DeepReduce: A Sparse-tensor Communication Framework for Federated Deep Learning

Hang Xu
KAUST
hang.xu@kaust.edu.sa

Kelly Kostopoulou
Columbia University
kelkost@cs.columbia.edu

Aritra Dutta
KAUST
aritra.dutta@kaust.edu.sa

Xin Li
University of Central Florida
xin.li@ucf.edu

Alexandros Ntoulas
NKUA
antoulas@di.uoa.gr

Panos Kalnis
KAUST
panos.kalnis@kaust.edu.sa

Abstract

Sparse tensors appear frequently in federated deep learning, either as a direct artifact of the deep neural network's gradients, or, as a result of an explicit sparsification process. Existing communication primitives are agnostic to the challenges of deep learning; consequently, they impose unnecessary communication overhead. This paper introduces DeepReduce, a versatile framework for the compressed communication of sparse tensors, tailored to federated deep learning. DeepReduce decomposes sparse tensors into two sets, values and indices, and allows both independent and combined compression of these sets. We support a variety of standard compressors, such as Deflate for values, and Run-Length Encoding for indices. We also propose two novel compression schemes that achieve superior results: curve-fitting based for values, and bloom-filter based for indices. DeepReduce is orthogonal to existing gradient sparsifiers and can be applied in conjunction with them, transparently to the end-user, to significantly lower the communication overhead. As a proof of concept, we implement our approach on TensorFlow and PyTorch. Our experiments with real models demonstrate that DeepReduce transmits 320% less data than existing sparsifiers, without affecting accuracy. Code is available at `https://github.com/hangxu0304/DeepReduce`.

1 Introduction

In federated learning [43, 47, 55, 72], the training is typically performed by a large number of resource-constrained client devices (e.g., smartphones), operating on their private data and computing a *local* model. Periodically, a subset of the devices is polled by a central server that retrieves their gradients, updates the *global* model and broadcasts it back to the clients. The most constrained resource in client devices is the network, either because the practically sustained bandwidth between remote clients and the server is low (typically, in the order of 25-50Mbps), or, the financial cost (e.g., data plans for 4G/5G mobile connections) is high. On the other hand, deep neural network model sizes have been steadily increasing at a much faster rate than the available bandwidth. Consequently, in federated learning, it is imperative to reduce the communicated data volume.

One key observation that can help with reducing this volume is that the data exchanged during the Deep Neural Network (*DNN*) training often correspond to *sparse* tensors, i.e., tensors with many zero-value elements. Sparse tensors may be: (*i*) direct artifacts of the training process; for instance, the gradients of the NCF [35] and DeepLight [18] models consist of roughly 40% and 99% zero elements, respectively; or (*ii*) explicitly generated by *sparsification* [6, 51, 69, 71, 76, 81],

35th Conference on Neural Information Processing Systems (NeurIPS 2021), Sydney, Australia.

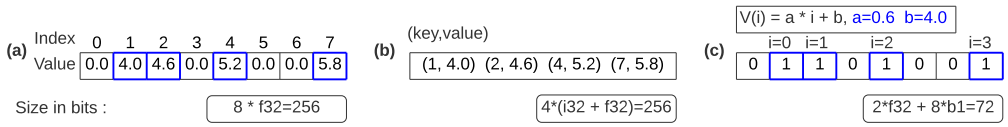


Figure 1: (a) Dense tensor format needs 256 bits; (b) sparse $\langle key, value \rangle$ form also needs 256 bits; (c) our method sends parameters, (a, b) as well as an 8-bit string, i.e., 72 bits in total.

a commonly used lossy (e.g., Top-r or Random-r) compression approach that selects only a few elements (with r typically < 1%).

Figure 1.a depicts an example tensor containing 8 real values, 4 of which are zero; its *dense* representation would require 256 bits. Typically, sparse tensors are represented as a set of $\langle key, value \rangle$ pairs (see Figure 1.b), where key is the index; notice, however, that the $\langle key, value \rangle$ representation also requires 256 bits, negating the benefit of sparsity. We show an example of our improved approach in Figure 1.c. We consider the indices as an ordered list represented by a Boolean array of 8 bits, such that the i^{th} bit is '1' if and only if the corresponding gradient element is non-zero. Moreover, we fit a function, $V(i) = a \cdot i + b$ to the gradient values, with parameters, $a = 0.6$ and $b = 4.0$. By transmitting only the bit string and parameters (a, b), we can reconstruct the original tensor while requiring only 72 bits.

The above example demonstrates a significant margin for additional compression for sparse tensors. Recent works (e.g., SKCompress [40]) take advantage of these opportunities, but rely on a tightly coupled index and value compression algorithm that benefits only some scenarios (see Section 6). In practice, there exist complex trade-offs among data volume, model accuracy, and computational overhead, in conjunction with system aspects, such as the network bandwidth and the communication library (e.g., NCCL [2], or Gloo [29]). Given that no single solution fits all scenarios, practitioners need the flexibility to adjust how sparse tensors are compressed and transmitted for each particular DNN model and system configuration.

In this paper, we propose DeepReduce, a framework for transmitting sparse tensors via a wide-area network, tailored for large-scale federated DNN training. Our contributions include:

(*i*) The **DeepReduce framework**, described in Section 3, that decomposes the sparse tensor into two sets, indices and values, and allows for independent and combined compression. By decoupling indices from values, our framework enables synergistic combination of a variety of compressors in a way that benefits each particular DNN and training setup. DeepReduce resides between the machine learning framework (e.g., Tensorflow, PyTorch) and the communication library. It exposes an easy-to-use API that encapsulates a variety of existing methods, such as Run Length [83] and Huffman encoding [38] for index compression; as well as Deflate [20] and QSGD [7] for value compression. DeepReduce also provides an index reordering abstraction, which is useful for combining value with index compressors.

(*ii*) **Two novel compressors** for sparse tensors: a Bloom-filter based *index* compressor (Section 4) that reduces the size of keys by 50%, compared to the traditional $\langle key, value \rangle$ sparse representation; and a curve-fitting based *value* compressor (Section 5) that reduces the large values array to a small set of parameters. Both of our compressors do not affect the quality of the trained model.

(*iii*) An **evaluation** of DeepReduce on a variety of DNN models and applications (Section 6). We demonstrate the practical applicability of our framework by realistic *federated learning* deployments on geographically remote, large-scale cloud infrastructures, and show that DeepReduce compresses already-sparse data by up to 320%, without affecting the training quality.

2 Background

Notations. By $[d]$ we denote the set of d natural numbers $\{1, 2, \cdots, d\}$. We denote the cardinality and complement of a set X, by $|X|$ and X^c, respectively; $x[i]$ is the i^{th} component of vector x. By $x_S \in \mathbb{R}^d$ we denote an $|S|$-sparse vector, where $S \subseteq [d]$ is its support. $\|x\|$ and $\|x\|_\infty$ are the ℓ_2 and ℓ_∞ norm of a vector x, respectively. Class C^1 consists of all differentiable functions with continuous derivative and $\text{Var}_{[a,b]}(f)$ denotes the variance of function $f \in C^1[a, b]$ over interval $[a, b]$.

Federated Learning (*FL*) trains models *collaboratively* on data stored in resource-constrained, heterogeneous and geographically remote client devices (e.g., smartphones). At each round, a fraction of the participating clients are selected. After synchronizing with the server-side *global* model, the selected clients perform *local* training[1] for some epochs over their private data. The server collects those updates, aggregates and applies them to the global model. Federated averaging (*FedAvg*) [55], is a popular instantiation of this process. Other variants exist, such as SCAFFOLD [44], FedNova [80], parallel SGD [85], local SGD [68], and FedProx [49]. Owing to insufficient communication bandwidth in FL, lossy compression is used to reduce the transmitted data volume; examples include FedBoost [31], FedCOM [30], and FedPAQ [60].

Compressor [69] is a random operator, $\mathcal{C}(\cdot) : \mathbb{R}^d \rightarrow \mathbb{R}^d$, that satisfies $\mathbb{E}_{\mathcal{C}}\|x - \mathcal{C}(x)\|^2 \leq \Omega\|x\|^2$, where $\Omega > 0$ is the compression factor and the expectation is taken over the randomness of $\mathcal{C}$. If $\Omega = 1 - \delta$ and $\delta \in (0, 1]$, $\mathcal{C}$ is a δ-*compressor*, denoted by $\mathcal{C}_\delta$.

Sparsification. A rank-1 tensor that has mostly zero components is said to be sparse. Many modern DNNs are inherently sparse [18, 35, 42], and so are their gradients. Sparse gradients can also be generated by a compressor that sparsifies [51, 71, 76, 81] the gradient g to generate $\tilde{g} \in \mathbb{R}^d$ via: $\tilde{g}[i] = \mathcal{C}(g[i]) = g[i]$, if $i \in S$, otherwise $\tilde{g}[i] = 0$. We assume that $S \subset [d]$ is the support set of $\tilde{g}$ such that $\tilde{g}$ is r-sparse if and only if $|S| = r$. Two commonly used sparsifiers, Random-r [69] and Top-r [6, 8], both are δ-compressors. We refer to further details in Appendix A.

Bloom filter [12] is a probabilistic data structure that represents the elements of a set S. Initially, it is a bit string $\mathcal{B}$ of m bits, all set to 0. To insert an element $y_j \in S$ in the Bloom filter, we apply k independent hash functions h_i on y_j. Each h_i yields a bit location in $\mathcal{B}$, and changes that bit to 1. To query if $y_j \in S$, we apply all k hash functions on it. If $h_i(y_j) = 0$ for any $i \in [k]$, then the element *surely* does not belong to S (i.e., no false negatives). In contrast, if $h_i(y_j) = 1$ for all $i \in [k]$, then y_j may or may not belong to S (i.e., possible *false positives*). The false positive rate (*FPR*) [12] of $\mathcal{B}$ is: $\epsilon \approx (1 - e^{-k|S|/m})^k$; see Lemma 3 in Appendix A.2.

3 System Architecture

DeepReduce (see Figure 12 in the Appendix) resides between the machine learning framework (e.g., TensorFlow, PyTorch) and the communication library, and is optimized for federated DNN training. It offers a simple API whose functions can be overridden to implement, with minimal effort, a wide variety of index and value compression methods for sparse tensors. At the transmitting side, the input to DeepReduce is a sparse tensor directly from the ML framework, for the case of inherently sparse models; or, is generated by an explicit sparsification process. In our implementation, we employ the GRACE [84] library for the sparsification operation, since it includes many popular sparsifiers; other libraries can also be used.

Sparse tensors are typically represented as $\langle key, value \rangle$ tuples. DeepReduce decouples the keys from the values and constructs two separate data structures. Let $\tilde{g} \in \mathbb{R}^d$ be the sparse gradient, where d is the number of model parameters and $\|\tilde{g}\|_0 = r$, is the number of nonzero gradient elements. Let S be the set of r indices corresponding to those elements. DeepReduce implements two equivalent representations of S: (*i*) an array of r integers; and (*ii*) a bit string B with d bits, where $\forall i \in [1, d], B[i] = 1$ if and only if $\tilde{g}[i] \neq 0$. These two representations are useful for supporting a variety of index compressors; e.g., the bit string representation is used in [14]. The Index Compression module encapsulates the two representations and implements several algorithms for index compression. It supports both *lossy* compressors (e.g., our Bloom-filter based proposal), as well as *lossless* ones, such as the existing Run Length (RLE) [83] and Huffman [38, 27] encoders (see Appendix A.1); there is also an option to bypass index compression.

The Value Compression module receives the sparse gradient values and compresses them independently. Several compressors, such as Deflate [20], QSGD [7], and our own curve-fitting based methods, are implemented. Again, there is an option to bypass value compression. Some compressors (e.g., our own proposals), require reordering of the gradient elements, which is handled by the Index reorder module. DeepReduce then combines, in one container, the compressed index and value

[1] In contrast to FL, conventional data-parallel, distributed training [87] synchronizes the updates from *all* workers at *each* iteration (see Appendix A).

structures, the reordering information and any required metadata. Then, the container is passed to the communication library.

The receiving side mirrors the structure of the transmitter, but implements the inverse functions, that is, index and value decompression, and index reordering. The reconstructed sparse gradient is routed to GRACE for de-sparsification, or passed directly to the ML framework. It is worth mentioning that DeepReduce is general enough to represent popular existing methods that employ proprietary combined value and index compression. For example, SKCompress [40] can be implemented in DeepReduce as follows: SketchML [39] plus Huffman for values, no index reordering, and delta encoding plus Huffman for indices.

4 Bloom Filter for Indices

This section introduces our novel Bloom-filter based, lossy index compressor. Recall that S is the set of r indices (i.e., $|S| = r$) corresponding to the nonzero components of sparse gradient $\tilde{g}$. Let us insert each item of S into a Bloom filter $\mathcal{B}$ of size m, using k hash functions.

Naïve Bloom filter. Let V be an array of size r containing the elements of $\tilde{g}$ that are indexed by S; formally: $\forall i \in [1, r] : V(i) = \tilde{g}[S[i]]$. DeepReduce transmits V and $\mathcal{B}$. The receiver initializes $ptr = 1$ and reconstructs the gradient as follows:

---

for $i = 1$ *to* d **do** /* all d elements of gradient $\tilde{g} \in \mathbb{R}^d$ */
  if $i \in \mathcal{B}$ **then** $\tilde{g}[i] = V[ptr]$; ptr++; **else** $\tilde{g}[i] = 0$

If $\mathcal{B}$ were *lossless*, the algorithm would have perfectly reconstructed the gradient. However, Bloom filters exhibit false positives (*FP*). Assume a single FP at $ptr = j$; then, for $j \leq ptr \leq r$, *every* $V[ptr]$ value will be assigned to the wrong gradient element. Therefore, FPs cause a disproportionately large error to the reconstructed gradient, significantly affecting the quality of the trained model.

No-error approach: Policy P0. To address this drawback of the naïve approach, we initialize a set $P = \varnothing$ and we modify the previous reconstruction algorithm as follows:

---

for $i = 1$ *to* d **do** /* all d elements of gradient $\tilde{g} \in \mathbb{R}^d$ */
  if $i \in \mathcal{B}$ **then** insert i in P

P contains the union of the true and false positive responses of $\mathcal{B}$. The transmitting worker can execute this algorithm and determine P prior to any communication. With that information, it constructs array V as follows: $\forall i \in [1, |P|] : V(i) = \tilde{g}[P[i]]$. Essentially, V contains the gradient elements that correspond to both true and false positives. Therefore, the receiving worker can reconstruct the sparse gradient *perfectly*. The trade-off is increased data volume compared to Naïve, since the size of V grows to $r \leq |P| \leq \lceil r + (\frac{1}{2})^{-\frac{\log(\epsilon)}{\log(2)}}(d - r)\rceil$; see Lemma 6 in Appendix B.2. We measure the compression error due to gradient, $\mathcal{C}_{P0,\delta}(g)$ resulted from policy $P0$ in Lemma 7 in Appendix B.2.

Random approach: Policy P1. To address the increased data volume issue of policy P0, this policy defines a new set of indices $\tilde{S} \subseteq P$, where $|\tilde{S}| = r$. $\tilde{S}$ is generated by randomly selecting r elements from P. Consequently, array V is constructed as $\forall i \in [1, r] : V(i) = g[\tilde{S}[i]]$. Since $\tilde{S} \neq S$ in general, we expected the error to be affected. Let $k_1 = |\tilde{S} \cap S|$ and assume that the input gradient is inherently sparse, i.e., sparsifier $\mathcal{C}_\delta$ is the identify function I_d. For the combined compressor $\mathcal{C}_{P1,I_d}$ with policy P1, we show in Appendix B.4 that $\mathbb{E}\|g - \mathcal{C}_{P1,I_d}(g)\|^2 = (1 - \frac{k_1}{r})\|g\|^2$. Specifically, policy P1 creates a lossy compressor with compression factor as good as Random-k_1. In practice, DeepReduce allows $\mathcal{C}_\delta$ to be any sparsifier (e.g., Top-r [6, 8]). In this case, policy P1 is essentially equivalent to a combined sparsifier, similar to Elibol et al. [24] and Barnes et al. [9]. We give the total compression error due to the compressed gradient $\mathcal{C}_{\mathcal{P}_R,\delta}(g)$ and other detailed analysis and proofs of error bounds in Lemma 9 in Appendix B.4.

Conflict sets: Policy P2. P0 and P1 represent two extremes: P0 eliminates errors but sends more data than P1, whereas P1 sends fewer data but may introduce errors. Here, we propose policy P2, which transmits the same amount of data as P1, but is closer to P0 in terms of error. Similar to P1, this policy generates a set $\tilde{S} \subseteq P$, but makes better probabilistic choices. Intuitively, false positives are due to collisions in the Bloom filter, resulting in conflicts. P2 groups all items of P into conflict sets. Two elements x and y belong to a conflict set $\mathcal{C}_j$ if $x, y \in P$ and $h_i(x) = h_{i'}(y) = j$ for $i, i' \in [k]$, where j is the j^{th} bit of $\mathcal{B}$ and $\bigcup_j C_j = P$. Figure 2 shows an example, with 4 items and 3 hash

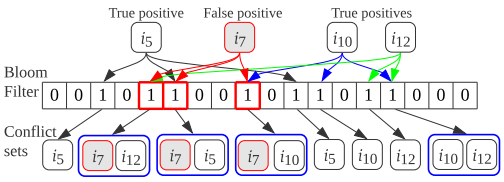


Figure 2: Policy P2, with 3 hash functions and 8 conflict sets.

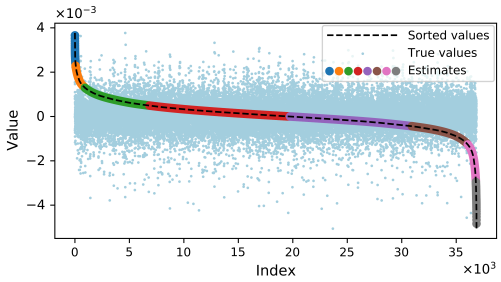


Figure 3: Piece-wise (8 pieces) value fitting on a convolution layer gradient of ResNet20 (CIFAR-10).

functions; item i_7 is a false positive and there exist 8 conflict sets. If a set C_j contains only one item, it is guaranteed to be a true positive, so it is added to $\tilde{S}$. Else, if C_j contains many items, a random subset is inserted into $\tilde{S}$. Therefore, it is possible that some false positives are included in $\tilde{S}$, but the probability is smaller, compared to policy P1. We present the pseudo-code of policy P2 in Algorithm 1, Appendix B.5. For Hash functions used and an efficient GPU implementation on PyTorch, refer to Appendix E.

