Compressed Communication for Distributed Deep Learning: Survey and Quantitative Evaluation

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

KAUST

ABSTRACT

Powerful computer clusters are used nowadays to train complex deep neural networks (DNN) on large datasets. Distributed training workloads increasingly become 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). We also propose a unified framework and API that allows for consistent and easy implementation of compressed communication on popular machine learning toolkits. We instantiate our API 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.

1. INTRODUCTION

Deep Neural Networks (DNNs) are becoming more complex. For example, ResNet-152 has 152 layers and 60.2M parameters [28], VGG-19 has 19 layers and 143M parameters [75], while BERT-Large has 24 layers, 16 attention heads and 340M parameters [20]. Combined with the large sizes of the training sets, parallelism during the training phase becomes a necessity. Consequently, popular deep learning toolkits, including TensorFlow [1], PyTorch [65] and Caffe [37], 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).

[1]Model and pipeline parallelism [56], which partition one replica of the model to multiple compute nodes, is orthogonal to data parallelism, but outside the scope of this paper.

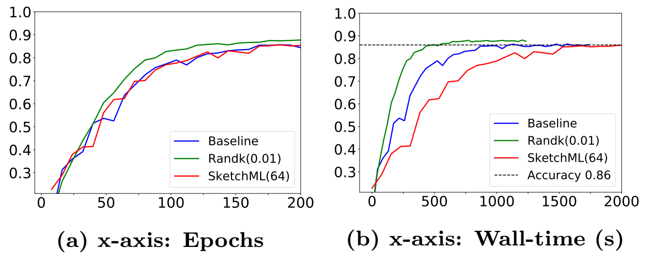


Figure 1: Top-1 accuracy for VGG16 on CIFAR-10 with TensorFlow on 8 workers via 25 Gbps network links. In (b) Randk converges in 450s, but SketchML needs 1500s.

Since the above-mentioned communication involves large amounts of data, the network becomes the bottleneck [52, 63, 71]. Luo et al. [52] 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.

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) [68] 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* [3, 10, 19, 41, 72, 87], which reduces the number of bits of each element in the gradient tensor (e.g., cast float32 to float8); (*ii*) *Sparsification* [2, 50, 76, 78, 82, 86], which transmits only a few elements per tensor (e.g., only the top-k largest elements); (*iii*) *Hybrid* methods [8, 22, 38, 48, 77], which combine quantization with sparsification; and (*iv*) *Low-rank* methods [14, 83, 84, 91], 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 §5 for details) running on 8 workers with NVIDIA V100 GPUs and 25 Gbps network. Two common compression methods, Random-k [76] and SketchML [38], 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. Supported by a back-of-the-envelop calculation of the reduced volume of transferred data, they conclude

Table 1: **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 15 methods on TensorFlow and PyTorch.**

	Compression	Ref.	Similar Methods	$\|\tilde{g}\|_0$	Nature of Q	EF-On	Implementation
Quantization	8-bit quantization	[19]		$\|g\|_0$	Det	✓	TFlow
	1-bit SGD	[72]	[10, 22, 77]	$\|g\|_0$	Det	✓	TFlow, PyTorch
	SignSGD	[10]	[72, 94]	$\|g\|_0$	Det	✗	TFlow, PyTorch
	SIGNUM	[11]	[10, 94]	$\|g\|_0$	Det	✗	TFlow, PyTorch
	QSGD	[3]	[32, 84, 87, 88, 92, 93]	$\|g\|_0$	Rand	✗	TFlow, PyTorch
	LPC-SVRG	[92]	[3, 32, 93]	$\|g\|_0$	Rand		
	Natural	[32]	[3, 92, 93]	$\|g\|_0$	Rand	✓	TFlow, PyTorch
	TernGrad	[87]	[3, 84, 92]	$\|g\|_0$	Rand	✗	TFlow, PyTorch
	EFsignSGD	[41]	[80, 94]	$\|g\|_0$	–NA–	✓	TFlow, PyTorch
	INCEPTIONN	[47]		$\|g\|_0$	Det	✗	TFlow
Sparsification	Random-k	[76]		k	Rand	✓	TFlow, PyTorch
	Top-k	[2]	[4, 76]	k	Det	✓	TFlow, PyTorch
	Threshold-v	[24]	[2]	Adaptive	Det	✓	TFlow, PyTorch
	Deep Gradient (DGC)	[50]	[78]	Adaptive	Det	✓	TFlow, PyTorch
	Sparse Gradient (SGC)	[78]	[50]	Adaptive	Det		
	Adaptive sparsification	[86]	[84]	Adaptive	Rand		
	Variance-based sparsification	[82]		Adaptive	Det		
	Sketched-SGD	[34]	[2, 4, 76]	k	Det		
Hybrid	Hard threshold SGD	[77]	[10, 22, 72]	Adaptive	Det		
	Adaptive threshold SGD	[22]	[10, 72, 77]	Adaptive	Det	✓	TFlow
	SketchML	[38]	[22, 77]	Adaptive	Rand	✓	TFlow
	3LC	[48]	[87]	Adaptive	Det		
	Qsparse-local-SGD	[8]		Adaptive	Rand		
LowRank	ATOMO	[84]	[86]	sparsity budget	Rand		
	GradiVeQ	[91]	[84]	$(m+L)r$	Det		
	PowerSGD	[83]	[14]	$(m+L)r$	Det	✓	TFlow, PyTorch
	GradZip	[14]	[83]	$(m+L)r$	Det		

that both compression methods perform well. 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 and is obviously preferable than the baseline that requires 850 s. Interestingly, SketchML converges after 1500 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. Our work can benefit researchers, who can easily implement novel methods using our API and evaluate them on a standard testbed, as well as practitioners who can investigate the trade-offs and select the method that suits the characteristics of their particular model and dataset. Our contributions are:

Survey. In §3, we present a comprehensive survey of the most influential works in compressed gradient communication, including quantization, sparsification, hybrid and low-rank methods, biased and unbiased, with or without memory. Refer to Table 1 for a summary.

Framework and API. In §4, we propose 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 our API in TensorFlow and PyTorch.

Implementation. With our API, we implement on TensorFlow and PyTorch 16 popular methods (see Table 1) that represent most of the spectrum. 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 compressed communication methods.

Quantitative evaluation. In §5, 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 compute overhead.

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 [61] or NCCL [57]) 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

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

computational overhead of compression/decompression; this issue is more pronounced for faster networks.

2. BACKGROUND

We focus on *data-parallel* distributed training [3, 10, 45, 79, 88], 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 [73]) in a peer-to-peer topology.[3]

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

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

where n is the number of workers. Each worker has a local copy of the model and has access to a partition of the training data. The workers jointly update the model parameters $x \in \mathbb{R}^d$, where d corresponds to the number of parameters. Typically, f_i is an expectation function defined on a random variable that samples the data.

Consider a deep neural network (DNN), and let $x \overset{\text{def}}{=} \{W, b\}$ 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

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

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; finally, uses the aggregated global state to update its parameters x.

