

# Synchronous Multi-GPU Deep Learning with Low-Precision Communication: An Experimental Study

**Author(s):**
Grubic, Demjan; Tam, Leo K.; Alistarh, Dan; Zhang, Ce

**Publication Date:**
2018

**Permanent Link:**

https://doi.org/10.3929/ethz-b-000319485 →

open proceedings

# Synchronous Multi-GPU Deep Learning with Low-Precision Communication: An Experimental Study

Demjan Grubic
Department of Computer Science
ETH Zurich
dgrubic@student.ethz.ch

Leo Tam
NVIDIA
leot@nvidia.com

Dan Alistarh
IST Austria
dan.alistarh@ist.ac.at

Ce Zhang
Department of Computer Science
ETH Zurich
ce.zhang@inf.ethz.ch

## ABSTRACT

Training deep learning models has received tremendous research interest recently. In particular, there has been intensive research on reducing the *communication cost* of training when using multiple computational devices, through reducing the precision of the underlying data representation. Naturally, such methods induce system trade-offs—lowering communication precision could decrease communication overheads and improve scalability; but, on the other hand, it can also reduce the accuracy of training. In this paper, we study this trade-off space, and ask: *Can low-precision communication consistently improve the end-to-end performance of training modern neural networks, with no accuracy loss?*

From the performance point of view, the answer to this question may appear deceptively easy: compressing communication through low precision should help when the ratio between communication and computation is high. However, this answer is less straightforward when we try to generalize this principle across various neural network architectures (e.g., AlexNet vs. ResNet), number of GPUs (e.g., 2 vs. 8 GPUs), machine configurations (e.g., EC2 instances vs. NVIDIA DGX-1), communication primitives (e.g., MPI vs. NCCL), and even different GPU architectures (e.g., Kepler vs. Pascal). Currently, it is not clear how a realistic realization of all these factors maps to the speed up provided by low-precision communication. In this paper, we conduct an empirical study to answer this question and report the insights.

## 1 INTRODUCTION

The system tradeoffs induced by training deep neural networks seem to be an endless discussion [9, 11, 15, 19, 24, 34, 46]. One reason for this sophistication is the level of diversity involved. At the algorithmic level, there are synchronous, asynchrhous, and hybrid approaches. At the workload level, different neural networks have different computation and data movement patterns. At the architecture level, different computation devices, such as CPUs, GPUs, and FPGAs offer sharply different tradeoffs. Making matters worse, these are not pure performance tradeoffs, which one could tackle with performance modeling. Instead, different point in the tradeoff space has different *accuracy* of the trained model, a property that is very difficult to predict, and for which has there currently exists little theoretical understanding [49].



**A Data Management Angle.** From a data management perspective, these trade-offs provide an opportunity to build a *automatic optimizers* for deep learning tasks. Just as with previous optimizers that our community has been building [6, 8], the understanding of the tradeoff is often the prerequisite. Given the limited current theoretical understanding of deep learning, such modeling is inevitably empirical for the immediate future. Therefore, we believe that there is timely necessity for a set of empirical, fair, comparison to reveal the tradeoffs behind building and optimizing deep learning systems.

In this paper, we present an in-depth empirical study which focuses on *a subspace* of the whole tradeoff, that is, the tradeoff introduced by *the precision of communication when training deep neural networks with a synchronous multi-GPUs system.*

**Low Precision Deep Learning.** An emerging topic in deep learning systems is lowering the precision of data representation throughout the whole system [5, 14, 18, 20, 23, 33, 36, 50, 51]. There has been recent success in representing *all* data movements involved in training a neural network with as little as 1-bit per dimension while still getting the same accuracy for *some, but not all, networks* [36]. However, none of these work depict a full tradeoff space — they either focus on extreme cases (e.g., 1-bit) with significant loss of accuracy and do not discuss the impact of adding more bits, or only focus on algorithmic aspects, without a well-implemented system. In this paper, we conduct an empirical study to understand the impact of lower precision of data representation both on the training accuracy and the speed.

**System Artefact.** A fair empirical study calls for a system that achieves optimized performance for each configuration in the above space. Surprisingly, such a system does not exist off-the-shelf: if we just take CNTK or TensorFlow, the performance of many configurations is not as optimized as they could be. Thus, we decide to start from CNTK but conduct intensive performance optimization to ensure the fairness of our study. We first developed a system prototype optimized for low-precision machine learning. In this paper, we report an array of system optimizations that leads to up to 3.5× speed up over CNTK, which allows us to understand the tradeoff as fair as possible.

**Summary of Contribution 1: Experimental Study.** We consider the task of training deep neural networks on a single system with multiple GPUs, in a setting where all these GPUs communicate in a synchronous manner [24, 34]. In this setting, our first contribution is a study of the impact of *varying the number of bits to communicate between these devices from 1bit to 32bits.* Our study contains the following axes in the tradeoff space:

(1) **Machine Learning Tasks/Datasets:** {Image, Speech}
(2) **Neural Network Architectures:** {AlexNet [27], VGG [40], ResNet [21], Inception [43], LSTM [22]}
(3) **Quantization Strategy:** {"1-bit SGD" [39], QSGD [5]}
(4) **Number of GPUs:** {1, 2, 4, 8, 16}
(5) **Type of Machines:** {Amazon EC2, NVIDIA DGX-1}
(6) **Programming Models:** {MPI, NCCL}
(7) **GPU Architectures:** {Kepler, Pascal}

The tradeoff space we studied contains a full cross products the above axes (whenever possible); for each configuration, we vary the precision of data communication and measure (1) end-to-end performance (seconds), (2) convergence speed (#iterations), (3) speed per iteration (seconds), and (4) accuracy (%).

To fully depict this tradeoff, our study used more than 1400 machine hours on Amazon EC2's recently released 16-GPU instance and more than 20 machine hours on NVIDIA DGX-1. These give us an overview of how recent hardware and software have impact on the importance of low precision communication. This is so far the most comprehensive study of the impact of low-precision communication to deep learning. The system insights we get go significantly beyond those of previous work.

**Summary of Contribution 2: Insights.** Our study not only reveals insights beyond those of previous work, but also provides the first comparable quantitative evaluation of the results scattered in previous work that was original evaluated on different platforms and settings. We now briefly summarize these insights.

#### *1. Does low-precision always hurt accuracy?*

***No***. Across all datasets and models that we evaluated, we found parameter settings under which low-precision variants are able to achieve the same accuracy as full precision. Specifically, on all networks we evaluated, when quantizing communication using the 1bitSGD algorithm, a low-precision system is able to achieve the same accuracy (within 0.2%) as a full precision run. Using QSGD usually requires slightly higher precision: using 4-bit gradients always preserves the same accuracy (within 0.1%), while the 8-bit variant always matches or even *improves* final accuracy (within 0.5%). We note a few caveats regarding this result:

(1) Different layers have different sensitivity to quantization: for convolutional neural networks, the convolutional layers require more bits than the fully connected layers.
(2) Quantizing too aggressively can lead to significant accuracy loss: we identify scenarios where 2bit QSGD can no longer train to state-of-the-art precision.

#### *2. Does low-precision always help performance?*

***Not always.*** The answer to this question depends on several factors. We postpone a detailed discussion to Section 5.2.

(1) The most surprising result regarding performance regards the impact of the communication primitives used. When we replace MPI with the NCCL, a restricted set of communication primitives optimized by NVIDIA for GPU-to-GPU communication, 32bit full precision becomes much faster than MPI. Consequently, the speedup we can obtain with low-precision communication is also limited. In fact, on most networks, the performance improvement we get when using 8GPUs and NCLL is almost negligible; only for one network (VGG), we only get up to 1.4× speedup.
(2) With slower MPI (which can be seen as as a proxy for a slower interconnect), the tradeoff becomes more significant. One key factor is the ratio between computation (time to calculate the gradient) and communication (time to communicate the gradient). For networks with a large number of parameters such as AlexNet, we observe up to 4× speedup using low-precision communication; For networks with small model, we observe almost no speedup.
(3) The number of GPUs also plays a role. Naturally, when the number of GPUs increases, the necessity of using low-precision communication increases.

#### *3. Is using extremely low precision ever helpful?*

***Rarely.*** We observe a trend of "diminishing returns" when lowering the precision of gradient to extreme cases such as a single bit, while such compression can lead to accuracy loss. Examining the time cost of a dataset iteration, even with MPI, using 1-bit rarely outperforms using 4-bit on our benchmarks. Through simulation, we can project a communication-to-computation regime where extreme low precision would have significant impact, but none of the existing networks fall into that regime.

#### *4. Have the current programming models unleashed the full potential of low-precision machine learning?*

***No.*** NCCL hardcodes the reduction semantics with 32-bit full precision, and only offers a limited set of binary operations. Hence, there are currently no easy ways to use the NCCL primitives to support low-precision reduction efficiently. We conduct a simulation which implies that a version of NCCL with low-precision support could lead to a low-precision system that is up to 1.4× faster than our current prototype. MPI is also cumbersome to use in the context of low precision, although its richer semantics allow us to implement efficient variants of quantized methods.

#### *5. Do we really need 16 GPUs on a single instance?*

***Rarely when training a single model.*** Amazon EC2 provides a single instance with 16 GPUs: `p2.16xlarge`, whose price price is 2× higher than a `p2.8xlarge` 8-GPU instance, and currently stands at $14/hour. We only find few scenarios where the speedup achieved with 16 GPUs versus 8 would justify the cost doubling.

**Limitations.** Our study has the following limitations.

The first is that our study assumes that the user has *zero error-tolerance*. That is, we focus on scenarios where the user wishes to obtain a model with the same accuracy as one trained at full precision, but wishes to save time (and money) on training by using quantized methods. If the user could tolerate higher errors (e.g., 10% accuracy loss), the results of our study might change, as we should be able to use more aggressive low-precision schemes.

