3LC: LIGHTWEIGHT AND EFFECTIVE TRAFFIC COMPRESSION FOR DISTRIBUTED MACHINE LEARNING

Hyeontaek Lim [1] **David G. Andersen** [2] **Michael Kaminsky** [3]

ABSTRACT

3LC is a lossy compression scheme for state change traffic in distributed machine learning (ML) that strikes a balance between multiple goals: traffic reduction, accuracy, computation overhead, and generality. It combines three techniques—*3-value quantization with sparsity multiplication*, *base-*3^5 *encoding*, and *zero-run encoding*—to leverage the strengths of quantization and sparsification techniques and avoid their drawbacks. 3LC achieves a data compression ratio of up to 39–107×, preserves the high test accuracy of trained models, and provides high compression speed. Distributed ML frameworks can use 3LC without modifications to existing ML algorithms. Our experiments show that 3LC reduces wall-clock training time of ResNet-110 for CIFAR-10 on a bandwidth-constrained 10-GPU cluster by up to 16–23× compared to TensorFlow's baseline design.

1 INTRODUCTION

Distributed machine learning (ML) harnesses the 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 the *accuracy* of a trained model and the *wall-clock training 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.

The goal of this paper is to apply systems and distributed ML insights to reduce substantially the amount of data that must be transmitted between workers during training. The utility of this approach is twofold. First, even modest reductions in bandwidth offer practical benefits, such as being able to use cheaper but slower interconnects that can help reduce costs even today. Second, we explore the extremes of reduced bandwidth use for distributed training: Understanding the achievable tradeoff between accuracy and bandwidth can help inform the design of future accelerators and clusters in the rapidly evolving ML ecosystem.

Training on local networks: Recent performance studies show that in-datacenter distributed training can demand more bandwidth than datacenter networks and GPU interconnects currently offer, identifying transmitting state changes as a major bottleneck in distributed ML (Strom, 2015; Alistarh et al., 2017; Wen et al., 2017; Zhang et al., 2017; Lin et al., 2018; Peng et al., 2018). Even high-bandwidth networks (e.g., 50 Gbps) can be insufficient for multi-tenant distributed training (Hazelwood et al., 2018).

Training on wide-area networks: Large-scale deployment of distributed ML often require the workers to communicate over a low-bandwidth wide-area network (WAN) for multi-datacenter distributed training or to conform to local laws that regulate transferring sensitive training data (e.g., personal photos) across regulatory borders (Vulimiri et al., 2015; Cano et al., 2016; European Commission, 2016; Hsieh et al., 2017; KPMG, 2017). Some data might be pinned to mobile devices (Konečnỳ et al., 2016; Hardy et al., 2017; McMahan et al., 2017), forcing distributed ML to use a slow and sometimes metered wireless network.

Following current hardware trends, the burden on communication will likely become heavier in future distributed ML. High-end GPUs' deep-learning performance measured in FLOPS (floating point operations per second) has increased by 25× between 2013 and 2018 (Nvidia Inside Pascal; Nvidia Tesla V100), while their interconnect bandwidth has increased by "only" 15× in the same period of time (Nvidia NVLink). With faster processors, distributed ML generates more state changes to transmit, but the network bandwidth may not grow fast enough to serve the increased state change traffic, aggravating distributed ML's network bottleneck problem.

[1]Google Brain, Mountain View, California, USA; work started while at Carnegie Mellon University [2]Computer Science Department, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA [3]Intel Labs, Pittsburgh, Pennsylvania, USA. Correspondence to: Hyeontaek Lim <hyeontaek@google.com>.

Proceedings of the 2nd SysML Conference, Palo Alto, CA, USA, 2019.

Communication reduction aims 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 (Ho et al., 2013; Li et al., 2014; Abadi et al., 2016; Hsieh et al., 2017). Unfortunately, existing schemes suffer one or more problems: They reduce network traffic only slightly, sacrifice the accuracy of the trained model, incur high computation overhead, and/or require modifications to existing ML algorithms.

We present 3LC (3-value lossy compression), a lightweight and efficient communication reduction scheme. 3LC balances between traffic reduction, accuracy, computation overhead, and generality, to provide a low-barrier 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., reduces traffic 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 lossy compression approaches commonly used for communication reduction: *Quantization* encodes state changes in low resolution, and *sparsification* sends only the likely important parts of state changes. We realize both mechanisms in a lightweight-yet-effective lossy compression scheme. We exploit domain knowledge on distributed ML that most state change values are close to zero: A quantization scheme that can represent zero and has very low resolution would produce many consecutive zero values, and such sparsity can be easily encoded by a simple and fast lossless compression technique.

3LC combines three 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. **Base-3^5 encoding** is a lossless transformation that folds each group of five 3-values into a single byte. **Zero-run encoding** is a lossless transformation that shortens consecutive runs of common bytes (groups of five zero values) that are abundant in base-3^5-encoded data.