Stochastic gradient descent (SGD). SGD [12, 68] 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*. At each iteration, Polyak [64] updates the momentum as: $z_{k+1} = \gamma z_k - \eta_k g_k$, where

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

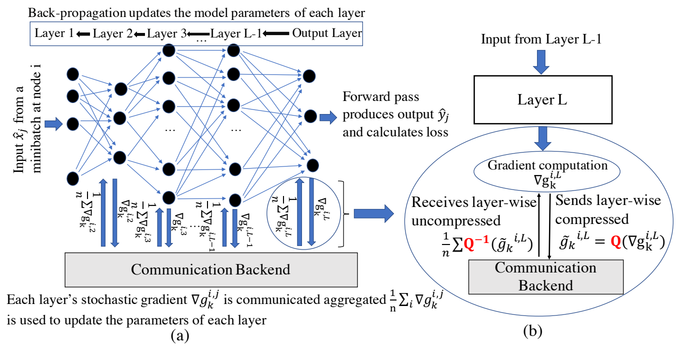

Back-propagation updates the model parameters of each layer
Layer 1 ⟵ Layer 2 ⟵ Layer 3 ⟵ … Layer L-1 ⟵ Output Layer
Input $\hat{x}_j$ from a minibatch at node i
Forward pass produces output $\hat{y}_j$ and calculates loss
$\frac{1}{n}\sum\nabla g_k^{i,2}$
$\nabla g_k^{i,2}$
$\frac{1}{n}\sum\nabla g_k^{i,3}$
$\nabla g_k^{i,3}$
$\frac{1}{n}\sum\nabla g_k^{i,L-1}$
$\nabla g_k^{i,L-1}$
$\frac{1}{n}\sum\nabla g_k^{i,L}$
$\nabla g_k^{i,L}$
Communication Backend
Each layer's stochastic gradient $\nabla g_k^{i,j}$ is communicated aggregated $\frac{1}{n}\sum_i \nabla g_k^{i,j}$ is used to update the parameters of each layer
(a)
Input from Layer L-1
Layer L
Gradient computation $\nabla g_k^{i,L}$
Receives layer-wise uncompressed $\frac{1}{n}\sum Q^{-1}(\tilde{g}_k{}^{i,L})$
Sends layer-wise compressed $\tilde{g}_k{}^{i,L} = Q(\nabla g_k^{i,L})$
Communication Backend
(b)

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

$0 \le \gamma \le 1$. Nesterov [58] proposes an alternative momentum update rule, where the gradient g is computed at a look-ahead point $x_k + \gamma z_k$ as: $z_{k+1} = \gamma z_k - \eta_k g(x_k + \gamma z_k)$. Both momentum approaches are used to modify Equation (3) of SGD to: $x_{k+1} = x_k + z_{k+1}$. In addition to SGD, several *accelerated* versions, such as ADAM [42], or ADAGrad [23], are used for DNN training.

3. GRADIENT COMPRESSION

This paper focuses 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 errors accumulated due to the lossy compression.

We classify gradient compression techniques into four categories, shown in Table 1: quantization, sparsification, hybrid and low-rank. The rest of this section presents the details.

3.1 Quantization

There are two broad classes of quantization compressors: (i) *limited-bit*, where each gradient element is mapped to fewer bits (e.g., truncation); and (ii) *codebook-based*, where Q projects the gradient coordinates into predefined code-words.

8-bit quantization. This is a limited-bit technique, proposed by Dettmers [19]. It maps each float32 element of the gradient to 8 bits: 1 sign, 3 exponent and 4 mantissa bits. To minimize precision loss, the work also proposed a dynamic scheme, where exponent bits range from 0 to 6.

1-bit SGD. Seide et al. [72] proposed this extreme form of limited-bit 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 introduced a memory mechanism m_k to compensate for the accumulated error. Let $\tilde{g}_k$ be the compressed gradient at iteration k. Then $\tilde{g}_k = Q(g_k + m_k)$, where $m_k = g_k - Q^{-1}(\tilde{g}_k)$.

SignSGD, SIGNUM and EFsignSGD. SignSGD [10] is also a limited-bit method that transmits the sign of gradient elements by quantizing the negative components to

[4]For the orthogonal topic of *parameter* compression, see §6.
[5]For simplicity, we will omit i, L from $g_k^{i,L}$ when possible.

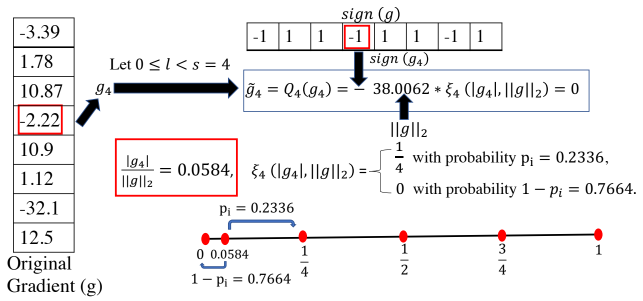


Figure 3: QSGD example with $s = 4$, $l = 3$. The possible code-words are $0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1$ and are represented by 3-bits.

-1 and the others to 1. SIGNUM [11] is a momentum version (see §2) of SignSGD. Each worker i updates the momentum term z_k^i via: $z_k^i = (1-\delta)g_k^i + \delta z_k^i$ and transmits $\text{sign}(z_k^i)$. All n workers receive the aggregated vectors: $\text{sign}\left(\sum_{i=1}^n \text{sign}(z_k^i)\right)$ and update the model parameter via: $x_k - \eta\left(\text{sign}\left(\sum_{i=1}^n \text{sign}(z_k^i) + \lambda x_k\right)\right)$, where $\lambda \geq 0$ is a weight decay factor. If $\delta = \lambda = 0$, then SIGNUM becomes SignSGD.

In some cases, SignSGD does not converge during DNN training. Karimreddy et al. [41] proposed EFsignSGD, that incorporates a memory mechanism; their method alleviates the convergence issues. Zheng et al. [94] extended the error feedback approach to a bidirectional [80] blockwise scheme with Nesterov momentum.

Quantized SGD. QSGD is a codebook-based stochastic compression scheme by Alistarh et al. [3]. Wu et al. [88] combined QSGD with error feedback. QSGD quantizes each component of the stochastic gradient via randomized rounding to a discrete set of values (i.e., code-words) that preserve the statistical properties of the original stochastic gradient. Formally, a gradient component $g[i]$ is quantized to:

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

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 [92] is a quantized version of the classic SVRG [40] algorithm. LPC-SVRG 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 either end-point of the interval:

$$\tilde{g}[i] = \begin{cases} \varepsilon & \text{with probability } p_i = \frac{\varepsilon + \delta - g[i]}{\delta} \\ \varepsilon + \delta & \text{otherwise} \end{cases} \tag{4}$$

where $\varepsilon \in \left\{-2^{w-1}\delta, \dots, -\delta, 0, \delta, \dots, (2^{w-1}-1)\delta\right\}$. The same authors proposed an accelerated version with Katyusha momentum [5]. Quantized-SVRG [3] is a related method with a variance reduction mechanism. Horváth et al. [32] proposed a similar scheme, called natural compression, that rounds the input to one of the two closest integer powers of 2.

INCEPTIONN. INCEPTIONN [47] is an adaptive quantization technique that leverages the characteristics of floating-point gradient values as well as the benefits of in-network acceleration. Based on the range of gradient values, each 32-bit floating-point gradient value is quantized into four

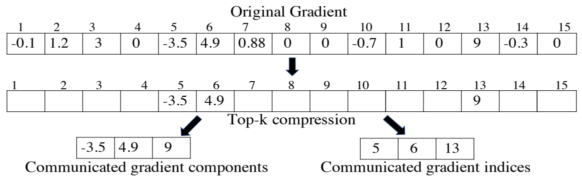


Figure 4: Example Top-k compression: 20% of the gradient components and corresponding indices are sent.

different levels (32bit, 16bit, 8bit, 0bit) with an extra 2-bit tag indicating the compression level. Notably, this work is implemented in FPGA-based Network Interface Cards (NICs) to reduce the compressor's computational overheads.

Ternary gradient. TernGrad [87] uses three numerical values $\{-1, 0, 1\}$ scaled by the infinite norm of the stochastic gradient to obtain the quantized stochastic gradient. First, given a stochastic gradient g, the elements of the bit-mask b is selected with probability: $P(b_i = 1|g[i]) = g[i]/\|g\|_\infty$. By using this bit-mask, the stochastic gradient g is quantized to $\tilde{g} = \|g\|_\infty \text{sign}(g) \odot b$ ($\odot$ is element-wise product).

Remark 1. The quantization schemes in [3, 32, 87, 93] are special cases of Equation (4). In particular, for $s = \|g\|_2/\|g\|_\infty$ TernGrad is equivalent to QSGD. TernGrad tends to achieve better convergence rate if $\|g\|_2/\|g\|_\infty \to \sqrt{d}$, that is, the gradient components are evenly distributed [39].