The second limitation is that our study only considers the communication overhead when exchanging gradients across devices. Other potential benefits of using low-precision data representation is to speed up computation (each register can hold more numbers). We do not consider these, as the current GPU platform does not yet provide enough flexibility (e.g., 4-bit computation). A related limitation is that we only focus on GPU platforms. The current work is part of a larger ongoing study examining computation-to-communication trade-offs across CPU, GPU, XEON PHI, and FPGA; however, these other platforms are currently out of the scope of a single paper.

The third is that our study focuses on the speed of *training*, which is currently the most computationally-intensive aspect of neural networks. When the objective changes, e.g., energy efficiency, or speed of inference, the results might also change.

The fourth limitation is that we build our artefact starting from a specific platform, Microsoft CNTK. However, we we expect similar observations to hold in the context of other deep learning systems, such as TensorFlow. All these systems use the same computational substrate (NVIDIA CUDA and cuDNN libraries [10]), and the semantics of the `allreduce`-based gradient exchange are similar across all synchronous systems. However, we do not wade into an in-depth quantitative analysis to support this claim. In the future, we plan to extend our study to a variety of systems, but this is out of the scope of this single paper.

The last limitation is that our study makes inherent assumptions of how different hyperparameters are tuned, e.g., learning rate and batch size. Although we believe our protocol of hyperparameters is reasonable and fair, it is also true that we did not fully explore all possible combinations of these hyperparameters, as the cost would increase significantly: by a rough estimate, an exhaustive search of the hyperparameter space would cost more than $1M on EC2.

**Reproducibility.** All of our experiments can be reproduced using a virtual machine availabe on EC2: `i-0a7a0521e93c329d9`. Our variant of CNTK is available under github.com/ZipML/CNTK.

**Overview.** The rest of this paper is organized as follows. We introduce the background material in Section 2 and describe our system prototype in Section 3. We describe our experiment setup in Section 4 and analyze our results in Section 5. We provide more discussion in Section 5.4 and survey related work in Section 7.

## 2 PRELIMINARIES

We present background for our study. We first describe the stochastic gradient descent (SGD) algorithm and the synchronous parallel version of it which is the focus of our study. We then describe two low-precision SGD algorithms and discuss two different communication primitives based on MPI and NCCL.

### 2.1 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is the workhorse algorithm in training deep learning models. SGD is developed for a more general class of optimization problems: Let $f : \mathcal{R}^n \to \mathcal{R}$, it solves $\min_x f(x)$. The assumption of SGD is that we have access to the *stochastic gradients* of this function. We denote the stochastic gradient by $\widetilde{g}$ which satisfies $\mathsf{E}[\widetilde{g}(\boldsymbol{x})] = \nabla f(\boldsymbol{x})$. SGD will converge towards the minimum by iterating the following procedure

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \eta_t \widetilde{g}(\boldsymbol{x}_t),$$

where $\boldsymbol{x}_t$ is the current state of the model, and $\eta_t$, also called the *step-size* or *learning rate* is a hyperparameter.

In the machine learning setting, we are given i.i.d. data points $X_1, \ldots, X_m$ generated from an unknown distribution $D$, and a loss function $\ell(X, \theta)$, which measures the loss of the model $\theta$ at data point $X$. We wish to find a model $\theta^*$ which minimizes $f(\theta) = \mathsf{E}_{X\sim D}[\ell(X, \theta)]$, the expected loss to the data. Since for each $i$, the function $\nabla\ell(X_i, \theta)$ is a stochastic gradient for $f$, we can use SGD to find $\theta^*$. This captures neural network training.

**Synchronous Parallel SGD.** In this paper, we focus on synchronous *data-parallel* SGD, modeling multi-GPU systems in which each GPU has a complete copy of the model. GPUs proceed in synchronous steps, and communicate using direct messages. Each of the $K$ processors maintains a local copy of the model $\boldsymbol{x}$, of dimension $n$. The algorithm is described in Algorithm 1.

Each processor aggregates the value of $\boldsymbol{x}$, then obtains random gradient updates for each component of $\boldsymbol{x}$, then communicates these updates to all other nodes, and finally aggregates the received updates and applies them to its local model. We have added *encoding* and *decoding* steps for the gradients before and after send/receive in lines 1 and 7. In the following, we assume the above pattern. Whenever describing a communication-efficient variant of SGD, we only specify the *encode/decode* functions. Note that these encoding process is often *lossy* that only recover an *approximate* gradient.

**Data:** Stochastic gradients
**Data:** Local copy of the parameter vector $\boldsymbol{x}$
1 **for** *each iteration* $t$ **do**
2  Let $\widetilde{g}_t^i$ be an independent stochastic gradient ;
3  $\boxed{M^i \leftarrow \mathsf{Encode}(\widetilde{g}^i(\boldsymbol{x}))}$ //encode gradients ;
4  broadcast $M^i$ to all peers;
5  **for** *each peer* $\ell$ **do**
6  receive $M^\ell$ from peer $\ell$;
7  $\boxed{\widetilde{g}^\ell \leftarrow \mathsf{Decode}(M^\ell)}$ //decode gradients ;
8  **end**
9 **end**

**Algorithm 1:** Synchronous Data-Parallel SGD Algorithm.

**Data:** Gradient vector $\boldsymbol{v}$ to be quantized
**Data:** Error $\boldsymbol{\epsilon}$ from the previous round
1 $\boxed{\boldsymbol{v} \leftarrow \boldsymbol{v} + \boldsymbol{\epsilon}}$ //add error from previous round ;
2 **for** *each component* $i$ **do**
3  $q_i \leftarrow avg_+$ if $v_i \geq 0$, $avg_-$ otherwise;
4  $\epsilon_i \leftarrow v_i - q_i$;
5 **end**
6 **return** $q$

**Algorithm 2:** 1bitSGD procedure.

When the encoding and decoding steps are trivial (i.e., no encoding/decoding), we refer to this algorithm as *full-precision (parallel) SGD*. In this case, at each processor, if $\boldsymbol{x}_t$ was the value of $\boldsymbol{x}$ that the processors held before iteration $t$, then the updated value of $\boldsymbol{x}$ by the end of this iteration is $\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - (\eta_t/K)\sum_{\ell=1}^K \widetilde{g}^\ell(\boldsymbol{x}_t)$, where each $\widetilde{g}^\ell$ is a stochastic gradient. The speedup comes from the fact that all the updates are computed in parallel.

### 2.2 One-Bit Stochastic Gradient Descent

We describe the 1bitSGD quantization scheme, introduced by Seide et al. [39]. We denote the quantization function by $Q^{1b}(\boldsymbol{v})$, mapping a vector $\boldsymbol{v}$ to its quantized version. The procedure first splits the vector $\boldsymbol{v}$ into its positive and negative components, respectively, and computes the average over positive values, which we denote by $avg_+$, and the average over negative values, which we denote by $avg_-$. Then,

$$Q_i^{1b}(\boldsymbol{v}) = \begin{cases} avg_+ & \text{if } v_i \geq 0\ ; \\ avg_- & \text{otherwise.} \end{cases}$$

Simply put, the procedure replaces each component with the average corresponding to its respective sign. Critically, the procedure also maintains an error correction vector $\boldsymbol{\epsilon}$, associated to the model. In each iteration, the error from the previous iteration is *added* to the current value, before the value is quantized. This *error correction* step is critical to preserve accuracy [39]. The resulting algorithm works as Algorithm 2, which can be used as the Encode function of standard SGD (Algorithm 1).

## 2.3 SGD with Stochastic Quantization

We present the QSGD stochastic quantization scheme, following the description from the original paper [5]. We denote the quantization function by $Q_s(\boldsymbol{v})$, where $s \geq 1$ is a tuning parameter, corresponding to the number of "quantization levels". Intuitively, $s$ will define uniformly distributed levels between 0 and 1, to which each real value is quantized such that: 1) the value is preserved in expectation, and 2) minimal variance is introduced.

Given a non-zero vector $\boldsymbol{v} \in \mathcal{R}^n$, $Q_s(\boldsymbol{v})$ is defined as

$$Q_s(v_i) = \|\boldsymbol{v}\|_2 \cdot \mathrm{sgn} v_i \cdot \xi_i(\boldsymbol{v}, s)\ , \tag{1}$$

where $\xi_i(\boldsymbol{v}, s)$'s are independent random variables defined as follows. Let $0 \leq \ell < s$ be an integer such that $|v_i|/\|\boldsymbol{v}\|_2 \in [\ell/s, (\ell+1)/s]$. That is, $[\ell/s, (\ell+1)/s]$ is the quantization interval corresponding to $|v_i|/\|\boldsymbol{v}\|_2$. Then

$$\xi_i(\boldsymbol{v}, s) = \begin{cases} \ell/s & \text{with probability } 1 - p\left(\frac{|v_i|}{\|\boldsymbol{v}\|_2}, s\right); \\ (\ell+1)/s & \text{otherwise.} \end{cases}$$

Here, $p(a, s) = as - \ell$ for any $a \in [0, 1]$. If $\boldsymbol{v} = \boldsymbol{0}$, then we define $Q(\boldsymbol{v}, s) = \boldsymbol{0}$. The quantization distribution $\xi_i(\boldsymbol{v}, s)$ is defined to have minimal variance over distributions with support $\{0, 1/s, \ldots, 1\}$. Its expectation satisfies $\mathsf{E}[Q_s(v_i)] = v_i$. The quantizer is an unbiased estimator of the original gradient, thus ensures convergence [5, 14].