Our empirical evaluation of 3LC and prior communication reduction techniques on our 10-GPU cluster shows that 3LC is more effective in reducing traffic while preserving high accuracy at low computation overhead. When training image classifiers based on ResNet-110 (He et al., 2015) for the CIFAR-10 dataset (Krizhevsky, 2009), 3LC reduces training time to reach similar test accuracy by up to 16–23× under severe network bandwidth constraints. To measure 3LC's practical performance gains over a strong baseline, we use a distributed training implementation on TensorFlow (Abadi et al., 2016; Yu et al., 2018) 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 (Chilimbi et al., 2014). Distributed ML reduces the total training time by parallelization.

The parameter server architecture is a common distributed training design (Ho et al., 2013; Chilimbi et al., 2014; Li et al., 2014; Cui et al., 2016). *Parameter servers*, or simply *servers*, store a part of the 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 a set of *tensors* (multidimensional arrays).

The workers train the model by repeatedly performing local computation and state change transmission via the servers. Each *training step* includes the following sub-steps: *Forward pass:* The workers evaluate a *loss 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.

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 (Valiant, 1990). 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") (Recht et al., 2011; Ho et al., 2013) make other workers waste computation resources, increasing training time.

To mitigate the straggler problem, researchers have capitalized upon the property that stochastic gradient descent

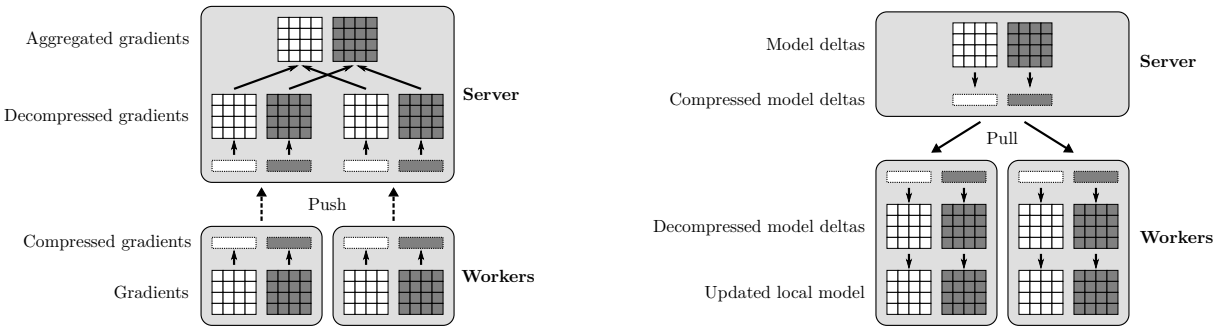

Figure 1. Point-to-point tensor compression for two example layers in 3LC (left: gradient push; right: model pull).

and its variants commonly used in distributed ML tolerate a small amount of inconsistency in the model across the workers (Recht et al., 2011). Asynchronous state change transmission permits a worker to submit an update based on a stale version of the model (Recht et al., 2011). Approaches such as stale synchronous parallel make a compromise between two extremes by limiting the staleness of the model for which an update is calculated (Ho et al., 2013).

A 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 (Recht et al., 2011; Ho et al., 2013; Li et al., 2014; Abadi et al., 2016; Hsieh et al., 2017). Thus, recent distributed ML frameworks often advocate synchronous state change transmission while using other techniques to mitigate stragglers. For instance, TensorFlow'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 (Chen et al., 2016).

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 (Zhang et al., 2017). 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 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 (Alistarh et al., 2017; Hsieh et al., 2017; Wen et al., 2017; Zhang et al., 2017; Lin et al., 2018). If the transmission takes excessive time, cluster nodes experience long stall time, harming the efficiency of distributed learning.

Quantization and sparsification reduce 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 discard the other information. *Quantization* uses low-resolution values to represent the approximate magnitude of the state change data (Seide et al., 2014; Alistarh et al., 2017; Wen et al., 2017; Wu et al., 2018). *Sparsification* discovers state changes with large magnitude and transmits a sparse version of tensors containing only these state changes (Li et al., 2014; Wei et al., 2015; Watcharapichat et al., 2016; Aji & Heafield, 2017; Hardy et al., 2017; Hsieh et al., 2017; Fang et al., 2018; Lin et al., 2018).

Note that the quantization and sparsification we discuss in this paper differ from model compression (Han et al., 2016; Jouppi et al., 2017; Lin et al., 2017; Venkatesh et al., 2017). *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 as 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 strike a balance between traffic reduction, accuracy, computation overhead, and generality.

3LC is a point-to-point tensor compression scheme. Figure 1 depicts how 3LC compresses, transmits, and decompresses state change tensors for two example layers. One compres-

sion 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 DNN.