3.2 Sparsification

Sparcification methods select only a subset of the elements of the original stochastic gradient g, resulting in a sparse vector. The selection process can be formalized as follows: 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.

We classify sparsification compressors into: (i) *fixed dimension*, where the number of selected elements is fixed (e.g., 20% of the original gradient); and (ii) *variable dimension*, where the number of components is adapted during runtime.

Random-k. This is a fixed-dimension sparsification method [76]. 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, Threshold-v and Sketched-SGD. Top-k [2, 4] is a fixed-dimension sparsification method. Using the above-mentioned formalization, bit mask b is selected 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. [76] also proposed a similar scheme with memory. Unlike selecting the k largest values in Top-k, Threshold-v selects the elements whose absolute values are larger than a fixed threshold value [24]. However, an appropriate threshold value is hard to determine as it depends on the model. Ivkin et al. [34] proposed Sketched-SGD, which uses count-sketch to select the "heavy hitters", which approximate the Top-k components of the gradient vectors. They also proposed a version of sketched-SGD with memory.

Deep gradient compression (DGC) and sparse gradient compression (SGC). In DGC [50] 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. 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. In the same spirit, in SGC [78], the local gradient g_k^i at each worker i is combined with the momentum via: $u_k^i = u_{k-1}^i + \epsilon g_k^i$, Then, the gradient is accumulated via: $v_k^i = \alpha u_{k-1}^i + u_k^i$, and sparsified similarly to DGC.

Adaptive and variance-based sparsification. Wangni et al. [86] observed that the variance of the gradient affects the convergence rate, and proposed an unbiased sparse coding to maximize sparsity and control the variance. They assign a probability p_i and a mask $Z_i \in \{0, 1\}$ to each component $g[i]$ of the gradient to obtain the compressed gradient element $\tilde{g}[i] = Z_i \frac{g[i]}{p_i}$, where $P\{Z_i = 1\} = p_i$. Tsuzuku et al. [82] also use the gradient element variance to sparsify. A stochastic gradient element is transmitted if the measurement of its variance is less than a predefined threshold.

3.3 Hybird compressors

Hybrid methods combine quantization with sparsification.

Qsparse local SGD. Basu et al. [8] combine quantization with Top-k or Random-k sparsification. They implement synchronous and asynchronous versions with error feedback.

Hard threshold SGD. Strom et al. [77] 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.

Adaptive threshold SGD (Adaptive). Instead of a fixed τ, Adaptive [22] uses a ratio $\alpha < 1$ of the proportion of negative and non-negative gradient elements that will be sent. The algorithm samples the gradient to determine dynamically for each mini-batch two thresholds τ^- and τ^+ that satisfy the α-ratio. The rest of the process is similar to [72]. Adaptive employs error feedback similar to [72, 77].

SketchML. Jiang et al. [38] proposed a sketch-based compression of the stochastic gradients. The algorithm selects only the non-zero elements of the gradient (i.e., sparsification) and builds a non-uniform quantile sketch [26]. 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. For a gradient vector g, 3LC [48] 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 $(1/\mathcal{M})g$. Larger s generates a sparser output which can be further compressed by an aggressive lossless encoding. Additionally, it uses memory compensation.

3.4 Low-rank decomposition

DNNs are over-parameterized and exhibit low-rank structure [7, 46]. Based on this observation [36, 60], 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.

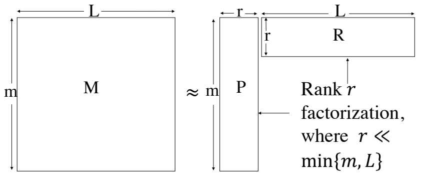


Figure 5: Low-rank compression: Matrix M is decomposed into two low-rank matrices P, R each of rank r.

ATOMO and GradiVeQ. ATOMO [84] 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) [91] is also based on SVD.

Remark 2. 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 [83] 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 [14], employs an extra regularizer $\|P\|_F^2 + \|R\|_F^2$ and uses an alternating direction method to obtain factors P and R.

3.5 General comment on convergence

For a non-convex function f (as it is the case with DNNs), it is typical to show that quantity $\min_{k\in[K]} \mathbb{E}(\|\nabla f_k\|^2) \to 0$ as $K \to \infty$. While some compressed distributed SGD methods were analyzed in the non-convex setup, some papers only provide the convergence guarantee when f is convex, under standard assumptions (e.g., see [24]). With compressed and distributed SGD, the majority of the aforementioned work showed the classical convergence rate $O(1/\sqrt{K})$ for non-convex functions. See [24] for a general convergence analysis of distributed SGD.

4. A UNIFIED FRAMEWORK

We now describe a *unified framework* for compressed communication for distributed SGD and the programming interface it offers. We then instantiate the framework within the two most popular ML toolkits, TensorFlow and PyTorch, and highlight implementation details. The main merit of our framework is that it encompasses a wide range of compression methods (capturing all the methods discussed in §3) and yet it exposes a simple programming interface with which we can implement compression methods succinctly. As such, our framework enables a reference for a fair comparative

Algorithm 1 Distributed Training Loop

Input: Number of nodes n, learning rate η_k, compression Q and decompression Q^{-1} operators, 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:         $[\tilde{g}_k^1, \tilde{g}_k^2, \cdots, \tilde{g}_k^n] = \texttt{Broadcast}(\tilde{g}_k^i) \mid \texttt{Allgather}(\tilde{g}_k^i)$
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}
```

evaluation across diverse methods and serves as a platform for rapid prototyping of new ones.

4.1 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 synchronizes with other nodes during communication phases. To ease presentation, we only give a high-level description of the process through simple notation. Additional technical details will follow in subsequent sections.

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

- $Q(\cdot)$ 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, the node combines g_k^i with its memory m_k^i via $\phi(\cdot)$.[6] Next, the node applies the compression operator Q on $\phi(g_k^i, m_k^i)$ to produce $\tilde{g}_k^i$ (Line 5). The 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 | 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 using 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 [23], ADAM [42], can be used as optimizer in Algorithm 1. Our experiments use different optimizers, including SGD, RMSProp and SGD with momentum.

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

$$\begin{aligned} \phi(m_k^i, g_k^i) &= \beta m_k^i + \gamma g_k^i \\ \psi(m_k^i, g_k^i, \tilde{g}_k^i) &= \phi(m_k^i, g_k^i) - \tilde{g}_k^i \end{aligned} \tag{5}$$

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; we consider a sensitivity analysis out of the scope.

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. We attempted to integrate with BytePS [13], which is meant to work as a drop-in replacement for Horovod; However, its API did not allow for easy customization of the aggregate function at the parameter server. We leave this extension to future work.

4.2 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 can carry 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, the context might carry the shape or 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 → [comp], ctx
```

Our framework provides defaults for `memory_compensate`, `memory_update` (Equation 5), and `aggregate` (the average function). The user needs to implement `compress` and `decompress` for a specific compression method, and indicate to the framework which communication strategy to use.

Compression typically produces tensors of different dimensions or data types (e.g., int8, float16) than the original ones. For instance, sparsification results in smaller tensors while quantization results in either different data types or bit-packed elements with the original data types. As these

[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 $\psi(m_k^i, g_k^i, \tilde{g}_k^i) = 0$.

Table 2: API utility 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

manipulations are common across several methods, our API implements the primitive functions of Table 2.

We aim to support both TensorFlow and PyTorch, which unfortunately 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. For simplicity, we adopt Python API; however, this does not prevent the user from integrating a custom C++ operation. We note that the Python API is what is typically used by model creators and is backed by efficient low-level kernels for GPUs or other HW accelerators.

Communication operations. We leverage the Horovod API [31] 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 [25].

Communication strategies. We support two types of communication strategies. In general, `Allreduce` is a more efficient operation than `Allgather | Broadcast.` 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, `Allgather` and `Broadcast` do not perform any aggregation and support input tensors of different forms.[7] 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.

4.3 Implemented compressors

Starting from the compression methods surveyed in §3, we now identify a subset for implementation within our framework. We focus on compressors that are representative of particular classes. We implement and evaluate 16 compressors (see list in Table 1).

