] J. Bernstein, Y.-X. Wang, K. Azizzadenesheli, and A. Anandkumar, "signSGD: Compressed Optimisation for Non-Convex Problems," in *ICML*, 2018.

[11] T. Dettmers, "8-Bit Approximations for Parallelism in Deep Learning," in *ICLR*, 2016.

[12] S. P. Karimireddy, Q. Rebjock, S. Stich, and M. Jaggi, "Error Feedback Fixes Sign SGD and other Gradient Compression Schemes," in *ICML*, 2019.

[13] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs," in *INTERSPEECH*, 2014.

[14] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning," in *NeurIPS*, 2017.

[15] A. F. Aji and K. Heafield, "Sparse Communication for Distributed Gradient Descent," in *EMNLP-IJCNLP*, 2017.

[16] Y. Lin, S. Han, H. Mao, Y. Wang, and W. Dally, "Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training," in *ICLR*, 2018.

[17] S. U. Stich, J.-B. Cordonnier, and M. Jaggi, "Sparsified SGD with Memory," in *NeurIPS*, 2018.

[18] Y. Tsuzuku, H. Imachi, and T. Akiba, "Variance-based Gradient Compression for Efficient Distributed Deep Learning," in *ICLR*, 2018.

[19] J. Wangni, J. Wang, J. Liu, and T. Zhang, "Gradient Sparsification for Communication-Efficient Distributed Optimization," in *NeurIPS*, 2018.

[20] D. Basu, D. Data, C. Karakus, and S. Diggavi, "Qsparse-local-SGD: Distributed SGD with Quantization, Sparsification and Local Computations," in *NeurIPS*, 2019.

[21] N. Dryden, S. A. Jacobs, T. Moon, and B. Van Essen, "Communication Quantization for Data-Parallel Training of Deep Neural Networks," *MLHPC*, 2016.

[22] J. Jiang, F. Fu, T. Yang, and B. Cui, "SketchML: Accelerating Distributed Machine Learning with Data Sketches," *SIGMOD*, 2018.

[23] H. Lim, D. Andersen, and M. Kaminsky, "3LC: Lightweight and Effective Traffic Compression for Distributed Machine Learning," in *MLSys*, 2019.

[24] N. Strom, "Scalable Distributed DNN Training using Commodity GPU Cloud Computing," in *INTERSPEECH*, 2015.

[25] M. Cho, V. Muthusamy, B. Nemanich, and R. Puri, "GradZip: Gradient Compression using Alternating Matrix Factorization for Large-scale Deep Learning," in *NeurIPS*, 2019.

[26] T. Vogels, S. P. Karimireddy, and M. Jaggi, "PowerSGD: Practical Low-Rank Gradient Compression for Distributed Optimization," in *NeurIPS*, 2019.

[27] H. Wang, S. Sievert, S. Liu, Z. Charles, D. Papailiopoulos, and S. Wright, "ATOMO: Communication Efficient Learning via Atomic Sparsification," in *NeurIPS*, 2018.

[28] M. Yu, Z. Lin, K. Narra, S. Li, Y. Li, N. S. Kim, A. Schwing *et al.*, "GradiVeQ: Vector Quantization for Bandwidth-Efficient Gradient Aggregation in Distributed CNN Training," in *NeurIPS*, 2018.

[29] S. Zheng, Z. Huang, and J. Kwok, "Communication-Efficient Distributed Blockwise Momentum SGD with Error-Feedback," in *NeurIPS*, 2019.

[30] J. Bernstein, J. Zhao, K. Azizzadenesheli, and A. Anandkumar, "signSGD with Majority Vote is Communication Efficient And Fault Tolerant," in *ICLR*, 2019.

[31] S. Horvath, C.-Y. Ho, L. Horvath, A. N. Sahu, M. Canini, and P. Richtárik, "Natural Compression for Distributed Deep Learning," *arXiv preprint arXiv:1905.10988v2*, 2019.

[32] J. Wu, W. Huang, J. Huang, and T. Zhang, "Error Compensated Quantized SGD and its Applications to Large-scale Distributed Optimization," in *ICML*, 2018.