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 (Wen et al., 2017) 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, they would perform redundant compression work. 3LC optimizes model pulls by sharing compression: Each server compresses model deltas to make a shared local copy that multiple workers may pull. Note that distributed ML frameworks that allow loosely synchronized local models on workers (Recht et al., 2011; Ho et al., 2013; Li et al., 2014; Hsieh et al., 2017) may require multiple copies of compressed model deltas, each of which is shared by a subset of the workers with the same local model.

3LC's tensor compression and decompression combines one lossy and two lossless transformations: 3-value quantization with sparsity multiplication (Section 3.1), base-3^5 encoding (Section 3.2), and zero-run encoding (Section 3.3).

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 and only a few state changes are far from zero (Wen et al., 2017). 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.

3-value quantization uses simple computation. For an input tensor T_{in}, the quantization output $T_{\text{quantized}}$ is computed by $\text{round}(T_{\text{in}}/m)$, where $m = \max(|T_{\text{in}}|) \cdot s$. The dequantization output T_{out} is $m \cdot T_{\text{quantized}}$.

s controls the compression level of 3LC. $s = 1$ is the default multiplier that preserves the maximum magnitude of values in the input tensor across quantization and dequantization. A larger s ($1 < s < 2$) makes the quantization output sparser (more zeros) because the magnitude of more values is smaller than $m/2$. The sparser output contains less state change information, but can be compressed more aggressively by zero-run encoding (Section 3.3).

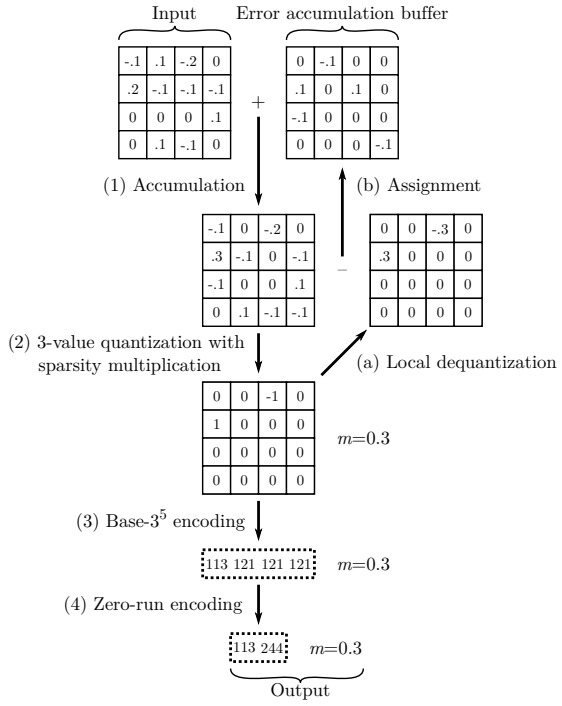

Figure 2. Tensor compression in 3LC.

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 (Seide et al., 2014; Wei et al., 2015; Watcharapichat et al., 2016; Aji & Heafield, 2017; Hsieh et al., 2017; Stich et al., 2018; Wu et al., 2018). It allows quantization errors to occur in the first place, but attempts to correct them in subsequent training steps. It keeps per-tensor local buffers to remember cumulative errors across training steps.

Figure 2 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.

3LC keeps a maximum absolute error smaller than the maximum magnitude of the input tensor at each training step. round() has a maximum absolute error of $1/2$. By the definition of $T_{\text{quantized}}$ and T_{out}, a maximum absolute error $\max(|T_{\text{in}} - T_{\text{out}}|) \leq m/2$. Because of the definition of m and $1 \leq s < 2$, $m/2 < \max(|T_{\text{in}}|)$.

The size of error accumulation buffers is proportional to the model size. The buffers can be pinned to less expensive host

memory and streamed to GPUs only when they are accessed. Compressing error accumulation buffers to further reduce memory costs is an interesting direction for future research.

Alternative quantization techniques: *Stochastic quantization* outputs randomized quantization values whose expectation matches their input value (Alistarh et al., 2017). It eliminates biases that exist in deterministic rounding. For instance, TernGrad (Wen et al., 2017) 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 and deterministic quantization instead of stochastic quantization for several reasons: (1) Biases from 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 4); designs using stochastic quantization require more bits for quantization (Alistarh et al., 2017) or additional accuracy-compensation techniques (Wen et al., 2017) for high accuracy. This result is consistent with prior work comparing error accumulation and stochastic quantization for gradient compression (Stich et al., 2018). (3) Using both error accumulation buffers and 3-value stochastic quantization caused training to fail to converge in our experiments. Combining them can require scaling accumulated errors by $0.1\times$ when adding the errors to the input tensor (Wu et al., 2018), which greatly limits the maximum amount of correctable errors per training step.