We implement the compressors by faithfully following the algorithm descriptions presented in their corresponding papers. Where available we draw from publicly available implementations of these methods; however, in many cases these are not released, although we reached out to the original authors to acquire them. As such, we remark that our implementation is a best-effort approach that reflects the intention of the respective methods, although it might not be as efficient as the original ones. We avoid creating custom C++ tensor operations for efficiency because we find that the Python API is functionally sufficient and because it would have required an excessive effort given the large number of methods we implement. Despite there might be some differences in implementation techniques, we are careful to reproduce the original accuracy results in their respective papers in order to validate our implementation. Below, we highlight noteworthy implementation details.

Implementing quantization. Consider a quantization method implemented through our API. `quantize` converts the original 32-bit floating-point values into a lower-bit representation. The context stores additional information (such as mean and different norms) needed to dequantize. In some scenarios, `pack` can further compress the representation by encoding several values originally stored in 32-bit into a single 32-bit value. For instance, a 1-bit quantization can pack 32 values into a single 32-bit integer. `dequantize` transforms quantized values to an approximation of the original values. When multiple values are tightly packed, `unpack` first decodes them into their original representation.

Implementing sparsification. Consider a sparsification method implemented through our API. To ease implementation, `sparsify` flattens the original tensor into a rank-1 tensor and indexes elements using this representation. The context stores the original tensor dimension. `sparsify` selects partial elements (m out of d) of the gradient, and creates two rank-1 tensors ($1 \times m$) to represent the values and indices of selected elements, respectively. `desparsify` stores the sparsified values into a rank-1 tensor of size d according to their indices, fills missing values with zeros, and reshapes the tensor as per the original shape (obtained by `ctx`).

Implementing Adaptive [22]. Adaptive splits the gradient into a positive- and a negative-value part.We apply `quantize` to encode values in a ternary format and we separate the +1 and -1 values. We use `sparsify` based on a dynamically determined threshold to select elements according to a sparsification ratio k. As values are all ones, we omit communicating them and send mean and selected indices of each part. Then, for decompression, we use `desparsify` to restore the dequantized values scaled by the mean for each part. Finally, we aggregate the gradient tensor as the sum of the positive and negative parts.

Implementing DGC [50]. The momentum correction used in DGC is similar to memory compensation. We implement it using custom 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.

5. EXPERIMENTAL EVALUATION

In this evaluation of 16 compressors, we focus on measuring and analyzing the performance gains across a range of representative benchmarks. Although it is not possible to generalize our conclusions to arbitrary conditions, our results cover 5 benchmarks, 7 model architectures and 4 ML tasks (image classification, segmentation, recommendation, and language modeling). We believe these are sufficient to offer insights and draw lessons that are broadly applicable.

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/s). 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, the extent to which the model architecture is communication bound).

[7]NCCL `Allgather` requires same type and dimension, which precludes us from using NCCL at times.

Table 3: Summary of the benchmarks used in this work.

Task	Model	Dataset	Training parameters	Gradient vectors	Epochs	Quality metric	Baseline quality
Image Classification	ResNet-20 [28]	CIFAR-10 [44]	269,467	51	328	Top-1 Accuracy	90.86%
	DenseNet40-K12 [33]	CIFAR-10 [44]	357,491	158	328		92.07%
	Custom ResNet-9 [62]	CIFAR-10 [44]	6,573,120	25	24		91.67%
	VGG16 [75]	CIFAR-10 [44]	14,982,987	30	328		86.32%
	ResNet-50 [28]	ImageNet [18]	25,559,081	161	90		75.37%
	VGG19 [75]	ImageNet [18]	143,671,337	38	90		68.90%
Recommendation	NCF [29]	Movielens-20M [55]	31,832,577	10	30	Best Hit Rate	95.98%
Language Modeling	LSTM [30]	PTB [54]	19,775,200	7	25	Test Perplexity	100.168
Image Segmentation	U-Net [69]	DAGM2007 [16]	1,850,305	46	2,500	Intersection over Union (IoU)	96.4%

For the above metrics, our evaluation explores their trade-off with model quality (e.g., accuracy). In particular, we address the following questions:

- Is there a definite trade-off between model quality and data volume or throughput?
- Does a larger data volume (intuitively, more information) result in higher model quality?
- To what extent do computational overheads of compression influence performance gains?
- How do number of workers, network link capacity, or the hosting ML framework (TensorFlow or PyTorch) affect performance gains?
- Does error feedback (EF) work throughout?
- Are there distinctly better or worse compression methods?
- What is the effect of using a different optimizer (e.g., SGD or SGD with momentum)?

Our main observations are as follows:

- There is no evidence that any particular compression method outperforms every other methods across all experimental scenarios.
- The computational overheads of compression are not negligible. In several cases, at relatively high communication link rates (10 Gbps or more), not doing any compression results in faster training times. This observation agrees with the results reported in [47, 53], which also show that computational overheads can make compression inefficient in practice.
- EF is widely applicable to sparsification and improves the accuracy by 4.38% on average, and up to 72.98%. However, it can lower accuracy in certain cases or even prevent model convergence. Further, EF comes with memory overheads, which can prevent using the default mini-batch sizes in some cases, thus lowering overall efficiency.
- A 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 (NCCL vs. OpenMPI).

We now describe our experimental setup and present the results. We then discuss them and draw our conclusions.

5.1 Experimental setup and methodology

Environment. We run most of the experiments on 8 dedicated server-grade machines while offloading a subset of accuracy experiments to a shared cluster. We only report time-insensitive metrics (e.g., accuracy, data volume) from the experiments on the shared cluster. The dedicated machines run Ubuntu 18.04.2 LTS and Kernel v.4.15.0-74, are equipped with 16-Core Intel Xeon Silver 4112 running at 2.6 GHz, 512 GB of RAM, one NVIDIA Tesla V100 GPU card with 16 GB of on-board memory, and 10 and 25 Gbps network interface cards. The shared cluster has a heterogeneous group of nodes and we use those 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, NCCL 2.4.8.

Benchmarks. We use publicly available, industry-standard benchmarks from TensorFlow [51, 81] and NVIDIA [59]. Table 3 gives an overview of the models we train, including their size and dataset dimensions. These benchmarks span 4 common ML 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 3.

Methodology. We run each experiment for a fixed number of training epochs (complete iterations over the training set) according to every benchmark's specification to reach convergence. 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*. By default, as the optimizer, "Baseline" uses SGD with momentum for image classification, RMSProp for segmentation, ADAM for recommendation, SGD for language modeling. In each benchmark, compressors use the same optimizer as the baseline, except for image classification whereby PowerSGD, Random-k, DGC, SignSGD, SIGNUM use vanilla SGD (compare to "Baseline(sgd)") as it gets better quality.

When reporting relative results, they are normalized to the relevant metrics measured for the baseline case. We ensure our baselines converge to state-of-the-art results (Table 3).

We report results from one representative experiment with each method. 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). However, we observe that the model quality only has small variations once training converges. Due to space limit, we focus mainly on TensorFlow results. We illustrate an example with PyTorch and comment on the differences where relevant.

For each compressor, we generally report only results with EF on or off according to which one worked best in our exhaustive analysis on the image classification task. Table 1 indicates where EF is used.

Unless otherwise noted, we use the default configuration with each benchmark. But the performance of compressors is sensitive to a range of factors such as the optimizer (e.g., SGD or SGD with momentum), standard hyperparameters (e.g, mini-batch size, learning rate) and compressor-specific parameters. For instance, several compressors allow for a varying degree of compression. Where practical, we experiment with multiple values of these factors and report on their effects (e.g., Random-$k(0.1)$ means $k = 10\%$). However, a complete sensitivity analysis to these factors is out of scope.

We report throughput as the average measured at steady state by considering the last 100 iterations during training as this metric is fairly stable due to performance predictability of GPUs (per-worker mini-batch size is kept constant). We measure data volume in bytes for each invocation of a collective communication primitive 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).