5 Curve Fitting for Values

Figure 3 shows, in light blue, the gradient values for one layer of ResNet20 on CIFAR-10. By sorting those values, we obtain a curve that can be approximated as a *smooth convex* curve. Taking advantage of this observation, we propose a novel curve fitting-based value compression approach that results into high compression ratios, even after considering the overhead of mapping the original to the sorted values. In contrast, modelling DNN gradients by using statistical distributions [26, 54, 66] is both restrictive and model specific.

Formally, we sort the nonzero values of the sparse gradient, $C_\delta(g_t)$ resulting from stochastic gradient, g_t at each iteration t, in descending order and denote as $C_S(g_t) \in \mathbb{R}^d$. Let $C_S(g_t)$ follow the hierarchical model: $\mathcal{C}_S(g_t) = \nabla f_t + \sigma\xi_t$, where $\xi_t \sim N(0, I_d)$, $\nabla f_t \in \mathbb{R}^d$ is the sorted oracle gradient and $\sigma \in \mathbb{R}^+$; the model is commonly used in signal processing for approximation problems [5, 41, 74]. With these assumptions, we show how to fit regression models on $\mathcal{D} := \{i, \mathcal{C}_S(g_t)[i]\}_{i=1}^r$ and calculate the fitting error.

Nonlinear approximation. We employ a nonlinear approximation by using splines with free knots, which consists of splitting the sorted curve into segments and fitting each segment individually. Finding the locations of the knots is where the nonlinearity and difficulties are (cf. [16]). Although existing numerical algorithms can find the best splines with free knots, they are computationally intensive and unsuitable for our use at each iteration of DNN training. Because we aim at approximating a smooth convex curve, we use the following procedure by selecting the knot one by one. We explain how this is done for the positive sorted values; similar idea applies to the negative sorted values. Let the whole gradient be sorted in descending manner; set $[l]$, with $l \leq d$ corresponds to indices of the sorted positive values. Let $y = mx + c$ be the line joining $(1, \mathcal{C}_S(g)[1])$ and $(l, \mathcal{C}_S(g)[l])$. Calculate $d_i := (y_i - \mathcal{C}_S(g)[i])^2$, where $y_i = mi + c$. Choose the sorted gradient component that corresponds to $\max_{i\in[l]} d_i$ as the segmentation point (a knot). The process continues until the desired number of segments is reached. We stop segmentation if the number of points in a segment is less than $(n' + 1)$, where n' is the degree of the polynomial used to fit on each segment. The following Proposition justifies our procedure above.

Lemma 1. *(Knot selection) If f is a differentiable convex function on $[a, b]$, then the point $x^* \in (a, b)$ that gives the best approximation to f using line segments from $(a, f(a))$ to $(x^*, f(x^*))$, and then from $(x^*, f(x^*))$ to $(b, f(b))$ is the same point x^+ that maximizes the difference between f and the line segment connecting $(a, f(a))$ to $(b, f(b))$.*

Polynomial regression. Over each segment, we apply polynomial regression. Due to limited space, we focus on the piece-wise linear fit, and have the following result with *explicit constants*. For the results on piece-wise constant approximation, see Lemma 11 in Appendix C.

Table 1: Benchmarks and datasets; last column shows the best quality achieved by the no-compression baseline.

Type	Model	Task	Dataset	Parameters	Optimizer	Platform	Metric	Baseline
CNN	ResNet-20 [34]	Image classif.	CIFAR-10 [48]	269,722	SGD-M [73]	TFlow	Top-1 Acc.	90.94%
	DenseNet40-K12 [37]	Image classif.	CIFAR-10 [48]	357,491	SGD-M [73]	TFlow	Top-1 Acc.	91.76%
	ResNet-50 [34]	Image classif.	ImageNet [17]	25,557,032	SGD-M [73]	TFlow	Top-1 Acc.	73.78%
MLP	NCF [35]	Recommendation	Movielens-20M [56]	31,832,577	Adam [46]	PyTorch	Best Hit Rate	94.97%
RNN	LSTM[59]	Next word pred.	Stack Overflow[67]	4,053,428	**FedAvg [55]**	PyTorch	Top-1 Acc.	18.56%

Lemma 2. *(Error of piece-wise linear fit) For $\mathcal{C}_S(g) \in C^1([1,d])$ with $\mathrm{Var}_{[1,d]}(\mathcal{C}'_S(g)) \leq M$, we have $\|s - \mathcal{C}_S(g)\|_\infty \leq \frac{2M}{p^2}$, for some $s \in \mathcal{S}^1_p$—the set of piece-wise linear splines with p knots.*

Note that, the least squares error is less than $\sqrt{d}\|s - \mathcal{C}_S(g)\|_\infty$, and can be controlled by using a large p (see Remark 5 in the Appendix). We provide here a heuristic to calculate p from Lemma 2. First, we calculate $M = |(\mathcal{C}_S(g)[1] - \mathcal{C}_S(g)[2]) - (\mathcal{C}_S(g)[d-1] - \mathcal{C}_S(g)[d])|$. By considering the error bound $\frac{2M}{p^2}$ as a function of p, we can find the closed-form solution for p as $p = \lceil 2\sqrt{M} \rceil$.

Nonlinear regression. We can use nonlinear regression for value fitting, e.g., through a double exponential model, $y = ae^{bx} + ce^{dx}$, where $(a, b, c, d) \in \mathbb{R}^4$ are the parameters; refer to Section 6.

For the theoretical analysis of compression error from regression, see Appendix C.1. In Appendix D, we provide the compression error from joint value and index compression. Also, see Appendix D.1 for comments regarding the convergence of the approach. Finally, refer to Appendix E for an efficient GPU and CPU implementation of our polynomial regression on PyTorch and TensorFlow, and the combined index and value compression. For overall complexities of the compression methods we also refer to Appendix E.

6 Experimental Evaluation

Implementation. DeepReduce supports TensorFlow and Pytorch. We provide various versions of our index and value compressors, on CPUs and GPUs. We also instantiate our framework with combinations of existing methods, namely Huffman and RLE for index compression, as well as Deflate and QSGD [7] for value compression; see Appendix F. We denote implementations that use DeepReduce by $\mathrm{DR}^{\mathrm{val}}_{\mathrm{idx}}$, where idx, val are the index and value compression methods, respectively.

Testbed. We set up a realistic federated learning testbed on Amazon AWS. The server is an EC2 instance located in Ohio, whereas the clients are 56 geographically remote instances spread across 7 regions throughout the globe (i.e., Tokyo, Central Canada, Northern California, Seoul, São Paulo, Paris and Oregon). Each instance is equipped with a 4-core Intel CPU @ 2.50GHz, 16GB RAM, and an NVIDIA Tesla T4 GPU with 16 GB on-board memory (see Appendix F.1 for details). We also run simulated deployments on a local cluster of 8 nodes, each with a 16-core Intel CPU @ 2.6GHz, 512GB RAM, one NVIDIA Tesla V100 GPU with 16 GB on-board memory and 100Gbps network.

Benchmarks. We employ the popular FedML [33] benchmark that uses an LSTM model [59] to perform next-word prediction in a federated learning setting, on the Stack Overflow [67] dataset with 135,818,730 training and 16,586,035 test examples; the dataset follows a real-life partitioning among 342,477 clients. We also use industry-standard benchmarks from TensorFlow [52, 75] and NVIDIA [57], on image classification and recomendation; refer to Table 1 for details.

6.1 Simulated deployment on a local testbed

First, we run a set of experiments on our local cluster to validate our index and value compressors.

Bloom filter-based index compression. Figure 4 depicts the convergence timeline for our three index compression polices for ResNet-20 on CIFAR-10; FPR is set to 10^{-3} (refer to Appendix F for the effect of varying FPR). We compare against the no-compression baseline, as well as the plain Top-r sparsifier ($r = 1\%$). All our policies converge to the same top-1 accuracy as the no-compression baseline, but BF-P0 converges in fewer training epochs. It is worth noting that BF-P0 converges faster than the plain Top-r sparsifier, despite transmitting 33% fewer data (refer to Figure 15c in the Appendix). Note that BF-naïve (Section 4) achieves much lower accuracy, justifying the need for our proposed policies. Similar results were observed for DenseNet40-K12; see Appendix F.3.

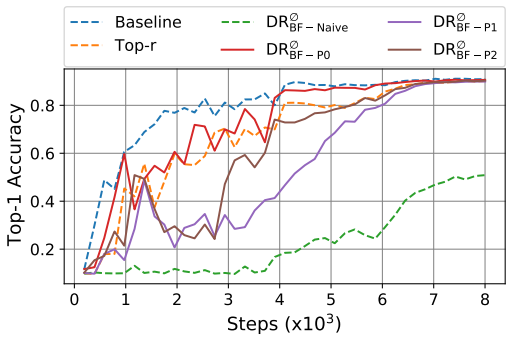


Figure 4: Convergence timeline of our bloom filter policies for ResNet-20 on CIFAR-10; FPR = 0.001, $r = 1\%$. $\mathrm{DR}^{\varnothing}_{\mathrm{BF-P0}}$ converges the fastest.

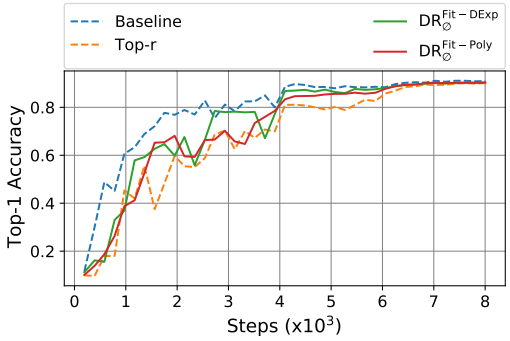


Figure 5: Convergence timeline of value compressors for ResNet-20 on CIFAR-10; $r = 1\%$. $\mathrm{DR}^{\mathrm{Fit-DExp}}_{\varnothing}$ is slightly better than $\mathrm{DR}^{\mathrm{Fit-Poly}}_{\varnothing}$.

Curve fitting-based value compression. Figure 5 shows the converge timeline for our two curve fitting-based value compressors, Fit-Poly and Fit-DExp, on ResNet-20 with CIFAR-10. The sparse input gradients were generated by Top-r ($r = 1\%$). Fit-Poly uses polynomials with degree 5 (i.e., 6 coefficients). Fit-DExp requires 4 coefficients without segmentation. Both methods converge to the same accuracy as the no-compression baseline; also, both converge in fewer steps than plain Top-r. Fit-DExp sends fewer data: it compresses the output of Top-r by roughly 50%, whereas Fit-Poly compresses it by around 40%. However, the computational overhead of compression for Fit-DExp is roughly 3.5× more than Fit-Poly's; see details about the data volume and runtime in Figure 8.

DeepReduce for an inherently sparse model. DeepReduce is designed to work either in conjunction with a sparsifier (e.g., Top-r), or be applied directly on models such as the NCF [35] and DeepLight [18], that exhibit inherently sparse gradients. In this experiment, we train NCF on ML-20m, with 10^6 local batch size. We test $\mathrm{DR}^{\mathrm{QSGD}}_{\mathrm{BF-P0}}$, which uses BF-P0 (FPR=0.6) for indices, but combines it with QSGD [7], an existing method for value compression. This demonstrates that DeepReduce is compatible with various existing compressors. We compare against SKCompress [40], an improved version of SketchML [39], optimized for sparse tensors. Table 6 in the Appendix shows the results. All methods achieve virtually the same best hit rate (i.e., the quality metric for NCF), and both $\mathrm{DR}^{\mathrm{QSGD}}_{\mathrm{BF-P0}}$ and SKCompress reduce the data volume by 5× compared with Baseline. However, in practice $\mathrm{DR}^{\mathrm{QSGD}}_{\mathrm{BF-P0}}$ can be more easily implemented on GPUs; in Figure 8b we show that it is 380× faster in terms of compression and decompression time. In Appendix F.3, we also compare DeepReduce against 3LC [50] and SketchML [39] on a larger benchmark, ResNet-50 on ImageNet.

6.2 Realistic Federated Learning deployment in the cloud

In this section, we use the FedML [33] benchmark to deploy a realistic federated learning system in the cloud. The real Stack Overflow [67] dataset is naturally partitioned among 342,477 clients. At each round, the server randomly selects 56 clients to communicate. Those clients are activated on physical EC2 instances in Amazon AWS, located at geographically remote data centers all over the world. Each client executes 1 local epoch; the learning rate is 0.3 and the batch size is 16. Sparsification is performed on tensors with more than 1 dimension, by bidirectional Top-r ($r = 10\%$) compression with error feedback on the model updates. We execute 200 rounds and achieve 18.56% test accuracy, which is consistent with the FedML benchmark. We give pseudocode of FedAvg with DeepReduce in Algorithm 2 in Appendix F.3.

Communication cost and computational overhead for compression. Since the most constrained resource in federated learning is the network, our main target is to *minimize the amount of transferred data*. We measure separately the amount of transferred data from server-to-client (S2C) and client-to-server (C2S). Our baseline is the popular FedAvg [55] algorithm; we also compare against the plain Top-r sparsifier ($r = 10\%$). Table 2 shows the results for three instantiations of DeepReduce. In all cases DeepReduce compresses significantly the already-sparse data. In particular, $\mathrm{DR}^{\mathrm{QSGD}}_{\mathrm{BF-P0}}$ transmits only 6.2% of the original data, which is 3.2× less than Top-r, while the test accuracy

Table 2: Time breakdown and data volume of DeepReduce variants, Top-r, and Baseline (FedAvg [55]) in a FL setting. (CLI, SER, S2C and C2S stand for Client, Server, Server-to-Client, and Client-to-Server, respectively.)

	Average Encoding/Decoding Time (s)				Avg. Comm. Time (s)		Avg. Data Volume (rel. to baseline)		Test Accuracy
	CLI_{decode}	CLI_{encode}	SER_{decode}	SER_{encode}	S2C	C2S	S2C	C2S	
Baseline	0	0	0	0	1.6014	1.6117	1.0	1.0	0.1856
Top-r (10%)	0.0035	0.0266	0.0045	0.0299	0.7853	0.8165	0.2033	0.2033	0.1840
$\text{DR}^{\varnothing}_{\text{BF-P0}}$	0.0181	0.0623	0.0161	0.0574	0.7763	0.7986	**0.1425**	**0.1426**	0.1841
$\text{DR}^{\text{Fit-Poly}}_{\varnothing}$	0.0187	0.1178	0.0175	0.1024	0.6858	0.6876	**0.1039**	**0.1039**	0.1838
$\text{DR}^{\text{QSGD}}_{\text{BF-P0}}$	0.0192	0.0754	0.0175	0.0691	0.6842	0.6864	**0.0621**	**0.0621**	0.1836

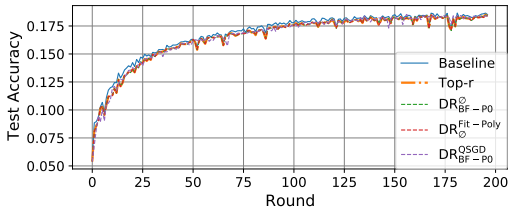


Figure 6: Convergence timeline of training an RNN to do next-word-prediction on Stack Overflow datasets. DeepReduce on Top-r exhibits the same convergence rate as Top-r in a FL setting.

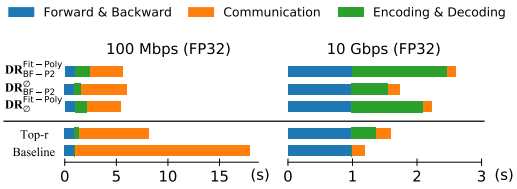


Figure 7: Time breakdown in one iteration training of NCF on ml-20m. We show the speedup of training by DeepReduce on 4 nodes with different network bandwidth: 100Mbps vs. 10Gbps

remains virtually unaffected. The high quality of the resulting models is also confirmed in Figure 6, which shows the convergence timeline of the training. For completeness, Table 2 also reports the computational overhead for encoding and decoding, although this is not our primary concern. DeepReduce is 1.5-3.4× slower than Top-r, but in practice the overhead is within acceptable limits (i.e., less than 19msec). Despite the higher computational overhead, the average communication time is still up to 15% lower for DeepReduce compared to Top-r, and up to 2.3× lower compared to Baseline; the moderate gain is due to the high latency between geographically remote data centers in the Amazon AWS cloud. See test-accuracy vs. wall clock time, and server to client and client to server communication time (with error bars) in Figures 20 and 21, respectively, in Appendix F.

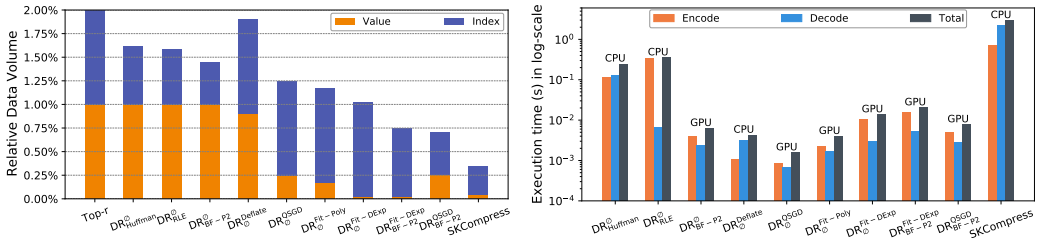
6.3 Practical applicability of DeepReduce

Any operation on the gradient imposes computational overheads that may exceed the benefits of the reduced data volume and affect the practical applicability, as discussed below.

Suitability for federated learning. We profile diverse deployments by training NCF on ML-20m and measuring the wall clock time of the various components. We employ gradient accumulation with 10 accumulations per iteration and 10^6 local batch size. We use NCCL `Allreduce` for baseline communication, and NCCL `Allgather` for Top-r ($r = 10\%$) and DeepReduce. We vary the network bandwidth from 100Mbps (typical for remote smartphones or IoT clients) up to 10Gbps (typical for local server clusters). Figure 7 shows the wall time, in three components: forward and back-propagation, encoding / decoding, and communication. Gradient compression is useful only when the ratio of communication over computation cost is high (i.e., lower bandwidth). This is consistent with the findings in [53, 58, 84] and reinforces our claim that DeepReduce is beneficial for federated learning deployments.

Data volume and computational overhead. In Figure 8a, we show the data volume (relative to the no-compression baseline) separately for values and indices, for various instantiations of DeepReduce. We test Resnet-20 on CIFAR-10 and generate sparse tensors by Top-r. We compare against SKCompress, which also operates on the sparse tensor. For fairness, parameters are selected such that all methods achieve similar accuracy. Although the ratio of index over value data volume differs for each combination of DeepReduce, all versions transmit fewer data than plain Top-r. Interestingly, SKCompress performs the best; however, this depends on the particular model.

