3LC: Lightweight and Effective Traffic Compression for Distributed Machine Learning

Hyeontaek Lim,[1] David G. Andersen,[1] Michael Kaminsky[2]

[1]*Carnegie Mellon University,* [2]*Intel Labs*

Abstract

The performance and efficiency of distributed machine learning (ML) depends significantly on how long it takes for nodes to exchange state changes. Overly-aggressive attempts to reduce communication often sacrifice final model accuracy and necessitate additional ML techniques to compensate for this loss, limiting their generality. Some attempts to reduce communication incur high computation overhead, which makes their performance benefits visible only over slow networks.

We present *3LC*, a lossy compression scheme for state change traffic that strikes balance between multiple goals: traffic reduction, accuracy, computation overhead, and generality. It combines three new techniques—*3-value quantization with sparsity multiplication*, *quartic encoding*, and *zero-run encoding*—to leverage strengths of quantization and sparsification techniques and avoid their drawbacks. It achieves a data compression ratio of up to 39–107×, almost the same test accuracy of trained models, and high compression speed. Distributed ML frameworks can employ 3LC without modifications to existing ML algorithms. Our experiments show that 3LC reduces wall-clock training time of ResNet-110–based image classifiers for CIFAR-10 on a 10-GPU cluster by up to 16–23× compared to TensorFlow's baseline design.

1 Introduction

Distributed machine learning (ML) harnesses high aggregate computational power of multiple worker nodes. The workers train an ML model by performing local computation and transmitting *state changes* to incorporate progress made by the local computation, which are repeated at each *training step*. Common metrics of interest in distributed ML include *accuracy* (how well a trained model performs) and *training time* (wall-clock time until a model reaches a trained state). To improve training time, distributed ML must be able to transmit large state change data quickly and avoid impeding local computation.

However, the network does not always provide sufficient bandwidth for rapid transmission of state changes. Large-scale deployment of distributed ML often require the workers to communicate over a low-bandwidth wide-area network (WAN) to conform to local laws that regulate transferring sensitive training data (e.g., personal photos) across regulatory borders [5, 10, 17, 22, 36]. Some data might be pinned to mobile devices [21, 28], forcing distributed ML to use a slow and sometimes metered wireless network. Recent performance studies show that in-datacenter distributed training can demand more bandwidth than local networks and even GPU interconnects currently offer [3, 25, 39, 41].

Communication reduction intends to mitigate the network bottleneck by reducing the overall communication cost. In particular, lossy compression schemes reduce the volume of state change data by prioritizing transmission of important state changes [1, 16, 17, 24, 30]. Unfortunately, existing schemes suffer one or more problems: They offer only a small amount of network traffic reduction, sacrifice the accuracy of the trained model, incur high computation overhead, and/or require modifications to existing ML algorithms.

We present 3LC[1] (3-value lossy compression), a lightweight and efficient communication reduction scheme. 3LC strikes a balance between traffic reduction, accuracy, computation overhead, and generality, to provide a "go-to" solution for bandwidth-constrained distributed ML. Our design (1) uses only 0.3–0.8 bits for each real-number state change on average (i.e., traffic reduction by 39–107× from original 32-bit floating point numbers), (2) causes small or no loss in accuracy when using the same number of training steps, (3) adds low computation overhead, and (4) runs with unmodified ML algorithms.

To achieve both high efficiency and high quality for distributed ML, 3LC unifies two well-known lossy compression approaches commonly used for communication reduction: *Quantization* encodes state changes in low resolution, and *sparsification* only picks likely important parts of state changes. We do not blindly combine two approaches because doing so might end up suffering drawbacks of both approaches; instead, we take their principle and reconstruct them as a lightweight-yet-effective lossy compression scheme.

---

[1]Read as "elk."
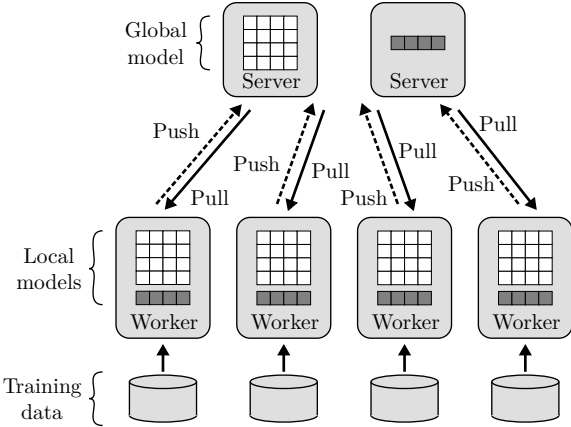
Figure 1: Distributed machine learning architecture using parameter servers.

3LC combines three new techniques:

3-value quantization with sparsity multiplication is a lossy transformation that maps each floating-point number representing a state change onto three values $\{-1, 0, 1\}$, with a knob that controls the compression level. It corrects resulting quantization errors over time by using error accumulation buffers. Since it makes a small impact on the trained model's accuracy, it does not require compensating for potential accuracy loss with ML algorithm changes.

Quartic encoding is a lossless transformation that folds each group of five 3-values into a single byte using fast vectorizable operations, which takes 20% less space use than simple 2-bit encoding of 3-value data. The quartic encoding output is easy to compress further.

Zero-run encoding is a lossless transformation that shortens consecutive runs of common bytes (groups of five zero values) by using a variant of run-length encoding [31] specialized for quartic encoded data. It achieves approximately a 2× or higher compression ratio, which varies by the distribution of state change values.

Our empirical evaluation of 3LC and prior communication reduction techniques on our custom 10-GPU cluster shows that 3LC is more effective in saving traffic reduction while preserving high accuracy at low computation overhead. When training image classifiers based on ResNet-110 [15] for the CIFAR-10 dataset [23], 3LC reduces training time to reach similar test accuracy by up to 16–23×. To measure 3LC's practical performance gains over a strong baseline, we use a production-level distributed training implementation on TensorFlow [1] that is already optimized for efficient state change transmission.

2 Distributed ML Background

Machine learning (ML) is a resource-heavy data processing task. Training a large-scale deep neural network (DNN) model may require tens of thousands of machine-hours [7]. Distributed ML reduces the total training time by parallelization [1, 24].

Figure 1 depicts typical distributed DNN training using parameter servers [7, 9, 16, 24]. *Parameter servers*, or simply *servers*, store a partition of the global model, which consists of *parameters* (trainable variables). *Workers* keep a local copy of the model and training dataset. The parameters (and their state changes) are often represented as *tensors* (multidimensional arrays). For example, the "weights" of a fully-connected layer (a matrix multiply) would be a single 2-D tensor of floats. The weights of a different layer would be a separate 2-D tensor.

The workers train the model by repeatedly performing local computation and state change transmission via the servers. Each *training step* includes the following substeps: *Forward pass:* The workers evaluate a *loss function* (objective function) for the current model using the local training dataset. *Backward pass:* The workers generate *gradients* that indicate how the model should be updated to minimize the loss function. *Gradient push:* The workers send the gradients to the servers. *Gradient aggregation and model update:* The servers average the gradients from the workers and update the global model based on the aggregated gradients. *Model pull:* The workers retrieve from the servers *model deltas* that record the model changes, and apply the deltas to the local model.