We keep all hyperparameters the same as with the baseline except for the cases where specific hyperparameter settings are stated in a compressor's original paper. Specifically, for EFsignSGD, we set $\beta = 1$ and γ equal to the initial learning rate.

Finally, we remark that the results shown below refer to experiments with 10 Gbps network links and using OpenMPI over TCP as collective library.[8] We also run experiments with 25 Gbps network links and we observe mild improvements in throughput (on average, 1.3%) whereas the relative improvement compared to the baseline in the two scenarios is minimal. Thus, 10 Gbps links are our default setting and we analyze later the effects of this configuration.

5.2 Model quality vs. training throughput

We first observe the effects of compression on model quality as a function of throughput achieved. Intuitively, we expect that more aggressive compressors would tend to make a larger trade-off with quality. Figure 6 quantifies this relation across different benchmarks and for each compressor. Note that VGG results are not shown in for brevity. Compressors that achieve poor quality (below the y-axis cut-off) or do not converge are not shown. Throughput is normalized to the baseline case (highlighted with a vertical line in red).

In general, we observe that training converges to solutions with quality metrics comparable to the respective baselines in most cases. In some cases, and perhaps surprisingly, the model quality is slightly higher than the baseline. We believe this can be attributed to the stochastic nature of the process and that compression can in some cases cancel out bad gradient directions. For example, DGC [50] also reported 0.37% better accuracy than baseline.

With respect to throughput, in many cases, most compressors achieve a throughput that is lower 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.

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, it illustrates that for both variable-dimension and fixed-dimension compressors with a 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. We elaborate more on the trade-off with data volume below.

Takeaways. No method consistently performs well across all benchmarks and there is no strong correlation between throughput and model quality. Hence, the quality-throughput trade-off is not straightforward to infer; implying that compression should be chosen carefully for a given benchmark so as to employ a compressor that achieves a desired trade-off.

5.3 Model quality vs. data volume

We now consider the model quality vs. data volume trade-off. This reflects the communication complexity of each compressor from a fundamental perspective – that is, we compare compressors in terms of communication cost[9] regardless of the associated computational overheads of compression. Figure 7 shows for each compressor its best model quality and the average communicated data volume per iteration to achieve that quality.

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 task as shown in Figure 7d. However, we observe that for some compressors, a higher data volume results in lower model quality. For instance, this is the case with Adaptive across all benchmarks; Sketch-ML and DGC for the CIFAR-10 image classification task (Figures 7a and 7b); Threshold-v for the image segmentation task (Figure 7f).

Takeaways. There are several cases where for a varying degree of compression in a given compressor, the model quality increases as more compression is performed. This suggests that the quality vs. data volume trade-off is non trivial and, as mentioned before, compression should be tuned carefully to deliver the best benefits for a given scenario.

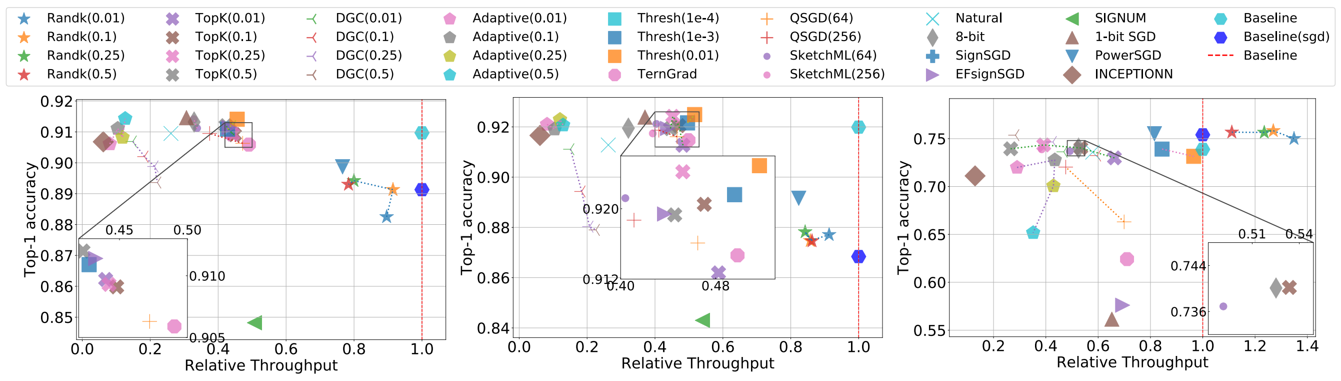
5.4 Model quality vs. error feedback

Table 1 indicates with a ✓ where EF is applied; this is because we find that EF improves accuracy in general 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, this issue is known and was fixed by design in EFsignSGD. That said, we observe that without EF, SignSGD and SIGNUM achieve poor

[8]NCCL is faster than OpenMPI but, as mentioned in §4.2, NCCL constrains input sizes and this prevents a fair comparison as NCCL is applicable for a subset of compressors.

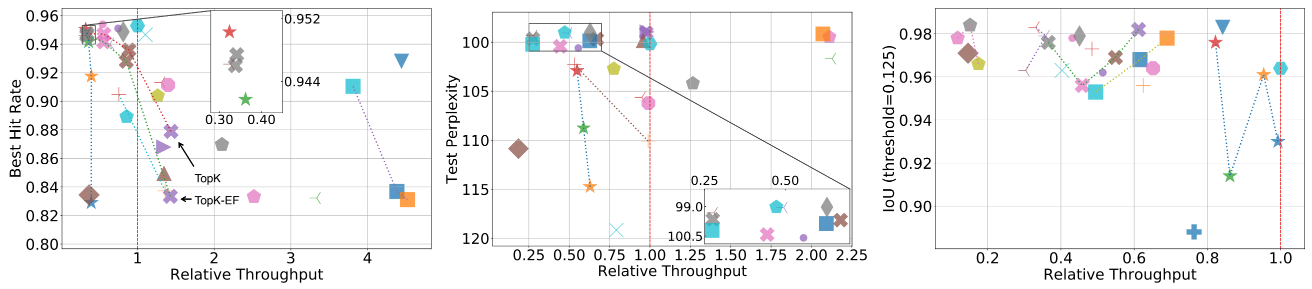
[9]Because we do not implement packing, the data volumes are inflated for quantization methods. However, in a relative sense our results still hold.



(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

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

quality solutions in many (when not every) benchmark. One reason for this is that these compressors seem to require a non-standard learning rate. For instance, we use 0.001 for CIFAR-10 as per the original paper. However, we did not customize it for other benchmarks and the results did not converge. EFsignSGD does not have this issue.

Interestingly, and exclusively for the recommendation task results, we see that 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 as it is the most severe case.

Takeaways. We show that EF is not a turn-key solution to improving convergence and we establish the need for further studying the impact of EF on compressors as its performance can vary across types of compressors and benchmarks.

5.5 Scaling efficiency

We measure the impact of compression on scaling efficiency. Figure 8 shows how throughput varies as we scale the workers from 2 to 8. The shaded gray bars are discussed in the next section. The horizontal dashed lines depict the baseline scalability, which exhibits less than ideal (linear) scaling behavior. We observe that all compressors scale with increasing workers for these benchmarks.

The results indicate that the scaling rate for most compressors is lower than the baseline, except for the communication-intensive VGG19 benchmark (Figure 8d). However, a few compressors exhibit scaling behavior close to the baseline: Random-k and PowerSGD with CIFAR-10 (Figures 8a and 8b); Random-k, Threshold-v and PowerSGD with ImageNet (Figure 8c) and image segmentation (Figure 8f).

Moreover, in the recommendation task, most compressors achieve better scaling rate than the baseline while PowerSGD and Threshold-v scale close to linear behavior (Figure 8e). Similarly, in language modeling tasks, many compressors achieve better scaling rate than the baseline while PowerSGD, DGC, Adaptive, and Threshold-v scale close to linear behavior (Figure 10).