The above algorithm assumes quantization levels are uniformly distributed. There are algorithms in which quantization levels are distributed to further minimize variance, and they can, in some cases, significantly improve accuracy for model compression [50]. We implemented this for gradient but does not observe significant improvement.

## 2.4 Communication Primitives

The major communication bottleneck in the parallel SGD algorithm is in line 4 of Algorithm 1, where the gradient is broadcast to all other machines. Since the operation just needs to collectively add all gradients and produce the output at each node, this can be implemented using a standard instance of the `allreduce` operator, the optimized version of which has been studied for decades [32]. This operation can be implemented differently in standard CNTK: via a reduce-and-broadcast pattern implemented on top of MPI, which is the default in CNTK, and via NVIDIA's NCCL extensions, which provide an allreduce-sum operation implementation.

*2.4.1 MPI Reduce-and-Broadcast.* The first aggregation algorithm is an MPI implementation of the classic reduce-and-broadcast pattern. More precisely, given a set of $K$ nodes, the model of dimension $n$ is split into $n/K$ consecutive ranges, where the $i$th processor is logically assigned range $i\lfloor n/K \rfloor$. At the end of each processing batch, each processor has a full set of gradients, and the goal of the procedure is to aggregate (sum) all these gradients so that each processor has the same global gradient value at the end of the procedure.

The first step in the aggregation process is that a processor sends each range in its current gradient to the corresponding processor. Thus, each processor sums up the values for its assigned range locally. In the final step, each processor broadcasts its aggregated range to all other processors. Note that, in the actual implementation, the gradients may be transmitted in quantized form. (That is, we add quantization and un-quantization steps before communication, and after the message is received.) This does not affect the pattern.

*2.4.2 NCCL Accumulation.* NCCL [7] (pronounced "Nickel") is an open-source communication library provided by NVIDIA, that is designed to provide efficient multi-GPU collective operations in a topology-aware way. In particular, NCCL provides a sum collective operation, which can be used to implement gradient aggregation. Internally, NCCL works by building a system-aware communication efficient ring topology, on top of which the collective operation is applied in a step-by-step manner. To minimize the memory impact of storing multiple copies, NCCL splits large data buffers into small slices (4-16KB), and uses efficient peer-to-peer access to push data between GPUs. NCCL currently supports several collective types (e.g., all-gather, all-reduce, broadcast) and a few binary operations (e.g., sum, product).

# 3 SYSTEM DESCRIPTION

We implemented our system on top of the Microsoft Cognitive Toolkit (CNTK) [3], version 2.0 beta 9. Our code is released both as open-source and as a docker instance at github.com/ZipML/CNTK. We first provide a brief technical overview of CNTK, and then focus on the additions brought by our artefact.

## 3.1 CNTK

The Microsoft Cognitive Toolkit (CNTK) is a computational platform optimized for deep learning. One general principle behind CNTK is that neural network operations are described by a directed computation graph, in which leaf nodes represent input values or network parameters, and internal nodes represent matrix operations on their children. CNTK supports and implements several popular network architecture types, such as feed-forward DNNs, convolutional nets (CNNs), and recurrent networks (RNNs/LSTMs). To train such networks, CNTK implements stochastic gradient descent (SGD) with automatic differentiation. CNTK supports parallelization across multiple GPUs and servers, with efficient MPI-based communication.

CNTK provides MPI-based GPU-to-GPU communication, and implements a CUDA-optimized version of the 1bit-SGD algorithm [39], as presented in Section 2.2. CNTK is optimized for usage with multi-server multi-GPU systems. CNTK exports APIs for C++, Python and C#/.NET. Additionally, CNTK specifies and implements a BrainScript scripting language, which can be used to define models such as neural networks, as well as to train and evaluate models without employing any lower-level programming languages. CNTK comes with many preinstalled scripts for training state-of-the art models for different tasks, of which one of the most documented is image classification. Futhermore, it offers a number of pre-trained state-of-the-art models.

## 3.2 Low-Precision Support

CNTK implements 1bitSGD, which is used by default in multi-GPU environments with SGD. We now describe CNTK's 1bitSGD implementation. Low-precision commununication between GPUs is implemented using MPI reduce-and-broadcast.

*3.2.1 Data Representation and 1bitSGD.* CNTK stores the computation intensive data (model and minibatch samples) in the GPU memory, to ensure fast access and to avoid expensive data copying between host (main) and device (GPU) memory. In synchronized SGD, after every GPU is finished with the computation of gradients using backpropagation for its batch of samples, GPUs collectively aggregate gradients, so that each is left with a coherent copy of the model.

Gradients are stored using a matrix datatype in GPU memory. Aggregation is performed using the MPI Reduce-and-Broadcast technique described above. Sending gradient matrices is done separately for each gradient. To reduce communication, each GPU performs quantization and un-quantization on each message sent/received.

Quantization is done over columns. That is, the gradient is divided into columns, and each column is quantized separately. Thus, each quantized column is represented by two numbers $avg_+$ and $avg_-$, as well as an array of bits of size equal to number of elements in the column. That array of bits is represented as an integer array, where 8 quantized values, each consisting of 1 bit, are packed into 1 byte of memory. For instance, if there are $n$ rows in a gradient matrix, then the quantized column is represented by two floating-point numbers and $\lceil n/8 \rceil$ bytes of memory.

In order for GPU memory to be aligned, if single-precision float-point numbers are used for calculations in a network, then $avg_+$ and $avg_-$ are represented as a float data type, and $n$ bits are packed in $\lceil n/32 \rceil$ C++ unsigned integers. If double-precision float-point numbers are used, $avg_+$ and $avg_-$ are doubles and $n$ bits are packed inside $\lceil n/64 \rceil$ C++ unsigned long long integers.

The quantization of a column is divided into 2 phases. The first phase launches GPU threads in order to calculate the numbers $avg_+$ and $avg_-$, where the number of threads is tuned for performance. The second phase uses calculated numbers $avg_+$ and $avg_-$ to quantize values and pack bits into integers. For each integer in the quantized array, a new GPU thread is launched, which means that each thread in second phase quantizes exactly 32 or 64 elements of a column, depending on the precision employed. All of a gradient's columns are quantized in parallel on the GPU. To optimize the performance, the *double buffering* techique is used, by which, while some gradients are being quantized, gradients that are finished with quantization are already being sent. That way communication overlaps with computation and GPU and CPU resources are maximally used. The current CNTK implementation uses MPI to exchange data between GPUs. Because of that, an additional transfer of the gradient between GPU device memory and host memory is required.

*3.2.2 QSGD Implementation.* In order to implement the QSGD quantization technique, we started from the 1bitSGD implementation. We re-wrote the quantization function, such that it follows the algorithm in Section 2.3. As in a case of 1bitSGD, we pack quantizated values in integers, but we stored only one additional floating point number (for scaling) instead of two. The algorithm works such that the number of bits used for quantization is specified as $gBits$, and each gradient matrix values are quantize into an integer range with a specified number of bits. We implemented two different quantization methods, which differ in terms of the distribution of quantization levels. The first method follows faithfully the described algorithm: we use one out of $gBits$ bits to store the sign, and the rest of the bits is used for the quantized value. A second approach was to divide the interval $[-\|\boldsymbol{v}\|_2, \|\boldsymbol{v}\|_2]$ into $2^{gBits} - 1$ intervals of equal size, where the quantization levels are the endpoints of those intervals.

Since the dimensions of the gradient matrix could be large, and the variance introduced by quantization depends on the dimension, we implemented a variant which splits the vector into *buckets* of consecutive scalar values, where each bucket is quantized independently. The bucket size will be tuned for accuracy.

| Dataset | # Training samples | # Validation samples | Size | # classes | Task |
|---|---|---|---|---|---|
| ImageNet | 1.3M | 50k | 145GB | 1000 | Image |
| CIFAR-10 | 50k | 10k | 1GB | 100 | Image |
| AN4 | 948 | 130 | 64MB | NA | Speech |

**Figure 1: Statistics of datasets.**

| | Instance | # CPU cores | GPU | TFLOPS (single) | $/hour |
|---|---|---|---|---|---|
| Amazon | p2.xlarge | 4 | $1 \times$ K80 | $1 \times 8.73$ | $0.9 |
| | p2.8xlarge | 32 | $8 \times$ K80 | $8 \times 8.73$ | $7.2 |
| | p2.16xlarge | 64 | $16 \times$ K80 | $16 \times 8.73$ | $14.4 |
| DGX1 | | $2 \times 16$ | $8 \times$ P100 | $8 \times 10.6$ | $50 (Nimbix) |

**Figure 2: Statistics of machines**

Importantly, the whole matrix is reshaped such that each column has a specified number of elements, where we always try to place consequtive elements from the same column in original matrix in the same bucket. Using this bucketing approach, we are able to control quantization variance and get significant accuracy improvements.

We also implemented possibility to specify how the scaling factor for a vector is done. Currently, we support normalization by 2-norm and maximal element of a vector (infinity norm). The former is useful if we wish to obtain sparse quantized vectors, while the latter introduces smaller variance. In our experiments, normalizing by the maximal element gave better results in terms of accuracy, since more information is preserved. We used the NVIDIA cuRAND library with a different seed for each thread to generate random numbers.

Additional improvement was not to quantize matrices with small number of elements compared to total number of learning parameters in the model, since for those matrices we lose time by setting GPU threads and quantizing. We use full precision pipeline to send those matrices. We choose a threshold for small matrices in such way so we always quantize more than 99% of all parameters.

**Reshaped 1bitSGD.** We noted that some of the design choices made in the CNTK implementation of 1BitSGD can adversely affect the performance of this algorithm. For objects without dynamic dimensions, the first tensor dimension is the "row," while the other dimensions are flattened onto "columns." The CNTK implementation of 1BitSGD always quantizes *per column*.