Distributed ML may observe two types of communication costs, training step barriers and state change traffic, which we discuss in the rest of this section.

2.1 Relaxing Barriers

One important pillar of distributed ML research is how to perform efficient synchronization of workers using barriers. Although relaxing barriers is not the main focus of our work, we briefly describe related techniques because modern distributed ML systems already employ these optimizations to partially hide communication latency.

In vanilla *bulk synchronous parallel* (BSP), workers train on an identical copy of the model [35]. BSP forces the servers to wait for all workers to push gradients, and the workers to wait for the servers to finish updating the global model before model pulls. In this model, slow or failed workers ("straggler") [16, 30] make other workers waste computation resources, increasing training time.

To reduce a straggler problem, researchers have capitalized upon the property that stochastic gradient descent and its variants commonly used in distributed ML tolerate a small amount of inconsistency in the model across the workers [30]. Fully asynchronous state change transmission permits a worker to submit an update based on an arbitrarily stale version of the model [30]. Approaches such as stale synchronous parallel make a compromise between two extremes by limiting the maximum asynchrony of the model for which an update is calculated [16].
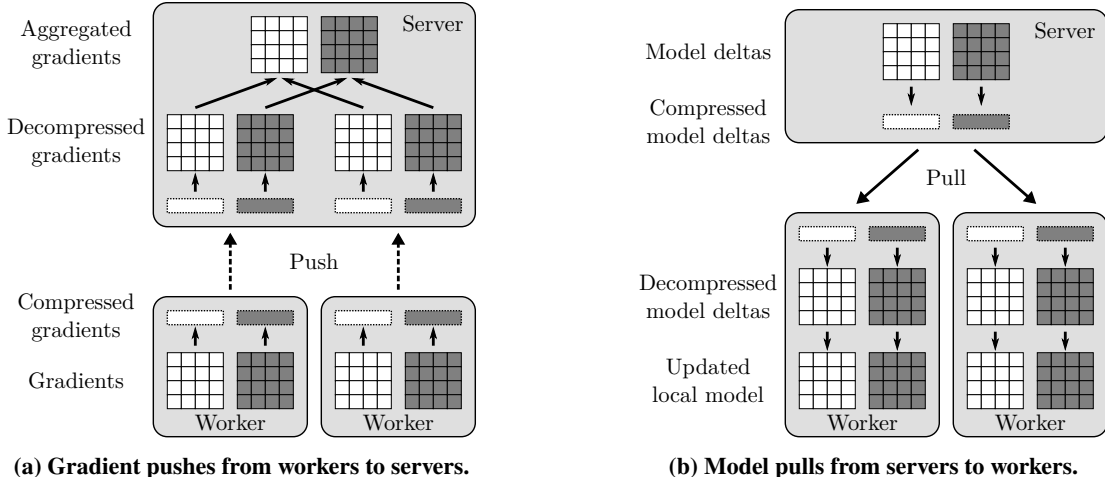
(a) Gradient pushes from workers to servers.

(b) Model pulls from servers to workers.

Figure 2: Point-to-point tensor compression for two example layers in 3LC.

A common downside of asynchronous state change transmission is that it may accomplish less useful work per training step because of desynchronized local models. Asynchronous state change transmission generally requires more training steps than BSP to train a model to similar test accuracy [1, 16, 17, 24, 30]. Thus, recent distributed ML frameworks often advocate synchronous state change transmission while using other techniques that mitigate stragglers. For instance, TensorFlow [1]'s stock distributed training implementation, SyncReplicasOptimizer, uses backup workers: A global training step can advance if a sufficient number of updates to the latest model have been generated regardless of the number of unique workers that calculated the updates [6].

Modern distributed ML frameworks split barriers into more *fine-grained barriers* that help hide communication latency. For example, Poseidon pushes individual layers' gradients, allowing the servers to update part of the model and let the workers pull that part instead of having to wait for the entire model to be updated [41]. TensorFlow's SyncReplicasOptimizer pulls updated model data for individual layers as they are evaluated in the forward pass. Such fine-grained barriers facilitate overlapping communication and computation and improve computational efficiency of distributed ML.

2.2 Compressing State Change Traffic

Relaxed barriers reduce communication costs, but they do not completely hide communication latency. Gradient pushes and model pulls are sensitive to the available network bandwidth, as these steps need to transmit large data quickly, and state change transmission can take longer as the model size grows and/or the network bandwidth is more constrained [3, 17, 25, 39, 41]. If the transmission takes excessive time, cluster nodes experience long stall time, harming the efficiency of distributed learning.

Quantization and sparsification techniques make state change transmission generate less network traffic by applying lossy compression to the state change data. They prioritize sending a small amount of likely important state change information and defer sending or even ignore unsent changes. *Quantization* uses low-resolution values to transmit the approximate magnitude of the state change data [3, 32, 39]. *Sparsification* discovers state changes with large magnitude and transmits a sparse version of tensors that contain these state changes [2, 17, 24, 25, 37, 38].

Note that quantization and sparsification we discuss in this paper differ from model compression [14, 20]. *Model compression* reduces the memory requirement and computation cost of DNN models by quantizing and reducing their parameters (not state changes). *Inference* with a compressed model can run faster without demanding much computation and memory resources. In contrast, our paper focuses on *distributed training* of a model that consists of full-precision parameters, which can be processed using model compression after training finishes.

3 Design

The design goal of 3LC is to achieve good balance between traffic reduction, accuracy, computation overhead, and generality. We present the high-level design of 3LC and its components in detail.

3LC is a point-to-point tensor compression scheme. Figure 2 depicts how 3LC compresses, transmits, and decompresses state change tensors for two example layers. One compression context encompasses the state for compression and decompression of a single tensor that represents gradients (a push from a worker to a server) or

model deltas (a pull from a server to a worker) of a single layer in a deep neural network.

This point-to-point design preserves the communication pattern of existing parameter server architectures. It adds no extra communication channels between servers or workers because it involves no additional coordination between them. Some designs [39] synchronize their compression parameters among workers before actual traffic compression, which adds round trips to communication between the workers for each training step.

A potential performance issue of this point-to-point compression is redundant work during model pulls. Servers send identical data to workers so that the workers update their local model to the same state. If the servers compress individual pulls separately, it would perform redundant compression work. 3LC optimizes model pulls by sharing compression: The servers compresses model deltas and make a shared local copy of the compressed model deltas, and the workers pull the compressed data as if they pull uncompressed model deltas (Figure 2b). Note that distributed ML frameworks that allow loosely synchronized local models on workers [16, 17, 24, 30] may require multiple copies of compressed model deltas, each of which is shared by a subset of the workers with the same local model.

For 3LC's tensor compression and decompression, we introduce one lossy and two lossless transformations: 3-value quantization with sparsity multiplication (Section 3.1), quartic encoding (Section 3.2), and zero-run encoding (Section 3.3). The rest of this section describes their designs and rationale.