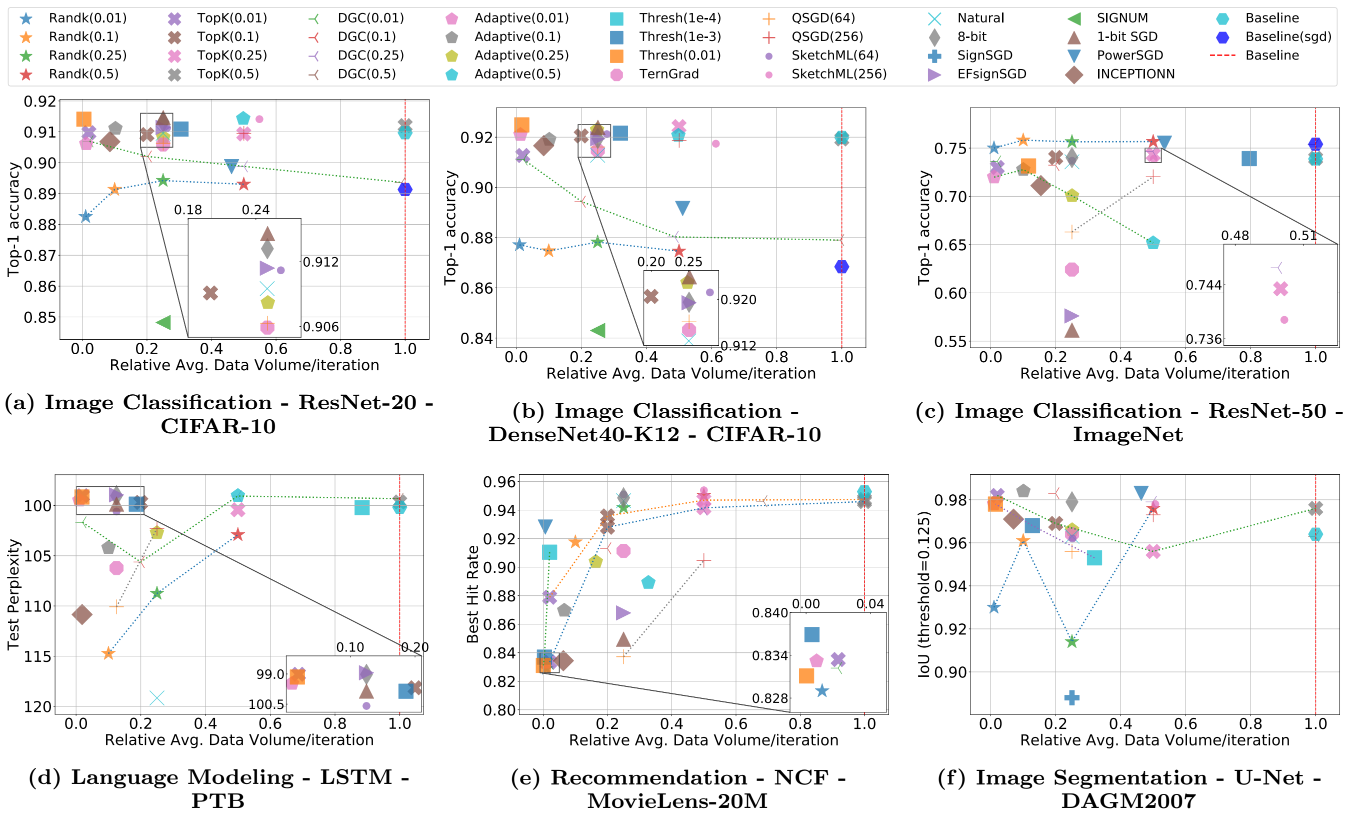
Finally, as mentioned, most compressors scale throughput better than the baseline for VGG19, recommendation, and language modeling benchmarks. Moreover, PowerSGD, DGC, and Adaptive not only significantly improve the training throughput but also have better scaling rate than the baseline.

Takeaways. These results support our previous observations that no method consistently performs well on all tasks.

5.6 Computational overheads of compression

Figure 8 shows gray bars that correspond to a throughput upper bound of each compressor. This is measured by skipping the `compress` and `decompress` calls while faithfully sending the data volume equivalent to what the compressor would produce. The difference between gray and corresponding colored bar captures overheads. We also contrast the results with a micro-benchmark experiment that measures the combined latency of `compress` and `decompress` in isolation; Figure 9 shows the results as a violin plot for 30 repetitions while placing operations both on GPU and on CPU.

Results show that compressors induce non-negligible overheads. Through profiling, we determine that sparsification compressors such as Random-k and Top-k predominantly im-



(a) Image Classification - ResNet-20 - CIFAR-10

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

(c) Image Classification - ResNet-50 - ImageNet

(d) Language Modeling - LSTM - PTB

(e) Recommendation - NCF - MovieLens-20M

(f) Image Segmentation - U-Net - DAGM2007

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

pose high overhead due to operations only available on CPU and (slow) sorting operations, respectively. Skech-ML imposes high overhead due to sketch operations. Natural Compression performs expensive bit-level operations. Other quantization methods (Terngrad, QSGD, 1-bit SGD, SignSGD) have mild overheads, but they are not the fastest during real training. This is in part because their data volume is inflated in our setting. Then, we observe that DGC and Adaptive have mild overheads but there is a large gap from their upper bound measures. We observe that Random-k generally performs well in terms of throughput in image classification (Figures 8a, 8b, and 8c) and segmentation (Figure 8f) tasks. However, it does poorly in recommendation (Figure 8e) and communication-intensive VGG19 (Figure 8d).

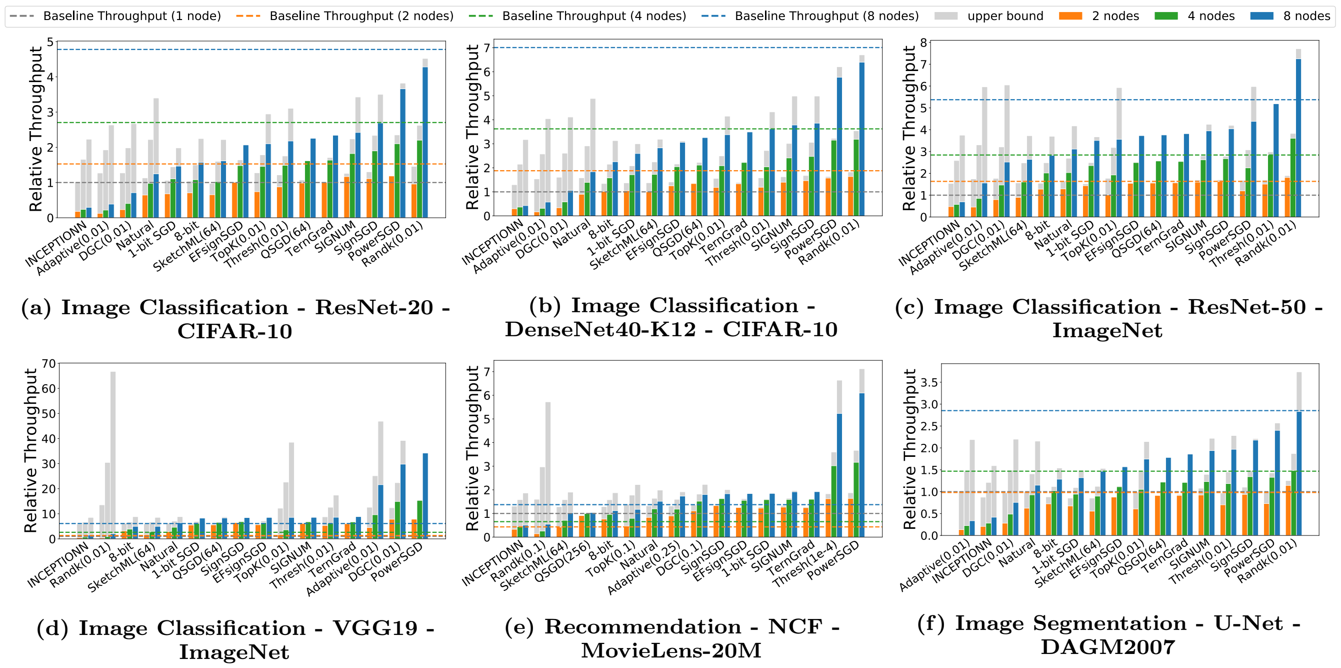
We investigate these cases using the timeline of low-level operations executing on GPU and find that: 1) Both Adaptive and DGC involve a loop (max 20 iterations) to adjust the threshold to best match the target ratio. This loop turns out to be very expensive; their throughput improved by $\approx 2\times$ by executing only one iteration. 2) As shown in Figure 9, Random-k shows high overhead as the `tf.random.shuffle` operation executes on the CPU due to lack of GPU kernel. However, during real training, Tensorflow can schedule device-host data transfer so that it overlaps with GPU computation and its overhead is at times mitigated. 3) 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. 4) 8-bit invokes a `find_bins` operation for each quantized value which, due to lack of a GPU implementation, is executed on the CPU. 5) We also observe that some methods rely on an 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).