Figure 8b shows, in logarithmic scale, the computational overhead of compression and decompression measured by the wall clock runtime. We implement all methods by utilizing the best available libraries either on CPUs or GPUs; we acknowledge there remains margin for improvement. Nonetheless, this



(a) Data volume (b) Run time

Figure 8: Comparing various compression methods on the Top-r(1%) values of a convolution gradient in ResNet-20 (gradient size: 36,864). (a) Data volume (b) Encoding and decoding runtime

experiment demonstrates the significant variation in terms of overhead among methods; for instance, SKCompress is 3 orders of magnitude slower, compared to $\mathrm{DR}_{\varnothing}^{\mathrm{QSGD}}$.

Discussion. The previous experiment demonstrates that the practical benefit of any compressor, including DeepReduce, depends on multiple factors that affect the communication over computation ratio. Those include the communication library (e.g., NCCL, Gloo), the implementation details of each compressor, the computing hardware (e.g., faster GPUs), possible accelerators (e.g., NICs with FPGAs), and others. The advantage of DeepReduce lies in its versatility to intermix various index and value compressors, to match the requirements of each specific model and system configuration.

7 Related Work

Gradient compression is commonly employed to alleviate the network bottleneck in distributed training. Four main families of compression methods exist (refer to [84] for a survey): (*i*) Quantization [7, 11, 19, 45, 63, 82], where each tensor element is replaced by a lower precision one (e.g., float8 instead of float32), to achieve in practice compression ratios in the order of $4\times$ - $8\times$ [84]. (*ii*) *Sparsification* [6, 51, 69, 71, 76, 81], where only a few elements (e.g., Top-r or Random-r) of the tensor are selected; it can achieve compression ratios of $100\times$ or more. (*iii*) Hybrid methods [10, 39, 50, 70], which combine quantization with sparsification to achieve a higher compression. (*iv*) Low-rank methods [15, 78, 79] that decompose the tensor into low-rank components.

SketchML and SKCompress. In SketchML [39], the nonzero gradient elements are quantized into buckets using a non-uniform quantile sketch. The number of buckets is further reduced via hash tables that resolve collisions by a Min-Max strategy. SKCompress [40] improves SketchML by additional Huffman coding on the bucket indices as well as the prefix of delta keys. Both of these methods can be viewed as special cases of DeepReduce.

Hybrid compressors. Qsparse local SGD [10] combines quantization with Top-r or Random-r sparsifiers. Strom et al. [70] and Dryden et al. [22] use a fixed and adaptive threshold, respectively, to sparsify. Elibol et al. [24] combine Top-r with randomized unbiased coordinate descent. Barnes et al. in [9], perform a Top-m selection of each local gradient and communicate $r < m$ randomly chosen components. Double quantization [86] is an asynchronous approach that integrates gradient sparsification with model parameter and gradient quantization. The output sparse gradient of hybrid methods can be the input to our framework; therefore, our work is orthogonal.

Sparse tensor communication. Communication libraries typically transmit sparse tensors via `Allgather` [1], because the more efficient `Allreduce` collective only supports dense tensors. In contrast, ScaleCom [13] tailors `Allreduce` to sparse data. OmniReduce [25] also implements sparse `Allreduce` that sends the non-zero blocks to the workers in an all-to-all manner. SparCML [61] adaptively switches between `Allreduce` and `Allgather` based on global gradient sparsity among the workers. SwitchML [62] is a hardware approach that aggregates the model updates in programmable network switches.

8 Conclusions

Sparse tensors are ubiquitous in federated DNN training. DeepReduce integrates seamlessly with popular machine learning frameworks and provides an easy-to-use API for the effortless implementation of a wide variety of sparse tensor communication methods. We instantiate DeepReduce both with existing index and value compressors, as well as with two novel methods: a Bloom filter-based index compressor and a curve fitting-based value compressor. We demonstrate its practical applicability by a realistic deployment in the cloud. DeepReduce is available as open-source and can be used to significantly lower the communication overhead in large-scale federated learning deployments.

References

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

[2] Nccl: Nvidia collective communication library. `https://developer.nvidia.com/nccl`.

[3] Numpy: API reference. `https://numpy.org/doc/stable/reference/generated/numpy.polyfit.html`.

[4] pybloomfilter library. `https://axiak.github.io/pybloomfiltermmap/`.

[5] Felix Abramovich, Yoav Benjamini, David L Donoho, Iain M Johnstone, et al. Adapting to unknown sparsity by controlling the false discovery rate. *The Annals of Statistics*, 34(2):584–653, 2006.

[6] Alham Fikri Aji and Kenneth Heafield. Sparse communication for distributed gradient descent. In *Proc. of EMNLP*, pages 440–445, 2017.

[7] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Proc. of NeurIPS*, pages 1709–1720, 2017.

[8] Dan Alistarh, Torsten Hoefler, Mikael Johansson, Sarit Khirirat, Nikola Konstantinov, and Cédric Renggli. The convergence of sparsified gradient methods. In *Proc. of NeurIPS*, pages 5977–5987, 2018.

[9] Leighton Pate Barnes, Huseyin A Inan, Berivan Isik, and Ayfer Özgür. rTop-k: A Statistical Estimation Approach to Distributed SGD. *IEEE Journal on Selected Areas in Information Theory*, 2020.

[10] Debraj Basu, Deepesh Data, Can Karakus, and Suhas Diggavi. Qsparse-local-SGD: Distributed SGD with quantization, sparsification, and local computations. In *Proc. of NeurIPS*, pages 14668–14679, 2019.

[11] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. SIGNSGD: Compressed optimisation for non-convex problems. In *Proc. of ICML*, pages 559–568, 2018.

[12] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.

[13] Chia-Yu Chen, Jiamin Ni, Songtao Lu, Xiaodong Cui, Pin-Yu Chen, Xiao Sun, et al. Scalecom: Scalable sparsified gradient compression for communication-efficient distributed training. In *NeurIPS*, pages 13551–13563, 2020.

[14] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-state Circuits*, 52(1):127–138, 2016.

[15] Minsik Cho1, Vinod Muthusamy, and Ruchir Nemanich1, Brad Puri. GradZip: Gradient Compression using Alternating Matrix Factorization for Large-scale Deep Learning. In *Proc. of NeurIPS Systems for ML Workshop*, 2019.

[16] Maurice Cox, Peter Harris, and Paul Kenward. Fixed- and free-knot univariate least-square data approximation by polynomial splines. In *Proc. of ISAAC*, pages 330–345, 2001.

[17] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *Proc. of CVPR*, pages 248–255, 2009.

[18] Wei Deng, Junwei Pan, Tian Zhou, Deguang Kong, Aaron Flores, and Guang Lin. Deeplight: Deep lightweight feature interactions for accelerating ctr predictions in ad serving. In *Proc. of WSDM*, pages 922–930, 2021.

[19] Tim Dettmers. 8-bit approximations for parallelism in deep learning. In *Proc. of ICLR*, 2016.

[20] Peter Deutsch. Deflate compressed data format specification version 1.3. *IETF RFC 1951*, 1996.

[21] Ronald A DeVore and George G Lorentz. *Constructive approximation*, volume 303. Springer Science & Business Media, 1993.

[22] Nikoli Dryden, Tim Moon, Sam Ade Jacobs, and Brian Van Essen. Communication quantization for data-parallel training of deep neural networks. In *Proc. of MLHPC*, pages 1–8, 2016.

[23] Aritra Dutta, El Houcine Bergou, Ahmed M. Abdelmoniem, Chen-Yu Ho, Atal Narayan Sahu, Marco Canini, and Panos Kalnis. On the Discrepancy between the Theoretical Analysis and Practical Implementations of Compressed Communication for Distributed Deep Learning. In *Proc. of AAAI*, volume 34, pages 3817–3824, 2020.

[24] Melih Elibol, Lihua Lei, and Michael I Jordan. Variance reduction with sparse gradients. In *Proc. of ICLR*, 2019.

[25] Jiawei Fei, Chen-Yu Ho, Atal Narayan Sahu, Marco Canini, and Amedeo Sapio. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *Proc. of SIGCOMM*, 2021.

[26] Fangcheng Fu, Yuzheng Hu, Yihan He, Jiawei Jiang, Yingxia Shao, Ce Zhang, and Bin Cui. Don't waste your bits! squeeze activations and gradients for deep neural networks via tinyscript. In *Proc. of ICML*, pages 3304–3314, 2020.

[27] Rishikesh R. Gajjala, Shashwat Banchhor, Ahmed M. Abdelmoniem, Aritra Dutta, Marco Canini, and Panos Kalnis. Huffman coding based encoding techniques for fast distributed deep learning. In *Proc. of ACM CoNEXT 1st Workshop on Distributed Machine Learning (DistributedML '20)*, pages 21–27, 2020.

[28] Shiming Ge, Zhao Luo, Shengwei Zhao, Xin Jin, and Xiao-Yu Zhang. Compressing deep neural networks for efficient visual inference. In *Proc. of ICME*, pages 667–672, 2017.

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

[30] Farzin Haddadpour, Mohammad Mahdi Kamani, Aryan Mokhtari, and Mehrdad Mahdavi. Federated learning with compression: Unified analysis and sharp guarantees. In *Proc. of AISTATS*, pages 2350–2358, 2021.

[31] Jenny Hamer, Mehryar Mohri, and Ananda Theertha Suresh. FedBoost: A communication-efficient algorithm for federated learning. In *Proc. of ICML*, pages 3973–3983, 2020.

[32] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *Proc. of ICLR*, 2016.

[33] Chaoyang He, Songze Li, Jinhyun So, Xiao Zeng, Mi Zhang, Hongyi Wang, Xiaoyang Wang, Praneeth Vepakomma, Abhishek Singh, Hang Qiu, et al. Fedml: A research library and benchmark for federated machine learning. *arXiv preprint arXiv:2007.13518*, 2020.

[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. of CVPR*, pages 770–778, 2016.

[35] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural Collaborative Filtering. In *Proc. of WWW*, pages 173–182, 2017.

[36] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[37] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely Connected Convolutional Networks. In *Proc. of CVPR*, pages 2261–2269, 2017.

[38] David A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc. of IRE*, volume 40, pages 1098–1101, 1952.

[39] Jiawei Jiang, Fangcheng Fu, Tong Yang, and Bin Cui. SketchML: Accelerating distributed machine learning with data sketches. In *Proc. of SIGMOD*, pages 1269–1284, 2018.

[40] Jiawei Jiang, Fangcheng Fu, Tong Yang, Yingxia Shao, and Bin Cui. Skcompress: compressing sparse and nonuniform gradient in distributed machine learning. *The VLDB Journal*, pages 1–28, 2020.

[41] Iain M. Johnstone. On minimax estimation of a sparse normal mean vector. *The Annals of Statistics*, pages 271–289, 1994.

[42] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

[43] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Bennis, et al. Advances and open problems in federated learning. *arXiv preprint arXiv:1912.04977*, 2019.

[44] Sai Praneeth Karimireddy, Satyen Kale, Mehryar Mohri, Sashank Reddi, Sebastian Stich, and Ananda Theertha Suresh. SCAFFOLD: Stochastic controlled averaging for federated learning. In *Proc. of ICML*, pages 5132–5143, 2020.

[45] Sai Praneeth Karimireddy, Quentin Rebjock, Sebastian Stich, and Martin Jaggi. Error feedback fixes SignSGD and other gradient compression schemes. In *Proc. of ICML*, pages 3252–3261, 2019.

[46] Diederik P. Kingma and Jimmy Ba. ADAM: A Method for Stochastic Optimization. In *Proc. of ICLR*, 2015.

[47] Jakub Konečný, H. Brendan McMahan, Felix X. Yu, Peter Richtarik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. In *Proc. of NeurIPS Workshop on Private Multi-Party Machine Learning*, 2016.

[48] Alex Krizhevsky, Geoffrey Hinton, et al. Learning Multiple Layers of Features From Tiny Images. *Technical report, University of Toronto*, 1(4), 2009.

[49] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. Federated optimization in heterogeneous networks. In *Proc. of MLSys*, pages 429–450, 2020.

[50] Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. 3LC: Lightweight and Effictive Traffic Compression for Distributed Machine Learning. In *Proc. of MLSys*, pages 53–64, 2019.

[51] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *Proc. of ICLR*, 2018.

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

[53] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy. PHub: Rack-Scale Parameter Server for Distributed Deep Neural Network Training. In *Proc. of SoCC*, pages 41–54, 2018.

[54] Ahmed M Abdelmoniem, Ahmed Elzanaty, Mohamed-Slim Alouini, and Marco Canini. An Efficient Statistical-based Gradient Compression Technique for Distributed Training Systems. In *Proc. of MLSys*, 2021.

[55] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proc. of AISTATS*, volume 54, pages 1273–1282, 2017.

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

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

[58] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *Proc. of SOSP*, pages 16–29, 2019.

[59] Sashank Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečnỳ, Sanjiv Kumar, and H Brendan McMahan. Adaptive federated optimization. In *Proc. of ICLR*, 2021.

[60] Amirhossein Reisizadeh, Aryan Mokhtari, Hamed Hassani, Ali Jadbabaie, and Ramtin Pedarsani. Fedpaq: A communication-efficient federated learning method with periodic averaging and quantization. In *Proc. of AISTATS*, pages 2021–2031, 2020.

[61] Cèdric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. Sparcml: High-performance sparse communication for machine learning. In *Proc.of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2019.

[62] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. Scaling Distributed Machine Learning with In-Network Aggregation. In *Proceedings of NSDI*, 2021.

[63] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. In *Proc. of INTERSPEECH*, pages 1058–1062, 2014.

[64] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[65] Anil Shanbhag, Holger Pirk, and Samuel Madden. Efficient Top-K Query Processing on Massively Parallel Hardware. In *Proceedings of International Conference on Management of Data (SIGMOD)*, page 1557–1570, 2018.

[66] Shaohuai Shi, Xiaowen Chu, Ka Chun Cheung, and Simon See. Understanding top-k sparsification in distributed deep learning. *arXiv:1911.08772*, 2019.

[67] TensorFlow Federated Stack Overflow dataset. `https://www.tensorflow.org/federated/api_docs/python/tff/simulation/datasets/stackoverflow/`.

[68] Sebastian U Stich. Local SGD converges fast and communicates little. In *Proc. of ICLR*, 2018.

[69] Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. Sparsified SGD with memory. In *Proc. of NeurIPS*, pages 4447–4458, 2018.

[70] N. Strom. Scalable distributed DNN training using commodity GPU cloud computing. In *Proc. of INTERSPEECH*, pages 1488–1492, 2015.

[71] Haobo Sun, Yingxia Shao, Jiawei Jiang, Bin Cui, Kai Lei, Yu Xu, and Jiang Wang. Sparse Gradient Compression for Distributed SGD. In *Proc. of DASFAA*, pages 139–155, 2019.

[72] Ananda Theertha Suresh, Felix X. Yu, Sanjiv Kumar, and H. Brendan McMahan. Distributed mean estimation with limited communication. In *Proc. of ICML*, pages 3329–3337, 2017.

[73] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proc. of ICML*, pages 1139–1147, 2013.

[74] T T. Cai, W. G. Sun, and W. Wang. Covariate-assisted ranking and screening for large-scale two-sample inference. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 81(2):187–234, 2019.

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

[76] Yusuke Tsuzuku, Hiroto Imachi, and Takuya Akiba. Variance-based gradient compression for efficient distributed deep learning. In *Proc. of ICLR*, 2018.

[77] Sharan Vaswani, Francis Bach, and Mark Schmidt. Fast and faster convergence of SGD for over-parameterized models and an accelerated perceptron. In *Proc. of AISTATS*, pages 1195–1204, 2019.

[78] Thijs Vogels, Sai Praneeth Reddy Karimireddy, and Martin Jaggi. PowerSGD: Practical low-rank gradient compression for distributed optimization. In *Proc. of NeurIPS*, pages 14259–14268, 2019.

[79] Hongyi Wang, Scott Sievert, Shengchao Liu, Zachary Charles, Dimitris Papailiopoulos, and Stephen Wright. Atomo: Communication-efficient learning via atomic sparsification. In *Proc. of NeurIPS*, pages 9850–9861. 2018.

[80] Jianyu Wang, Qinghua Liu, Hao Liang, Gauri Joshi, and H Vincent Poor. Tackling the objective inconsistency problem in heterogeneous federated optimization. *Advances in Neural Information Processing Systems*, pages 7611–7623, 2020.

[81] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. Gradient sparsification for communication-efficient distributed optimization. In *Proc. of NeurIPS*, pages 1306–1316, 2018.

[82] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Proc. of NeurIPS*, pages 1508–1518, 2017.

[83] Ross N Williams. *Adaptive Data Compression*, volume 110. Springer Science & Business Media, 2012.

[84] Hang Xu, Chen-Yu Ho, Ahmed M Abdelmoniem, Aritra Dutta, Konstantinos Karatsenidis El Houcine Bergou, Marco Canini, and Panos Kalnis. GRACE: A Compressed Communication Framework for Distributed Machine Learning. In *Proc. of ICDCS*, 2021.

[85] Hao Yu, S. Yang, and Shenghuo Zhu. Parallel restarted sgd with faster convergence and less communication: Demystifying why model averaging works for deep learning. In *Proc. of AAAI*, pages 5693–5700, 2019.

[86] Yue Yu, Jiaxiang Wu, and Longbo Huang. Double quantization for communication-efficient distributed optimization. In *Proc. of NeurIPS*, pages 4438–4449, 2019.

[87] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Proc. of NeurIPS*, pages 2595–2603, 2010.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [N/A]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes] ; see Section 5, Appendix B, C, D, and D.1.
   (b) Did you include complete proofs of all theoretical results? [Yes] ; see Section 5, Appendix B, C, D, and D.1.
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [Yes]
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Background

This section complements Section 2 in the main paper by incorporating extra details (Sections A.1 and A.2) and technicalities (Section A.3 and A.4). These are instrumental for the rest of the paper.

Compressed distributed training via SGD. One can consider the traditional data-center DNN training as a special form of FL training, but without privacy. That is, the dataset is partitioned into all participating compute nodes. Moreover, in contrast to a fraction of participating clients in FL, all nodes update the model parameter, x. Additionally, the synchronization happens at each iteration, instead of a few local epochs at the nodes. Formally, during back-propagation with n workers (i.e., compute nodes), to update the model parameter x, each worker i, at each iteration, calculates a stochastic gradient g_t^i by processing an independent batch of data, D_i with $\bigcup_i D_i = D$, the global dataset. Often, for efficient communication, the gradient is compressed to $\tilde{g}_t^i$ and is communicated to all workers, either through a parameter server [58], or through a peer-to-peer collective, like `Allreduce` [64]. The aggregated gradient, $\tilde{g}_t = \frac{1}{n}\sum_{i=1}^n \tilde{g}_t^i$ is then transmitted to all workers, who update the parameters of their local model via: $x_{t+1} = x_t - \eta_t \tilde{g}_t$, where $\eta_t > 0$ is the learning rate; the process repeats until convergence. Figure 9 shows an example of distributed training for DNNs with compressed communication.