3.1 3-value Quantization with Sparsity Multiplication

3-value quantization compresses a state change tensor by leveraging the distribution of state changes that are centered around zero [39]. It transforms a full-precision input tensor into a new tensor of three discrete values $\{-1, 0, 1\}$ that has the same shape (dimensions) as the input tensor, and a full-precision scalar M that is the maximum magnitude of the input tensor values scaled by a *sparsity multiplier* s ($1 \le s < 2$).

Suppose T_{in} is an input tensor. The output of 3-value quantization is

$$M = \max(|T_{\text{in}}|) \cdot s \qquad (1)$$

$$T_{\text{quantized}} = \text{round}\left(\frac{T_{\text{in}}}{M}\right) \qquad (2)$$

Dequantization is a simple multiplication:

$$T_{\text{out}} = M \cdot T_{\text{quantized}} \qquad (3)$$

s controls the compression level of 3LC. $s = 1$ is the default multiplier that preserves the maximum magnitude
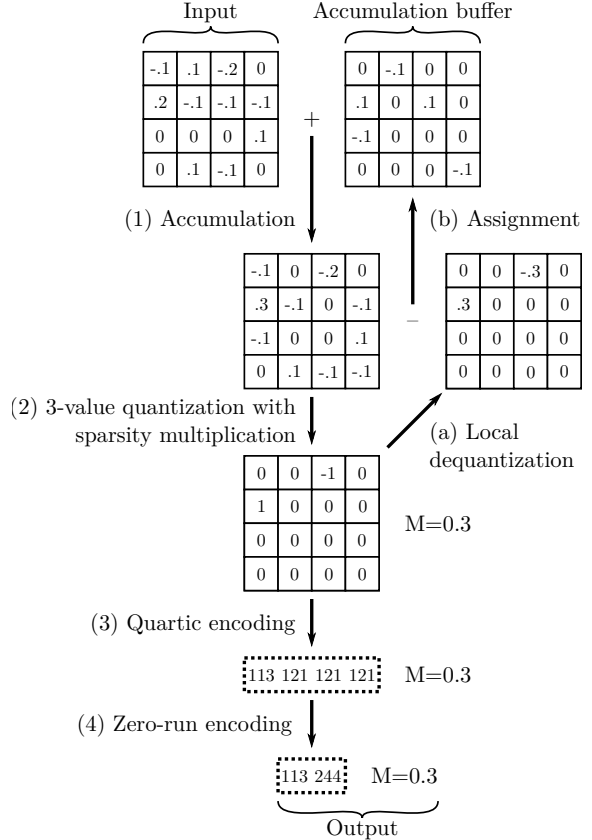

Figure 3: Tensor compression in 3LC.

of values in the input tensor across quantization and dequantization. With a larger s ($1 < s < 2$), the quantization output is sparser (more zeros) because the magnitude of more values are smaller than $M/2$. The sparser output may contain less state change information, but can be compressed more aggressively by zero-run encoding.

Quantization followed by dequantization returns a slightly different tensor from the input tensor, causing *quantization errors*. 3LC can experience relatively larger quantization errors especially when s is larger because dequantization can make a value farther from its original value (but within a certain limit to ensure convergence).

3LC corrects quantization errors using error accumulation buffers [2, 17, 32, 37, 38]. It allows quantization errors to occur in the first place, but attempts to correct in quantization at later training steps. It keeps a local per-tensor error accumulation buffer to remember the errors across training steps.

Figure 3 depicts 3-value quantization with error accumulation, using $s = 1$. Step (1) accumulates the input tensor into a local buffer. Step (2) applies 3-value quantization to the sum. Step (a) dequantizes the quantized data locally. Step (b) calculates remaining quantization errors and stores them in the local buffer.

Alternative quantization techniques: *Stochastic quantization* outputs randomized quantization values whose expectation matches their input value [3]. It eliminates biases that exist in deterministic rounding. For instance, TernGrad [39] uses three values for quantization similarly to 3-value quantization (without the sparsity multiplication), but uses stochastic selection of output values. We decided to use error accumulation buffers instead of stochastic quantization for several reasons: (1) Biases that are caused by non-stochastic quantization can be corrected over time by using error accumulation buffers. (2) When used alone, error correction with error accumulation buffers achieves better accuracy than stochastic quantization in our evaluation (Section 5); designs using stochastic quantization require more bits for quantization [3] or additional accuracy-compensation techniques [39] for high accuracy. (3) Using both error accumulation buffers and stochastic quantization caused training fail to converge in our experiments.

Squared quantization error minimization is a deterministic method that picks magnitude values that minimize the squared sum of quantization errors. For instance, 1-bit stochastic gradient descent maps non-negative values and negative values of an input tensor into two values $\{0, 1\}$, and each of these two values are dequantized using a different M value that is the average of non-negative or negative values in the input tensor [32]. In designing 3LC, we avoid reducing the magnitude of quantized values instead of pursuing minimum squared quantization errors because (1) low quantization errors do not necessarily lead to high accuracy in empirical evaluation (Section 5) and (2) other lossy compression techniques for state change traffic also preserve the approximate magnitude of input tensors for better accuracy even though doing so may provide weaker theoretic guarantees [3, 39].

Alternative sparsification techniques: The sparsity multiplier plays a role similar to the *threshold* knob in sparsification-based compression techniques [17, 25]. Both affect how many distinct state changes are chosen for transmission. However, thresholding makes a decompressed tensor have much smaller average values than the input tensor by omitting many input values (even though they are small); overly-aggressive thresholding can result in lower accuracy, and compensating for it requires changing ML algorithms such as modified momentum calculation [25] that does not generalize well to non-gradient data transmission such as model pulls. In contrast, dequantization using sparsity multiplication enlarges (now scarcer) large values, better preserving the average magnitude of the input tensor.

3-value quantization always uses a dense form (array) of tensors. Dense tensor operations are easier to accelerate than sparsification-based compression techniques that requires dense-to-sparse and sparse-to-dense tensor conversion whose vectorization is often unavailable (e.g., TensorFlow [1] has only a non-vectorized CPU implementation and no GPU implementation for sparse-to-dense conversion as of February 2018).

Prior lossy traffic reduction schemes often employ custom rounding function [2, 17, 24, 25, 32] that often makes vectorization difficult. 3-value quantization instead uses simple round() whose vectorized version is readily available on modern CPUs and GPUs [8, 18].

Convergence: 3-value quantization with sparsity multiplication retains convergence of state change tensors. round() adds a maximum absolute error of $1/2$. By Equations 2 and 3, the maximum absolute error $max(|T_{\text{in}} - T_{\text{out}}|)$ is bounded by $M/2$. Note $M/2 < \max(|T_{\text{in}}|)$ because of Equation 1 and $1 \leq s < 2$. Let α be a decaying learning rate (if T_{in} is a gradient tensor) or 1 (if T_{in} is a model delta tensor). Under an assumption that αT_{in} converges to zero, $\alpha M/2$ converges to zero, and αT_{out} also converges to zero.