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 (Seide et al., 2014). In designing 3LC, we avoid reducing the magnitude of quantized values instead of pursuing minimum squared quantization errors because (1) minimizing quantization errors does not necessarily minimize accuracy loss in empirical evaluation (Section 4), and (2) other quantization techniques also preserve the approximate magnitude of input tensors for better empirical accuracy even though doing so may provide weaker theoretic guarantees (Alistarh et al., 2017; Wu et al., 2018).

Alternative sparsification techniques: The sparsity multiplier plays a role similar to the *threshold* knob in sparsification-based compression techniques (Hardy et al., 2017; Hsieh et al., 2017; Lin et al., 2018). 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); aggressive thresholding can result in lower accuracy, and compensating for it requires changing ML algorithms such as modified learning rate calculation (Hardy et al., 2017) or custom momentum calculation (Lin et al., 2018) that is specific to "allreduce"-style gradient-only traffic and 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.

3.2 Base-3^5 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 (Wen et al., 2017) is larger than the theoretic minimum of $\log_2 3 \approx 1.585$ by 26%.

Base-3^5 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 2 Step (3)], using *1.6 bits per 3-value*, which is only 0.95% higher than the theoretic bound. Base-3^5 encoding exploits the fact that an expression $a \cdot 3^4 + b \cdot 3^3 + c \cdot 3^2 + d \cdot 3 + e$ has only $3^5 = 243$ distinct values (≤ 256) if $a, \ldots, e \in \{0, 1, 2\}$. One way to perform base-3^5 encoding is as follows (decoding reverses encoding steps): (1) Element-wise add 1 to the 3-value quantized tensor. (2) Type cast it to an unsigned 8-bit integer tensor. (3) Flatten it into a 1-D array. (4) Pad it with zeros to make its length a multiple of 5. (5) Partition the array into 5 equal-sized arrays $P_0, \ldots, P_4$, and compute $P_0 \cdot 3^4 + P_1 \cdot 3^3 + P_2 \cdot 3^2 + P_3 \cdot 3 + P_4$.

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

3.3 Zero-run Encoding

The input to base-3^5 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 base-3^5 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 (Robinson & Cherry, 1967), but is specialized to base-3^5–encoded data. Observe that base-3^5 encoding maps a group of five zero values from the 3-value quantized tensor into a byte value `121`. Also recall that base-3^5 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 2 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)`.

Our experience with 3LC thus far is that applying zero-run encoding to multiple tensors in parallel suffices (e.g., from different layers or parallel towers in a DNN). We sketch below how a parallel version of zero-run encoding can operate on large tensors.

Parallel compression divides the input tensor into partitions, encoding each one with the serial zero-run encoding algorithm in parallel, and combines the compressed partitions by conditionally coalescing their boundaries. If the last byte of a compressed partition and the first byte of the subsequent compressed partition are byte values representing zeros (`121`) or a zero-run length (`243`–`255`), these two bytes can become a single byte if the total zero-run length is between 2 and 14. *Parallel decompression* simply partitions the compressed tensor, decodes each partition in parallel, and concatenates the decoded partitions.

Compared to general-purpose compression algorithms or entropy coding schemes (Gailly & Adler, 2013; Øland & Raj, 2015; Alistarh et al., 2017), zero-run encoding is simpler to vectorize and requires no preprocessing by avoiding unaligned bit-level operations and table lookups.

4 Evaluation

We experimentally evaluate 3LC to quantify its effectiveness against other communication reduction schemes. Our experiments investigate multiple aspects: traffic reduction, wall-clock training time reduction, test accuracy, convergence speed, and computation overhead.

4.1 Implementation

We implement a prototype of 3LC on TensorFlow. 3-value quantization with sparsity multiplication and base-3^5 encoding use TensorFlow's built-in vectorized operators. Zero-run encoding uses custom operators 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.

4.2 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 8-bit quantization with error accumulation. Our implementation uses 255 distinct values, $[-127, 127]$, leaving -128 unused. `Stoch 3-value b-3-5` uses a stochastic version of 3-value quantization similar to that of TernGrad (Wen et al., 2017) without "gradient clipping," and our base-3^5 encoding for 1.6-bit quantization, which is smaller than TernGrad's 2-bit quantization. `MQE 1-bit int` performs 1-bit quantization with minimum squared quantization errors and error feedback (Seide et al., 2014). `5% sparsification` chooses 5% of the largest state changes in each tensor and accumulates unsent changes in buffers for later transmission, which performs the common tensor sparsification (Li et al., 2014; Wei et al., 2015; Aji & Heafield, 2017; Hardy et al., 2017; Hsieh et al., 2017; Fang et al., 2018; Lin et al., 2018). Note that 5% corresponds to 1.6 (bits) / 32 (bits). We use the magnitude, not relative magnitude, of values to find largest changes for better test accuracy in our experiments. To avoid exhaustive sorting while finding a threshold, we only sort sampled input values (Aji & Heafield, 2017). Our prototype uses a bitmap to indicate which state changes sparsification has selected, which adds 1 bit per state change as traffic overhead regardless of the input tensor size. `2 local steps` transmits state changes every 2 local steps. Unsent updates are accumulated locally and sent at the next training step. It reduces the traffic almost by half and effectively doubles the global batch size of distributed training. `3LC` is the full 3LC design with empirically chosen sparsity multipliers: $s \in \{1.00, 1.50, 1.75, 1.90\}$.