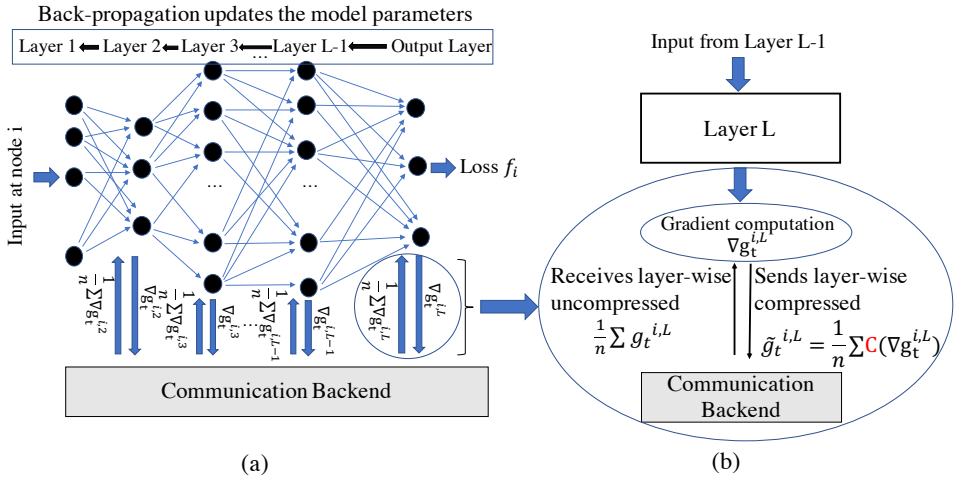


Figure 9: Distributed training from the perspective of i^{th} computing node.

Random-r, Top-r sparsifiers, and δ-compressors. Based on the selection criteria of the elements in S, some of the most commonly used sparsifiers such as, Random-r [69] and Top-r [6, 8] are defined. For Random-r, the elements of S are randomly selected out of $[d]$, whereas, for Top-r, the elements of S correspond to the indices of the r highest magnitude elements in g.Moreover, sparsifiers that follow (6) with $\Omega = 1 - \delta$, and $\delta \in (0, 1]$, are known as δ-*compressors* and denoted by $\mathcal{C}_\delta$. That is,

$$\mathbb{E}\|g - \mathcal{C}_\delta(g)\|^2 \leq (1-\delta)\|g\|^2. \tag{1}$$

Remark 1. Both Top-r and Random-r are δ-*compressors* with $\delta = \frac{r}{d}$, and $\mathbb{E}\|g - \text{Top}r(g)\|^2 \leq \mathbb{E}\|g - \text{Random}r(g)\|^2 = (1 - r/d)\|g\|^2$, for all $g \in \mathbb{R}^d$.

A.1 Lossless encoding strategies

In this section, we explain two *lossless* strategies that can be used in the DeepReduce framework. Discussion pertaining to their implementation is given in Section F.2.

Run Length Encoding (RLE) [83] is a *lossless* compressor in which consecutive occurrences of symbols are encoded as $\langle frequency, symbol\rangle$ tuples. For example, string "$aaaabaa$" is encoded as: $(4, \text{“}a\text{”}), (1, \text{“}b\text{”}), (2, \text{“}a\text{”})$. RLE is used to compress large sequences of repetitive data. In this work, we employ bit-level RLE, where symbols are 0 or 1, for index compression.

Huffman encoding [38] is a *lossless* scheme that assigns the optimal average decode-length prefix codes, using a greedy algorithm to construct a Huffman code tree. Higher frequency symbols are encoding with fewer bits. For instance, string "$aaaabaacaabaa$" generates mapping ("a", "b", "c") $\rightarrow (0, 10, 11)$ resulting to the following encoding: 0000**10**00**11**00**1**000. Huffman encoding has been used to compress DNN weights [28, 32], as well as sparse gradient indices (e.g., SKCompress [40]).

A.2 Further details on classic Bloom filter

The following Lemma characterizes the probability of the false-positive rates in a Bloom filter and Figure 10 is an example of a Bloom filter.

Lemma 3. *[12] Let k denote the number of independent hash functions, m the dimension of the bit-string, and r the cardinality of the index set, S. Then the probability of the false-positive rate (FPR) is $\epsilon \approx (1 - e^{-kr/m})^k$.*

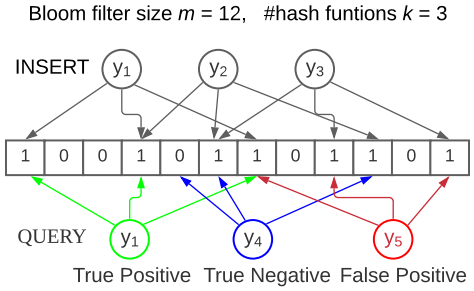


Figure 10: The figure illustrates an example Bloom filter $\mathcal{B}$ with $m = 12$ bits and $k = 3$ hash functions, representing a set, $S = \{y_1, y_2, y_3\}$. During querying, y_1 and y_4 are correctly identified as belonging (i.e., true positive) and not belonging (i.e., true negative) to $\mathcal{B}$, respectively. In contrast, y_5 is wrongly identified (i.e., false positive) as belonging to $\mathcal{B}$.

Remark 2. Given ϵ and r, the optimal $m = -\frac{r \log \epsilon}{(\log 2)^2}$ and $k = -\frac{\log \epsilon}{\log 2}$. Given m and r, the number of hash functions that minimizes the probability of false positives is $k = \frac{m}{r} \log 2$. This k results in the probability of false positive, ϵ as $\log \epsilon = -\frac{m}{r}(\log 2)^2$. In practice, we need to calculate the bits in the filter by using the relation $m = \frac{-r \log \epsilon}{(\log 2)^2}$ and the number of hash functions by $k = \frac{-\log \epsilon}{\log 2}$.

A.3 Inequalities used in this paper

1. If $a, b \in \mathbb{R}^d$ then the Peter-Paul inequality is: There exists a $\xi > 0$ such that
$$\|a + b\|^2 \leq (1 + \xi)\|a\|^2 + (1 + \frac{1}{\xi})\|b\|^2. \tag{2}$$
We generally use a relaxed version of the above inequality as follows:
$$\|a + b\|^2 \leq 2\|a\|^2 + 2\|b\|^2. \tag{3}$$
2. If $a, b \in \mathbb{R}^d$ then we have
$$2\langle a, b\rangle \leq 2\|a\|^2 + \frac{1}{2}\|b\|^2. \tag{4}$$
3. For $x_i \in \mathbb{R}^d$ we have:
$$\|\sum_{i=1}^{n} x_i\|^2 \leq n \sum_{i=1}^{n} \|x_i\|^2. \tag{5}$$
4. If the operator $\mathcal{C} : \mathbb{R}^d \rightarrow \mathbb{R}^d$ is a *compressor* then there exists $\Omega > 0$ such that
$$\mathbb{E}\|g - \mathcal{C}(g)\|^2 \leq \Omega \|g\|^2. \tag{6}$$
5. If X is a random variable then:
$$\mathbb{E}\|X\|^2 = \|\mathbb{E}[X]\|^2 + \underbrace{\mathbb{E}[\|X - E[X]\|^2]}_{\text{Var}(X)}. \tag{7}$$

A.4 Preliminary results

The next two Lemmas are instrumental in proving other compression related results.

Lemma 4. *Let $x \in \mathbb{R}^d$ and x_S be a vector that has the components of x arranged in ascending/descending order of magnitude. If $0 \leq \theta < \pi/2$ be the angle between x and x_S, then $\|x - x_S\|^2 = 2(1 - \cos\theta)\|x\|^2$.*

Proof. We have

$$\|x - x_S\|^2 = \|x\|^2 + \|x_S\|^2 - 2\langle x, x_S\rangle \overset{\|x\|=\|x_S\|}{=} 2\|x\|^2 - 2\|x\|^2\cos\theta = 2(1 - \cos\theta)\|x\|^2.$$

Hence the result. □

Lemma 5. *Let $\mathcal{C}(\cdot) : \mathbb{R}^d \to \mathbb{R}^d$ be a δ-compressor.*

*(*i*) If $\mathcal{C}_\delta(g)$ is unbiased then $\mathbb{E}\|\mathcal{C}_\delta(g)\|^2 \leq (2 - \delta)\|g\|^2$.*

*(*ii*) If $\mathcal{C}_\delta(g)$ is biased then $\mathbb{E}\|\mathcal{C}_\delta(g)\|^2 \leq 2(2 - \delta)\|g\|^2$.*

Proof. (*i*) Recall from (1), for δ-compressors, we have $\mathbb{E}\|g - \mathcal{C}_\delta(g)\|^2 \leq (1 - \delta)\|g\|^2$. Since $\mathbb{E}(\mathcal{C}_\delta(g)) = g$, from (7) we have,

$$\mathbb{E}\|\mathcal{C}_\delta(g)\|^2 \overset{\text{By (7)}}{=} \mathbb{E}\|g - \mathcal{C}_\delta(g)\|^2 + \|g\|^2 \overset{\text{By (1)}}{\leq} (1 - \delta)\|g\|^2 + \|g\|^2 = (2 - \delta)\|g\|^2.$$

(*ii*) On the other hand, for biased compressors, by (3) we have,

$$\mathbb{E}\|\mathcal{C}_\delta(g)\|^2 = \mathbb{E}\|g - g + \mathcal{C}_\delta(g)\|^2 \overset{\text{By (3)}}{\leq} 2\mathbb{E}\|g - \mathcal{C}_\delta(g)\|^2 + 2\|g\|^2 \overset{\text{By (1)}}{\leq} 2(1-\delta)\|g\|^2 + 2\|g\|^2 = 2(2-\delta)\|g\|^2.$$

□

B Bloom filter based index compression

In this section, we discuss in details different Bloom filter policies.

Overview. This Section serves as an addendum to Section 4 in the main paper and incorporates detailed discussions, examples, pseudocode, theoretical results, and their proofs. We start with an example in B.1 to illustrate the Naïve compression. Section B.2 provides proofs the Lemmas discussed in the main paper related to policy $P0$. In Section B.3, we discussed a new policy, called deterministic policy that sets the stage for more complex policies, random approach, Policy $P1$ (Section B.4) and conflict sets policy, Policy $P2$ (Section B.5).

B.1 Naïve compression

In this scope we explain Naïve compression by a simple illustration, see Figure 11. We have a dense gradient represented as a sparse tensor in a key-value format. Notice that, the values in the sparse representation are sorted by their indices in an increasing order. The set, S of keys is represented as a bloom filter. To communicate the sparse tensor we send both the values and the bloom filter. During the phase of decompression, we try to reconstruct S by following the process we described. However, in this case, we manage to retrieve only 4 out of the 5 elements of S. Index 4 does not belong to S and corresponds to a FP response. The mapping, $\mathcal{M}$ scans the communicated values in the order they arrive and assigns each one of them to the next larger index from the set of decoded indices. Notice how the selection of one wrong index affects the decompression by causing re-arrangements or shifts of the reconstructed gradient components with respect to their true positions.

B.2 Policy P0

We provide the theoretical results involving policy $P0$ stated in the main paper.

Lemma 6. *The cardinality of the set P is at most $\lceil r + (\frac{1}{2})^{-\frac{\log(\epsilon)}{\log(2)}}(d - r)\rceil$ and approaches to r as $\epsilon \to 0$.*

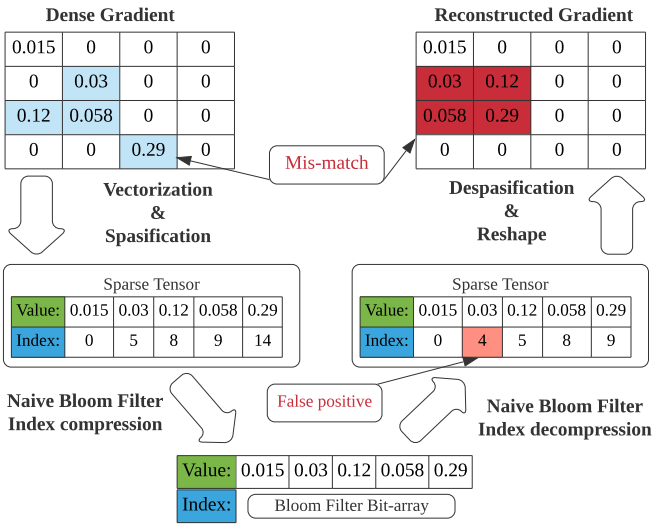


Figure 11: Naïve Bloom filter example that demonstrates how the FP elements in the Bloom filter, can cause re-arrangements or shifts of the reconstructed gradient components with respect to their true positions.

Proof. For given ϵ, r, d the cardinality of the set of positives P follows

$$r \leq |P| \leq \lceil r + \epsilon(d-r) \rceil \overset{\text{By Lemma 3}}{\approx} \lceil r + (1 - e^{-kr/m})^k (d-r) \rceil.$$

Given ϵ and r, the optimal $m = -\frac{r \log \epsilon}{(\log 2)^2}$ and $k = -\frac{\log \epsilon}{\log 2}$. Therefore, plugging them in the above expression, we get

$$r \leq |P| \leq \lceil r + \left(\frac{1}{2}\right)^{-\frac{\log(\epsilon)}{\log(2)}} (d-r) \rceil,$$

which after taking limit $\epsilon \to 0$, gives the desired result. □

The next Lemma measures compression error due to compressed gradient, $\mathcal{C}_{\mathcal{P}_0,\delta}(g)$.

Lemma 7. *(*i*) For a general δ-compressor, $\mathcal{C}_\delta$, there exists a $\beta \in [0,1)$, $\beta \geq \delta$ such that the compression error due to a compressed gradient, $\mathcal{C}_{\mathcal{P}_0,\delta}(g)$ resulted from $\mathcal{P}_0$ is $\mathbb{E}\|g - \mathcal{C}_{\mathcal{P}_0,\delta}(g)\|^2 \leq (1-\beta)\|g\|^2$. (*ii*) For inherently sparse gradient, g, with $\mathcal{C}_\delta = I_d$, we have $\beta = \delta = 1$.*

Proof. (*i*) Consider a δ sparsifier, $\mathcal{C}_\delta$ such that $\|\mathcal{C}_\delta(g)\|_0 = r$. If $\mathcal{P}_0$ is used then, by Lemma 6, $|P| = r + (\frac{1}{2})^{-\frac{\log \epsilon}{\log 2}}(d-r) \geq r$ for $\epsilon \geq 0$, ($|P| = r$ for $\epsilon = 0$). Therefore, for $\mathcal{C}_{\mathcal{P}_0,\delta}$, we have $\beta = |P|/d > \delta$ resulting $\mathbb{E}\|g - \mathcal{C}_{\mathcal{P}_0,\delta}(g)\|^2 \leq (1-\beta)\|g\|^2$.

(*ii*) For inherently sparse gradient, g, with $\mathcal{C}_\delta = I_d$, we have $\delta = 1$, and $\|\mathcal{C}_{I_d}(g)\|_0 = r$. Therefore, for policy $\mathcal{P}_0$, we have $\|\mathcal{C}_{\mathcal{P}_0,I_d}(g)\|_0 = r$ resulting $\beta = \delta = 1$.

□

Remark 3. We consider two extreme cases of δ sparsifier. For Random-r, $\delta = r/d \leq 1$. If $\mathcal{P}_0$ is used then $|P| = r + (\frac{1}{2})^{-\frac{\log \epsilon}{\log 2}}(d-r) > r$ for $\epsilon > 0$. In this case, for $\mathcal{C}_{\mathcal{P}_0,\delta}$ we have $\beta = |P|/d > \delta$.

In another extreme case, for $\mathcal{C}_\delta$ to be Top-r, by Remark 1, $\delta \leq r/d \leq 1$. For $g \in \mathbb{R}^d$, similar argument as above gives us:

$$\mathbb{E}\|g - \mathcal{C}_{\mathcal{P}_0,\delta}\|^2 < \mathbb{E}\|g - \text{Top}r(g)\|^2 \leq \mathbb{E}\|g - \text{random}r(g)\|^2 \leq (1 - r/d)\|g\|^2.$$

Lemma 6 show that for small ϵ, no policy sends negligible amount of extra data compared to the other policies. The GRACE [84] sparsification library allows to use the original dense gradient g, instead of $\tilde{g}$, to populate V. Consequently, all elements corresponding to false positives (i.e., set

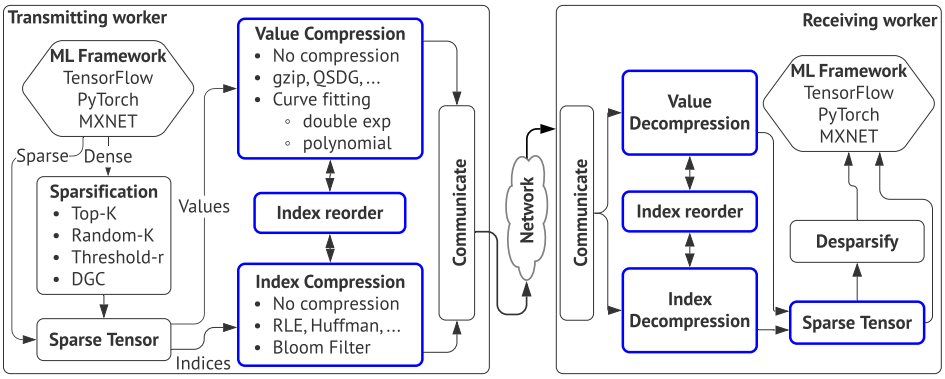


Figure 12: DeepReduce system architecture (highlighted in blue). DeepReduce resides between the machine learning framework (e.g., TensorFlow, PyTorch) and the communication library, and is optimized for federated DNN training. It offers a simple API whose functions can be overridden to implement, with minimal effort, a wide variety of index and value compression methods for sparse tensors. At the transmitting worker side, the input to DeepReduce is a sparse tensor, which is fed directly from the ML framework, for the case of inherently sparse models; or, is generated by an explicit sparsification process. Sparse tensors are typically represented as $\langle key, value \rangle$ tuples. DeepReduce decouples the keys from the values and constructs two separate data structures. Let $\tilde{g} \in \mathbb{R}^d$ be the sparse gradient, where d is the number of model parameters and $\|\tilde{g}\|_0 = r$, is the number of nonzero gradient elements. Let S be the set of r indices corresponding to those elements. DeepReduce implements two equivalent representations of S: (*i*) an array of r integers; and (*ii*) a bit string B with d bits, where $\forall i \in [1, d], B[i] = 1$ if and only if $\tilde{g}[i] \neq 0$. These two representations are useful for supporting a variety of index compressors. The Index Compression module encapsulates the two representations and implements several algorithms for index compression. It supports both *lossy* compressors (e.g., our Bloom filter-based proposal), as well as *lossless* ones, such as the existing Run Length (RLE) [83] and Huffman [38, 27] encoders; there is also an option to bypass index compression. The Value Compression module receives the sparse gradient values and compresses them independently. Several compressors, such as Deflate [20] and QSGD [7], are implemented, in addition to our own curve fitting-based method. Again, there is an option to bypass value compression. Some value compressors (e.g., our own proposals), require reordering of the gradient elements, which is handled by the Index reorder module. DeepReduce then combines in one container the compressed index and value structures, the reordering information and any required metadata; the container is passed to the communication library. The receiving worker, at the right of the figure, mirrors the structure of the transmitter, but implements the reverse functions, that is, index and value decompression, and index reordering. The reconstructed sparse gradient is routed for de-sparsification, or passed directly to the ML framework. DeepReduce is general enough to represent popular existing methods that employ proprietary combined value and index compression. E.g., SKCompress [40] can be implemented in DeepReduce as follows: SketchML [39] plus Huffman for values, no index reordering, and delta encoding plus Huffman for indices.