[33] Y. Yu, J. Wu, and J. Huang, "Exploring Fast and Communication-Efficient Algorithms in Large-Scale Distributed Networks," in *AISTATS*, 2019.

[34] H. Zhang, J. Li, K. Kara, D. Alistarh, J. Liu, and C. Zhang, "ZipML: Training Linear Models with End-to-End Low Precision, and a Little Bit of Deep Learning," in *ICML*, 2017.

[35] Y. Li, J. Park, M. Alian, Y. Yuan, Z. Qu, P. Pan, R. Wang *et al.*, "A Network-Centric Hardware/Algorithm Co-Design to Accelerate Distributed Training of Deep Neural Networks," in *Micro*, 2018.

[36] A. Dutta, E. Bergou, A. Abdelmoniem, C. Ho, A. Sahu, M. Canini, and P. Kalnis, "On the Discrepancy between the Theoretical Analysis and Practical Implementations of Compressed Communication for Distributed Deep Learning," in *AAAI*, 2020.

[37] N. Ivkin, D. Rothchild, E. Ullah, V. Braverman, I. Stoica, and R. Arora, "Communication-efficient Distributed SGD with Sketching," in *NeurIPS*, 2019.

[38] "Open MPI," https://www.open-mpi.org/.

[39] "NCCL," https://developer.nvidia.com/nccl.

[40] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized Stochastic Gradient Descent," in *NeurIPS*, 2010.

[41] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato *et al.*, "Large scale distributed deep networks," in *NeuRIPS*, 2012.

[42] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long *et al.*, "Scaling Distributed Machine Learning with the Parameter Server," in *OSDI*, 2014.

[43] H. Tang, S. Gan, C. Zhang, T. Zhang, and J. Liu, "Communication Compression for Decentralized Training," in *NeurIPS*, 2018.

[44] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," *arXiv preprint arXiv:1802.05799*, 2018.

[45] Y. Nesterov, "Gradient Methods for Minimizing Composite Functions," *Mathematical Programming*, vol. 140, no. 1, 2013.

[46] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *ICLR*, 2015.

[47] J. Duchi, E. Hazan, and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *JMLR*, vol. 12, 2011.

[48] H. Xu, C.-Y. Ho, A. M. Abdelmoniem, A. Dutta, E. H. Bergou, K. Karatsenidis, M. Canini, and P. Kalnis, "Compressed Communication for Distributed Deep Learning: Survey and Quantitative Evaluation," KAUST, Tech. Rep., 2020. [Online]. Available: http://hdl.handle.net/10754/662495

[49] A. M. Abdelmoniem, A. Elzanaty, M.-S. Alouini, and M. Canini, "An Efficient Statistical-based Gradient Compression Technique for Distributed Training Systems," in *MLSys*, 2021.

[50] M. Greenwald and S. Khanna, "Space-Efficient Online Computation of Quantile Summaries," in *SIGMOD*, 2001.

[51] S. Arora, R. Ge, B. Neyshabur, and Y. Zhang, "Stronger Generalization Bounds for Deep Nets via a Compression Approach," in *ICML*, 2018.

[52] Y. Li, T. Ma, and H. Zhang, "Algorithmic Regularization in Over-parameterized Matrix Sensing and Neural Networks with Quadratic Activations," in *COLT*, 2018.

[53] P. Jain, C. Jin, S. M. Kakade, P. Netrapalli, and A. Sidford, "Streaming PCA: Matching Matrix Bernstein and Near-Optimal Finite Sample Guarantees for Oja's Algorithm," in *COLT*, 2016.

[54] E. Oja, "Simplified Neuron Model as a Principal Component Analyzer," *Journal of Mathematical Biology*, vol. 15, no. 3, 1982.

[55] D. Alistarh, T. Hoefler, M. Johansson, S. Khirirat, N. Konstantinov, and C. Renggli, "The Convergence of Sparsified Gradient Methods," in *NeurIPS*, 2018.

[56] "Gloo," https://github.com/facebookincubator/gloo.