Similar to prior work (Alistarh et al., 2017), we exclude state changes for small layers—in our experiments, batch normalization layers (Ioffe & Szegedy, 2015)—from compression because avoiding computation overhead far outweighs compacting already small tensors.

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. `Sparsification` does not use modified learning rate or momentum calculation (Hardy et al., 2017; Lin et al., 2018) because TensorFlow sends not only gradients, but also model deltas to which their modifications of ML algorithms are inapplicable directly.

4.3 Evaluation Setup

Workload: Our experiments train image classifiers based on ResNet-110 (He et al., 2015) for the CIFAR-10 dataset (Krizhevsky, 2009). CIFAR-10 contains 50,000 training images and 10,000 test images, each of which has one of 10 labels. ResNet-110 is a 110-layer convolutional neural network (CNN) for CIFAR-10.

Our evaluation focuses on CNN training because it is by far

the most common non-convex optimization problem used in recent work (Watcharapichat et al., 2016; Alistarh et al., 2017; Hardy et al., 2017; Hsieh et al., 2017; Wen et al., 2017; Zhang et al., 2017; Lin et al., 2018; Wu et al., 2018). In particular, we choose to train a ResNet, which 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 (Zoph et al., 2017). Compared to earlier neural network architectures such as VGG (Simonyan & Zisserman, 2014), ResNet models typically have small parameter count to computation ratios (Fang et al., 2018), 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 performance gains on the ResNet architecture are likely to transfer to other neural network architectures.

Training configuration: We reuse the local optimizer type and hyperparameters for ResNet-110 training from the original ResNet paper (He et al., 2015) except for the learning rate schedule. The local optimizer is stochastic gradient descent with a 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 (Loshchilov & Hutter, 2017) instead of the stepwise decay because the cosine decay achieves better accuracy and has fewer hyperparameters to tune. We apply the standard data augmentation that randomly crops and horizontally flips original images to generate training examples (He et al., 2015).

Our distributed training configuration follows the guideline for large-batch stochastic training (Goyal et al., 2017). We scale the learning rate proportionally to the worker count and make one worker responsible for updating batch normalization parameters. We use a per-worker batch size of 32 images; 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. Our accuracy matches or exceeds the accuracy of a ResNet-110 trained using a similar batch size but stepwise decay (Lin et al., 2018).

Hardware and network: Our distributed training runs on a GPU cluster. It uses 10 workers, each with a single 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 (linux-tc) on all 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 test images* as test accuracy.

To reduce computation resource use, 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 (He et al., 2015); 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 $t_{\text{full}} \cdot s_{\text{short}}/s_{\text{full}}$. 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 configuration is run 5 times for full measurement, and 3 times for accelerated measurement.

4.4 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)`, which performs aggressive traffic compression. Other designs require longer training or harm accuracy.

Table 1. Speedup over the baseline and test accuracy using standard training steps.

Design	Speedup (×) @ 10 Mbps	@ 100 Mbps	@ 1 Gbps	Accuracy (%)	Difference
`32-bit float`	1	1	1	93.37	
`8-bit int`	3.62	3.47	1.51	93.33	−0.04
`Stoch 3-value b-3-5`	12.3	7.51	1.53	92.06	−1.31
`MQE 1-bit int`	14.6	7.40	1.30	93.21	−0.16
`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

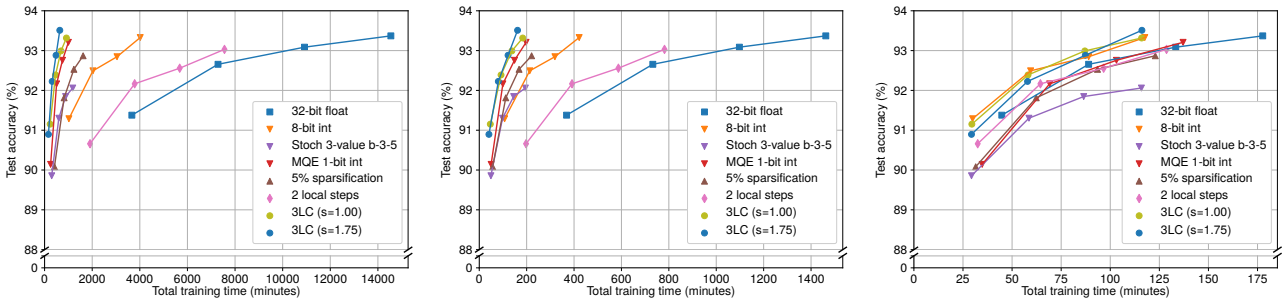

Figure 3. Training time and test accuracy with a varied number of training steps (left: 10 Mbps; center: 100 Mbps; right: 1 Gbps).