$P - S$) receive the original, instead of zero values. Lemma 7 (*i*) shows that for sparsified vectors, no policy achieves a better compression factor than the original sparsifier, $\mathcal{C}_\delta$. However, for inherently sparse tensors, Lemma 7 (*ii*) shows that no policy is *lossless* and is the best choice.

B.3 Deterministic policy

This policy deterministically selects a subset of r elements from P and is denoted by $\mathcal{P}_D$. One can select the first r, the middle r, or the last r elements from P, and based on this denote them as, leftmost-r, middle-r, and rightmost-r policy, respectively. For implementation, the set $\tilde{S}$ can be created while iterating and posing queries on the universe U and once it has r elements, the querying is stopped. Let $\mathcal{C}_\delta$ be a general δ-compressor that selects r gradient components. However, with a policy, $\mathcal{P}_D$, not all the r selected indices are due to $\mathcal{C}_\delta$. Let I_1 denote the set of indices that are selected via policy $\mathcal{P}_D$ originally resulted from $\mathcal{C}_\delta$ sparsifier and let I_2 denote the set of the rest of the $(r - |I_1|)$ indices. Therefore, $\tilde{S} = I_1 \bigcup I_2$ and let $\mathcal{C}_{\mathcal{P}_D,\delta}(g)$ be the compressed gradient whose indices are drawn via policy $\mathcal{P}_D$ and has support $\tilde{S}$. The following lemma quantifies the compression error.

Deterministic policy error. Lemma 8 gives the compression error bound for Deterministic policies.

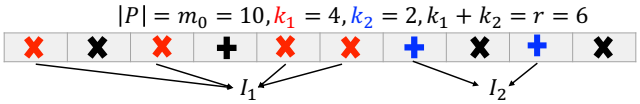


Figure 13: Random policy, policy $P0$ example with size of $|P| = m_0 = 10$, ✖ and ✚ denote TP and FP, respectively. Policy $P1$ selects set I_1 (in ✖), with $k_1 = 4$ elements and I_2 (in ✚), with $k_2 = 2$ elements, and $r = 6$.

Lemma 8. *(i) For all deterministic policies, $\mathcal{P}_D$ and for an inherently sparse gradient, g, the compression error due to a compressed gradient, $\mathcal{C}_{\mathcal{P}_D,I_d}(g)$ is given by $\mathbb{E}_{\tilde{S}}\|g - \mathcal{C}_{\mathcal{P}_D,I_d}(g)\|^2 = (1 - \frac{|I_1|}{r})\|g\|^2$.*

(ii) For all deterministic policies, $\mathcal{P}_D$ and a general $\mathcal{C}_\delta$, the compression error due to a compressed gradient, $\mathcal{C}_{\mathcal{P}_D,\delta}(g)$ is given by $\mathbb{E}_{\tilde{S}}\|g - \mathcal{C}_{\mathcal{P}_D,\delta}(g)\|^2 = (1 - \frac{r}{d})\|g\|^2$.

The proof of Lemma 8 follows from the standard procedure of taking expectation with respect to all possible $|I_1|$ cardinality subsets formed from the set P by using policy, $\mathcal{P}_D$ and follows the same structure as the proof of Lemma 9 (*i*) and (*ii*). We omit the proof.

Lemma 8 (*i*) shows that by adopting a deterministic policy, $\mathcal{P}_D$ on the support of a general δ-compressor, the compression error in expectation is as good as using a Random-r sparsifier on the original gradient. Moreover, Lemma 8 (*ii*) shows that by adopting a deterministic policy, $\mathcal{P}_D$ on the support of a inherently sparse vector, the compression error in expectation is as good as using a Random-$|I_1|$ sparsifier on the original gradient. Therefore, it creates a *lossy* compression whose compression factor is as good as a Random-$|I_1|$ compressor on r elements.

B.4 Random Policy: Policy P1

First, we define the random policy, Policy $P1$ in detail, and then inspect the error incurred due to $P1$.

Random approach: Policy P1. A deterministic policy, $\mathcal{P}_D$ may be prone to bias, based on how the gradient components are distributed and the sparsifier used. E.g. If the support set of the sparsifier, $\mathcal{C}_\delta$ is concentrated at the beginning of P, then except leftmost-r, other deterministic policies such as, middle-r and rightmost-r incur more bias as they select more elements in I_2 than I_1. Similarly, if the support set of the sparsifier, $\mathcal{C}_\delta$, concentrated at the center, then only middle-r policy is expected to incur the least bias compare to the others as it selects more elements in I_1 than in I_2. Without knowing the distribution of the gradient components, it is hard to invoke a deterministic policy. A random policy, $\mathcal{P}_R$ forms $\tilde{S}$ by picking r indices randomly from P. Without loss of generality, consider the only source of randomness here is due to the random selection of r indices and is unaffected by other source of randomness—randomness in the i.i.d. data, independence of the hash functions, etc.

Let $|P| = m_0$. Let policy $\mathcal{P}_R$ chooses a set I_1 of k_1 elements from the support set of a δ-compressor $\mathcal{C}_\delta$ without replacement. Let the rest k_2 elements belong to the set I_2 such that, $k_1 + k_2 = r$. We illustrate this in Figure 13. By this, we incur two types of errors with respect to the vector g_P. The first error, E_1, is due to the compressed gradient g_{I_1}. The second error, E_2, is due to the compressed gradient g_{I_2}. We have, $\mathcal{C}_{\mathcal{P}_R,\delta}(g) := g_{I_1} \bigoplus g_{I_2}$. In the following Lemma, we measure the compression error due to compressed gradient $\mathcal{C}_{\mathcal{P}_R,\delta}(g)$ with respect to g_P.

Lemma 9. *With the notations mentioned above, we have the following measures of the compression error:*
(i) $E_1 = \mathbb{E}_{\mathcal{P}_R}\|g_P - g_{I_1}\|^2 = (1 - \frac{k_1}{r})\|g_P\|^2$.
(ii) $E_2 = \mathbb{E}_{\mathcal{P}_R}\|g_P - g_{I_2}\|^2 = (1 - \frac{k_2}{m_0 - r})\|g_P\|^2$.
(iii) Denote $E := \mathbb{E}_{\mathcal{P}_R}\|g_P - \mathcal{C}_{\mathcal{P}_R,\delta}(g)\|^2$ be the total compression error due to the compressed gradient $\mathcal{C}_{\mathcal{P}_R,\delta}(g)$ with respect to g_P. Then, $E \leq E_1 + E_2$.

Proof. Let Ω_{k_1} and Ω_{k_2} denote the set of all k_1 and k_2 elements subsets of the sets having cardinality r and $m_0 - r$, respectively. The first error, E_1, is due to the compressed gradient g_{I_1} whose support belongs to Ω_{k_1}. The second error, E_2, is due to the compressed gradient g_{I_2} whose support belongs to Ω_{k_2}. With the notations mentioned above, we have

(*i*)

$$E_1 = \mathbb{E}_{\mathcal{P}_R}\|g_P - g_{I_1}\|^2 = \frac{1}{|\Omega_{k_1}|}\sum_{I_1\in\Omega_{k_1}}\sum_{i=1}^{m_0} g_i^2\mathbb{I}\{i\notin I_1\} = \|g_P\|^2\left(\frac{1}{\binom{r}{k_1}}\frac{r-k_1}{r}\binom{r}{k_1}\right) = (1-\frac{k_1}{r})\|g_P\|^2.$$

(*ii*) Similarly, we have $E_2 = \mathbb{E}_{\mathcal{P}_R}\|g_P - g_{I_2}\|^2 = \frac{1}{|\Omega_{k_2}|}\sum_{I_2\in\Omega_{k_2}}\sum_{i=1}^{m_0} g_i^2\mathbb{I}\{i \notin I_2\} = \|g_P\|^2\left(\frac{1}{\binom{m_0-r}{k_2}}\frac{m_0-r-k_2}{m_0-r}\binom{m_0-r}{k_2}\right) = (1-\frac{k_2}{m-r})\|g_P\|^2$.

(*iii*) Denote $E := \mathbb{E}_{\mathcal{P}_R}\|g_P - \mathcal{C}_{\mathcal{P}_R,\delta}(g)\|^2$ be the total compression error due to the compressed gradient $\mathcal{C}_{\mathcal{P}_R,\delta}(g)$ with respect to g_P. Then, by using the linearity of expectation, we have

$$\begin{aligned}
E &= \mathbb{E}_{\mathcal{P}_R}\|g_P - \mathcal{C}_{\mathcal{P}_R,\delta}(g)\|^2 \\
&= \mathbb{E}_{\mathcal{P}_R}\|g_P - g_{I_1}\bigoplus g_{I_2}\|^2 \\
&\overset{\langle g_P, g_{I_1}\bigoplus g_{I_2}\rangle = \sum_{i\in I_1\bigcup I_2} g_i^2}{=} \mathbb{E}_{\mathcal{P}_R}\|g_P\|^2 + \mathbb{E}_{\mathcal{P}_R}\|g_{I_1}\bigoplus g_{I_2}\|^2 - 2\mathbb{E}_{\mathcal{P}_R}\left(\sum_{i\in I_1\bigcup I_2} g_i^2\right) \\
&\overset{\langle g_{I_1}, g_{I_2}\rangle = 0}{=} \mathbb{E}_{\mathcal{P}_R}\|g_P\|^2 + \mathbb{E}_{\mathcal{P}_R}\|g_{I_1}\|^2 + \mathbb{E}_{\mathcal{P}_R}\|g_{I_2}\|^2 - 2\mathbb{E}_{\mathcal{P}_R}\left(\sum_{i\in I_1\bigcup I_2} g_i^2\right).
\end{aligned}$$

On the other hand,

$$\begin{aligned}
E_1 + E_2 &= 2\mathbb{E}_{\mathcal{P}_R}\|g_P\|^2 + \mathbb{E}_{\mathcal{P}_R}\|g_{I_1}\|^2 + \mathbb{E}_{\mathcal{P}_R}\|g_{I_2}\|^2 - 2\mathbb{E}_{\mathcal{P}_R}\left(\sum_{i\in I_1} g_i^2\right) - 2\mathbb{E}_{\mathcal{P}_R}\left(\sum_{i\in I_2} g_i^2\right) \\
&= 2\mathbb{E}_{\mathcal{P}_R}\|g_P\|^2 + \mathbb{E}_{\mathcal{P}_R}\|g_{I_1}\|^2 + \mathbb{E}_{\mathcal{P}_R}\|g_{I_2}\|^2 - 2\mathbb{E}_{\mathcal{P}_R}\left(\sum_{i\in I_1\bigcup I_2} g_i^2\right),
\end{aligned}$$

together with $\mathbb{E}_{\mathcal{P}_R}\|g_P\|^2 \geq 0$ implies $E \leq E_1 + E_2$. □

To measure the compression error due to the compressed gradient $\mathcal{C}_{\mathcal{P}_R,\delta}(g)$ with respect to the full gradient vector g, by Lemma 9, there exists an $\alpha \in \mathbb{R}^+$ such that the total expected compression error:

$$\mathbb{E}\|g - \mathcal{C}_{\mathcal{P}_R,\delta}(g)\|^2 = E + \sum_{i\in P^c} g_i^2 \leq \alpha\|g\|^2. \tag{8}$$

But there is no guarantee that $\alpha \in [0, 1)$. On the other hand, by Remark 1 we have:

$$\mathbb{E}\|g - \mathcal{C}_{\mathcal{P}_R,\delta}(g)\|^2 = \mathbb{E}\|g - \text{Random}r(g)\|^2 \leq (1 - \tfrac{r}{d})\|g\|^2, \tag{9}$$

which guarantees $\mathcal{C}_{\mathcal{P}_R,\delta}$ to be a δ-compressor. Additionally, the sparsifier, $\mathcal{C}_{\mathcal{P}_R,\delta}(g)$ is an hybrid sprasifier—It has some attributes of the original sparsifier, $\mathcal{C}_\delta$, but we are unsure which k_1 and k_2 random elements are selected via policy $\mathcal{P}_R$. If $k_1 = 0$, then $\mathcal{C}_{\mathcal{P}_R,\delta}(g)$ is Random-r [69] sparsifier. If $k_2 = 0$, then $\mathcal{C}_{\mathcal{P}_R,\delta}(g)$ is $\mathcal{C}_\delta$ sparsifier. Furthermore, if $\mathcal{C}_\delta$ is Top-r, then $\mathcal{C}_{\mathcal{P}_R,\delta}(g)$ is similar to hybrid random-Top-r sparsifier by Elibol et al. [24]. If $\mathcal{C}_\delta$ is Top-r, $k_1 \leq r, k_2 = 0$, then it is random-Top-k_1 sparsifier of Barnes et al. [9].

For inherently sparse vectors g, with $\mathcal{C}_\delta = I_d$, we have $\mathcal{C}_{\mathcal{P}_R,I_d}(g) = g_{I_1}$ and Lemma 9 holds. Moreover, by (8) we have:

$$\mathbb{E}\|g - \mathcal{C}_{\mathcal{P}_R,I_d}(g)\|^2 = (1 - \frac{k_1}{r})\|g\|^2. \tag{10}$$

That is, the policy creates a *lossy* compression with compression factor as good as a Random-k_1 compressor.

Algorithm 1 Construct bloom filter for policy BF-P2

Input: Bloom filter $\mathcal{B}$ of size m, k-hash functions h_i, gradient dimensionality d, empty set P, empty conflict sets C_j, empty set $\tilde{S}$, target number of decompressed indices r
Output: A set of decompressed indices $\tilde{S}$
for $i = 1$ *to* d **do** /* all d elements of gradient $\tilde{g} \in \mathbb{R}^d$ */
  if $i \in \mathcal{B}$ **then** insert i in P
for *each* $x \in P$ **do**
  for $i = 1$ *to* k **do** insert x in $C_{h_i(x)}$
Sort conflict-sets in C by their sizes in ascending order
while $size(\tilde{S}) < r$ **do**
  for *each* $C_j \in C$ **do**
    if $|C_j| = 1$ **then** Insert C_j in $\tilde{S}$; Remove C_j from C
    else
      Remove from C_j items that exist in $\tilde{S}$ Insert into $\tilde{S}$ a random item from C_j
return $\tilde{S}$

B.5 Algorithmic details of conflict set policy: Policy P2

In the following, we explain the conflict set policy, P2. Pseudocode in Algorithm 1 presents the details. Lines 1-2 construct set P, i.e., the union of the true and false positive responses of $\mathcal{B}$. Lines 3-4 re-hash the items of P into $\mathcal{B}$ to construct conflict sets $C_{1...j}$, where j is equal to the number of "1"s in $\mathcal{B}$. For lack of better information, we assume that the true positives are uniformly distributed across the conflict sets; therefore, the probability of drawing a true positive out of a smaller conflict set is higher. To prioritize such sets, Line 5 sorts $C_{1...j}$ in ascending size order. Then, lines 6-11 repeatedly draw items out of the conflict sets until the size of $\tilde{S}$ reaches our target size r. If a set C_j is initially a singleton, its item is a true positive; thus it is added to $\tilde{S}$. Else, we remove from C_j any items that already exist in $\tilde{S}$ (observe there may be duplicates among conflict sets), and add randomly a remaining item to $\tilde{S}$.

C Approximation error due to polynomial fit

We discuss the missing details of value fitting (Section 5) in the following and then provide the compression error from value fitting in Section C.1. The first result is concerning the knot selections.

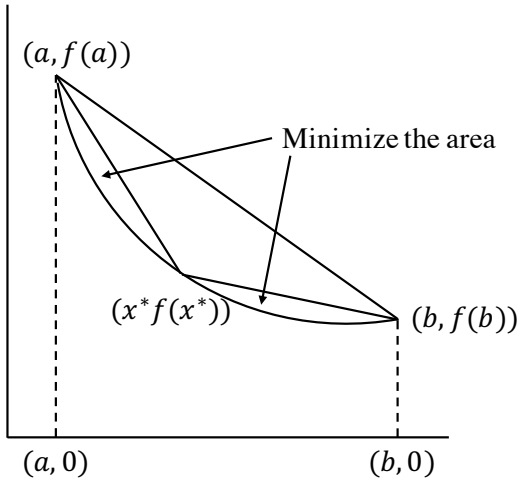


Figure 14: Knot selection for $y = f(x)$ on the interval $[a, b]$.

Lemma 10. *(Knot selection) If f is a differentiable convex function on $[a, b]$, then the point $x^* \in (a, b)$ that gives the best approximation to f using line segments from $(a, f(a))$ to $(x^*, f(x^*))$, and then*

from $(x^*, f(x^*))$ to $(b, f(b))$ is the same point x^+ that maximizes the difference between f and the line segment connecting $(a, f(a))$ to $(b, f(b))$.

Proof. The goal is to find x^* such that it minimizes $\int_a^{x^*} \left[\frac{f(x^*)-f(a)}{x^*-a}(x-a)+f(a)-f(x)\right]dx + \int_{x^*}^b \left[\frac{f(b)-f(x^*)}{b-x^*}(x-b)+f(b)-f(x)\right]dx$ among all choices of $x^* \in (a,b)$; see Figure 14.

Let

$$F(t) = \int_a^t \left[\frac{f(t)-f(a)}{t-a}(x-a)+f(a)-f(x)\right]dx + \int_t^b \left[\frac{f(b)-f(t)}{b-t}(x-b)+f(b)-f(x)\right]dx.$$

Then

$$\begin{aligned} F'(t) &= \int_a^t \frac{d}{dt}\left[\frac{f(t)-f(a)}{t-a}(x-a)\right]_t dx + \int_t^b \frac{d}{dt}\left[\frac{f(b)-f(t)}{b-t}(x-b)\right]_t dx \\ &= = \frac{1}{2}[f'(t)(b-a)-(f(b)-f(a))]. \end{aligned}$$

So, the critical point satisfies

$$f'(t)(b-a)-(f(b)-f(a)) = 0,$$
$$\text{that is,} \qquad f'(t) = \tfrac{f(b)-f(a)}{b-a}. \tag{11}$$

Now consider,

$$A(x) = \max_{x\in[a,b]} \left[\frac{f(b)-f(a)}{b-a}(x-a)+f(a)-f(x)\right].$$

It is easy to see that the critical point of the function $A(x)$ alone satisfies $f'(x) = \frac{f(b)-f(a)}{b-a}$, which is the same solution as in (11). □

Polynomial regression. As mentioned in Section 5, over each sorted segment, we apply a polynomial regression. In our experiments, we usually take the degree of the polynomial as 5. The following result concerns piece-wise constant approximation.