Practically, on networks with many convolutional layers, this can lead to the following performance artefact: quantization is often applied to a column of very small dimension (1–3), which is computationally expensive, and leads to practically no communication reduction. (At the same time, this artefact leads to practically no accuracy loss for this version of the algorithm, since the gradients are almost unchanged.) Given this artefact, the standard implementation of 1BitSGD can be slower than even the full-precision version on heavily convolutional networks such as ResNet and Inception.

We correct this issue by always reshaping tensors with a reshaping technique applied in QSGD. We observe up to 4× speedup compared with the original CNTK implementation. In all experiments we will denote this version of 1bitSGD with 1bitSGD*.

# 4 EXPERIMENTAL SETUP

## 4.1 DataSets

We test various quantization techniques on two types of tasks: image classification, and automated speech recognition.

| Task | Network | Dataset | Params. | # epochs to run | Initial Learning Rate |
|---|---|---|---|---|---|
| Image | AlexNet | ImageNet | 62M | 112 | 0.07 |
| | BN-Inception | ImageNet | 11M | 300 | 3.6 |
| | ResNet50 | ImageNet | 25M | 120 | 1 |
| | ResNet110 | CIFAR-10 | 1M | 160 | 0.1 |
| | ResNet152 | ImageNet | 60M | 120 | 1 |
| | VGG19 | ImageNet | 143M | 80 | 0.1 |
| Speech | LSTM | AN4 | 13M | 20 | 0.5 |

**Figure 3: Statistics of networks.**

| | 1 GPU | 2 GPUs | 4 GPUs | 8 GPUs | 16 GPUs |
|---|---|---|---|---|---|
| AlexNet | 256 | 256 | 256 | 256 | 256 |
| BN-Inception | 64 | 128 | 256 | 256 | 256 |
| VGG19 | 32 | 64 | 128 | 128 | 128 |
| ResNet50 | 32 | 64 | 128 | 256 | 256 |
| ResNet152 | 16 | 32 | 64 | 128 | 256 |
| ResNet110 | 128 | 128 | 128 | 128 | 128 |
| LSTM | 16 | 16 | NA | NA | NA |

**Figure 4: Batch sizes used for each network and # GPUs. See Section 4.4 for the tuning protocol that results in these choices.**

**ImageNet.** The ILSRVC 2015 dataset [37] (ImageNet) consists of 1.3 million images, each labelled into one of 1000 categories (classes), which cover a wide variety of objects, animals, scenes, or geometric concepts. To evaluate our models, we consider a standard scenario where validation set that consists of 100K images is used. The classifier has to predict either a single label (top-1) or five labels (top-5); an image is considered to be correct if at least one of the outputs matches the ground truth.

**CIFAR-10.** We also consider the small-scale CIFAR-10 object classification dataset [26]. The training set consists of 50,000 32×32 images. Images are labelled into 10 categories. The classifier has to output a single label (top-1); an image is considered to be correct if the prediction matches the ground truth.

**AN4.** For speech recognition, we use the CMU Alphanumeric (AN4) dataset [2]. AN4 consists of recordings of subjects spelling out their census data (name, address), as well as randomly generated sequences. It contains 948 training utterances, and 130 test utterances, sampled at 16 kHz, with 16-bit linear sampling.

## 4.2 Networks

We conducted our experiments on state-of-the-art networks for image classification and speech recognition.

For image classification we tested all the winning entries from the ImageNet competition from 2012 to 2015: AlexNet, VGG, BN-Inception, and ResNet. These architectures are state-of-the-art for this task. For experiments with VGG we used an instance with 19 layers, called VGG19. We performed experiments on the ImageNet dataset using ResNet networks with 50 and 152 layers (called ResNet50 and ResNet152), and used an instance of ResNet110 for the CIFAR10 dataset. All of these networks have different properties and different types of layers. BN-Inception is optimized for low number of parameters in network, whereas ResNet is built entirely of convolutional layers. For automated speech recognition we used a network that consists of 3 long short-term memory (LSTM) components. In all experiments we used architectures built and optimized by CNTK. More details about each network are in Figure 3.

## 4.3 Machines

For experiments, we used Amazon AWS EC2 P2 instances, designed for high-performance computing using GPUs. Instances we used are p2.xlarge with 1 GPU, p2.x8large with 8 GPUs on a single machine and p2.x16large with 16 GPUs on a single machine. The above instances have Nvidia Tesla K80 GPUs, based on the Kepler architecture. These GPUs support GPUDirect, which enables peer-to-peer GPU communication without using CPU.

Another machine we used is DGX1 that consists of 8×P100 GPUs. These GPUs are based on Pascal architecture and have NVlink high-bandwidth interconnection between GPUs. This results in highly-optimized GPU communication. The DGX machine also benefits from a customized interconnect, which provides high throughput and low latency.

## 4.4 Tuning and Experimental Protocol

We used nvidia-docker to conduct our experiments. One complicating factor of benchmarking deep learning systems is a large set of hyperparameters one needs to tune to conduct fair comparison. In this paper, we use the following tuning protocol.

**Batch Size.** The principle behind our hyperparameters tuning protocol is to always start from CNTK's default hyperparameters, which have already been optimized by the CNTK developers for a specific network. For each configuration that we run, if CNTK does not run with the default hyperparameter setting, we vary the hyperparameter until it runs. For example, with the default batch size and 1GPU, CNTK cannot run ResNet because the whole batch does not fit in memory. In this case, we decrease the batch size until CNTK runs. Figure 4 shows the batch sizes we used.

**Learning Rate.** We always use the default learning rate, which is fine tuned by CNTK for 32-bit full precision. We use an SGD optimizer with default momentum (0.9 for most architectures). Note that, because we focus on synchronous communication, the number of GPUs has negligible impact on the convergence rate and final quality given a fixed value of the learning rate and momentum. This has been observed previously, in e.g. [24].

**Bucket Size.** QSGD and 1bitSGD with reshaped matrices introduce new hyperparameters, such as bucket size and normalization scaling factor. We picked these parameters that optimize for accuracy on these networks. We used QSGD 2bit with bucket size 128, QSGD 4bit and 8bit with bucket sizes 512, and QSGD 16bit with bucket size 8192. Also, for our version of 1bitSGD with reshaped matrices we tried multiple bucket sizes and used bucket size 64 for all performance and scalability numbers.

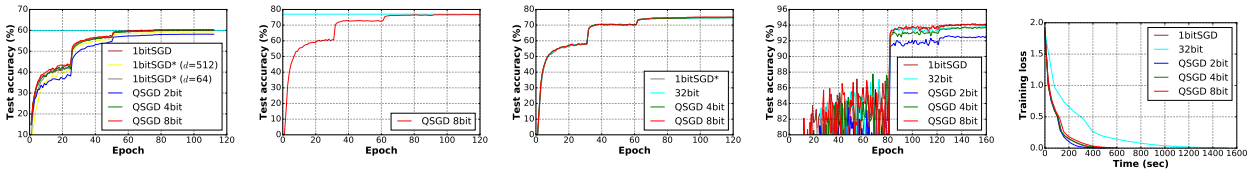
**NCCL Simulation.** CNTK uses NCCL only for full-precision data parallel SGD. When trying to twist it for low-precision, we find that its `sum` primitive only supports full precision and therefore, cannot be used for low-precision communication (as scaling factors cannot be taken into account). Thus, we implement a version where we simulated gradient aggregation with NCCL calls—we send the same number of bits using NCCL as we would if NCCL would have had support for low-precision. In the NCCL experiments, we use this simulated version to compare the performance of different low-precision setups.

# 5 ANALYSIS

We now analyze the experiment results and discuss the insights.

## 5.1 Accuracy

***Does low-precision always hurt accuracy?***




**(a) Test accuracy per epoch, for AlexNet ImageNet.**

**(b) Test accuracy per epoch, for ResNet152 on ImageNet.**

**(c) Test accuracy per epoch, for ResNet50 on ImageNet.**

**(d) Accuracy per epoch, for ResNet110 on CIFAR-10.**

**(e) Accuracy per time, for LSTM on AN4.**

**Figure 5: Accuracy results for various networks and datasets. Cyan = 32bit.**

In the first set of experiments, we study the impact of training deep neural networks with low precision communication on the training accuracy. For each network and data set, we vary the precision and report the test accuracy[1] for each epoch. Figure 5 illustrates the result for AlexNet, ResNet50 and ResNet152 on ImageNet, ResNet110 on CIFAR, and an LSTM network on AN4.

**Accuracy: Positives.** There always exist settings of parameters for which quantized variants converge to the same or better final accuracy, compared to the full-precision version. For ResNet50/ImageNet, the full precision variant converges to 59.90% top-5 accuracy, whereas 1bitSGD converges to 60.31% accuracy. Similarly, QSGD with 4bit and 8bit quantization converge to 60.37% and 60.05% final accuracy, respectively. The accuracy improvement, of up to 0.47% is statistically significant. The reason for this small, but statistically significant, accuracy gain is not clear. We suspect it is caused by similar reasons with recent work observing that adding noise during the training process can improve accuracy [35]. Additional experiments show that, for this dataset, quantization also improves *training loss* (see Figure 5e), which is perhaps surprising. It is not clear what is the definitive reason of this behavior.

**Accuracy: Negatives.** It is also important to note that quantizing too aggresively can lead to significant accuracy loss. For example, QSGD where gradient values are quantized to two bits (levels 0, 1, and −1) has accuracy loss of at least 1 percentage point, consistently across image classification datasets. We note that this is not the case for non-convolutional networks (LSTMs), which appear to be able to handle quantization to very low precision. This finding is consistent with the theory [5, 50]: aggressive quantization increases the variance of the convergence process, and as such, the theory predicts that one would have to run for more iterations in order to converge to the same quality results.