[57] A. Krizhevsky and G. Hinton, "Learning Multiple Layers of Features From Tiny Images," *Technical report, UToronto*, vol. 1, no. 4, 2009.

[58] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely Connected Convolutional Networks," in *CVPR*, 2017.

[59] D. Page, "CIFAR10-fast," https://github.com/davidcpage/cifar10-fast.

[60] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and F. F. Li, "ImageNet: a Large-Scale Hierarchical Image Database," in *CVPR*, 2009.

[61] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua, "Neural Collaborative Filtering," in *WWW*, 2017.

[62] "Movielens," https://grouplens.org/datasets/movielens/.

[63] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computing*, vol. 9, no. 8, 1997.

[64] M. Marcus, B. Santorini, M. Marcinkiewicz, and A. Taylor, "Treebank-3," https://catalog.ldc.upenn.edu/LDC99T42.

[65] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," in *MICCAI*, 2015.

[66] "29th Annual symposium of the German association for pattern recognition," https://resources.mpi-inf.mpg.de/conference/dagm/2007/index.html.

[67] "LSTM-PTB," https://github.com/tensorflow/models/tree/master/tutorials/rnn.

[68] "TensorFlow benchmark," https://github.com/tensorflow/benchmarks.

[69] "NVIDIA deep learning examples," https://github.com/NVIDIA/DeepLearningExamples.

[70] L. Luo, P. West, A. Krishnamurthy, L. Ceze, and J. Nelson, "PLink: Discovering and Exploiting Datacenter Network Locality for Efficient Cloud-based Distributed Training," in *MLSys*, 2020.

[71] T. Yang, "Trading Computation for Communication: Distributed Stochastic Dual Coordinate Ascent," in *NeurIPS*, 2013.

[72] T. Ben-Nun and T. Hoefler, "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis," *ACM Computing Surveys*, vol. 52, no. 4, 2019.

[73] A. Koloskova, S. Stich, and M. Jaggi, "Decentralized Stochastic Optimization and Gossip Algorithms with Compressed Communication," in *ICML*, 2019.

[74] M. Jaggi, V. Smith, M. Takac, J. Terhorst, S. Krishnan, T. Hofmann, and M. I. Jordan, "Communication-Efficient Distributed Dual Coordinate Ascent," in *NeurIPS*, 2014.

[75] J. Wang and G. Joshi, "Adaptive Communication Strategies to Achieve the Best Error-Runtime Trade-off in Local-Update SGD," in *MLSys*, 2019.

[76] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent," in *NeurIPS*, 2011.

[77] C. De Sa, M. Feldman, C. Ré, and K. Olukotun, "Understanding and Optimizing Asynchronous Low-precision Stochastic Gradient Descent," in *ISCA*, 2017.

[78] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-Network Computation is a Dumb Idea Whose Time Has Come," in *HotNets*, 2017.

[79] C. Renggli, S. Ashkboos, M. Aghagolzadeh, D. Alistarh, and T. Hoefler, "SparCML: High-Performance Sparse Communication for Machine Learning," in *SC*, 2019.

[80] J. Fei, C.-Y. Ho, A. N. Sahu, M. Canini, and A. Sapio, "Efficient Sparse Collective Communication and its application to Accelerate Distributed Deep Learning," KAUST, Tech. Rep., 2020. [Online]. Available: http://hdl.handle.net/10754/665369

[81] R. R. Gajjala, S. Banchhor, A. M. Abdelmoniem, A. Dutta, M. Canini, and P. Kalnis, "Huffman Coding Based Encoding Techniques for Fast Distributed Deep Learning," in *ACM CoNEXT Dist. ML*, 2020.

[82] K. Kostopoulou, H. Xu, A. Dutta, X. Li, A. Ntoulas, and P. Kalnis, "DeepReduce: A Sparse-tensor Communication Framework for Distributed Deep Learning," *arXiv preprint arXiv:2102.03112*, 2021.

[83] Y. Guo, "A Survey on Methods and Theories of Quantized Neural Networks," *arXiv preprint arXiv:1808.04752v2*, 2018.