3.2 Quartic Encoding

Compactly encoding 3-values is nontrivial because CPU and GPU architectures do not provide native data types for base-3 numbers. The space requirement of a simple encoding for 3 discrete values using 2 bits [39] is larger than the theoretic minimum of $\log_2 3 \approx 1.585$ by approximately 26%.

Quartic encoding is a fixed-length representation for a 3-value quantized tensor. It takes five 3-values and packs them into a single byte [Figure 3 Step (3)], using *1.6 bits per 3-value* that is only 0.95% higher than the theoretic bound. Quartic encoding exploits the fact that a quartic-form expression $a \cdot 3^4 + b \cdot 3^3 + c \cdot 3^2 + d \cdot 3^1 + e$ has only $3^5 = 243$ distinct values (≤ 256) if $a, \ldots, e \in \{0, 1, 2\}$. Quartic encoding of a 3-value quantized tensor takes the following steps:

1. Element-wise add 1 to the 3-value quantized tensor
2. Type cast it to an unsigned 8-bit integer array
3. Flatten it into a 1-D array
4. Pad it with zeros to make its length a multiple of 5
5. Divide the array into 5 partitions: p_0, p_1, p_2, p_3, p_4
6. Compute $a = p_0 \cdot 81 + p_1 \cdot 27 + p_2 \cdot 9 + p_3 \cdot 3 + p_4$

Decoding reverses encoding steps:

1. Restore p_0, p_1, p_2, p_3, p_4 by dividing a by a power of 3 and taking the remainder (a base-3 conversion)
2. Concatenate p_0, p_1, p_2, p_3, p_4
3. Unpad, reshape, and type cast
4. Element-wise subtract 1

These encoding and decoding steps can be easily vectorized on CPUs and GPUs using operations provided by ML frameworks.

3.3 Zero-run Encoding

The input to quartic encoding is sparse (even though the data structure is dense), containing a large number of zeros. The number of zeros increases as the sparsity multiplier s increases. Although quartic encoding is compact, it always generates a fixed-length representation, which does not take advantage of the sparseness in the input.

Zero-run encoding is a variant of run-length encoding [31], but is specialized to quartic-encoded data. Note that quartic encoding maps a group of five zero values from the 3-value quantized tensor into a byte value 121. Also recall that quartic encoding only outputs byte values of 0–242. Zero-run encoding finds a run of 121 and replaces it with a new byte value between 243 and 255, inclusive [Figure 3 Step (4)]. In other words, k consecutive occurrences of 121 ($2 \leq k \leq 14$) are replaced with a single byte value of 243+(k-2). In a hypothetical case of compressing a zero 32-bit floating-point tensor, the combination of all techniques in 3LC reaches a compression ratio of 280×.

Compared to general-purpose compression algorithms or entropy coding schemes [3, 12, 29], zero-run encoding is simple to implement and fast to run by avoiding any bit-level operation and lookup tables, which helps 3LC keep low computation overhead.

4 Implementation

We implement a prototype of 3LC on TensorFlow [1]. 3-value quantization with sparsity multiplication and quartic encoding use TensorFlow's built-in vectorized operators. Zero-run encoding uses a custom operator written in C++.

Our prototype includes a distributed optimizer that retains the interface of SyncReplicasOptimizer, which is TensorFlow's stock distributed training implementation. The distributed optimizers augment any local optimizer with distributed training by providing gradient aggregation and training step barriers. To replicate TensorFlow's tensor caching and incremental pull behavior that copies each remote tensor into a local cache before local access to that tensor, our prototype ensures that first-time access to a tensor at each training step executes extra operators that pull, decompress, and apply model deltas to the tensor.

One user-facing change is tensor allocation. Our prototype asks the user program to call a helper function that provides the same interface as get_variable(), which is a TensorFlow function that allocates a single tensor. This helper function reserves buffers for error accumulation and compressed model deltas, and assigns a correct physical location to the buffers. The user program can keep using the default get_variable() for the tensors that do not require compression (e.g., tensors for small layers); our distributed optimizer falls back to SyncReplicasOptimizer's behavior for distributed training of these tensors.

5 Evaluation

We experimentally evaluate 3LC to quantify its effectiveness against other communication reduction schemes. Our experiments investigate the following aspects:

- Traffic: How much traffic does each scheme save?
- Training time: How much wall-clock training time do they save?
- Accuracy: What is the highest test accuracy each scheme can achieve using standard training steps?
- Convergence speed: What is the highest test accuracy they achieve using much fewer training steps?[2]
- Computation overhead: How low is their computation overhead?

5.1 Compared Designs

Our evaluation compares representative communication reduction schemes that we implement on TensorFlow:

32-bit float is the baseline that transmits 32-bit floating-point state changes without compression.

8-bit int is an 8-bit quantization scheme that approximates Google Tensor Processing Unit's internal 8-bit quantization [20]. Our implementation uses 255 distinct values ($[-127, 127]$, leaving -128 unused).

Stoch 3-value + QE uses stochastic 3-value quantization similar to TernGrad (but without "gradient clipping") [39], and our quartic encoding for 1.6-bit quantization (smaller than TernGrad's 2-bit quantization).

MQE 1-bit int performs 1-bit quantization with minimum squared quantization errors and error feedback [32].

25% sparsification and 5% sparsification choose 25% and 5% of the largest state changes in each tensor, respectively, and accumulate unsent changes in buffers, which reproduce common sparsification techniques [2, 17, 24, 25, 38]. We use the magnitude (not relative magnitude [17]) of values to find largest values for better accuracy in our experiments. To avoid exhaustive sorting while finding a threshold, we only sort sampled input values [2]. We use a bitmap to indicate which state changes sparsification has selected, which adds 1 bit per state change as traffic overhead regardless of input size.

2 local steps transmits state changes every 2 local steps. Unsent updates are accumulated locally and sent at the next training step using error accumulation buffers. It reduces the traffic almost by half and effectively doubles the global batch size of distributed training.

3LC is the full 3LC design. s is the sparsity multiplier.

Similar to prior work [3], we exclude state changes for small layers (batch normalization [19] in our experiments) from compression because avoiding computation overhead far outweighs compacting already small tensors.

---

[2]High convergence speed with few training steps can be useful for accurate and fast hyperparameter (training configuration) optimizations using small computational resources [11, 34].

6

Note that the implementation of some compared designs are not identical to prior proposed designs because their design is incompatible with our workload and the TensorFlow parameter server architecture. For instance, `sparsification` does not use modified momentum algorithms [25] because and TensorFlow sends not only gradients, but also model deltas to which their modifications of ML algorithms are inapplicable.

5.2 Evaluation Setup

Workload: Our experiments train image classifiers based on ResNet-110 [15] for the CIFAR-10 dataset [23]. CIFAR-10 contains 50,000 training images and 10,000 testing images, each of which has one of 10 labels. ResNet-110 is a 110-layer convolutional neural network for CIFAR-10.