**Convergence Rate.** Another aspect of accuracy is the *convergence rate*, i.e., how many epoches do the system need to converge to the same accuracy. We observe that 8bit bit QSGD with 512 bucket size is always enough, on all data sets considered, to guarantee effectively the same convergence rate. For 1bitSGD, we note that the reshaped variant requires relatively small bucket size (64) to converge to the same target accuracy. Even so, we note that the convergence rate of 1bitSGD (i.e., training error versus epoch) is lower per epoch compared to that of the full precision variant, but the best accuracy achieved across at the end of the training process is comparable for the two processes.

[1]We also conduct similar experiments for training accuracy. We leave the result to the full version as it does not change any claims.

**Impact of Layer Types.** We also find *convolutional layers* are more "sensitive" to the noise induced by quantization. This phenomenon becomes apparent when we study the accuracy of two variants: (1) only quantize convolutional layers, and (2) quantize all layers. For example, on AlexNet, we can compare the accuracy of 1bitSGD with reshaping, which quantizes all layers, with the accuracy of standard 1bitSGD, which effectively does not quantize convolutional layers, as previously discussed. We observe that the reshaped variant has end accuracy that is lower; a closer examination of the loss-per-epoch graph shows that the reshaped version has consistently lower accuracy throughout the training process, although it almost catches up in terms of final accuracy across all epochs. The accuracy difference is starker (1 accuracy point) if we examine reshaped 1bitSGD with 512 bucket size.

**Impact of Bucket Size.** Another factor that has impact on the training accuracy is the bucket size of quantization, an additional parameter which we implemented for reshaped 1bitSGD and QSGD, and tuned for accuracy. For QSGD, this parameter can be used to directly throttle the added variance of the quantization process, at the cost of extra communication (since we need to send an extra floating-point number per each bucket). We observed that this can make a significant difference in terms of accuracy. For instance, on AlexNet / ImageNet, 4bit QSGD with 8192 bucket size has end accuracy that is $> 0.6\%$ inferior to the full-precision variant. Adjusting the bucket size to 512 allows it to *improve* accuracy over the full precision variant.

**Further Lowering Accuracy.** We also experimented with variants of QSGD with even lower communication overhead (such as 1-bit per location, or two-norm nomalization). These variants are theoretically justified, in that they will eventually converge to a local optimum, at the cost of additional running time. However, in our experiments, these variants did not provide good accuracy results when run for the same number of epochs as the full precision version. These accuracy results are therefore omitted.

**Discussion.** One final observation we make is the impressive accuracy of the 1bitSGD error-correction techniques. It is worth emphasizing that this technique only sends the *signs* of the components, plus two scaling factors. On large-scale image classification datasets, it only loses relatively small amounts of accuracy ($< 0.3\%$). It is interesting future work to develop the theoretically foundation of its convergence and correctness, which is still an open question [39].

## 5.2 Performance

***Does low-precision always help performance?***

We now study the impact of low-precision training on end-to-end training time. We measure the time per iteration for the

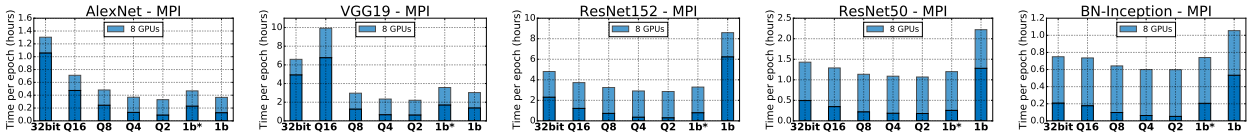



**Figure 6: Performance: Amazon EC2 Instance with MPI. QN = QSGD with $N$ bits. 1b = 1bitSGD, 1b* = 1bitSGD with reshaping.**

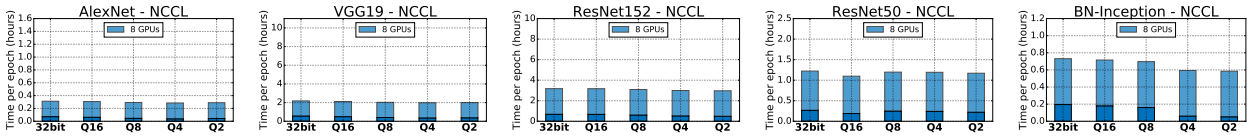



**Figure 7: Performance: Amazon EC2 Instance with NCCL. QN = QSGD with $N$ bits. 1b = 1bitSGD, 1b* = 1bitSGD with reshaping.**

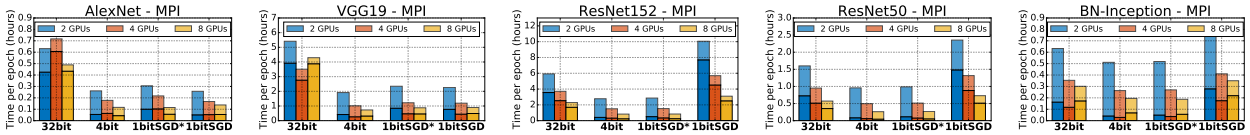



**Figure 8: Performance: NVIDIA DGX-1 with MPI.**

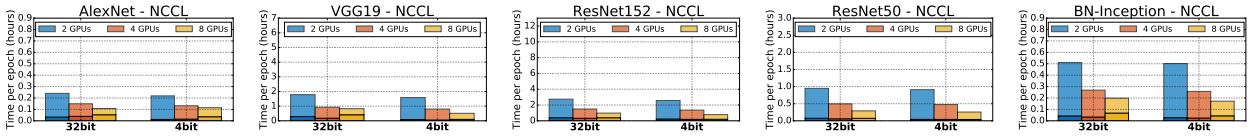



**Figure 9: Performance: NVIDIA DGX-1 with NCCL.**

full-precision version (32bit), the standard CNTK implementation of 1bitSGD, our variant with reshaping and 64 bucket size (whose parameters are chosen so as to guarantee no accuracy loss), and QSGD with 2, 4 and 8-bit precision. (As observed, 512 bucket size guarantees state-of-the-art accuracy across all networks for the QSGD 4bit and 8bit variants. Even lowering the bucket size down to 32 for 2bit QSGD does not recover the full precision accuracy.). We break down the epoch time into communication time (bottom of each bar), and computation time (top of each bar, which also includes time spent compressing and uncompressing gradients). Figure 6 and Figure 7 show the result on the 16GPU Amazon instance, and Figure 8 and Figure 9 show the results on the 8GPU NVIDIA DGX-1. (Recall that DGX-1 machine has newer Pascal GPUs, as well as a faster, custom interconnect.)

**Slow Inter-connections with Slow Primitives.** We start with a setting that *favors* low precision communication the most — Amazon instances have 16 GPUs and all communications are conducted over MPI/PCIe. Figure 6 illustrates the result of using MPI as the communication primitives. We see that, in this setting, using low-precision communication significantly improves the performance — with 8GPUs on VGG network, the speedup of using 2-bit / 4-bit precision is almost 3× compared with using 32-bit full precision. On 16GPUs, the speedup is of > 5×.

This speedup does not hold across all network types, as these networks have different ratio between communication and computation: there are communication-dominated networks (such as AlexNet and VGG), and computation-dominated networks (such as BN-Inception and ResNet50). ResNet152 balances these two.

**Impact on Communication Overhead.** We observe that on all networks, lowering the precision of communication significantly decreases the communication time. On AlexNet, the reduction in communication time with 4-bit quantization is almost 5×. For other networks, we observer similar speedups.

**Impact on End-to-end Performance.** As the computation time stays the same across different precision settings, the end-to-end performance speedup is smaller that our amount of savings we have in communication. For example, on AlexNet, the 5× speedup on communication results in 2× speedup overall. For computation-dominating network such as BN-Inception, we get only 1.3× speedup when lowering the precision by 16×.

**Extremely Low Precision.** This "diminishing returns" phenomenon implies that, even on machines with slow connection and slow communication, we rarely observe a case where using 1-2bit precision resulting in significant performance benefit as using 8-bits and 4-bits. In practice, this suggests that it may be reasonable to only lower the precision up to 8-bit gradients, since that going even lower does not really provide significant performance benefit on the model sizes we trained.

