Master’s Thesis Nr. 163
Systems Group, Department of Computer Science, ETH Zurich

Communication-Scalable Machine Learning

by
Demjan Grubic

Supervised by
Prof. Dr. Ce Zhang, Dr. Dan Alistarh

October 2016 - April 2017
Abstract

Training deep learning models has received tremendous research interest recently. In particular, there has been intensive research on reducing the communication cost of the training process 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 GPUs vs. 8 GPUs), machine configurations (e.g., EC2 instances vs. NVIDIA DGX-1), programming models (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 we obtain.
Acknowledgements

First of all, I would like to thank my supervisors Dan Alistarh and Ce Zhang for giving me opportunity to work with them on this great project. I would like to express gratitude for all the discussions we had, their suggestions, patience and help, and for sharing their knowledge and ideas. I cannot state how less stressful all the deadlines were because of them. They were always there for me and supported my ideas. Furthermore, I would like to thank them for providing budget and hardware for this project.

I am grateful for the opportunity to collaborate with Leo Tam who provided additional hardware and helped with the experiments.

I must also mention my study mentor Angelika Steger who always supported my decisions throughout my master studies.

I would like to thank my great friend and an ETH colleague Marko whom I known for many years and with whom I shared many great memories. My thesis and my master studies would not be the same without him. My gratitude also goes to Milos who was very helpful since my first day at ETH.

Acknowledgements also go to Sonja, who proofread my master thesis and supported me on this journey.

Finally, I thank my parents Milan and Jasna, and my brothers and sisters Marina, Jurica, Maja, Boris, Miroslav for their love and support.
Some results from this master thesis appear in research papers that are currently in preparation. Most of the results presented here are from reference [17], which is currently in preparation. Some results also overlap with reference [4].
# Contents

## 1 Introduction

## 2 Preliminaries

### 2.1 Stochastic Gradient Descent

### 2.2 One-Bit Stochastic Gradient Descent

### 2.3 SGD with Stochastic Quantization

### 2.4 Communication Primitives

#### 2.4.1 MPI Reduce-and-Broadcast

#### 2.4.2 NCCL Accumulation

## 3 System Description

### 3.1 CNTK

### 3.2 Low-Precision Support

#### 3.2.1 Data Representation and 1bitSGD

#### 3.2.2 QSGD Implementation

## 4 Experimental Setup

### 4.1 DataSets

### 4.2 Networks

### 4.3 Machines

### 4.4 Tuning and Experimental Protocol

## 5 Analysis

### 5.1 Accuracy

### 5.2 Performance

### 5.3 Scalability

### 5.4 Discussion

## 6 Outlook
Chapter 1

Introduction

The system tradeoffs induced by training deep neural networks seem to be an endless discussion [52, 14, 39, 24, 9, 54, 19]. One fundamental reason for this sophistication is the level of diversity involved. At the algorithmic level, there are synchronous, asynchronous, 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 there currently exists little theoretical understanding [57].

From a data management perspective, these trade-offs provide an opportunity to build automatic optimizers for deep learning tasks. Just as with previous optimizers that our community has been building [5, 7], 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.

Furthermore, we will present new quantization technique “TopK” which we do not compare to other quantization techniques since this is still preliminary work. We plan to do that in near future.

Low Precision Deep Learning. A recently emerging research topic in deep learning systems is lowering the precision of data representation throughout the whole system [59, 58, 41, 38, 23, 18, 13, 4, 20]. In particular, there has been recent success in representing all data movements involved in training a neural network with as little
1. Introduction

as 1-bit per dimension while still getting the same accuracy for some, but not all, networks [31]. 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 system performance.

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. We start from CNTK but report an array of system optimizations. In some cases, our system prototype can be up to 3.5× faster than 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, 39]. 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:

2. Neural Network Architectures: {AlexNet [31], ResNet [21], VGG [47], Inception [50], LSTM [22]}
3. Quantization Strategy: {"1-bit SGD" [45], QSGD [4]}
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’s recently released 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 evalua-
tion 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 a full precision run. 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 leave 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 specially optimized by NVIDIA for GPU-to-GPU communication, the baseline 32bit full precision becomes much faster than MPI. Consequently, the speed-up we can obtain with low-precision communication is also limited, a standard instance of diminishing returns. In fact, on most networks, the performance improvement we get when using 8GPUs and NCCL is almost negligible; only for one network (VGG), we only get up to $1.4 \times$ 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 $4 \times$ speedup using low-precision communication; For networks with small model, such as Inception, 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 $1.4 \times$ 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 is $2 \times$ 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 of Our Study. Like all scientific studies, our study has limitations. We outline them here.

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 limitation 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. Here, we emphasize the fact that 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 available on EC2: i-0a7a0521e93c329d9. Our variant of CNTK is available under [github.com/ZipML/CNTK](https://github.com/ZipML/CNTK).