Figure 3 (left) plots total training time and final 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 b-3-5` 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, Figure 3 (center) shows that the benefit of reducing traffic begins to diminish and preserving high accuracy becomes important. For example, `5% sparsification` now provides always better time-to-accuracy than `Stoch 3-value b-3-5`.

On a 1 Gbps network, maintaining high accuracy and low computation overhead becomes even more important. Figure 3 (right) shows the time vs. accuracy curves using this network. When training to high accuracy, 3LC equals the accuracy of standard training using less time than other designs. For runs that do not aim to achieve the highest accuracy (and, thus, produce results faster), `8-bit int` achieves slightly better accuracy-per-time, though 3LC is a close second. In contrast, `MQE 1-bit int` is slower in time-to-accuracy than `8-bit int`, despite using much less network bandwidth; the long training time of `MQE 1-bit int` comes from its computation overhead of selective averaging for error minimization. By leveraging efficient vectorized operations, 3LC does not add such high overhead.

We also examine how designs perform during a training run in detail. Figure 4 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.

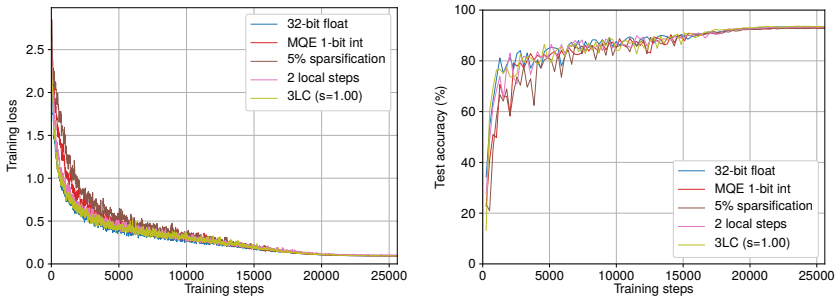

Figure 4. Training loss (left) and test accuracy (right) using standard training steps.

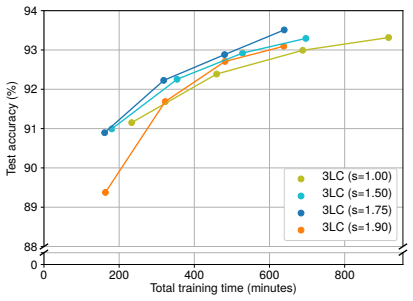

Figure 5. Training time and test accuracy with a varied sparsity multiplier (s).

Table 2. Average traffic compression of 3LC using standard training steps with a varied sparsity multiplier (s).

s	Comp. ratio	Avg. bits
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

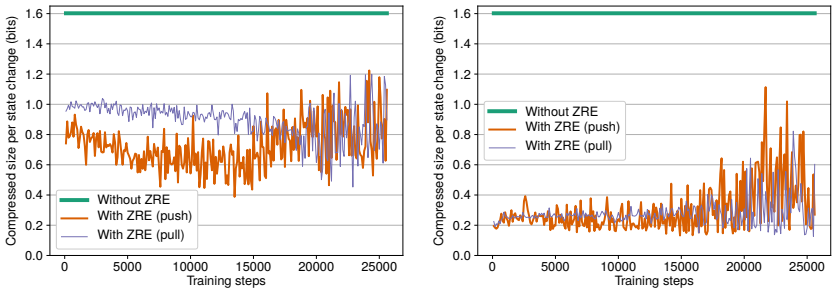

Figure 6. Compressed size per state change value (left: s=1.00; right: s=1.75).

4.5 Sensitivity Analysis

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

Figure 5 compares tradeoffs between total training time and test accuracy using 25%, 50%, 75%, and 100% of standard training steps at 10 Mbps. In general, a high sparsity multiplier reduces training time, but it can also reduce 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.

Table 2 examines the average traffic reduction of 3LC. Without zero-run encoding ("No ZRE"), the base-3^5–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; $s = 1.90$ realizes a 160× end-to-end compression ratio and 0.2 bits per state change while it achieves lower test accuracy. This high compression ratio can be useful for metered and/or highly bandwidth-constrained network connections.