**Precision versus Bucket Size.** We observe that in many cases 1bitSGD is actually slower than 2bit QSGD. This is because, to preserve accuracy, we have to significantly decrease bucket size

| AlexNet ImageNet | | | | | | |
|---|---|---|---|---|---|---|
| Setup | | Samples per second (MPI) | | | | |
| Precision | Bucket size | 1 GPU | 2 GPUs | 4 GPUs | 8 GPUs | 16 GPUs |
| 32bit | / | 240.80 | 301.45 | 328.00 | 272.90 | 192.10 |
| QSGD 16bit | 8192 | / | 388.80 | 508.80 | 500.90 | 335.60 |
| QSGD 8bit | 512 | / | 424.90 | 544.60 | 739.10 | 535.00 |
| QSGD 4bit | 512 | / | 466.50 | 598.70 | 964.90 | 748.50 |
| QSGD 2bit | 128 | / | 449.20 | 609.15 | 1076.50 | 889.80 |
| 1bitSGD | / | / | 424.05 | 564.30 | 971.10 | 849.40 |
| 1bitSGD* | 64 | / | 370.80 | 476.50 | 761.20 | 712.70 |
| 1bitSGD* | 512 | / | / | / | / | / |

| ResNet50 ImageNet | | | | | | |
|---|---|---|---|---|---|---|
| Setup | | Samples per second (MPI) | | | | |
| Precision | Bucket size | 1 GPU | 2 GPUs | 4 GPUs | 8 GPUs | 16 GPUs |
| 32bit | / | 47.20 | 80.80 | 142.40 | 247.90 | 272.30 |
| QSGD 16bit | 8192 | / | 90.20 | 156.30 | 275.80 | 348.70 |
| QSGD 8bit | 512 | / | 92.60 | 162.70 | 313.70 | 416.80 |
| QSGD 4bit | 512 | / | 93.90 | 165.70 | 326.10 | 461.20 |
| QSGD 2bit | 128 | / | 93.30 | 178.35 | 330.45 | 472.25 |
| 1bitSGD | / | / | 45.10 | 81.70 | 160.15 | 155.20 |
| 1bitSGD* | 64 | / | 88.10 | 156.50 | 296.70 | 442.40 |

| ResNet110 CIFAR10 | | | | | | |
|---|---|---|---|---|---|---|
| Setup | | Samples per second (MPI) | | | | |
| Precision | Bucket size | 1 GPU | 2 GPUs | 4 GPUs | 8 GPUs | 16 GPUs |
| 32bit | / | 343.70 | 555.00 | 957.70 | 1229.10 | 831.60 |
| QSGD 16bit | 8192 | / | 551.00 | 942.70 | 1164.20 | 763.40 |
| QSGD 8bit | 512 | / | 550.20 | 960.10 | 1193.10 | 759.70 |
| QSGD 4bit | 512 | / | 571.10 | 957.40 | 1257.10 | 784.30 |
| QSGD 2bit | 128 | / | 557.20 | 973.10 | 1227.90 | 780.40 |
| 1bitSGD | / | / | 465.60 | 643.30 | 610.90 | 406.90 |
| 1bitSGD* | 64 | / | 550.40 | 884.80 | 1156.70 | 757.70 |

| ResNet152 ImageNet | | | | | | |
|---|---|---|---|---|---|---|
| Setup | | Samples per second (MPI) | | | | |
| Precision | Bucket size | 1 GPU | 2 GPUs | 4 GPUs | 8 GPUs | 16 GPUs |
| 32bit | / | 16.90 | 26.10 | 45.00 | 73.90 | 113.50 |
| QSGD 16bit | 8192 | / | 31.20 | 54.50 | 95.50 | 151.00 |
| QSGD 8bit | 512 | / | 32.80 | 62.70 | 109.20 | 182.50 |
| QSGD 4bit | 512 | / | 33.60 | 60.20 | 121.90 | 203.20 |
| QSGD 2bit | 128 | / | 33.50 | 64.35 | 123.55 | 208.50 |
| 1bitSGD | / | / | 10.55 | 22.10 | 41.40 | 63.15 |
| 1bitSGD* | 64 | / | 30.40 | 55.50 | 108.10 | 193.50 |

| VGG19 ImageNet | | | | | | |
|---|---|---|---|---|---|---|
| Setup | | Samples per second (MPI) | | | | |
| Precision | Bucket size | 1 GPU | 2 GPUs | 4 GPUs | 8 GPUs | 16 GPUs |
| 32bit | / | 12.40 | 20.40 | 36.30 | 53.95 | 40.60 |
| QSGD 16bit | 8192 | / | 24.80 | 46.40 | 35.80 | 67.80 |
| QSGD 8bit | 512 | / | 24.20 | 47.50 | 119.50 | 106.60 |
| QSGD 4bit | 512 | / | 27.00 | 52.30 | 151.65 | 143.80 |
| QSGD 2bit | 128 | / | 24.60 | 49.35 | 160.35 | 170.50 |
| 1bitSGD | / | / | 22.20 | 43.15 | 117.35 | 120.60 |
| 1bitSGD* | 64 | / | 22.90 | 44.80 | 99.15 | 134.30 |

| BN-Inception ImageNet | | | | | | |
|---|---|---|---|---|---|---|
| Setup | | Samples per second (MPI) | | | | |
| Precision | Bucket size | 1 GPU | 2 GPUs | 4 GPUs | 8 GPUs | 16 GPUs |
| 32bit | / | 88.30 | 164.80 | 316.75 | 473.75 | 500.40 |
| QSGD 16bit | 8192 | / | 171.80 | 337.10 | 482.70 | 592.30 |
| QSGD 8bit | 512 | / | 173.60 | 342.50 | 552.90 | 696.30 |
| QSGD 4bit | 512 | / | 174.80 | 346.90 | 593.40 | 743.30 |
| QSGD 2bit | 128 | / | 173.40 | 343.70 | 591.80 | 747.50 |
| 1bitSGD | / | / | 127.60 | 236.25 | 336.15 | 321.30 |
| 1bitSGD* | 64 | / | 170.30 | 335.10 | 480.50 | 700.40 |

**Figure 10: Speed of using MPI on Amazon EC2 instance.**

for the reshaped 1bitSGD algorithm. This renders it both more expensive in terms of communication, and increases the computational complexity of the GPU encoding/decoding operations.

**Slow Inter-connections with Fast Primitives.** Another way to implement the communication is to use NCCL instead of MPI. In our experiment, for the primitives we used in training, NCCL provides significantly faster communication. The reasons are two-fold: NCCL provides faster messaging, but also less memory overhead. Figure 7 shows the result of using NCCL on EC2.

**Implementation Notes.** We note the following caveats when using NCCL. First, NCCL does not currently support more than 8 GPUs, and we therefore only report numbers on up to 8 GPUs. Second, the current `allreduce` primitive in NCCL does not support low-precision and therefore all low-precision numbers of

| AlexNet ImageNet | | | | | |
|---|---|---|---|---|---|
| Setup | | Samples per second (NCCL) | | | |
| Precision | Bucket size | 1 GPU | 2 GPUs | 4 GPUs | 8 GPUs |
| 32bit | / | 240.80 | 458.20 | 625.00 | 1138.30 |
| QSGD 16bit | 8192 | / | 462.80 | 632.10 | 1157.60 |
| QSGD 8bit | 512 | / | 458.40 | 641.80 | 1214.80 |
| QSGD 4bit | 512 | / | 471.90 | 659.40 | 1247.70 |
| QSGD 2bit | 128 | / | 471.00 | 661.60 | 1229.70 |

| ResNet50 ImageNet | | | | | |
|---|---|---|---|---|---|
| Setup | | Samples per second (NCCL) | | | |
| Precision | Bucket size | 1 GPU | 2 GPUs | 4 GPUs | 8 GPUs |
| 32bit | / | 47.20 | 93.80 | 164.80 | 291.10 |
| QSGD 16bit | 8192 | / | 93.70 | 164.50 | 324.20 |
| QSGD 8bit | 512 | / | 94.00 | 165.80 | 297.40 |
| QSGD 4bit | 512 | / | 95.60 | 167.90 | 298.40 |
| QSGD 2bit | 128 | / | 95.50 | 168.20 | 304.10 |

| ResNet152 ImageNet | | | | | |
|---|---|---|---|---|---|
| Setup | | Samples per second (NCCL) | | | |
| Precision | Bucket size | 1 GPU | 2 GPUs | 4 GPUs | 8 GPUs |
| 32bit | / | 16.90 | 33.60 | 60.10 | 112.10 |
| QSGD 16bit | 8192 | / | 33.40 | 59.80 | 112.20 |
| QSGD 8bit | 512 | / | 33.70 | 60.80 | 115.10 |
| QSGD 4bit | 512 | / | 34.20 | 62.10 | 118.70 |
| QSGD 2bit | 128 | / | 34.30 | 62.20 | 119.90 |

| VGG19 ImageNet | | | | | |
|---|---|---|---|---|---|
| Setup | | Samples per second (NCCL) | | | |
| Precision | Bucket size | 1 GPU | 2 GPUs | 4 GPUs | 8 GPUs |
| 32bit | / | 12.40 | 24.90 | 48.70 | 163.10 |
| QSGD 16bit | 8192 | / | 24.90 | 49.10 | 168.00 |
| QSGD 8bit | 512 | / | 25.50 | 50.50 | 175.20 |
| QSGD 4bit | 512 | / | 25.60 | 51.00 | 179.50 |
| QSGD 2bit | 128 | / | 25.60 | 51.10 | 177.80 |

| BN-Inception ImageNet | | | | | |
|---|---|---|---|---|---|
| Setup | | Samples per second (NCCL) | | | |
| Precision | Bucket size | 1 GPU | 2 GPUs | 4 GPUs | 8 GPUs |
| 32bit | / | 88.30 | 175.30 | 342.00 | 486.70 |
| QSGD 16bit | 8192 | / | 174.30 | 342.70 | 497.10 |
| QSGD 8bit | 512 | / | 174.50 | 345.30 | 510.10 |
| QSGD 4bit | 512 | / | 178.60 | 349.00 | 598.90 |
| QSGD 2bit | 128 | / | 177.20 | 349.00 | 608.20 |

**Figure 11: Speed of using NCCL on Amazon EC2 instance.**

Figure 7 are simulated by using the NCCL `allreduce` primitive to send the same amount of data that would be sent in a low-precision implementation using NCCL. (The rest of the algorithm remains the same; in this case, the GPUs will converge at a lower rate or could diverge, but this is irrelevant for the experiment.)

**NCCL vs. MPI.** We draw the reader's attention to the performance difference between MPI (Figure 6) and NCCL (Figure 7). It is clear that the MPI implementation is slower than the NCCL — the computation time stays the same but the communication time differs significantly. One result we found especially surprising is that NCCL with full precision can be *faster* than MPI with low precision. Thus, on systems where NCCL is available, the fastest approach may be simply to run full precision with NCCL.