**Overview.** The rest of this paper is organized as follows. We introduce the background material in Chapter 2 and describe our system prototype in Chapter 3. We describe our experiment setup in Chapter 4 and analyze our results in Chapter 5. We provide more discussion in Section 5.4, introduce new quantization technique in Section 7, and survey related work in Chapter 8.
In this chapter, 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 \( f : \mathbb{R}^n \rightarrow \mathbb{R} \), 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 \( \tilde{g} \) which satisfies \( \mathbb{E}[\tilde{g}(x)] = \nabla f(x) \). SGD will converge towards the minimum by iterating the following procedure

\[
x_{t+1} = x_t - \eta_t \tilde{g}(x_t),
\]

where \( x_t \) 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 \( f(\theta) = \mathbb{E}_{X \sim D}[\ell(X, \theta)] \), 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 naturally 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 \( x \), of dimension \( n \). The algorithm is described in Algorithm 1.
2. Preliminaries

Data: Stochastic gradients
Data: Local copy of the parameter vector \( x \)

for each iteration \( t \) do
  1. Let \( \tilde{g}_i^t \) be an independent stochastic gradient;
  2. \( M^t \leftarrow \text{Encode}(\tilde{g}_i(x)) \) //encode gradients;
  3. broadcast \( M^t \) to all peers;
  4. for each peer \( \ell \) do
      5. receive \( M^\ell \) from peer \( \ell \);
      6. \( \hat{g}_i^\ell \leftarrow \text{Decode}(M^\ell) \) //decode gradients;
  7. end
  8. end

Algorithm 1: Synchronous Data-Parallel SGD Algorithm.

Each processor aggregates the value of \( x \), then obtains random gradient updates for each component of \( x \), 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.

When the encoding and decoding steps are trivial (i.e., no encoding / decoding), we will refer to this algorithm as full-precision (parallel) SGD. In this case, at each processor, if \( x_t \) was the value of \( x \) that the processors held before iteration \( t \), then the updated value of \( x \) by the end of this iteration is \( x_{t+1} = x_t - (\eta_t/K) \sum_{\ell=1}^K \tilde{g}_\ell(x_t) \), where each \( \tilde{g}_\ell \) 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 now describe the 1bitSGD quantization scheme, introduced by Seide et al. \[45\]. We denote the quantization function by \( Q_{1b}(v) \), mapping a vector \( v \) to its quantized version. The procedure first splits the vector \( v \) into its positive and negative components, respectively, and computes the average over positive values, which we denote by \( \text{avg}_+ \), and the average over negative values, which we denote by \( \text{avg}_- \). Then,

\[
Q_{1b}(v) = \begin{cases} 
\text{avg}_+ & \text{if } v_i \geq 0; \\
\text{avg}_- & \text{otherwise}.
\end{cases}
\]

Simply put, the procedure replaces each component with the average corresponding to its respective sign. Critically, the procedure also maintains an error correction vector \( \varepsilon \), associated to the model. In each iteration, the error from the previous iteration is added
2.3. SGD with Stochastic Quantization

We now present the stochastic quantization scheme known as QSGD, following the description from the original paper [4]. We denote the quantization function by $Q_s(v)$, 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 $v \in \mathbb{R}^n$, $Q_s(v)$ is defined as

$$Q_s(v_i) = \|v\|_2 \cdot \text{sgn} v_i \cdot \xi_i(v, s), \quad (2.1)$$

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

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

Here, $p(a, s) = as - \ell$ for any $a \in [0,1]$. If $v = 0$, then we define $Q(v, s) = 0$. See below for an illustration.

The quantization distribution $\xi_i(v, s)$ is defined to have minimal variance over distributions with support $\{0, 1/s, \ldots, 1\}$. Its expectation satisfies $E[Q_s(v_i)] = v_i$. The quantizer is an unbiased estimator of the original gradient, thus ensures convergence [4, 13].

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 [58]. We implemented this for gradient but does not observe significant improvement. Thus, we do not consider this extension in this paper.
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 [37]. 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 [6] (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).
Chapter 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 unified 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 error backpropagation, 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 [45], 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. Furthermore, it offers a number of pre-trained state-of-the-art models.
3. System Description

3.2 Low-Precision Support

CNTK implements 1bitSGD, which is used by default in multi-GPU environments with SGD. The algorithm behind 1bitSGD is described in Section 2.2 below we provide a detailed description of its CNTK C++ implementation. Low-precision communication 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 \( \text{avg}_+ \) and \( \text{avg}_- \), 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 \( \text{avg}_+ \) and \( \text{avg}_- \) 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, \( \text{avg}_+ \) and \( \text{avg}_- \) 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 \( \text{avg}_+ \) and \( \text{avg}_- \), where the number of threads is tuned for performance. The second phase uses calculated numbers \( \text{avg}_+ \) and \( \text{avg}_- \) 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 technique 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.
3.2. Low-Precision Support