The compression ratio of zero-run encoding changes over time because the sparsity of gradients and model deltas affects zero-run encoding's effectiveness. Figure 6 plots the compressed size of gradient pushes and model pulls at 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 less sparsity (fewer zeros in the quantization output) because these changes reflect aggregated gradient pushes from multiple workers. After finishing approximately 70% of training (when runtime training loss becomes more stable), compressed pushes become larger, which shows that workers now generate less sparse gradients. 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.

5 Related Work

Quantization: 1-bit stochastic gradient descent (Seide et al., 2014) 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 achieves 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 provides a sparsity multiplier that can change its compres-

sion level. 3LC's quantization and encoding methods are easier to accelerate by using existing vectorized operations.

QSGD (Alistarh et al., 2017) and TernGrad (Wen et al., 2017) 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 (Wen et al., 2017) 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.

ECQ-SGD (Wu et al., 2018) combines quantization error accumulation and stochastic quantization. It scales the accumulated errors by $0.1\times$ for convergence, which permits less error correction per training step than 3LC provides.

Quantization methods often employ entropy coding schemes such as Huffman coding and Elias coding for compact binary representations (Øland & Raj, 2015; Alistarh et al., 2017). 3LC's zero-run encoding offers high compression ratios (up to $8\times$) by using byte-level operations and no lookup tables, which helps achieve low computation overhead.

Sparsification: The parameter server (Li et al., 2014) discusses filtering zero gradients for small-value model parameters. 3LC can compress both gradients and model deltas regardless of the magnitude of the model parameters.

Bösen (Wei et al., 2015) 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 (Hsieh et al., 2017), a global threshold (Aji & Heafield, 2017), per-tensor thresholds (Hardy et al., 2017; Fang et al., 2018; Lin et al., 2018), or round-robin selection (Watcharapichat et al., 2016) for low-overhead sparsification. Among these, Gaia (Hsieh et al., 2017) 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 (Aji & Heafield, 2017) and Deep Gradient Compression (Lin et al., 2018) achieve high compression by transmitting a very few gradients. Such aggressive gradient reduction, however, necessitates recovering accuracy by modifying machine learning algorithms (Hardy et al., 2017; Lin et al., 2018), which reduces their generality when compressing non-gradient state changes such as model pulls in parameter server architectures. 3LC tackles the traffic compression problem with a constraint of preserving existing machine learning algorithms to achieve better applicability.

Poseidon (Zhang et al., 2017) reduces network traffic by transmitting small "sufficient factors" that contain enough information to construct full gradient tensors for certain types of neural network layers (Xie et al., 2014). 3LC pursues a general tensor compression scheme that can compress gradients and model deltas for any type of layer.

Infrequent communication: Federated learning (Konečnỳ et al., 2016; McMahan et al., 2017) runs multiple training steps between state change transmission. Our experiments show that using infrequent transmission alone can lead to lower accuracy when using the same number of training steps. Combining 3LC and infrequent communication to maximize their benefits is future work.

Task and operation scheduling: Intelligent scheduling of distributed ML tasks and their internal operations can reduce contention at the network, help computation and communication overlap, and reduce expensive communication between devices (Zhang et al., 2017; Hazelwood et al., 2018; Mirhoseini et al., 2018). This approach is orthogonal to the traffic compression we discuss in this paper. We ensure that computation and communication overlap in our distributed training experiments so that we can examine practical benefits of 3LC beyond the current state of the art.

6 Conclusion

Achieving system balance is an everlasting challenge for computer systems. This paper investigated the balance between algorithmic efficiency (accuracy), computational efficiency, communication bandwidth, and applicability for machine learning (ML) systems. We described 3LC, a new lossy compression scheme for distributed training of ML models that can reduce wall-clock training time on bandwidth-constrained networks by up to $23\times$ without impairing training or changing ML algorithms. 3LC's key contributions arise from combining the strengths of tensor quantization and sparsification approaches together with customized encoding techniques that are both fast and effective. By offering a lower-communication operating point for distributed training, we hope to further the discussion of what hardware capabilities may be required for future balanced-system cluster and hardware designs for ML.

Acknowledgments

This work was supported by funding from National Science Foundation under Award IIS-1409802, Intel via the Intel Science and Technology Center for Visual Cloud Systems (ISTC-VCS), and Google. We thank Mike Burrows, Christopher Canel, Giulio Zhou, and anonymous reviewers for their invaluable feedback.

REFERENCES

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

Aji, A. F. and Heafield, K. Sparse communication for distributed gradient descent. In *Proc. EMNLP*, 2017.

Alistarh, D., Grubic, D., Li, J., Tomioka, R., and Vojnovic, M. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Proc. NIPS*, 2017.

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

Chen, J., Monga, R., Bengio, S., and Jozefowicz, R. Revisiting distributed synchronous SGD. In *Proc. ICLR Workshop Track*, 2016.

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

Cui, H., Zhang, H., Ganger, G. R., Gibbons, P. B., and Xing, E. P. GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proc. EuroSys*, 2016.