**Super-Linear Scaling.** The attentive reader may have noticed that, for the VGG19 network, the scaling is super-linear at 8GPUs. This phenomenon appears to be an artefact due to the minibatch size: at 8 GPUs, the batch for VGG19 consists of just 16 images, which the K80 GPUs are able to process in less than half of the time needed to process batches of 32 images. We were able to reproduce this behaviour on a single GPU with the same batch size, and we speculate that it may be due to better caching and less data movement at smaller sample size.

**End-to-end Performance.** Since NCCL could be modified to supports a low-precision `allreduce` in the future, we might be able to take advantage of low-precision communication to get even faster systems. As illustrated in Figure 7, on communication-dominated networks such as VGG, we get up to 1.5× speedup with 4-bit or 8-bit precision NCCL. This end-to-end speedup is

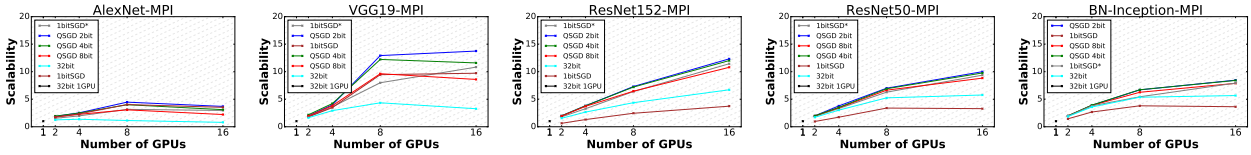



**Figure 12: Scalability: Amazon EC2 Instance with MPI.**

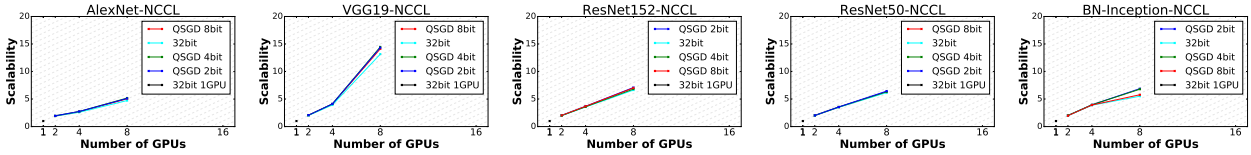



**Figure 13: Scalability: Amazon EC2 Instance with NCCL.**

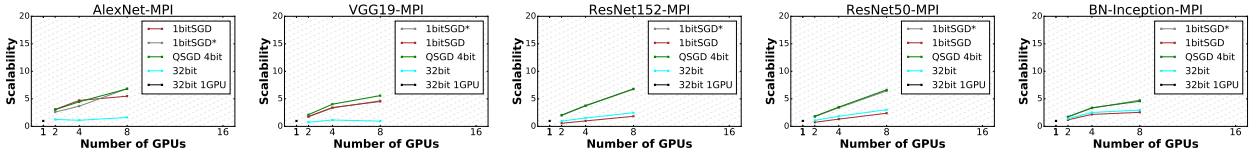



**Figure 14: Scalability: NVIDIA DGX-1 with MPI.**

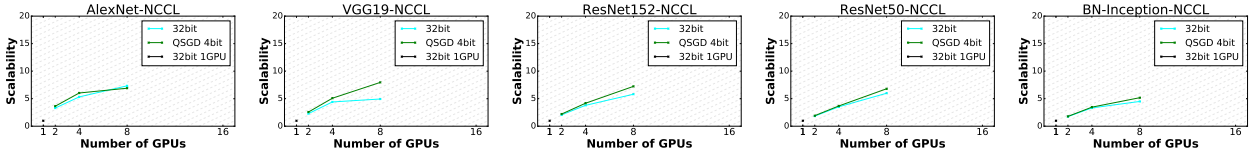



**Figure 15: Scalability: NVIDIA DGX-1 with NCCL.**

considerably lower than what we got in the MPI implementation, due to the balanced communication-to-computation ratio.

**Fast Interconnect with Slow/Fast Primitives.** On NVIDIA DGX-1, all GPUs are connected by a custom interconnect, which is significantly faster than PCIe. Further, the GPU is about 40% faster than in the Amazon instances. In this case, we observe similar results as the Amazon instances — when using slow primitives such as MPI, we get significant speedup by lowering the precision of communication, sometimes by up to 5× (VGG). However, with NCCL, the achievable speedup is again limited significantly. In fact, with NCCL, on VGG, we get up only 1.6× speedup when using 4-bits precision and relatively minor speedups for other networks and precision levels. It is interesting to note by examining the scalability graphs that the low precision brings the implementations close to linear scalability.

## 5.3 Scalability

***Do we really need 16 GPUs on a single instance?***

We now study scalability: how does the performance change with respect to the number GPUs we use? We define scalability as the number of samples per second in a certain configuration, divided by the number of samples per second processed by a single GPU for a training epoch. Figure 12-15 shows the result.

**Scalability of Full Precision.** We see that, using 32-bit full precision is able to scale, to some extent. For computation-dominated networks, 32-bit full precision is able to scale up to 6× with MPI and 7× with NCLL on 8 GPUs. However, for communication dominated networks, 32-bit suffers. For example, for AlexNet, 32-bit full precision with MPI only achieves 2× scale up with 16 GPUs. Comparing NCCL with MPI, NCCL scales up better than MPI for 32-bit full precision — This is not surprising, as NCCL is more efficient in dealing with communications.

**Impact of Low Precision Communication.** We see that using low-precision quantized communication consistently improves the scalability over 32-bit for all experiments. When using MPI on Amazon instances, the scalability for AlexNet is improved from < 3× to 8× by using 1bitSGD. Networks such as ResNet152 scale almost *linearly* once quantization is applied even with MPI: with quantization, scalability at 16GPUs is 2×, and 5× without.

When using NCCL, the difference between quantized communication and full precision decreases significantly. As shown in Figure 13 and Figure 15, quantization only introduces at most 50% scale up compared with full precision. The only exception is for VGG, which has the most number of parameters, in which 32-bit saturated the memory bandwidth. On the other hand, on AlexNet, we note that 32bit NCCL is *faster* than the 4bit variant. This explanation for this perhaps surprising result is the following: the computational overhead of AlexNet is relatively low; 4bit quantization adds to it, since we need to compress and

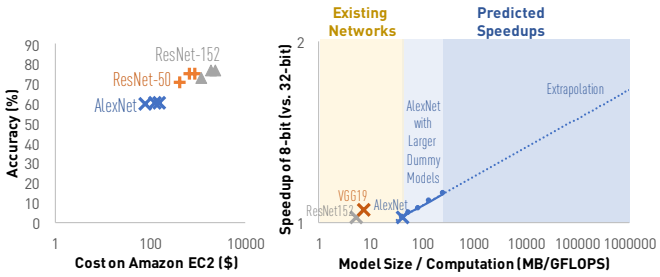



**Figure 16: Left: Price and accuracy of training different neural networks with different # epochs on Amazon EC2 with 8-bit and NCCL. Right: The performance improvement of using low-precision communication w.r.t. the communication and computation ratio.**

de-compress the data. This overhead is not compensated by the reduction in communication for this network.

**Impact on Reshaping.** One optimization we did to improve OneBitSGD is reshaping. The scalability of the reshaped version of 1bitSGD is initially lower than the original on e.g. AlexNet, due to the higher computational cost of quantizing into buckets, but this cost is amortized at higher thread counts. We note however that overall the less computationally expensive QSGD variant may perform better, although it sends more data per iteration.

## 5.4 Discussion

**Accuracy vs. Cost.** The experimental results we get in this paper also allow us to understand a tradeoff between the model accuracy and the dollar cost of training. Such tradeoff could be useful for scenarios like the following: assume a user with a large dataset (e.g., ImageNet), wishing to train a large-scale neural network to full precision, but on a limited budget. What is the most cost-effective way to do so, assuming current Amazon pricing?

Figure 16 illustrates the results of the price and accuracy for training different networks to full accuracy, according to its published recipe (accuracy and # of epochs to convergence). We use the cheapest EC2 solution for each network, which we derive from the scalability graphs. We see that there is an almost monotonic correlation between $ cost and accuracy — the more $ the user spent, the higher the quality s/he can expect. On the other hand, it is also clear that there exists a diminishing return phenomenon — spending $600 to switch from AlexNet to ResNet-50 bumps the accuracy by 15 percentage points, but another $1500 to switch to ResNet-152 only gets 2 percentage points improvement. It would be interesting to study an automatic management system that is both budget-aware and error tolerance-aware.

**MPI vs. NCCL.** The answer to this question may be now obvious to the reader. While low-precision techniques can render the MPI implementation competitive, the gap in performance is clear, since the NCCL implementation is heavily optimized. One issue is that NCCL is currently not fully supported for large GPU deployments, such as multi-node or supercomputer setups. In these cases, an MPI-based implementation is necessary. An interesting topic for future work would be to add or improve MPI support for such reduction operations, as well as support for low-precision data representations.

**1bitSGD vs QSGD.** Given our experiments, there is no clear winner between the two quantized methods. However, it is worth noting that the 8bit QSGD variant provides stable accuracy and performance for all the architectures we consider, and therefore may be a good entry-level compressor. Further, we note that QSGD wins in most of the head-to-head comparisons, albeit by small margins. The 1bitSGD algorithm communicates the least data of all the methods we tried, and can provide impressive accuracy results. Thus, in a setting where tuning is possible or minimal communication is necessary, 1bitSGD may be a good alternative. Unfortunately, there is no easy way of predicting whether a quantized method will achieve the target accuracy for a network without actually running it.

# 6 OUTLOOK

One general way to interpret our results is that the neural network training workloads we considered are currently *not communication-bottlenecked*. More precisely, either by using compressed communication, NCCL, or both, all the networks we considered can achieve significant speedup on multiple GPUs.