Lemma 11. *[21] (Error of piece-wise constant fit) For $\mathcal{C}_S(g) \in C([1,d])$ and $M > 0$ the following are equivalent:*
*(*i*) $\mathrm{Var}_{[1,d]}(\mathcal{C}_S(g)) \le M$ and (*ii*) $\|s - \mathcal{C}_S(g)\|_\infty \le \frac{M}{2p+2}$, for some $s \in \mathcal{S}_p^0$— the set of piece-wise constant splines with p knots.*

Proof of the above Lemma follows from [21].

Remark 4. We can also give a heuristic to calculate p from Lemma 11. First, calculate $M = |(\mathcal{C}_S(g)[1] - \mathcal{C}_S(g)[2]) - (\mathcal{C}_S(g)[d-1] - \mathcal{C}_S(g)[d])|$. By considering the error bound $\frac{M}{2p+2}$ as a function of p, we can find closed-form solution for p as $p = \lceil \frac{M}{\sqrt{2}} - 1 \rceil$.

However, for piece-wise linear fit, we propose the following result with an explicit constant.

Lemma 12. *(Error of piece-wise linear fit with explicit constants) For $\mathcal{C}_S(g) \in C^1([1,d])$ with $\mathrm{Var}_{[1,d]}(\mathcal{C}_S'(g)) \le M$, we have $\|s - \mathcal{C}_S(g)\|_\infty \le \frac{2M}{p^2}$, for some $s \in \mathcal{S}_p^1$— the set of piece-wise linear splines with p knots.*

Proof. Let $p' = \lceil p/2 \rceil$, the integer part of $p/2$, and let $s^0 \in S_p^0$ satisfy $|\mathcal{C}_S'(g)(t) - s^0(t)| \le \frac{M}{2p'+2}$, as guaranteed by Lemma 11. Denote $V^1 := \int_1^r |\mathcal{C}_S'(g)(t) - s^0(t)|dt$. Choose p' knots $\{t_j\}$ in $(1,d)$ such that $\int_{t_j}^{t_{j+1}} |\mathcal{C}_S'(g) - s^0(t)|dt = \frac{V^1}{p'+1}$. Define $s(x) = \int_{t_j}^x s^0(t)dt + f(t_j)$ for $x \in [t_j, t_{j+1})$, $j = 0, 1, ..., p'$. Then s is a piece-wise linear spline function with possible knots at $2p' \le p$ points such that, for $x \in [t_j, t_{j+1})$, $|\mathcal{C}_S(g)(x) - s(x)| \le \int_{t_j}^x |\mathcal{C}_S'(g)(t) - s^0(t)|dt \le \int_{t_j}^{t_{j+1}} |\mathcal{C}_S(g)(t) - s^0(t)|dt = \frac{2M}{(2p'+2)^2} \le \frac{2M}{p^2}$, which implies the result. □

Remark 5. Let $s \in \mathcal{S}_p^1$. Denote $\hat{\sigma} := \|s - \mathcal{C}_S(g)\|$. Then

$$\hat{\sigma} := \|s - \mathcal{C}_S(g)\| \le \sqrt{d}\|s - \mathcal{C}_S(g)\|_\infty \overset{\text{By Lemma 12}}{\le} \frac{2\sqrt{d}M}{p^2}.$$

C.1 Compression error from value fitting

Now we are set to discuss about the compression error from value fitting. Let $\hat{\mathcal{C}}(g)$ be the approximation of the sparse vector, $\mathcal{C}_\delta(g)$ resulted from a δ-compressor. In the intermediate step, we consider the sparse vector, $\mathcal{C}_S(g)$ that has the components of $\mathcal{C}_\delta(g)$ arranged in descending order of magnitude. Let $s \in \mathcal{S}_p^1$ be the approximation of $\mathcal{C}_S(g)$. Assume no orthogonality and let the angle between $\mathcal{C}_\delta(g)$ and $\mathcal{C}_S(g)$ be $0 \leq \theta < \pi/2$, and the angle between s and $\hat{\mathcal{C}}(g)$ be $0 \leq \theta' < \pi/2$. We aim to calculate the bound on $\mathbb{E}\|\hat{\mathcal{C}}(g)\|$, (where $\hat{\mathcal{C}}(g)$ is considered to be iteration and worker agnostic) and the Lemma follows.

Lemma 13. *(*i*) If $\mathcal{C}_\delta(g)$ is unbiased, that is, $\mathbb{E}(\mathcal{C}_\delta(g)) = g$ then*

$$\mathbb{E}\|\hat{\mathcal{C}}(g)\|^2 \leq 2(2-\delta)(21 - 4\cos\theta - 16\cos\theta')\|g\|^2 + (5 - 4\cos\theta')\frac{32dM^2}{p^4}. \tag{12}$$

*(*ii*) If $\mathcal{C}_\delta(g)$ is biased, that is, $\mathbb{E}(\mathcal{C}_\delta(g)) \neq g$, then*

$$\mathbb{E}\|\hat{\mathcal{C}}(g)\|^2 \leq 2(3-2\delta)\|g\|^2 + 16(2-\delta)(5 - \cos\theta - 4\cos\theta')\|g\|^2 + (5 - 4\cos\theta')\frac{32dM^2}{p^4}. \tag{13}$$

Proof. We have

$$\mathbb{E}\|\hat{\mathcal{C}}(g)\|^2 = \mathbb{E}\|\hat{\mathcal{C}}(g) - g + g\|^2 \overset{\text{By (3)}}{\leq} 2\mathbb{E}\|\hat{\mathcal{C}}(g) - g\|^2 + 2\|g\|^2.$$

Case 1: Consider $\mathcal{C}_\delta(g)$ be unbiased, that is, $\mathbb{E}(\mathcal{C}_\delta(g)) = g$. Therefore,

$$\begin{aligned}
\mathbb{E}\|g - \hat{\mathcal{C}}(g)\|^2 &= \mathbb{E}\|g - \mathcal{C}_\delta(g) + \mathcal{C}_\delta(g) - \hat{\mathcal{C}}(g)\|^2 \\
&\overset{\mathbb{E}(\mathcal{C}_\delta(g))=g}{=} \mathbb{E}\|g - \mathcal{C}_\delta(g)\|^2 + \mathbb{E}\|\mathcal{C}_\delta(g) - \hat{\mathcal{C}}(g)\|^2 \\
&\overset{\text{By (1)}}{\leq} (1-\delta)\|g\|^2 + \mathbb{E}\|\mathcal{C}_\delta(g) - \hat{\mathcal{C}}(g)\|^2.
\end{aligned}$$

Further we need to bound $\mathbb{E}\|\mathcal{C}_\delta(g) - \hat{\mathcal{C}}(g)\|^2$. We have

$$\begin{aligned}
\mathbb{E}\|\mathcal{C}_\delta(g) - \hat{\mathcal{C}}(g)\|^2 &= \mathbb{E}\|\mathcal{C}_\delta(g) - \mathcal{C}_S(g) + \mathcal{C}_S(g) - \hat{\mathcal{C}}(g)\|^2 \\
&\overset{\text{By (3)}}{\leq} 2\mathbb{E}\|\mathcal{C}_\delta(g) - \mathcal{C}_S(g)\|^2 + 2\mathbb{E}\|\mathcal{C}_S(g) - \hat{\mathcal{C}}(g)\|^2.
\end{aligned}$$

If the angle between $\mathcal{C}_\delta(g)$ and $\mathcal{C}_S(g)$ be $0 \leq \theta < \pi/2$, then

$$\mathbb{E}\|\mathcal{C}_\delta(g) - \mathcal{C}_S(g)\|^2 \overset{\text{Lemma 4}}{=} 2(1-\cos\theta)\mathbb{E}\|\mathcal{C}_\delta(g)\|^2 \overset{\text{Lemma 5}(i)}{\leq} 2(1-\cos\theta)(2-\delta)\|g\|^2. \tag{14}$$

We pause here and quantify: $\|\mathcal{C}_S(g) - \hat{\mathcal{C}}(g)\|^2$. Let $s \in \mathcal{S}_p^1$ be the approximation of $\mathcal{C}_S(g)$. Then

$$\begin{aligned}
\mathbb{E}\|\mathcal{C}_S(g) - \hat{\mathcal{C}}(g)\|^2 &= \mathbb{E}\|\mathcal{C}_S(g) - s + s - \hat{\mathcal{C}}(g)\|^2 \\
&\overset{\text{By (3)}}{\leq} 2\mathbb{E}\|\mathcal{C}_S(g) - s\|^2 + 2\mathbb{E}\|s - \hat{\mathcal{C}}(g)\|^2 \\
&\overset{\text{By Remark 5}}{\leq} \frac{8dM^2}{p^4} + 2\mathbb{E}\|s - \hat{\mathcal{C}}(g)\|^2.
\end{aligned} \tag{15}$$

Similarly, if the angle between s and $\hat{\mathcal{C}}(g)$ be $0 \leq \theta' < \pi/2$, then

$$\begin{aligned}
\mathbb{E}\|s - \hat{\mathcal{C}}(g)\|^2 &\overset{\text{By Lemma 4}}{=} 2(1-\cos\theta')\mathbb{E}\|s\|^2 \\
&\leq 2(1-\cos\theta')\mathbb{E}\|s - \mathcal{C}_S(g) + \mathcal{C}_S(g)\|^2 \\
&\overset{\text{By (3)}}{\leq} 4(1-\cos\theta')\|s - \mathcal{C}_S(g)\|^2 + 4(1-\cos\theta')\mathbb{E}\|\mathcal{C}_S(g)\|^2 \\
&\overset{\mathbb{E}\|\mathcal{C}_\delta(g)\|^2 = \mathbb{E}\|\mathcal{C}_S(g)\|^2}{=} (1-\cos\theta')\frac{16dM^2}{p^4} + 4(1-\cos\theta')\mathbb{E}\|\mathcal{C}_\delta(g)\|^2 \\
&\overset{\text{By Lemma 5}(i)}{\leq} (1-\cos\theta')\frac{16dM^2}{p^4} + 4(1-\cos\theta')(2-\delta)\|g\|^2.
\end{aligned} \tag{16}$$

Therefore,

$$\begin{aligned}\mathbb{E}\|\mathcal{C}_\delta(g)-\hat{\mathcal{C}}(g)\|^2 &\leq 2\mathbb{E}\|\mathcal{C}_\delta(g)-\mathcal{C}_S(g)\|^2+2\mathbb{E}\|\mathcal{C}_S(g)-\hat{\mathcal{C}}(g)\|^2\\ &\leq 4(1-\cos\theta)(2-\delta)\|g\|^2+\frac{16dM^2}{p^4}+\frac{64dM^2}{p^4}(1-\cos\theta')\\ &\quad +16(1-\cos\theta')(2-\delta)\|g\|^2.\end{aligned}$$

Combining all together we have

$$\begin{aligned}&\mathbb{E}\|\hat{\mathcal{C}}(g)\|^2\\ \leq\ & 2(2-\delta)\|g\|^2+8(1-\cos\theta)(2-\delta)\|g\|^2+\frac{32dM^2}{p^4}+\frac{128dM^2}{p^4}(1-\cos\theta')\\ &+32(1-\cos\theta')(2-\delta)\|g\|^2.\end{aligned}$$

Arranging the terms, we get the result.

Case 2: If $\mathcal{C}_\delta(g)$ be biased, that is, $\mathbb{E}(\mathcal{C}_\delta(g))\neq g$ then

$$\begin{aligned}\mathbb{E}\|g-\hat{\mathcal{C}}(g)\|^2 &\leq 2\mathbb{E}\|g-\mathcal{C}_\delta(g)\|^2+2\mathbb{E}\|\mathcal{C}_\delta(g)-\hat{\mathcal{C}}(g)\|^2\\ &\overset{\text{By (1)}}{\leq} 2(1-\delta)\|g\|^2+2\mathbb{E}\|\mathcal{C}_\delta(g)-\hat{\mathcal{C}}(g)\|^2.\end{aligned}$$

For biased compressor $\mathcal{C}_\delta(g)$, by using Lemma 5 (*ii*) we have

$$\begin{aligned}\mathbb{E}\|\mathcal{C}_\delta(g)-\mathcal{C}_S(g)\|^2 &\overset{\text{By Lemma 4}}{=} 2(1-\cos\theta)\mathbb{E}\|\mathcal{C}_\delta(g)\|^2\\ &\overset{\text{By Lemma 5}(ii)}{\leq} 4(1-\cos\theta)(2-\delta)\|g\|^2.\end{aligned}\tag{17}$$

and

$$\begin{aligned}\mathbb{E}\|s-\hat{\mathcal{C}}(g)\|^2 &\overset{\text{By Lemma 4}}{=} 4(1-\cos\theta')\|s-C_S(g)\|^2+4(1-\cos\theta')\mathbb{E}\|\mathcal{C}_\delta(g)\|^2\\ &\overset{\text{By Lemma 5}(ii)}{\leq} (1-\cos\theta')\frac{16dM^2}{p^4}+8(1-\cos\theta')(2-\delta)\|g\|^2.\end{aligned}\tag{18}$$

Finally,

$$\begin{aligned}\mathbb{E}\|\mathcal{C}_\delta(g)-\hat{\mathcal{C}}(g)\|^2 &\leq 2\mathbb{E}\|\mathcal{C}_\delta(g)-\mathcal{C}_S(g)\|^2+2\mathbb{E}\|\mathcal{C}_S(g)-\hat{\mathcal{C}}(g)\|^2\\ &\leq 8(1-\cos\theta)(2-\delta)\|g\|^2+\frac{16dM^2}{p^4}+(1-\cos\theta')\frac{64dM^2}{p^4}\\ &\quad +32(1-\cos\theta')(2-\delta)\|g\|^2.\end{aligned}$$

Combining all together we have

$$\begin{aligned}&\mathbb{E}\|\hat{\mathcal{C}}(g)\|^2\\ \leq\ & 2\mathbb{E}\|\hat{\mathcal{C}}(g)-g\|^2+2\|g\|^2\\ \leq\ & 2(3-2\delta)\|g\|^2+16(1-\cos\theta)(2-\delta)\|g\|^2+\frac{32dM^2}{p^4}+(1-\cos\theta')\frac{128dM^2}{p^4}\\ &+64(1-\cos\theta')(2-\delta)\|g\|^2.\end{aligned}$$

Arranging the terms, we get the result. □

D Compression error from combined index and value fitting

In this section, we discuss about the compression error from joint index and value fitting. Let $\bar{\mathcal{C}}(g)$ be the sparse approximation of the vector, g after sparse vector, $\mathcal{C}_\delta(g)$ resulted from a δ-compressor, goes through consequent index compression via $\mathcal{P}_R$,[2] and value compression via piecewise polynomial fit.

[2]We give the result by using random policy, $\mathcal{P}_R$. Also, similar bounds hold for deterministic policy. For P_0, the quantity, $\frac{r}{d}$ in the proofs will be replaced by β with $0<\beta\leq 1$.

Let $\mathcal{C}_{\mathcal{P}_R,\delta}(g)$ be the sparse vector whose indices are resulted from policy $\mathcal{P}_R$ applied to $\mathcal{C}_\delta(g)$. In the intermediate step, we consider the sparse vector, $\mathcal{C}_S(g)$ that has the components of $\mathcal{C}_{\mathcal{P}_R,\delta}(g)$ arranged in descending order of magnitude. Let $s \in \mathcal{S}_p^1$ be the approximation of $\mathcal{C}_S(g)$. Let the angle between $\mathcal{C}_{\mathcal{P}_R,\delta}(g)$ and $\mathcal{C}_S(g)$ be $0 \leq \theta < \pi/2$, and the angle between s and $\bar{\mathcal{C}}(g)$ be $0 \leq \theta' < \pi/2$. We aim to calculate the bound on $\mathbb{E}\|\bar{\mathcal{C}}(g)\|$, (where $\bar{\mathcal{C}}(g)$ is considered to be iteration and worker agnostic) and the Lemma follows.