Takeaways. Implementing compressors requires careful engineering to account for their intrinsic computational overheads, which cannot be discounted. In several cases, custom GPU code or well-optimized CPU code might be necessary to achieve high performance.

5.7 ML toolkit, transport and links

Figure 11 shows the throughput of different compressors in CIFAR-10 image classifcation task using PyTorch with different communication protocols (TCP and RDMA). We report that the training throughput is mostly consistent yet higher than what we observe for most compressors in the TensorFlow image classification tasks. For instance, throughput-wise, Random-k and PowerSGD are among the highest performing methods. When using RDMA instead of TCP as transport protocol, although the absolute throughput has decent gains for most compressors, we observe that the performance of Random-k relative to the baseline worsens.

In contrast, Figure 12 shows the relative throughput for the same experiment 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



(a) Image Classification - ResNet-20 - CIFAR-10

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

(c) Image Classification - ResNet-50 - ImageNet

(d) Image Classification - VGG19 - ImageNet

(e) Recommendation - NCF - MovieLens-20M

(f) Image Segmentation - U-Net - DAGM2007

Figure 8: Performance of compressors in terms of scaling efficiency and throughput upper bound (gray bars).

obtain a throughput speedup over the baseline.

Takeaways. Not all compression methods perform consistently when implemented in different frameworks and different transport protocols (i.e., throughput gains are impacted).

5.8 Summary and new research directions

We posit that gradient compression should only be used after extensive evaluation in a profiling environment before committing to it for production workloads.

We conclude that this area of research would benefit from increased attention into the following directions:

- Optimizing the scheduling of gradient compression by maximizing the overlap between the compression computations and forward pass of training iterations.
- Employing compression in network-constrained scenarios such as geo-distributed or federated learning settings.
- Offloading the compression overhead to a hardware accelerator (e.g., FPGA NICs [47]).
- Revisiting the underlying communication libraries to work efficiently with compressed data (e.g., NCCL `Allreduce` for sparse vectors).

6. RELATED WORK

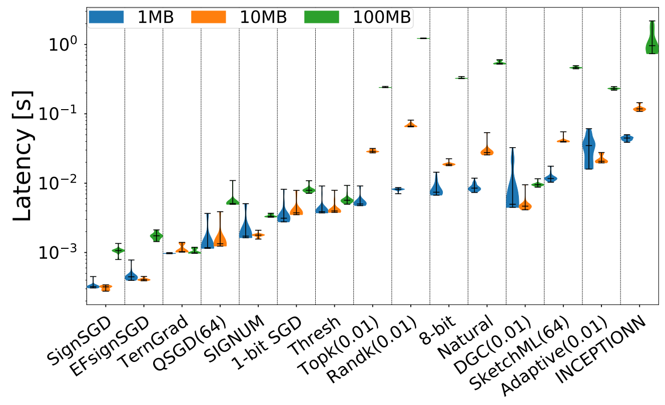
Yang [89] was one of the first to study the trade-off between computation and communication for distributed stochastic optimization. Since then, numerous related approaches have been proposed. We refer to the survey by Ben-Nun and Hoefler [9] for an in-depth understanding. Below we cite those that are more relevant to our work.

Compression for ad-hoc P2P overlays. Unlike our work, which assumes all-to-all aggregation semantics (e.g., AllReduce), others [39, 43, 79, 90] consider an ad-hoc peer-to-peer network overlay, where nodes communicate only with neighbours. These methods differ from the one we directly support primarily because they redifine the aggregation semantics to involve only a subset of workers at a time. Nevertheless, some use similar compression techniques, like sparsification and quantization. We leave it as future work to integrate in our framework's communication primitives that accommodate the P2P overlay setting.

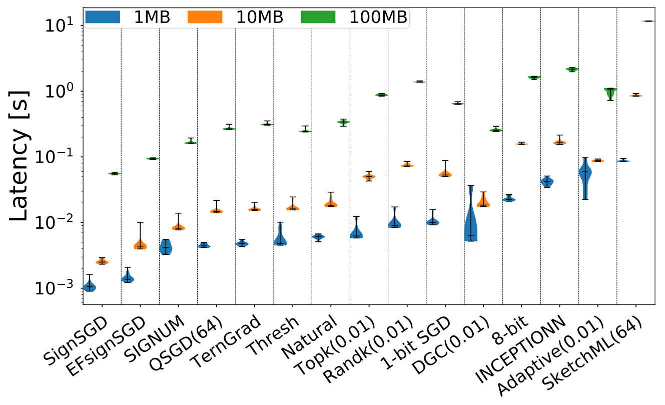
Fewer communication rounds. Some methods reduce the volume of transmitted data by communicating less often. CoCoA [35] is dual coordinate ascent algorithm that performs several local steps before communicating with other workers. Shamir et al. [74] do not compute or communicate explicitly second-order information for Newton-type optimization algorithm. Wang and Joshi [85] propose periodic averaging SGD, to update the local model at each worker node and then use periodic average to update the final parameters. For a non-asymptotic analysis for local-SGD with large step sizes we refer the readers to [21].

Asynchronous communication. Hogwild! [66] proposed an *asynchronous* parallel SGD where the computing nodes have access to shared memory and can modify the parameters at any time without locking. De Sa et al. [17] proposed a low-precision asynchronous SGD method. They develop various techniques to increase the throughput and improve the speed of low-precision SGD. They also implemented the low-precision SGD on an FPGA. Asynchronous communication is outside the scope of our paper.

Communication primitives. Popular communication libraries (OpenMPI, NCCL, etc.) do not support natively sparse data structures that are necessary for many compression methods. Renggli et al. propose a comprehensive framework, known as SparcML [67] that implements a stream structure to support sparse tensors. There also exists work that focus on lower-level communication primitives: Sapio et al. [71] proposed SwitchML, which uses a programmable network switch to implement in-network aggregation while packets are transmitted through the network. Without using



(a) Latency of GPU execution.



(b) Latency of CPU execution.

Figure 9: Latency of `compress` **and** `decompress` **(combined) for different compressors with a range of input sizes.**

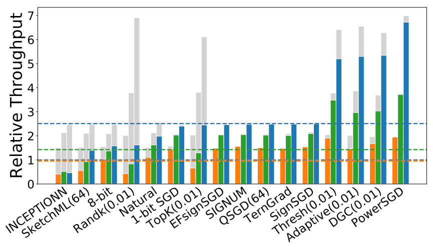


Figure 10: Performance of compressors for Language Modeling - LSTM - PTB. Legend in Figure 8.

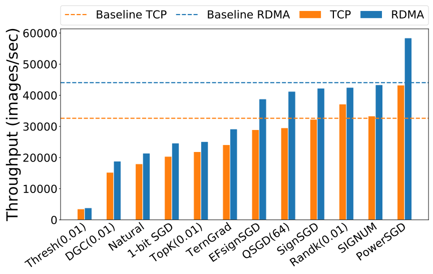


Figure 11: Throughput for ResNet-9 on CIFAR-10 contrasting TCP vs. RDMA performance in PyTorch.