Detailed training configuration: The following paragraphs provide an exhaustive description of our ML parameters and environment for completeness. We use standard configurations and values from the literature [15, 27]. Readers can feel free to skip these ML-focused details and resume at the "Hardware and Network" paragraph.

We reuse the local optimizer type and hyperparameters for ResNet-110 training from the original ResNet paper [15] except for the learning rate schedule. The local optimizer is TensorFlow's `MomentumOptimizer` with the momentum of 0.9. The weight decay is 0.0001. We vary the learning rate from 0.1 to 0.001, following the original learning rate range, but we use cosine decay without restarts [27] instead of the original stepwise decay because the cosine decay achieves better accuracy [4, 27] and has fewer hyperparameters to tune. We apply the standard data augmentation that randomly crops and horizontally flips original images to generate training examples [15].

Our distributed training configuration follows the guideline for large-batch stochastic training [13]. We use a per-worker batch size of 32 images [25]; using the original batch size of 128 reduces accuracy for all designs because it produces a large global batch size of 1,280 on a 10-worker cluster. We scale the learning rate proportionally to the worker count and make one worker responsible for updating batch normalization parameters [13]. Our accuracy matches or exceeds the accuracy of a ResNet-110 trained using a similar batch size but stepwise decay [25].

We choose to train a ResNet because it is both a representative and *challenging* workload for communication reduction schemes to show their performance benefits. The ResNet architecture's "identity mappings" are commonly found in high-accuracy neural network architectures [42]. Compared to traditional neural network architectures such as VGG [33], ResNet models typically have small parameter count to computation ratios [42], generating *less state change traffic* for the same amount of communication. Its very deep network structure permits *efficient incremental transmission* of state changes (Section 2.1),

facilitating overlapping computation and communication and hiding communication latency. Therefore, we believe that a capability to show performance gains on the ResNet architecture is likely to be transferable to other neural network architectures.

Hardware and Network: Our distributed training runs on a custom GPU cluster. It uses 10 workers with a GPU; each pair of workers shares a physical machine equipped with two Intel Xeon E5-2680 v2 CPUs (total 20 physical cores), 128 GiB DRAM, and two Nvidia GTX 980 GPUs. Our experiments use `numactl` for CPU and memory isolation between worker pairs and `CUDA_VISIBLE_DEVICES` for a dedicated per-worker GPU. A separate machine acts as a parameter server. We use the Linux Traffic Control [26] on worker and server nodes to emulate constrained network bandwidth.

Measurement Methodology: A dedicated node reads the snapshot of the global model and calculates the *top-1 score of the testing images* as test accuracy.

Due to limited computation resources, we divide the experiments into two categories of full measurement and accelerated measurement. *Full measurement* measures training time, average per-step training time, and accuracy on 1 Gbps by executing *standard training steps* (163.84 epochs [15], which is equivalent to 25,600 steps for 10 workers with a batch size of 32). *Accelerated measurement* only obtains average per-step time on 10 Mbps and 100 Mbps links by executing 100 and 1000 steps, respectively (about 1 hour of training for `32-bit float`); one exception is that any design with zero-run encoding runs 10% of standard training steps to faithfully reflect its compression ratios changing over time. The learning rate schedule uses adjusted training steps as the total training steps (as usual) to ensure each training run to sweep the entire learning rate range.

We predict the training time on 10 Mbps and 100 Mbps by scaling the training time from the 1 Gbps full measurement based on per-step training time differences between full and accelerate measurement results while reusing the accuracy from the full measurement. Suppose a full measurement result for 1 Gbps is training time of t_{full}, per-step training time of s_{full}, and an accelerated measurement result for 10 Mbps is per-step training time of s_{short}. We estimate the training time of 10 Mbps to be $t_{full} \cdot s_{short}/s_{full}$. We take test accuracy obtained in the full measurement as-is because network bandwidth changes do not affect test accuracy. Without training time extrapolation, obtaining a single datapoint on a slow network takes approximately 10 days on our cluster, which would make it hard for us to compare many designs extensively at high confidence.

We show the average of measurement results from multiple independent runs. Each experiment configuration is run 5 times for full measurement, and 3 times for accelerated measurement.

Design	Speedup (×)			Accuracy (%)	Difference
	@ 10 Mbps	@ 100 Mbps	@ 1 Gbps		
`32-bit float`	1	1	1	93.37	
`8-bit int`	3.62	3.47	1.51	93.33	−0.04
`Stoch 3-value + QE`	12.3	7.51	1.53	92.06	−1.31
`MQE 1-bit int`	14.6	7.40	1.30	93.21	−0.16
`25% sparsification`	3.25	3.11	1.33	93.40	+0.03
`5% sparsification`	8.98	6.62	1.44	92.87	−0.50
`2 local steps`	1.92	1.87	1.38	93.03	−0.34
`3LC (s=1.00)`	15.9	7.97	1.53	93.32	−0.05
`3LC (s=1.50)`	20.9	8.70	1.53	93.29	−0.08
`3LC (s=1.75)`	22.8	9.04	1.53	93.51	+0.14
`3LC (s=1.90)`	22.8	9.22	1.55	93.10	−0.27

Table 1: Speedup over the baseline and test accuracy using standard training steps (graphs in the next page).

5.3 Macrobenchmark

We examine the tradeoff between total training time and accuracy of compared schemes. Each datapoint on the graph represents a separate experiment configuration; the learning schedule (the cosine decay) depends on total training steps, requiring a new experiment for accuracy measurement for a different number of total training steps.

Table 1 summarizes training time speedups over the baseline and test accuracy when using standard training steps. 3LC achieves the best speedup across all network configurations, and its accuracy remains similar to the baseline, except `3LC (s=1.90)` that performs highly aggressive traffic compression. Other designs offer less training time reduction or suffer lower accuracy.

Figure 4 plots total training time and test accuracy on 10 Mbps when varying the total number of training steps to 25%, 50%, 75%, and 100% of standard training steps. An experiment using 100% training steps gives the accuracy of fully trained models, while using fewer training steps indicates the convergence speed of a design.

The result shows that designs that achieve high accuracy with many training steps do not always yield high accuracy with fewer training steps. `3LC (s=1.75)` provides the best training time and maintains high accuracy when using 100% training steps because of its effective traffic compression. When using fewer training steps, `3LC (s=1.00)` achieves better accuracy. 3LC's sparsity multiplication affects tradeoffs between traffic reduction and convergence speed, but it does not necessarily harm accuracy obtained using sufficient training steps (e.g., executing as many training steps as standard no-compression training uses).

Note that `Stoch 3-value + QE` has lower accuracy than 3LC. This accuracy loss by stochastic quantization supports our design decision of using error accumulation buffers to correct quantization errors.

With a faster network of 100 Mbps, as shown in Figure 5, the benefit of reducing traffic begins to diminish and preserving high accuracy becomes more important. For example, `5% sparsification` provides always better speed-accuracy tradeoffs than `Stoch 3-value + QE`, which is different on 10 Mbps.