*What would happen to scalability if we extrapolate the trend of increasing the model size?* In particular, *in what model-size-to-GFLOPS regime would the scalability impact of quantization be significant?* These are the questions we address in Figure 16, where we examine the performance improvement (in terms of speedup) of the 8-bit quantized variant over the full-precision one, as we (artificially) increase model size for the AlexNet architecture, for the 8GPU NCCL version, on which we only noted minimal improvements using quantization. The experiment shows that, the performance improvement due to low-precision communication depends on a key factor, i.e., the ratio between communication and computation (MB/GFLOPS). As the MB/GFLOPS ratio increases, running time becomes driven by the cost of communication, and hence the speedup because of quantization is more apparent. Notice that the overall speedup will always be upper bounded by the difference in bandwidth usage, which is 4×.

Another trend is the increased computational power of GPUs, and the presence of increasingly many GPUs on a single machine. The first factor decreases the computational cost of networks, while the second increases the overall communication bandwidth of the workload. We can therefore speculate that these factors will bring quantized methods closer to the fore in terms of ways to reduce total running time, although it is also likely that the speed of communication between GPUs will improve in the future.

# 7 RELATED WORK

Changing the precision of data representation is machine learning systems has received tremendous interest recently [5, 13, 14, 16–18, 20, 23, 28, 29, 33, 36, 39, 45, 50, 51]. These works cover a whole spectrum of machine learning tasks, from training to inference, as well as models, from deep learning to linear models. This paper builds upon this prior art, but focuses on the system perspective, to understand to what extent these approaches help.

**Low-Precision Deep Learning Training** The research area closest to our work is *training* deep learning models with low precision [3, 5, 18, 23, 39]. Most algorithms use *quantizing* different data channels to lower the precision of the data representation.

One of the first references to recognize the performance impact of gradient communication when training large speech models was 1BitSGD [39]. This algorithm is inspired by delta-sigma modulation [38] for analog-to-digital conversion. In further work, variants of the same algorithm are benchmarked and refined on large-scale Amazon proprietary datasets by Strom [42]. The

1BitSGD algorithm is available by default in the Microsoft Cognitive Toolkit (CNTK) [3]. Another algorithm we consider is QSGD [5], based on the idea of *stochastically rounding* floating-point values to a small set of integer levels. For both algorithms, it is not clear how well they generalize to other deep learning models and what are their system tradeoffs; if fact, 1BitSGD is even observed to diverge in some large-scale experiments by the original authors. The goal of this paper is to provide such a fair and well-optimized system benchmark.

Recent work considered alternative quantization strategies. Aji and Heafield [4] proposed to *truncate* the gradients to only the top few percentage of components–sorted by magnitude–and to store the remaining components locally, in an accumulation vector. This enables *sparse* communication to be employed, and the authors show that extremely small densities (<0.5%) are sufficient for convergence for neural machine translation tasks. This scheme is promising, and can be theoretically justified by relating it to *asynchronous* SGD. However, we believe that this method requires further research to be widely applicable, for the following two reasons. First, in ImageNet experiments on the Inception architecture, we noticed that the density levels required for convergence to the same accuracy level as the full communication variant were large (>10%); due to the extra cost of transmitting indices, it is not clear that the reduction in communication is sufficient to ensure scalability on such tasks. Second, sparse communication is not efficiently supported by communication primitives such as NCCL or MPI.

Another approach to reduced communication is to *factorize* large network layers, and communicate the factors instead of large matrices [11, 47]. These methods are effective at reducing communication for fully-connected layers, but are less useful for conv layers, where weights are typically smaller than activations. As many modern architectures, e.g. ResNet, are almost entirely convolutional, this could limit the usefulness of this approach.

**Distributed Machine Learning** There is certainly no shortage of distributed systems for machine learning, and deep learning in particualar, such as TensorFlow [1], CNTK [3], Theano [44], Torch [12], Caffe [25], and MXNet [9]. The database community has also been contributing to this list intensively, and examples include MLlib [31] built upon Spark [48], KeystoneML [41], SystemML [8], and GraphLab [30]. However, only a few of these systems support low precision training. We hope our study helps clarify the system tradeoff introduced by low precision communication and that the insights can help to improve these systems.

**(Acknowledgment)** DA is supported in part by the SNF Ambizione Fellowship. CZ and the DS3Lab gratefully acknowledge the support from the Swiss National Science Foundation NRP 75 407540_167266, IBM Zurich, Mercedes-Benz Research & Development North America, Oracle Labs, Swisscom, Zurich Insurance, Chinese Scholarship Council, and the Department of Computer Science at ETH Zurich, the GPU donation from NVIDIA Corporation, the cloud computation resources from Microsoft Azure for Research award program.

## REFERENCES

[1] M. Abadi et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *ArXiv*, 2016.
[2] A. Acero. *Acoustical and environmental robustness in automatic speech recognition*, volume 201. Springer Science & Business Media, 2012.
[3] A. Agarwal et al. An introduction to computational networks and the computational network toolkit. Technical report, Tech. Rep. MSR, 2014.
[4] A. F. Aji et al. Sparse communication for distributed gradient descent. In *EMNLP*, 2017.
[5] D. Alistarh, J. Li, R. Tomioka, and M. Vojnovic. QSGD: Randomized Quantization for Communication-Optimal Stochastic Gradient Descent. *NIPS*, 2017.
[6] M. M. Astrahan et al. System R: relational approach to database management. *TODS*, 1976.
[7] A. A. Awan et al. Efficient large message broadcast using NCCL and CUDA-aware MPI for deep learning. In *EuroMPI*, 2016.
[8] M. Boehm et al. SystemML. *PVLDB*, 9(13):1425–1436, 2016.
[9] T. Chen et al. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *Arxiv*, 2015.
[10] S. Chetlur et al. cuDNN: Efficient Primitives for Deep Learning. 2014.
[11] T. Chilimbi et al. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, 2014.
[12] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, NIPS Workshop*, 2011.
[13] C. Cortes, M. Mohri, and A. Talwalkar. On the Impact of Kernel Approximation on Learning Accuracy. In *AISTATS*, pages 113–120, 2010.
[14] C. M. De Sa, C. Zhang, K. Olukotun, and C. Ré. Taming the wild: A unified analysis of hogwild-style algorithms. In *NIPS*, 2015.
[15] J. Dean et al. Large scale distributed deep networks. In *NIPS*, 2012.
[16] Y. Gong, L. Liu, M. Yang, and L. Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.
[17] S. Gopi, P. Netrapalli, P. Jain, and A. V. Nori. One-Bit Compressed Sensing: Provable Support and Vector Recovery. In *ICML (3)*, pages 154–162, 2013.
[18] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep Learning with Limited Numerical Precision. In *ICML*, pages 1737–1746, 2015.
[19] S. Hadjis, C. Zhang, I. Mitliagkas, D. Iter, and C. Ré. Omnivore: An Optimizer for Multi-device Deep Learning on CPUs and GPUs. *ArXiv*, 2016.
[20] S. Han et al. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv*, 2015.
[21] K. He et al. Deep Residual Learning for Image Recognition. *ArXiv*, 2015.
[22] S. Hochreiter and J. Schmidhuber. *Neural Computation*, 1997.
[23] I. Hubara et al. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv*, 2016.
[24] F. N. Iandola et al. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *CVPR*, 2016.
[25] Y. Jia et al. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv*, 2014.
[26] A. Krizhevsky et al. Learning multiple layers of features from tiny images, 2009.
[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *NIPS*, 2012.
[28] B. Lesser et al. Effects of reduced precision on floating-point SVM classification accuracy. *Procedia Computer Science*, 4, 2011.
[29] D. D. Lin, S. S. Talathi, and V. S. Annapureddy. Fixed point quantization of deep convolutional networks. *arXiv, page*, 2015.
[30] Y. Low et al. Distributed GraphLab. *PVLDB*, 5(8):716–727, 2012.
[31] X. Meng et al. MLlib: machine learning in apache spark. *JMLR*, 2016.
[32] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tennessee, 1994.
[33] D. Miyashita, E. H. Lee, and B. Murmann. Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025*, 2016.
[34] P. Moritz et al. SparkNet: Training Deep Networks in Spark. *ArXiv*, 2015.
[35] A. Neelakantan et al. Adding gradient noise improves learning for very deep networks. *arXiv*, 2015.
[36] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.
[37] O. Russakovsky et al. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3), 2015.
[38] R. Schreier and G. C. Temes. *Understanding delta-sigma data converters*, volume 74. IEEE Press, Piscataway, NJ, 2005.
[39] F. Seide et al. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns. In *Interspeech*, 2014.
[40] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *ICLR*, 2015.
[41] E. R. Sparks et al. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. *ICDE*, 2017.
[42] N. Strom. Scalable distributed DNN training using commodity GPU cloud computing. In *INTERSPEECH*, 2015.
[43] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. *ArXiv*, 2016.
[44] The Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *Arxiv*, 2016.
[45] V. Vanhoucke et al. Improving the speed of neural networks on CPUs. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, 2011.
[46] W. Wang et al. Deep Learning At Scale and At Ease. *ArXiv*, 2016.
[47] P. Xie et al. Lighter-communication distributed machine learning via sufficient factor broadcasting. In *UAI*, 2016.
[48] M. Zaharia et al. Apache Spark. *CACM*, 2016.
[49] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning requires rethinking generalization. *ICLR*, 2017.
[50] H. Zhang et al. ZipML: An End-to-end Bitwise Framework for Dense Generalized Linear Models. *ICML*, 2017.
[51] S. Zhou et al. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv*, 2016.