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, February 2016.

Fang, J., Fu, H., Yang, G., and Hsieh, C.-J. RedSync : Reducing synchronization traffic for distributed deep learning. Technical Report arXiv:1808.04357, arXiv, 2018.

Gailly, J.-L. and Adler, M. zlib. http://www.zlib.net/, 2013.

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

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

Hardy, C., Merrer, E. L., and Sericola, B. Distributed deep learning on edge-devices: Feasibility via adaptive compression. In *Proc. IEEE NCA*, 2017.

Hazelwood, K., Bird, S., Brooks, D., Chintala, S., Diril, U., Dzhulgakov, D., Fawzy, M., Jia, B., Jia, Y., Kalro, A., Law, J., Lee, K., Lu, J., Noordhuis, P., Smelyanskiy, M., Xiong, L., and Wang, X. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *Proc. IEEE HPCA*, 2018.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. Technical Report arXiv:1512.03385, arXiv, 2015.

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

Hsieh, K., Harlap, A., Vijaykumar, N., Konomis, D., Ganger, G. R., Gibbons, P. B., and Mutlu, O. Gaia: Geo-distributed machine learning approaching LAN speeds. In *Proc. USENIX NSDI*, 2017.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proc. ICML*, 2015.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-datacenter performance analysis of a tensor processing unit. In *Proc. ISCA*, 2017.

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

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

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

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

Lin, X., Zhao, C., and Pan, W. Towards accurate binary convolutional neural network. In *Proc. NIPS*, 2017.

Lin, Y., Han, S., Mao, H., Wang, Y., and Dally, B. Deep Gradient Compression: Reducing the communication bandwidth for distributed training. In *Proc. ICLR*, 2018.

linux-tc. Linux Traffic Control. http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html, 2017.

Loshchilov, I. and Hutter, F. SGDR: Stochastic gradient descent with warm restarts. In *Proc. ICLR*, 2017.

McMahan, H. B., Moore, E., Ramage, D., Hampson, S., and y Arcas, B. A. Communication-efficient learning of deep networks from decentralized data. In *Proc. AISTATS*, 2017.

Mirhoseini, A., Goldie, A., Pham, H., Steiner, B., Le, Q. V., and Dean, J. A hierarchical model for device placement. In *Proc. ICLR*, 2018.

Nvidia Inside Pascal. Inside Pascal: Nvidia's newest computing platform. https://devblogs.nvidia.com/inside-pascal/, April 2016.

Nvidia NVLink. Nvidia NVLink fabric. https://www.nvidia.com/en-us/data-center/nvlink/, April 2018.

Nvidia Tesla V100. Nvidia Tesla V100. https://www.nvidia.com/en-us/data-center/tesla-v100/, March 2018.

Øland, A. and Raj, B. Reducing communication overhead in distributed learning by an order of magnitude (almost). In *Proc. IEEE ICASSP*, 2015.

Peng, Y., Bao, Y., Chen, Y., Wu, C., and Guo, C. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *Proc. EuroSys*, 2018.

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

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

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

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. Technical Report arXiv:1409.1556, arXiv, 2014.

Stich, S. U., Cordonnier, J., and Jaggi, M. Sparsified SGD with memory. In *Proc. NeurIPS*, 2018.

Strom, N. Scalable distributed DNN training using commodity GPU cloud computing. In *Proc. INTERSPEECH*, 2015.

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

Venkatesh, G., Nurvitadhi, E., and Marr, D. Accelerating deep convolutional networks using low-precision and sparsity. In *Proc. IEEE ICASSP*, 2017.

Vulimiri, A., Curino, C., Godfrey, P. B., Jungblut, T., Padhye, J., and Varghese, G. Global analytics in the face of bandwidth and regulatory constraints. In *Proc. USENIX NSDI*, 2015.

Watcharapichat, P., Morales, V. L., Fernandez, R. C., and Pietzuch, P. Ako: Decentralised deep learning with partial gradient exchange. In *Proc. ACM SoCC*, 2016.

Wei, J., Dai, W., Qiao, A., Ho, Q., Cui, H., Ganger, G. R., Gibbons, P. B., Gibson, G. A., and Xing, E. P. Managed communication and consistency for fast data-parallel iterative analytics. In *Proc. ACM SoCC*, 2015.

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

Wu, J., Huang, W., Huang, J., and Zhang, T. Error compensated quantized SGD and its applications to large-scale distributed optimization. In *Proc. ICML*, 2018.

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

Yu, Y., Abadi, M., Barham, P., Brevdo, E., Burrows, M., Davis, A., Dean, J., Ghemawat, S., Harley, T., Hawkins, P., Isard, M., Kudlur, M., Monga, R., Murray, D., and Zheng, X. Dynamic control flow in large-scale machine learning. In *Proc. EuroSys*, 2018.

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

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