On a 1 Gbps network, in Figure 6, the most influential factors to speed-accuracy tradeoffs are high accuracy and low computation overhead, and traffic reduction becomes less important. 3LC provides high accuracy using 75% to 100% of standard training steps and slightly lower accuracy than `8-bit int` using fewer training steps. `MQE 1-bit int` is slower than `8-bit int` that transmits 8× more traffic; the long training time of `MQE 1-bit int` is attributable to its high computation overhead of using an unconventional rounding function. 3LC does not add such high overhead because it leverages existing vectorized operations.

We also examine how designs perform during a training run in detail. Figure 7 depicts runtime (not final) training loss and test accuracy of the baseline, the most representative quantization, sparsification, and multiple local steps designs, and 3LC with the default sparsity multiplier; the result of omitted designs is similar to that of a close design (e.g., `8-bit int` is similar to the baseline). Except for 3LC, traffic reduction designs tend to have higher training loss, and their accuracy also increases slowly. In contrast, 3LC achieves small training loss and high accuracy that are close to those of the baseline.

5.4 Sensitivity Analysis

The control knob of 3LC is a sparsity multiplier s. With a high s, 3-value quantization emits more zeros that can make zero-run encoding more effective. We vary s and measure training time, traffic reduction, and accuracy.

Figure 8 compares tradeoffs between total training time and test accuracy. In general, a high sparsity multiplier reduces training time, but it can also lower convergence speed with fewer training steps. Most s values lead to high accuracy when using 100% of standard training steps, but $s = 1.90$ exhibits lower accuracy than others.
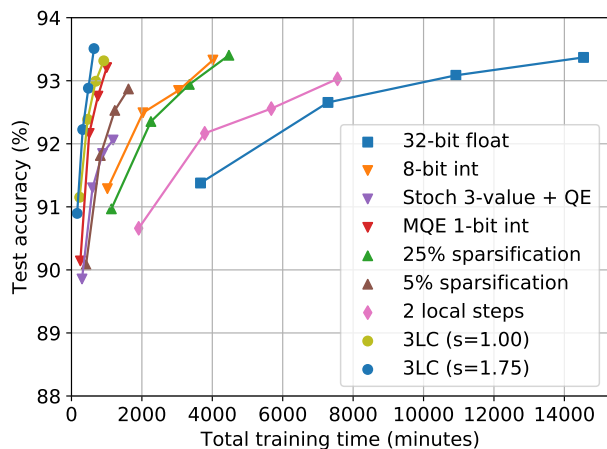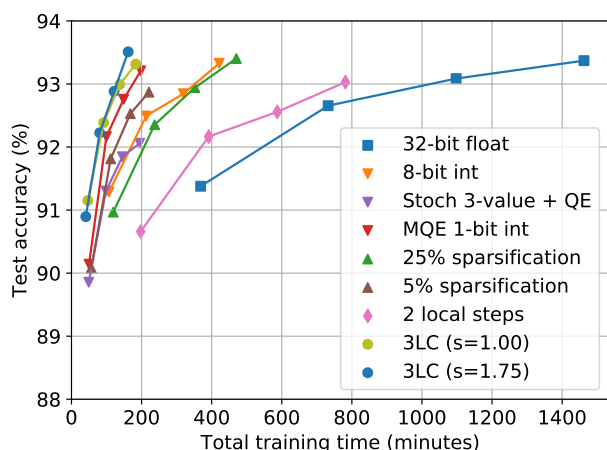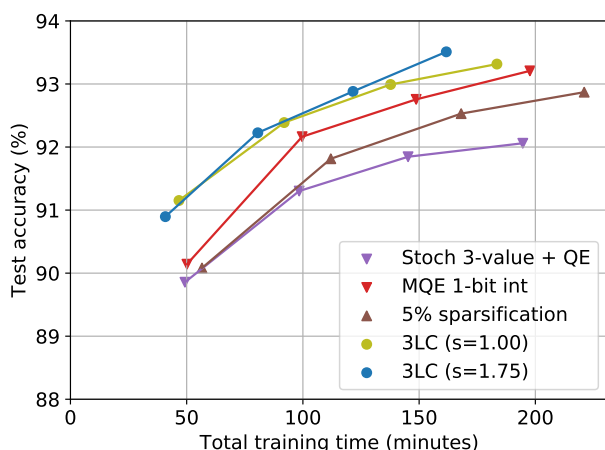
(a) Overview
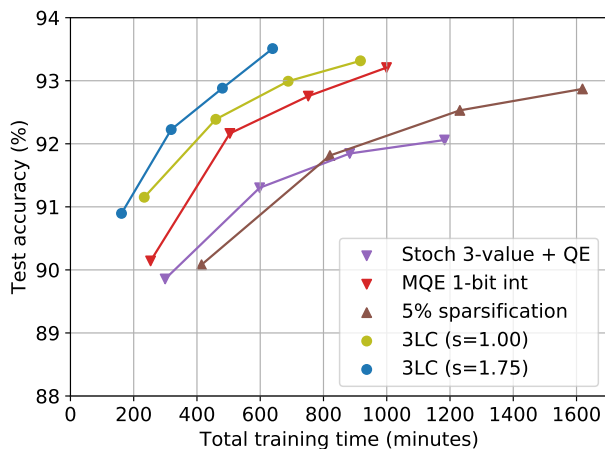
(b) Fast designs

Figure 4: Training time and test accuracy using 25/50/75/100% of standard training steps @ 10 Mbps.
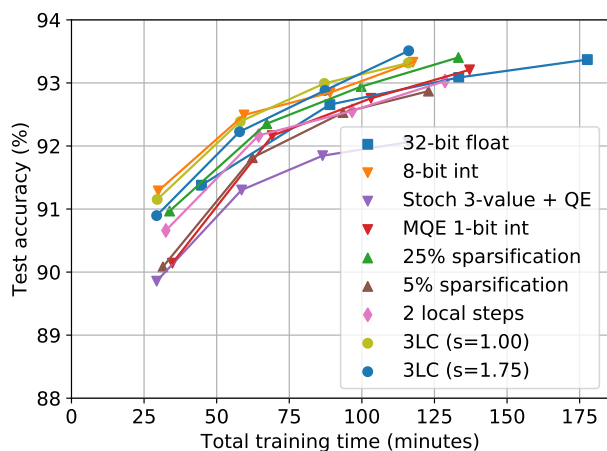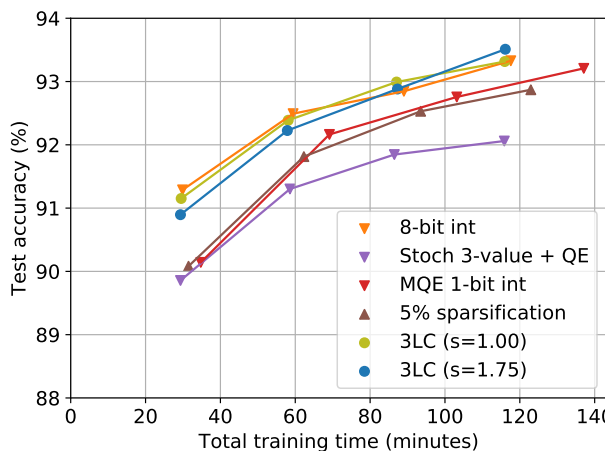

(a) Overview

(b) Fast designs

Figure 5: Training time and test accuracy using 25/50/75/100% of standard training steps @ 100 Mbps.