Lemma 14. *(*i*) If $\mathcal{C}_\delta(g)$ is unbiased, that is, $\mathbb{E}(\mathcal{C}_\delta(g)) = g$ then*

$$\begin{aligned}\mathbb{E}\|\bar{\mathcal{C}}(g)\|^2 \quad &\leq \quad 2(2-\delta)\|g\|^2 + 4(2-\frac{r}{d}-\delta)\|g\|^2 + 32(1-\cos\theta)(2-\frac{r}{d})\|g\|^2 + \frac{64dM^2}{p^4} \\ &\quad +(1-\cos\theta')\frac{256dM^2}{p^4} + 64(1-\cos\theta')(2-\delta)\|g\|^2. \end{aligned} \tag{19}$$

*(*ii*) If $\mathcal{C}_\delta(g)$ is biased, that is, $\mathbb{E}(\mathcal{C}_\delta(g)) \neq g$, then*

$$\begin{aligned}\mathbb{E}\|\bar{\mathcal{C}}(g)\|^2 \quad &\leq \quad 2(3-\delta)\|g\|^2 + 16(2-\frac{r}{d}-\delta)\|g\|^2 + 32(1-\cos\theta)(2-\frac{r}{d})\|g\|^2 + \frac{64dM^2}{p^4} \\ &\quad +(1-\cos\theta')\frac{256dM^2}{p^4} + 128(1-\cos\theta')(2-\delta)\|g\|^2. \end{aligned} \tag{20}$$

Proof. We have

$$\mathbb{E}\|\bar{\mathcal{C}}(g)\|^2 = \mathbb{E}\|\bar{\mathcal{C}}(g) - g + g\|^2 \overset{\text{By (3)}}{\leq} 2\mathbb{E}\|\bar{\mathcal{C}}(g) - g\|^2 + 2\|g\|^2.$$

Case 1: Consider $\mathcal{C}_\delta(g)$ be unbiased, that is, $\mathbb{E}(\mathcal{C}_\delta(g)) = g$. Therefore,

$$\begin{aligned}\mathbb{E}\|g - \bar{\mathcal{C}}(g)\|^2 \quad &= \quad \mathbb{E}\|g - \mathcal{C}_\delta(g) + \mathcal{C}_\delta(g) - \bar{\mathcal{C}}(g)\|^2 \\ &\overset{\mathbb{E}(\mathcal{C}_\delta(g))=g}{=} \quad \mathbb{E}\|g - \mathcal{C}_\delta(g)\|^2 + \mathbb{E}\|\mathcal{C}_\delta(g) - \bar{\mathcal{C}}(g)\|^2 \\ &\overset{\text{By (1)}}{\leq} \quad (1-\delta)\|g\|^2 + \mathbb{E}\|\mathcal{C}_\delta(g) - \bar{\mathcal{C}}(g)\|^2.\end{aligned}$$

Further, we need to bound $\mathbb{E}\|\mathcal{C}_\delta(g) - \bar{\mathcal{C}}(g)\|^2$. We have

$$\begin{aligned}\mathbb{E}\|\mathcal{C}_\delta(g) - \bar{\mathcal{C}}(g)\|^2 \quad &= \quad \mathbb{E}\|\mathcal{C}_\delta(g) - \mathcal{C}_{\mathcal{P}_R,\delta}(g) + \mathcal{C}_{\mathcal{P}_R,\delta}(g) - \bar{\mathcal{C}}(g)\|^2 \\ &\overset{\text{By (3)}}{\leq} \quad 2\mathbb{E}\|\mathcal{C}_\delta(g) - \mathcal{C}_{\mathcal{P}_R,\delta}(g)\|^2 + 2\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g) - \bar{\mathcal{C}}(g)\|^2 \\ &= \quad 2\mathbb{E}\|g - \mathcal{C}_{\mathcal{P}_R,\delta}(g) - (g - \mathcal{C}_\delta(g))\|^2 + 2\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g) - \bar{\mathcal{C}}(g)\|^2 \\ &\overset{\text{By (9) (3),and (1)}}{\leq} \quad 2(1-\tfrac{r}{d})\|g\|^2 + 2(1-\delta)\|g\|^2 + 2\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g) - \bar{\mathcal{C}}(g)\|^2 \\ &= \quad 2(2-\tfrac{r}{d}-\delta)\|g\|^2 + 2\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g) - \bar{\mathcal{C}}(g)\|^2.\end{aligned}$$

Now,

$$\begin{aligned}\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g) - \bar{\mathcal{C}}(g)\|^2 \quad &= \quad \mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g) - \mathcal{C}_S(g) + \mathcal{C}_S(g) - \bar{\mathcal{C}}(g)\|^2 \\ &\overset{\text{By (3)}}{\leq} \quad 2\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g) - \mathcal{C}_S(g)\|^2 + 2\mathbb{E}\|\mathcal{C}_S(g) - \bar{\mathcal{C}}(g)\|^2.\end{aligned}$$

If the angle between $\mathcal{C}_{\mathcal{P}_R,\delta}(g)$ and $\mathcal{C}_S(g)$ be $0 \leq \theta < \pi/2$, then

$$\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g) - \mathcal{C}_S(g)\|^2 \overset{\text{Lemma 4}}{=} 2(1-\cos\theta)\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g)\|^2 \overset{\text{By (9) and (3)}}{\leq} 2(1-\cos\theta)(2-\frac{r}{d})\|g\|^2. \tag{21}$$

We pause here and quantify: $\|\mathcal{C}_S(g) - \bar{\mathcal{C}}(g)\|^2$. Let $s \in \mathcal{S}_p^1$ be the approximation of $\mathcal{C}_S(g)$. Then

$$\begin{aligned}\mathbb{E}\|\mathcal{C}_S(g) - \bar{\mathcal{C}}(g)\|^2 \quad &= \quad \mathbb{E}\|\mathcal{C}_S(g) - s + s - \bar{\mathcal{C}}(g)\|^2 \\ &\overset{\text{By (3)}}{\leq} \quad 2\mathbb{E}\|\mathcal{C}_S(g) - s\|^2 + 2\mathbb{E}\|s - \bar{\mathcal{C}}(g)\|^2 \\ &\overset{\text{By Remark 5}}{\leq} \quad \frac{8dM^2}{p^4} + 2\mathbb{E}\|s - \bar{\mathcal{C}}(g)\|^2.\end{aligned} \tag{22}$$

Similarly, if the angle between s and $\bar{\mathcal{C}}(g)$ be $0 \leq \theta' < \pi/2$, then

$$\begin{aligned}
\mathbb{E}\|s - \bar{\mathcal{C}}(g)\|^2 &\overset{\text{By Lemma 4}}{=} 2(1-\cos\theta')\mathbb{E}\|s\|^2 \\
&\leq 2(1-\cos\theta')\mathbb{E}\|s - \mathcal{C}_S(g) + \mathcal{C}_S(g)\|^2 \\
&\overset{\text{By (3)}}{\leq} 4(1-\cos\theta')\|s - \mathcal{C}_S(g)\|^2 + 4(1-\cos\theta')\mathbb{E}\|\mathcal{C}_S(g)\|^2 \\
&\overset{\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g)\|^2 = \mathbb{E}\|\mathcal{C}_S(g)\|^2}{=} (1-\cos\theta')\frac{16dM^2}{p^4} + 4(1-\cos\theta')\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g)\|^2 \\
&\overset{\text{By Lemma 5}(i)}{\leq} (1-\cos\theta')\frac{16dM^2}{p^4} + 4(1-\cos\theta')(2-\frac{r}{d})\|g\|^2. \quad (23)
\end{aligned}$$

Therefore,

$$\begin{aligned}
&\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g) - \bar{\mathcal{C}}(g)\|^2 \\
\leq\ & 2\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g) - \mathcal{C}_S(g)\|^2 + 2\mathbb{E}\|\mathcal{C}_S(g) - \bar{\mathcal{C}}(g)\|^2 \\
\leq\ & 4(1-\cos\theta)(2-\frac{r}{d})\|g\|^2 + \frac{16dM^2}{p^4} + \frac{64dM^2}{p^4}(1-\cos\theta') + 16(1-\cos\theta')(2-\frac{r}{d})\|g\|^2.
\end{aligned}$$

Combining all together we have

$$\begin{aligned}
&\mathbb{E}\|\bar{\mathcal{C}}(g)\|^2 \\
\leq\ & 2(2-\delta)\|g\|^2 + 4(2-\frac{r}{d}-\delta)\|g\|^2 + 16(1-\cos\theta')(2-\frac{r}{d})\|g\|^2 + \frac{64dM^2}{p^4} \\
&+\frac{256dM^2}{p^4}(1-\cos\theta') + 64(1-\cos\theta')(2-\delta)\|g\|^2.
\end{aligned}$$

Arranging the terms, we get the result.

Case 2: If $\mathcal{C}_\delta(g)$ be biased, that is, $\mathbb{E}(\mathcal{C}_\delta(g)) \neq g$ then

$$\begin{aligned}
\mathbb{E}\|g - \bar{\mathcal{C}}(g)\|^2 &\leq 2\mathbb{E}\|g - \mathcal{C}_\delta(g)\|^2 + 2\mathbb{E}\|\mathcal{C}_\delta(g) - \bar{\mathcal{C}}(g)\|^2 \\
&\overset{\text{By (1)}}{\leq} 2(1-\delta)\|g\|^2 + 2\mathbb{E}\|\mathcal{C}_\delta(g) - \bar{\mathcal{C}}(g)\|^2.
\end{aligned}$$

Further, we have

$$\begin{aligned}
\mathbb{E}\|\mathcal{C}_\delta(g) - \bar{\mathcal{C}}(g)\|^2 &= \mathbb{E}\|\mathcal{C}_\delta(g) - \mathcal{C}_{\mathcal{P}_R,\delta}(g) + \mathcal{C}_{\mathcal{P}_R,\delta}(g) - \bar{\mathcal{C}}(g)\|^2 \\
&\overset{\text{By (3)}}{\leq} 2\mathbb{E}\|\mathcal{C}_\delta(g) - \mathcal{C}_{\mathcal{P}_R,\delta}(g)\|^2 + 2\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g) - \bar{\mathcal{C}}(g)\|^2 \\
&= 2\mathbb{E}\|g - \mathcal{C}_{\mathcal{P}_R,\delta}(g) - (g - \mathcal{C}_\delta(g))\|^2 + 2\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g) - \bar{\mathcal{C}}(g)\|^2 \\
&\overset{\text{By (9) (3),and (1)}}{\leq} 4(1-\tfrac{r}{d})\|g\|^2 + 4(1-\delta)\|g\|^2 + 2\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g) - \bar{\mathcal{C}}(g)\|^2 \\
&= 4(2-\tfrac{r}{d}-\delta)\|g\|^2 + 2\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g) - \bar{\mathcal{C}}(g)\|^2.
\end{aligned}$$

For biased compressor $\mathcal{C}_\delta(g)$, by using Lemma 5 (*ii*) we have

$$\begin{aligned}
\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g) - \mathcal{C}_S(g)\|^2 &\overset{\text{By Lemma 4}}{=} 2(1-\cos\theta)\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g)\|^2 \\
&\overset{\text{By Lemma 5}(ii)}{\leq} 4(1-\cos\theta)(2-\frac{r}{d})\|g\|^2; \quad (24)
\end{aligned}$$

and

$$\begin{aligned}
\mathbb{E}\|s - \bar{\mathcal{C}}(g)\|^2 &\overset{\text{By Lemma 4}}{=} 4(1-\cos\theta')\|s - C_S(g)\|^2 + 4(1-\cos\theta')\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g)\|^2 \\
&\overset{\text{By Lemma 5}(ii)}{\leq} (1-\cos\theta')\frac{16dM^2}{p^4} + 8(1-\cos\theta')(2-\frac{r}{d})\|g\|^2. \quad (25)
\end{aligned}$$

Finally,

$$\begin{aligned}
&\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g) - \bar{\mathcal{C}}(g)\|^2 \\
\leq\ & 2\mathbb{E}\|\mathcal{C}_{\mathcal{P}_R,\delta}(g) - \mathcal{C}_S(g)\|^2 + 2\mathbb{E}\|\mathcal{C}_S(g) - \bar{\mathcal{C}}(g)\|^2 \\
\leq\ & 8(1-\cos\theta)(2-\frac{r}{d})\|g\|^2 + \frac{16dM^2}{p^4} + (1-\cos\theta')\frac{64dM^2}{p^4} + 32(1-\cos\theta')(2-\frac{r}{d})\|g\|^2.
\end{aligned}$$

Combining all together we have

$$\begin{aligned}
& \mathbb{E}\|\bar{\mathcal{C}}(g)\|^2 \\
\leq\ & 2\mathbb{E}\|\bar{\mathcal{C}}(g)-g\|^2+2\|g\|^2 \\
\leq\ & 2(3-\delta)\|g\|^2+16(2-\frac{r}{d}-\delta)\|g\|^2+32(1-\cos\theta)(2-\frac{r}{d})\|g\|^2+\frac{64dM^2}{p^4} \\
& +(1-\cos\theta')\frac{256dM^2}{p^4}+128(1-\cos\theta')(2-\delta)\|g\|^2.
\end{aligned}$$

Arranging the terms, we get the result. □

D.1 Convergence of distributed compressed SGD without error feedback

We comment on the convergence of compressed distributed SGD without error feedback [23]. We consider the following scenarios:

1. **Approximation by using only value compression.** One can find the bound on compressed aggregated gradient $\tilde{g}_k$ resulting at k^{th} iteration. Denote $\hat{\mathcal{C}}(g_k^i)$ be the approximation of the sparse vector $\mathcal{C}_\delta(g_k^i)$ resulted from a δ-compressor at i^{th} worker, at the k^{th} iteration. Denote the compressed aggregated gradient at k^{th} iteration to be $\tilde{g}_k := \frac{1}{n}\sum_{i=1}^n \hat{\mathcal{C}}(g_k^i)$. By using Lemma 13, we can find bound on $\mathbb{E}\|\tilde{g}_k\|^2$ is for both biased and unbiased $\mathcal{C}_\delta(g)$. [3]
2. **Approximation by using both value and index compression.** Similarly, denote $\bar{\mathcal{C}}(g_k^i)$ be the approximation of the sparse vector $\mathcal{C}_\delta(g_k^i)$ resulted from a δ-compressor at i^{th} worker, at the k^{th} iteration by consequent index compression via $\mathcal{P}_R$, and value compression via piecewise polynomial fit. Denote the compressed aggregated gradient at k^{th} iteration to be $g_k^\perp := \frac{1}{n}\sum_{i=1}^n \bar{\mathcal{C}}(g_k^i)$. By using Lemma 14, we can find bound $\mathbb{E}\|g_k^\perp\|^2$ for both biased and unbiased $\mathcal{C}_\delta(g)$.

With the above, based on the strong growth condition of stochastic gradients [23, 77], for a lower bounded, Lipschitz smooth, and non-convex loss function f, following [23], the distributed SGD with an δ sparsifier converges, that is, $\min_{k\in[T]} \mathbb{E}(\|\nabla f_k\|^2) \to 0$ as $T\to\infty$. The convergence with error-feedback [45] is a more mathematically involved problem that requires independent investigation, and left for future research.

For the convergence of compressed FedAvg algorithm, we refer to the recent unified analysis in [30] (also see [60, 31]). However, convergence of bidirectional compressed FedAvg with error feedback is an open problem and not the scope of this paper.

E Implementation details

This section highlights the implementation of different Bloom policies for index compression, polynomial regression for value compression, and joint index and value compression on GPUs and CPUs by using popular deep learning toolkits, TensorFlow and PyTorch (see Table 3).

Hash-Functions used in Bloom Filter implementation. We use MurmurHash (MurmurHash3) to construct the hash table in the GPU implementation of the Bloom filter; see Python library https://pypi.org/project/mmh3/. We determine the number of hash functions, k, and the length of Bloom filter bit-string, m, by using Lemma 3 and Remark 2 in the Appendix.

Implementation of Bloom Filter on GPUs and CPUs. We provide an efficient GPU implementation of Bloom filters on PyTorch. During construction, many items can be inserted in parallel without locking, since collisions do not cause inconsistency. Since the domain $[d]$ of the hash functions is finite, we precompute a 2D lookup table $\mathbb{H}^{d,k}$, for each possible input of all hash functions. We store $\mathbb{H}$ in the GPU memory, allowing us to insert items in the Bloom filter using only lookup operations. $\mathbb{H}$ occupies around 1.5MB for ResNet-20 and 1GB for NCF; note that this optimization may not be feasible for very large models. Querying is also implemented in the GPU. If an item i belongs

[3] Approximation error from index compression in distributed case, is a simple consequence of (9).

to the Bloom filter, then $\mathcal{B}[h_1(i)] + \mathcal{B}[h_2(i)] + \mathcal{B}[h_3(i)] + \cdots + \mathcal{B}[h_k(1)] == k$. The summation can be executed in parallel with each hash function reduced to a lookup in $\mathbb{H}$. Moreover, many such queries can run concurrently. Although the basic Bloom filter is implemented on GPUs, complex policies, like P2, require programming flexibility. For this reason, we provide CPU implementations on PyTorch, using library pybloomfilter [4]; and on TensorFlow using the C++ extension to create custom operators.

Implementation of polynomial regression on GPUs and CPUs. The piece-wise polynomial regression can be solved as a linear problem, once the segments are determined. Our GPU implementation uses Least-Square fitting, which can be trivially expressed with tensor operations. We also provide a CPU implementation using `polyfit` from the NumPy [3] library. We implement the nonlinear double exponential regression on TensorFlow, using tensor operations.

Combined index and value compression. To combine Bloom filter-based with curve fitting-based compression, first, observe that neither method is order preserving. Therefore, we need a mapping from the original to the final position of each value. This corresponds to a 1D vector with 1~d, where d is the size of sparse gradient. Since now the maximum element in this mapping vector is d, we encode their each element using $\lceil \log_2 d \rceil$ bits. For our experiments, this corresponds to 16 bits for ResNet50 and 19 bits for NCF, which is a significant gain compared to the usual int32 format.

Complexity of the methods. For each policy of the Bloom filter, if we have r elements and we use k hash functions, then the time and space complexity of each policy is $O(rk) = O(r \log_2(1/\epsilon))$, because the optimal $k = -\frac{\log \epsilon}{\log 2}$, see Remark 2.

We use radix sort which takes $O(d \log_2(d))$, to sort d gradient components. In general, to perform a Top-r selection on CPU on a d dimensional vector, the computational complexity is $O(d \log_2 r)$; for GPUs, other optimized implementations exist [65]. Therefore, to sort these r-components further, we require $O(r \log_2(r))$ time, and as $r \ll d$, the total time complexity remains $O(d \log_2 r)$.

For a polynomial fitting with degree n' on each sorted segment with d_p data points, the overall complexity of finding the least-squares solution is $O(d_p n'^2 + n'^3)$. For our application, $n' \ll d_p$, hence, overall complexity is $O(d_p n'^2)$.

We do not use segmentation for nonlinear regression. For nonlinear regression through a double exponential model, $y = ae^{bx} + ce^{dx}$, involves solving a 4×4 linear system followed by solving a 2×2 linear system. By using modern solvers (that use Cholesky factorization), the complexity of solving these linear systems are negligible. However, the linear regression in the first system is a proxy to solving integral equations. Therefore, the entries of the coefficient matrix of the first system are approximated via numerical integration (as they cannot be computed by analytical integration) and each of them requires d exact operations, where d is the total number points to be fitted. Similarly, for calculating each entry of the second linear system requires d exact operations.

F Additional experimental results

Due to limited space, we were unable to discuss many experimental details as well as many results in Section 6 of the main paper. In this scope, we discuss them in details.

F.1 Details of the Testbed

Simulation on local cluster. For conventional data center experiments, we use 8 dedicated machines with Ubuntu 18.04.2 LTS and Linux v.4.15.0-74, 16-Core Intel Xeon Silver 4112 @ 2.6GHz, 512 GB RAM, one NVIDIA Tesla V100 GPU card with 16 GB on-board memory and 100Gbps network. We deploy CUDA 10.1, TensorFlow 1.14, PyTorch 1.7.1, Horovod 0.21.0, OpenMPI 4.0 and NCCL 2.4.8.

Realistic federated learning deployment. For Federated Learning experiments, we use 57 EC2 instances (g4dn.xlarge) from Amazon Web Service(AWS). The central server is located in Ohio (USA) and 56 clients are spread across 7 different regions globally, each with 8 clients, including Tokyo, Central Canada, Northern California, Seoul, São Paulo, Paris and Oregon. Each client is independently connected to the server with high speed international network. Each instance is

equipped with Ubuntu 16.04.12 LTS, 4-Core Intel(R) Xeon(R) Platinum 8259CL CPU @ 2.50GHz, 16 GB RAM, one NVIDIA Tesla T4 GPU card with 16 GB on-board memory and 128 GB NVMe SSD. We deploy CUDA 11.0, Intel MKL, PyTorch 1.7.1 and MPICH 3.4.1.