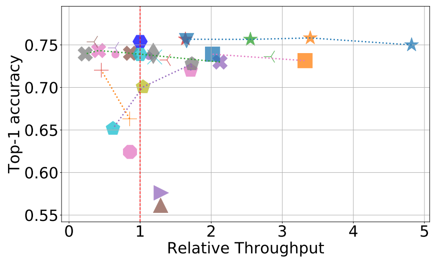


Figure 12: Performance of compressors for ResNet-50 on ImageNet via 1 Gbps network. Legend in Figure 6.

compression, SwitchML reduces the transmitted data because of the computation on the network switch. A similar idea is explored in DAIET [70], for general-purpose aggregation.

Model compression. Instead of compressing the communicated gradient, many papers (e.g., [6, 15, 49, 93]) propose to compress the model parameters. ZIPML [93], 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 [27].

7. CONCLUSION

We present a survey and systematic classification of the most influential methods on gradient compression for distributed, data-parallel DNN training. We propose a unified programming framework with the corresponding TensorFlow and PyTorch API, and implement 16 representative compression methods. We use both convolutional and recurrent DNNs, as well as a variety of datasets and system configurations, to perform thorough experimental evaluation and report metrics that include accuracy, throughput, scalability, communication volume and wall-time. We observe that the effectiveness of each method depends on the architecture of the trained DNN. Moreover, 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 at `https://github.com/sands-lab/grace`. 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.

8. REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.

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

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

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

[5] Z. Allen-Zhu. Katyusha: The First Direct Acceleration of Stochastic Gradient Methods. *STOC*, 2017.

[6] S. Anwar, K. Hwang, and W. Sung. Fixed Point Optimization of Deep Convolutional Neural Networks for Object Recognition. In *ICASSP*, 2015.

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

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

[9] T. Ben-Nun and T. Hoefler. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *ACM Computing Surveys*, 52(4), 2019.

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

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

[12] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, 1st edition, 1996.

[13] BytePS: A High Performance and Generic Framework for Distributed DNN Training. `https://github.com/bytedance/byteps`.

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

[15] M. Courbariaux, Y. Bengio, and J.-P. David. BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations. In *NeurIPS*, 2015.

[16] 29th Annual symposium of the German association for pattern recognition. `https://resources.mpi-inf.mpg.de/conference/dagm/2007/index.html`.

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

[18] 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.

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

[20] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Annual Conf. North American Assoc. for Computational Linguistics (NAACL)*, 2018.

[21] A. Dieuleveut and K. K. Patel. Communication Trade-offs for Local-SGD with Large Step Size. In *NeurIPS*, 2019.

[22] N. Dryden, S. A. Jacobs, T. Moon, and B. Van Essen. Communication Quantization for Data-Parallel Training of Deep Neural Networks. *Workshop on ML in HPC (MLHPC)*, 2016.

[23] J. Duchi, E. Hazan, and Y. Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12, 2011.

[24] A. Dutta, E. Bergou, A. M. Abdelmoniem, C. Y. Ho, A. N. 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.

[25] Gloo: Collective communications library with various primitives for multi-machine training. `https://github.com/facebookincubator/gloo`.

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

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

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

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

[30] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computing*, 9(8), 1997.

[31] Horovod API Documentation. `https://horovod.readthedocs.io/en/latest/api.html#module-horovod.tensorflow`.

[32] 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.

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

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

[35] 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.

[36] 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 *Conf. on Learning Theory (COLT)*, 2016.

[37] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *MM*, 2014.

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

[39] P. Jiang and G. Agrawal. A Linear Speedup Analysis of Distributed Deep Learning with Sparse and Quantized Communication. In *NeurIPS*, 2018.

[40] R. Johnson and T. Zhang. Accelerating Stochastic Gradient Descent using Predictive Variance Reduction. In *NeurIPS*, 2013.

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

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

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

[44] A. Krizhevsky and G. Hinton. Learning Multiple Layers of Features From Tiny Images. *Technical report, University of Toronto*, 1(4), 2009.

[45] M. Li, D. G. Andersen, J. W. Park, A. J. Smola,

A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI*, 2014.

[46] Y. Li, T. Ma, and H. Zhang. Algorithmic Regularization in Over-parameterized Matrix Sensing and Neural Networks with Quadratic Activations. In *Conference On Learning Theory (COLT)*, 2018.

[47] Y. Li, J. Park, M. Alian, Y. Yuan, Z. Qu, P. Pan, R. Wang, A. Gerhard Schwing, H. Esmaeilzadeh, and N. Sung Kim. A Network-Centric Hardware/Algorithm Co-Design to Accelerate Distributed Training of Deep Neural Networks. In *Micro*, 2018.

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

[49] D. Lin, S. Talathi, and S. Annapureddy. Fixed Point Quantization of Deep Convolutional Networks. In *ICML*, 2016.

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

[51] LSTM-PTB. `https://github.com/tensorflow/models/tree/master/tutorials/rnn`.

[52] 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.

[53] 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.

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

[55] Movielens. `https://grouplens.org/datasets/movielens/`.

[56] 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.

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

[58] Y. Nesterov. Gradient Methods for Minimizing Composite Functions. *Mathematical Programming*, 140(1), 2013.

[59] NVIDIA deep learning examples. `https://github.com/NVIDIA/DeepLearningExamples`.

[60] E. Oja. Simplified Neuron Model as a Principal Component Analyzer. *Journal of Mathematical Biology*, 15(3), 1982.

[61] Open MPI: Open Source High Performance Computing. `https://www.open-mpi.org/`.

[62] D. Page. CIFAR10-fast. `https://github.com/davidcpage/cifar10-fast`.

[63] 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.

[64] B. T. Polyak. Some Methods of Speeding up the Convergence of Iteration Methods. *URSS Computational Mathematics and Mathematical Physics*, 4(5), 1964.

[65] PyTorch. `https://pytorch.org/`.

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

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

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

[69] O. Ronneberger, P. Fischer, and T. Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *Int. Conf. on Medical Image Computing and Computer-assisted Intervention*, 2015.

[70] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-Network Computation is a Dumb Idea Whose Time Has Come. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2017.

[71] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtárik. Scaling Distributed Machine Learning with In-network Aggregation. *arXiv preprint arXiv:1903.06701*.

[72] 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.

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

[74] O. Shamir, N. Srebro, and T. Zhang. Communication-Efficient Distributed Optimization using an Approximate Newton-type Method. In *ICML*, 2014.

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

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

[77] N. Strom. Scalable Distributed DNN Training using Commodity GPU Cloud Computing. In *INTERSPEECH*, 2015.

[78] H. Sun, Y. Shao, J. Jiang, B. Cui, K. Lei, Y. Xu, and J. Wang. Sparse Gradient Compression for Distributed SGD. In *DASFAA*, 2019.

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

[80] H. Tang, C. Yu, X. Lian, T. Zhang, and J. Liu. DoubleSqueeze: Parallel Stochastic Gradient Descent with Double-pass Error-Compensated Compression. In *ICML*, 2019.

[81] TensorFlow benchmark. `https://github.com/tensorflow/benchmarks`.

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

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

[84] H. Wang, S. Sievert, S. Liu, Z. Charles,

D. Papailiopoulos, and S. Wright. ATOMO: Communication Efficient Learning via Atomic Sparsification. In *NeurIPS*, 2018.

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

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

[87] 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.

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

[89] T. Yang. Trading Computation for Communication: Distributed Stochastic Dual Coordinate Ascent. In *NeurIPS*, 2013.

[90] H. Yu, R. Jin, and S. Yang. On the Linear Speedup Analysis of Communication Efficient Momentum SGD for Distributed Non-Convex Optimization. In *ICML*, 2019.

[91] M. Yu, Z. Lin, K. Narra, S. Li, Y. Li, N. S. Kim, A. Schwing, M. Annavaram, and S. Avestimehr. GradiVeQ: Vector Quantization for Bandwidth-Efficient Gradient Aggregation in Distributed CNN Training. In *NeurIPS*, 2018.

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

[93] 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.

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