(a) Overview

(b) Fast designs

Figure 6: Training time and test accuracy using 25/50/75/100% of standard training steps @ 1 Gbps.
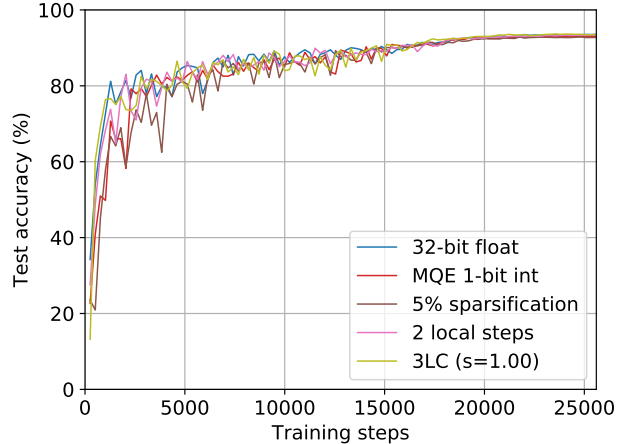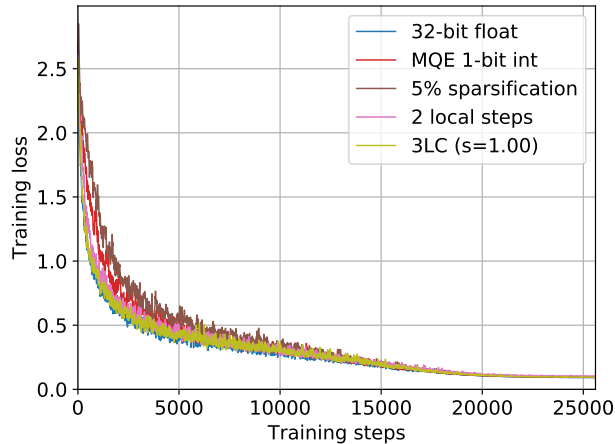
9

Figure 7: Training loss (left) and test accuracy (right) using standard training steps.
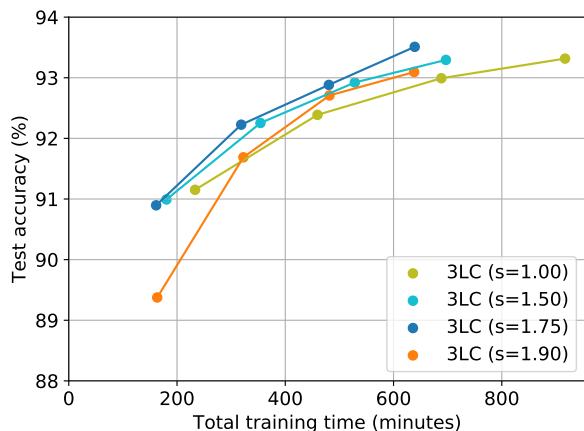

Figure 8: Training time and test accuracy with a varied sparsity multiplier (s) using 25/50/75/100% of standard training steps @ 10 Mbps.

s	Compression ratio ($\times$)	bits per state change
No ZRE	20.0	1.60
1.00	39.4	0.812
1.50	70.9	0.451
1.75	107	0.298
1.90	160	0.200

Table 2: Average traffic compression of 3LC using standard training steps.

each training step when executing standard training steps. Compressed pushes tend to be smaller than compressed pulls until the later stage of training, which indicates that state changes in model pulls have lower variance (including fewer zeros in the quantization output) because these changes reflect aggregated gradient pushes from multiple workers. After finishing approximately 70% of training, compressed pushes become larger, which shows that workers now generate gradients with lower variance. 3LC does not forcefully limit how many state changes can be transmitted at each training step; it permits transmitting important state changes as much as needed, which can help achieve fast convergence and high accuracy.

6 Related Work

Quantization: 1-bit stochastic gradient descent [32] represents each state change with two values, which can be dequantized using two floating-point numbers that minimize squared quantization errors. It accumulates quantization errors for later error correction. 3LC provides more effective traffic reduction that transmits approximately 1.6-bit worth information in a sub-1-bit representation without reducing the maximum magnitude of state change values (important for fast convergence and high accuracy). 3LC also provides a sparsity multiplier that can

Table 2 examines the average traffic reduction of 3LC. Without zero-run encoding ("ZRE"), the quartic-encoded size of each state change is 1.6 bits. Applying zero-run encoding halves the traffic volume for the default sparsity multiplier ($s = 1.00$). With a higher s, 3LC can compress traffic more aggressively; 3LC (s=1.90) realizes a 160× end-to-end compression ratio and 0.2 bits per state change. This high compression ratio can be useful for metered and/or highly bandwidth-constrained network connections where reducing the number of bytes required for state change transmission is crucial for cost-effective distributed ML.

The compression ratio of zero-run encoding changes over time because nodes generate different gradients and model deltas as the model changes. Figure 9 plots the compressed size of gradient pushes and model pulls at
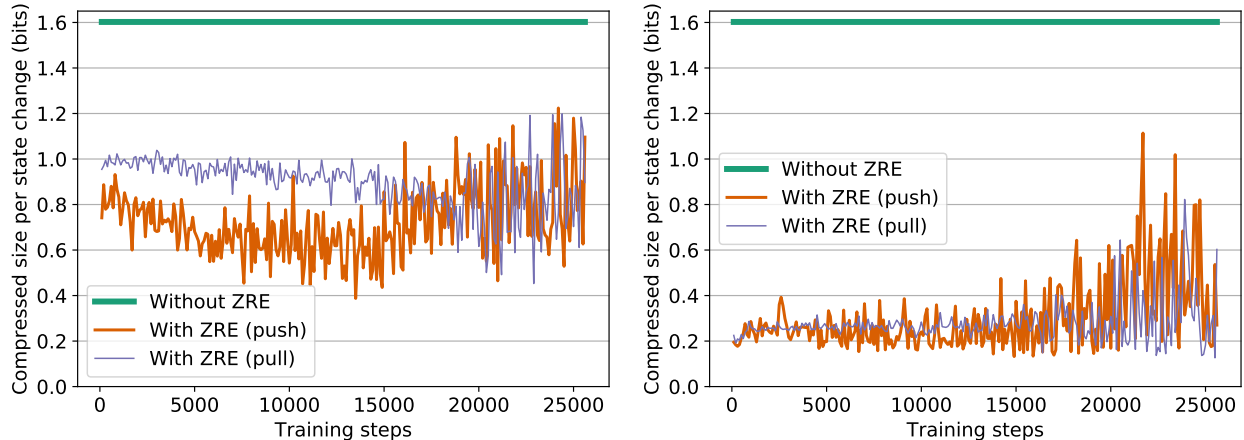
Figure 9: Compressed size per state change value using standard training steps (left: s=1.00; right: s=1.75).

change its compression level. 3LC's quantization and encoding methods are easier to vectorize by using existing vectorizable operations.