F.2 Implemented methods

We implement DeepReduce [4] as an extension of GRACE [84], a framework that supports many popular sparsification techniques and interfaces with various low-level communication libraries for distributed deep learning. Table 3 presents a summary of the methods we implement.

Table 3: Summary of implementations. $\mathrm{DR}^{\mathrm{val}}_{\mathrm{idx}}$ denotes instantiation of DeepReduce with idx and val as index and value compression method, respectively.

Method	Idx	Val	Device	Framework
$\mathrm{DR}^{\varnothing}_{\mathrm{BF-Naive}}$	✓		CPU	TFlow
$\mathrm{DR}^{\varnothing}_{\mathrm{BF-P0}}$, $\mathrm{DR}^{\varnothing}_{\mathrm{BF-P1}}$	✓		GPU	PyTorch
$\mathrm{DR}^{\varnothing}_{\mathrm{BF-P0}}$, $\mathrm{DR}^{\varnothing}_{\mathrm{BF-P1}}$, $\mathrm{DR}^{\varnothing}_{\mathrm{BF-P2}}$	✓		CPU	TFlow, PyTorch
$\mathrm{DR}^{\varnothing}_{\mathrm{RLE}}$	✓		CPU	TFlow, PyTorch
$\mathrm{DR}^{\varnothing}_{\mathrm{Huffman}}$	✓		CPU	PyTorch
$\mathrm{DR}^{\mathrm{Fit-Poly}}_{\varnothing}$		✓	GPU, CPU	TFlow, PyTorch
$\mathrm{DR}^{\mathrm{Fit-DExp}}_{\varnothing}$		✓	GPU	TFlow
$\mathrm{DR}^{\mathrm{Deflate}}_{\varnothing}$		✓	CPU	PyTorch
$\mathrm{DR}^{\mathrm{QSGD}}_{\varnothing}$		✓	GPU	PyTorch
$\mathrm{DR}^{\mathrm{Fit-Poly}}_{\mathrm{BF-P0}}$, $\mathrm{DR}^{\mathrm{Fit-Poly}}_{\mathrm{BF-P1}}$	✓	✓	GPU	PyTorch
$\mathrm{DR}^{\mathrm{QSGD}}_{\mathrm{BF-P2}}$	✓	✓	GPU	PyTorch
3LC		✓	GPU	TFlow
SketchML		✓	CPU	PyTorch
SKCompress	✓	✓	CPU	PyTorch

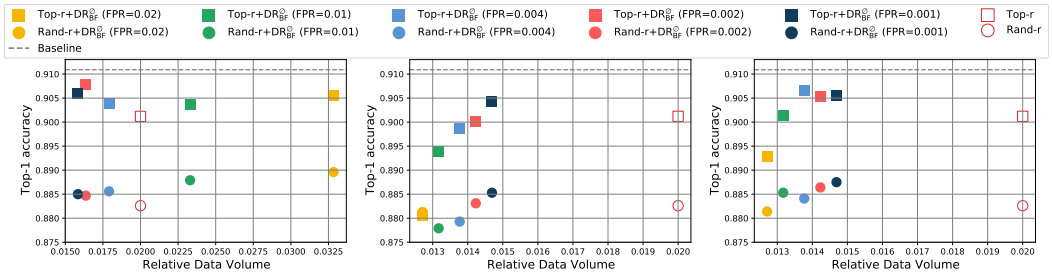
Run Length Encoding (RLE). Since RLE is a lossless method designed for continuous repetitive symbols, it is not directly applicable to non-repetitive gradient indices. We convert gradient indices into bitmap format, which is a boolean bit string indicating which elements are selected. In this way, RLE can be used to encode the continuous zeros and ones in the bitmap. Note that, the compression rate is highly dependent on the distribution of the indices. That is, RLE is more beneficial if gradient indices contain more continuous integers.

Huffman Encoding. The key idea of Huffman Coding is to use fewer bits to represent more frequent symbols. We note that most indices are much smaller than 2^{32}, and consequently their binary format start with continuous zero bits. Based on this observation, we can use Huffman Coding to compress the binary format of each index to remove the redundancy. The codec is constructed from all possible indices of the target model. (i.e. If the largets gradient size is d, then we use $0 \sim d-1$ for codec construction). The encoding phase contains 2 steps: unpack each 32-bit integer gradient key into Byte format and then encode each index with the pre-defined codec. The decoding phase is just a reversed process.

F.3 Additional results

Bloom filter-based index compression: Effect of false positive rate (FPR). We train ResNet-20 on CIFAR-10 on 8 nodes for 328 epochs and measure the top-1 accuracy and transferred data volume. Our baseline transmits the original uncompressed gradients. To generate sparse gradients, we employ the Top-r [6] and Rand-r [69] sparsifiers; each achieves different accuracy [84]. We vary FPR and measure its effect; smaller FPR corresponds to larger bloom filter. The results for our three index compression policies are shown in Figure 15. Recall (Section 4) that policy BF-P0 transmits extra data for each false positive index. The advantage, as shown in Figure 15a is that accuracy is only marginally affected by FPR, irrespective of the gradient sparsifier (i.e., Top-r or Rand-r). The disadvantage is that the amount of transferred data increases with higher FPR; if it is high enough

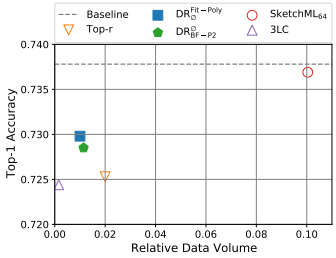
[4] Available at: `https://github.com/hangxu0304/DeepReduce`

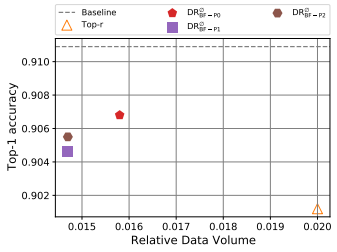


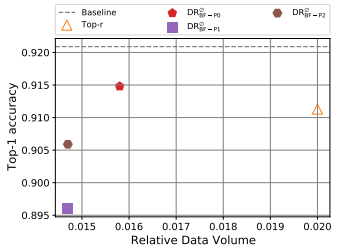
(a) Policy P0: $\mathrm{DR}^{\varnothing}_{\mathrm{BF-P0}}$ (b) Policy P1: $\mathrm{DR}^{\varnothing}_{\mathrm{BF-P1}}$ (c) Policy P2: $\mathrm{DR}^{\varnothing}_{\mathrm{BF-P2}}$

Figure 15: Effect of FPR on top-1 accuracy for the three Bloom filter policies, for ResNet-20 on CIFAR-10. The sparse input gradients were generated by Top-r and Random-r sparsification methods. Data volume is relative to the no-compression baseline.

(e.g., more than 0.004 in our figure), then BF-P0 transfers more data than the sparse input gradient. Policy BF-P1, on the other hand, resolves bloom filter conflicts randomly; as expected, Figure 15b confirms that the amount of transferred data decreases when FPR increases. The trade-off is that accuracy also decreases because more erroneous gradient elements are received. Fortunately, our next policy, BF-P2, improves this issue, as shown in Figure 15c; by resolving conflicts in an informed way.







a: ResNet-20 - CIFAR-10 b: DenseNet40-K12 - CIFAR-10

Figure 16: Top-1 test accuracy of ResNet-50 on ImageNet. We compare $\mathrm{DeepReduce}_{\mathrm{idx}}$ and $\mathrm{DeepReduce}_{\mathrm{val}}$ against Top-r, $\mathrm{SketchML}_{64}$, 3LC and baseline.

Figure 17: (a) Data volume vs. accuracy of ResNet-20 on CIFAR-10. (b) Data volume vs. accuracy of DensNet40 on CIFAR-10. We compare $\mathrm{DR}^{\varnothing}_{\mathrm{BF}}$ (with P0, P1, P2, Naïve policy) against Top-r and baseline. Ratios of Top-r are 1% for ResNet20, and 0.5% for DenseNet40. FPR is set to 0.001.

Bloom filter-based index compression: Trade-offs for different polices. We show the trade-off between accuracy and data volume for different policies of bloom filter in Fig17. BF-P0 can always maintain the accuracy while sending marginally more data than BF-P1 and BF-P2. BF-P2 has consistently better accuracy than BF-P0 due to the sophisticated conflict-set algorithm.

DeepReduce on top of the Top-r sparsifier v.s. *stand-alone* compressors. To contrast with the common practice, we compare DeepReduce against state-of-the-art *stand-alone* gradient compressors [84] that are applied directly on the original gradient. We consider two instantiations of DeepReduce: (*i*) $\mathrm{DR}^{\varnothing}_{\mathrm{BF-P2}}$ that uses Bloom filter index compression with policy P2 and FPR=0.001; and (*ii*) $\mathrm{DR}^{\mathrm{Fit-Poly}}_{\varnothing}$ that uses value compression with polynomial fit. Both operate on the sparse tensor generated by Top-r, with $r = 1\%$. We compare against two stand-alone gradient compressors: 3LC [50] with spasification multiplier set to 1, and SketchML [39]; we use 2^6 quantile buckets, since we opt for best accuracy. For the latter the number of quantile buckets affects accuracy and data volume; for instance, with 2^1 buckets SketchML achieves only 56.05% Top-1 accuracy. We opt for best accuracy; therefore we use 2^6 quantile buckets. Memory compensation is enabled for all methods. For this experiment, we employ a much larger benchmark: ResNet-50 on ImageNet. The results are shown in Figure 16, where data volume is relative to the no-compression baseline. Both DeepReduce instantiations provide a good balance between data volume and accuracy, whereas each of the stand-alone methods is biased towards one of the two metrics.

Table 4: Time breakdown and data volume of DeepReduce variants, Top-r, and Baseline (FedAvg [55]) in a simulated FL setting. (CLI, SER, S2C and C2S stand for Client, Server, Server-to-Client, and Client-to-Server, respectively.)

	Avg. Encoding/Decoding Time (s)				Avg. Comm. Time (s)		Avg. Relative Data Volume		Test Accuracy
	CLI_{decode}	CLI_{encode}	SER_{decode}	SER_{encode}	S2C	C2S	S2C	C2S	
***Baseline**	0	0	0	0	1.3980	1.3609	1.0	1.0	0.1856
***Top-r(0.1)**	0.0044	0.0501	0.0032	0.0258	0.3167	0.3904	0.2033	0.2033	0.1840
* $\text{DR}^{\emptyset}_{\text{BF-P0}}$	0.0179	0.0707	0.0183	0.0629	0.1936	0.1957	0.1425	0.1426	0.1841
* $\text{DR}^{\text{Fit-Poly}}_{\emptyset}$	0.0288	0.1586	0.0183	0.1170	0.1742	0.1429	0.1039	0.1039	0.1838
* $\text{DR}^{\text{QSGD}}_{\text{BF-P0}}$	0.0190	0.0713	0.0198	0.0706	0.0852	0.0859	0.0621	0.0621	0.1836

This is evident in Figure 16 as SketchML compresses less aggressively than the other compressors, thus it achieves the highest accuracy. In contrast, 3LC sends the least data and the accuracy suffers the most. This is because 3LC is a hybrid method [84] that quantizes the values into $(-1, 0, 1)$ and selects the non-zero values for encoding. We use the default setting of 3LC that gives the least sparsification (but fails to recover the baseline accuracy). DeepReduce can be applied on top of any sparsifier and the sparsification ratio is flexible to choose. Figure 16 shows that DeepReduce on Top-r not only reduces the data volume, but also improves accuracy by 0.7%.

Simulated FL experiments in a bandwidth-limited local environment. Apart from the realistic multi-region deployment, we also test DeepReduce with clients and the server in the same region connected with 100 Mbps network to simulate the low bandwidth scenario. The results are shown is Table 4. The total encoding/decoding overhead of DeepReduce is about 1.5-3.4× higher than Top-r. In contrast, the communication time is decreased by 1.8-4.0× for different DeepReduce variants, compared with Top-r. However, the extra overhead of DeepReduce is relatively low compared to their communication time reduction. Even with compression overhead taken into account, $\text{DR}^{\text{QSGD}}_{\text{BF-P0}}$ is 2.2× faster than Top-r and 7.8× faster than the Baseline. Unlike the multi-region case which is suffering from the high latency network, the communication time of DeepReduce here is proportional to the transmitted data volume.

FL training of MobileNet with DeepReduce. We report the FL training of MobileNet [36] on CIFAR-10 [48] dataset by using 10 clients. The CIFAR-10 dataset is partitioned into totally 10 clients by Latent Dirichlet Allocation (LDA). The experiment follows the same training procedure as the standard FedAVG algorithm. We use 64 local batch size, 0.001 learning rate, 1 local epoch and ADAM [46] optimizer for the clients. We train MobileNet for 800 rounds and achieve 88.17% Top-1 accuracy for the baseline, which is consistent with the FedML benchmark [33]. We use Top-r (r=10%) as the sparsifier to generate sparse tensors for DeepReduce, and compression is applied bidirectionally with error feedback. Figure18 shows that Top-r slightly affects the convergence rate compared with the baseline. Nonetheless, applying DeepReduce on Top-r does not compromise the convergence behavior and the final accuracy while largely reducing the data volume (see Table 5).

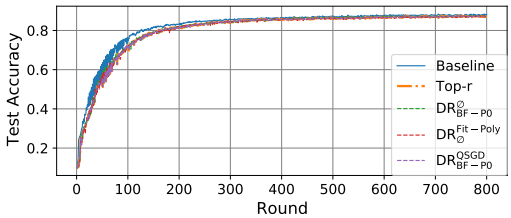


Figure 18: Convergence timeline of training MobileNet on CIFAR-10 datasets.

Method	Relative Data Volume	Top1 Test Acc.
Baseline	1.0	0.8817
Top-r	0.2069	0.8708
$\text{DR}^{\text{Fit-Poly}}_{\emptyset}$	0.1087	0.8700
$\text{DR}^{\emptyset}_{\text{BF-P0}}$	0.1475	0.8758
$\text{DR}^{\text{QSGD}}_{\text{BF-P0}}$	0.0713	0.8740

Table 5: Relative data volume and Top-1 test accuracy of MobileNet on CIFAR-10 in FL setup.

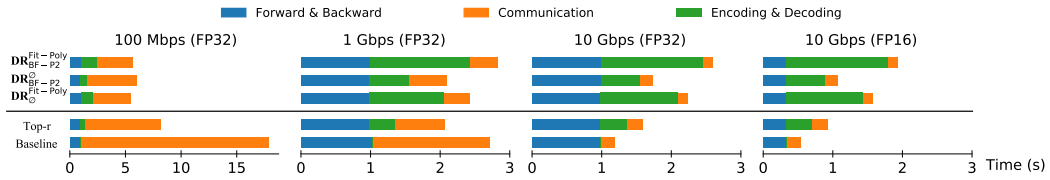


Figure 19: Time breakdown in one iteration training of NCF on ml-20m. We show the speedup of training by DR on 4 nodes with different network bandwidth: 100Mbps vs. 1Gbps vs. 10Gbps, and also with FP16 mixed precision training.

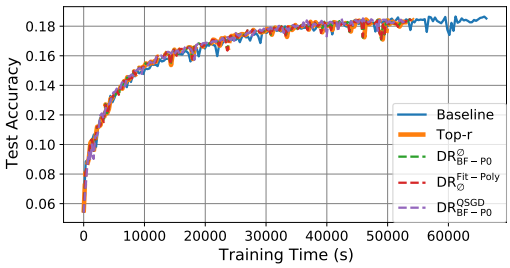


Figure 20: Convergence timeline of training an RNN to do next-word-prediction on Stack Overflow datasets.

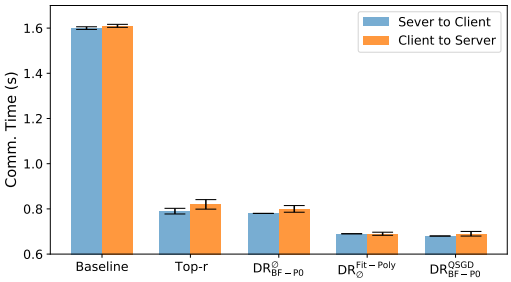


Figure 21: Communication time (with error bars) for realistic Federated Learning deployments.

Method	Relative Data Volume	Best Hit Rate
Baseline	1.0	0.9497
SKCompress	0.2175	0.9513
$\text{DR}^{\text{QSGD}}_{\text{BF-P0}}$	0.2063	0.9496

Table 6: We train NCF, an inherently sparse model, on ML-20m, with 10^6 local batch size. We test $\text{DR}^{\text{QSGD}}_{\text{BF-P0}}$, which uses BF-P0 (FPR=0.6) for indices, but combines it with QSGD [7], an existing method for value compression. This demonstrates that DeepReduce is compatible with various existing compressors. We compare against SKCompress [40], an improved version of SketchML, optimized for sparse tensors. We configure QSGD and SKCompress for 7-bits quantization and set the QSGD bucket size to 512. For SKCompress, we omit the grouped MinMaxSketch and separation of positive/negative gradients, as they have only minor effects. All methods achieve virtually the same best hit rate (i.e., the quality metric for NCF), and both $\text{DR}^{\text{QSGD}}_{\text{BF-P0}}$ and SKCompress reduce the data volume by 5× compared with Baseline. However, in practice $\text{DR}^{\text{QSGD}}_{\text{BF-P0}}$ can be more easily implemented on GPUs. Therefore, in our experiments (see Figure 8b) it is 380× faster in terms of compression and decompression time.

Algorithm 2 FedAvg with DeepReduce

Input: Number of clients K indexed by k, local minibatch size b, number of local epochs E, learning rate η, DeepReduce compression:=DR, DeepReduce decompression:=DR^{-1}

Output: Trained model x

On server side:

Initialize x_0

for *round* $t = 1, 2, \ldots,$ **do**

  $S_t \leftarrow$ (random set of m clients out of K clients)

  $g_t \leftarrow \text{DR}(x_t - x_0)$

  for *each client* $k \in S_t$ ***in parallel*** **do**

    $g^k_{t+1} \leftarrow \text{CLIENTUPDATE}(k, g_t)$

  $g_{t+1} \leftarrow \frac{1}{m}\sum_{k=1}^{m} \text{DR}^{-1}(g^k_{t+1})$

  $x_{t+1} \leftarrow x_t - \eta g_{t+1}$

On client k **side:**

Pull x_0 from server

while *training* **do**

  $\text{CLIENTUPDATE}(k, g_t)$

function $\text{CLIENTUPDATE}(k, g_t)$

Pull g_t from server

$x_t \leftarrow x_0 + \text{DR}^{-1}(g_t)$

$g_{t+1} \leftarrow \mathbf{0}$

for *each local epoch* i *from* 1 *to* E **do**

  for *batch* $b \in$ *local training data* **do**

    $g_{t+1} \leftarrow g_{t+1} + \nabla \ell(x_t; b)$

Push $\text{DR}(g_{t+1})$ to server