QSGD [3] and TernGrad [39] use stochastic quantization that makes quantized values an unbiased estimator of the original values. 3LC uses error accumulation buffers that empirically provide better accuracy without introducing changes to machine learning algorithms for accuracy compensation.

TernGrad [39] uses 3-values to quantize state changes, which is similar to 3LC's 3-value quantization. However, TernGrad lacks a knob to control the compression level and introduces a barrier to synchronize quantization parameters across all workers. TernGrad uses 2-bit encoding, which is far less compact than 3LC's encoding that requires only 0.3–0.8 bits per state change.

Quantization methods often employ entropy coding schemes such as Huffman coding and Elias coding for compact binary representations [3, 29]. 3LC's zero-run encoding offers high compression ratios (up to 8×) by using byte-level operations and no lookup tables, which helps achieve low computation overhead.

Sparsification: The parameter server [24] discusses filtering zero gradients for small-value model parameters. 3LC provides compression for both gradients and model deltas regardless of the magnitude of the model parameters.

Bösen [38] can prioritize sending large gradients and model deltas by sorting them. Because sorting millions of state change values is expensive, there are proposals that use a relative threshold [17], a global threshold [2], per-tensor thresholds [25], or round-robin selection [37] for low-overhead sparsification. Among these, Gaia [17] changes the relative threshold to send more state changes as training progresses. 3LC transmits larger compressed data in the later stage of training without having to control the compression level explicitly.

Gradient dropping [2] and Deep Gradient Compression [25] achieve high compression by selecting only 0.1% of gradients. This very aggressive gradient reduction, however, has worse accuracy. Recovering accuracy necessitates modifying machine learning algorithms [25], which reduces their generality and makes it hard to compress non-gradient state changes such as model deltas.

Project ADAM [7] and Poseidon [41] reduce network traffic by transmitting small "sufficient factors" that contain enough information to construct full gradient tensors for certain types of neural network layers [40]. 3LC pursues a general tensor compression scheme that can compress gradients and model deltas for any type of layers.

Infrequent communication: Federated learning [21, 28] runs multiple training steps before each global state change transmission. Our evaluation shows that infrequent transmission of state changes can lead to lower accuracy when using the same number of training steps.

7 Conclusion

A key challenge in modern, large-scale machine learning is marrying the demands of systems (reducing communication, overlapping computation and communication, and so on) and learning algorithms (algorithmic efficiency, accuracy, and convergence). In this paper, we described a new lossy compression scheme for distributed training of machine learning models that reduces network traffic by up to 107× without impairing training or altering machine learning algorithms. The key contribution is a new traffic compression scheme that combines the strengths of tensor quantization and sparsification approaches. 3LC introduces three new lightweight, yet effective lossy and lossless transformation techniques, resulting in greater balance between traffic compression, accuracy, computation overhead, and generality in distributed training under a large range of available network bandwidths.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A system for large-scale machine learning. In *Proc. 12th USENIX OSDI*, Nov. 2016.

[2] A. F. Aji and K. Heafield. Sparse communication for distributed gradient descent. In *Proc. Empirical Methods in Natural Language Processing (EMNLP)*, 2017.

[3] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Advances in Neural Information Processing Systems 30*, 2017.

[4] I. Bello, B. Zoph, V. Vasudevan, and Q. V. Le. Neural optimizer search with reinforcement learning. In *Proc. ICML 2017*, 2017.

[5] I. Cano, M. Weimer, D. Mahajan, C. Curino, and G. M. Fumarola. Towards geo-distributed machine learning. Technical Report arXiv:1603.09035, arXiv, 2016.

[6] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting distributed synchronous SGD. In *International Conference on Learning Representations Workshop Track*, 2016.

[7] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: Building an efficient and scalable deep learning training system. In *Proc. 11th USENIX OSDI*, Oct. 2014.

[8] Nvidia cuda c programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Jan. 2018.

[9] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, 2016.

[10] European Commission. EU Commission and United States agree on new framework for transatlantic data flows: EU-US Privacy Shield. http://europa.eu/rapid/press-release_IP-16-216_en.htm, Feb. 2016.

[11] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. E. Karro, and D. Sculley. Google Vizier: A service for black-box optimization. In *Proc. KDD 2017*, 2017.

[12] Google. Snappy. https://github.com/google/snappy, 2017.

[13] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch SGD: training ImageNet in 1 hour. Technical Report arXiv:1706.02677, arXiv, 2017.

[14] S. Han, H. Mao, and W. J. Dally. Deep Compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *Proc. ICLR 2016*, 2016.

[15] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[16] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *NIPS*, 2013.

[17] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu. Gaia: Geo-distributed machine learning approaching LAN speeds. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

[18] Intel 64 and IA-32 architectures developer's manual. https://software.intel.com/en-us/articles/intel-sdm, 2017.

[19] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, July 2015.

[20] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, 2017.

[21] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon. Federated learning: Strategies for improving communication efficiency. Technical Report arXiv:1610.05492, arXiv, 2016.

[22] KPMG. Overview of China's Cybersecurity Law. https://home.kpmg.com/cn/en/home/insights/2017/02/overview-of-chinas-cybersecurity-law.html, Feb. 2017.

[23] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[24] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *Proc. 11th USENIX OSDI*, Oct. 2014.

[25] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally. Deep Gradient Compression: Reducing the communication bandwidth for distributed training. Technical Report arXiv:1712.01887, arXiv, 2017.

[26] Linux Traffic Control. http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html, 2017.

[27] I. Loshchilov and F. Hutter. SGDR: stochastic gradient descent with restarts. In *Proc. ICLR 2017*, 2017.

[28] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2017.

[29] A. Øland and B. Raj. Reducing communication overhead in distributed learning by an order of magnitude (almost). In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015.

[30] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.

[31] A. H. Robinson and C. Cherry. Results of a prototype television bandwidth compression scheme. In *Proc. IEEE*, 1967.

[32] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech DNNs. In *Proc. Interspeech 2014*, 2014.

[33] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[34] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *NIPS*, 2012.

[35] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[36] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[37] P. Watcharapichat, V. L. Morales, R. C. Fernandez, and P. Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, 2016.

[38] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC)*, 2015.

[39] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li. TernGrad: Ternary gradients to reduce communication in distributed deep learning. In *Proc. NIPS 2017*, 2017.

[40] P. Xie, J. K. Kim, Y. Zhou, Q. Ho, A. Kumar, Y. Yu, and E. Xing. Distributed machine learning via sufficient factor broadcasting. Technical Report arXiv:1409.5705, arXiv, 2014.

[41] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017.

[42] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. Technical Report arXiv:1707.07012, arXiv, 2017.