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 quantized 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 \( g\text{Bits} \), 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 \( g\text{Bits} \) 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 \( [-\|v\|_2, \|v\|_2] \) into \( 2^{g\text{Bits}} - 1 \) 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. Importantly, the whole matrix is reshaped such that each column has a specified number of elements, where we always try to place consecutive 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.** During testing, 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.*
3. System Description

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Training samples</th>
<th># Validation samples</th>
<th>Size</th>
<th># classes</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImageNet</td>
<td>1.3M</td>
<td>50k</td>
<td>145GB</td>
<td>1000</td>
<td>Image</td>
</tr>
<tr>
<td>CIFAR-10</td>
<td>50k</td>
<td>10k</td>
<td>1GB</td>
<td>100</td>
<td>Image</td>
</tr>
<tr>
<td>AN4</td>
<td>948</td>
<td>130</td>
<td>64MB</td>
<td>NA</td>
<td>Speech</td>
</tr>
</tbody>
</table>

Figure 3.1: Statistics of datasets.

<table>
<thead>
<tr>
<th>Task</th>
<th>Network</th>
<th>Dataset</th>
<th># epochs to run</th>
<th>Initial Learning Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>AlexNet</td>
<td>ImageNet</td>
<td>62M</td>
<td>112</td>
</tr>
<tr>
<td></td>
<td>BN-Inception</td>
<td>ImageNet</td>
<td>11M</td>
<td>300</td>
</tr>
<tr>
<td></td>
<td>ResNet20</td>
<td>CIFAR-10</td>
<td>270k</td>
<td>160</td>
</tr>
<tr>
<td></td>
<td>ResNet50</td>
<td>ImageNet</td>
<td>25M</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td>ResNet110</td>
<td>CIFAR-10</td>
<td>1M</td>
<td>160</td>
</tr>
<tr>
<td></td>
<td>ResNet152</td>
<td>ImageNet</td>
<td>60M</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td>VGG19</td>
<td>ImageNet</td>
<td>143M</td>
<td>80</td>
</tr>
<tr>
<td>Speech</td>
<td>LSTM</td>
<td>AN4</td>
<td>13M</td>
<td>20</td>
</tr>
</tbody>
</table>

Figure 3.2: Statistics of networks.

<table>
<thead>
<tr>
<th>Instance</th>
<th># CPU cores</th>
<th>GPU</th>
<th>TFLOPS (single)</th>
<th>$/hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon</td>
<td>p2.xlarge</td>
<td>4</td>
<td>1 × K80</td>
<td>$0.9</td>
</tr>
<tr>
<td></td>
<td>p2.8xlarge</td>
<td>32</td>
<td>8 × K80</td>
<td>$7.2</td>
</tr>
<tr>
<td></td>
<td>p2.16xlarge</td>
<td>64</td>
<td>16 × K80</td>
<td>$14.4</td>
</tr>
<tr>
<td>DGX1</td>
<td>2 × 16</td>
<td>8 × P100</td>
<td>8 × 10.6</td>
<td>$50 (Nimbix)</td>
</tr>
</tbody>
</table>

Figure 3.3: Statistics of machines

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 $4 \times$ speedup compared with the original CNTK implementation. In all experiments we will denote this version of 1bitSGD with 1bitSGD*.
3.2. Low-Precision Support

<table>
<thead>
<tr>
<th>Network</th>
<th>1 GPU</th>
<th>2 GPUs</th>
<th>4 GPUs</th>
<th>8 GPUs</th>
<th>16 GPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>BN-Inception</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>VGG19</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>ResNet50</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>ResNet152</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>ResNet110</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>LSTM</td>
<td>16</td>
<td>16</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Figure 3.4: Batch sizes used for each network and # GPUs. See Section 4.4 for the tuning protocol that results in these choices.
### AlexNet ImageNet

<table>
<thead>
<tr>
<th>Setup</th>
<th>Precision</th>
<th>Bucket size</th>
<th>Accuracy</th>
<th>Samples per second (MPI)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>56</td>
<td>89</td>
<td>112</td>
</tr>
<tr>
<td>QSGD 16bit</td>
<td>8192</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>QSGD 8bit</td>
<td>512</td>
<td>60.37%</td>
<td>60.38%</td>
<td>60.38%</td>
</tr>
<tr>
<td>QSGD 4bit</td>
<td>512</td>
<td>59.63%</td>
<td>60.36%</td>
<td>60.37%</td>
</tr>
<tr>
<td>QSGD 2bit</td>
<td>512</td>
<td>59.20%</td>
<td>59.97%</td>
<td>60.05%</td>
</tr>
<tr>
<td>1bitSGD</td>
<td>/</td>
<td>59.57%</td>
<td>60.23%</td>
<td>60.31%</td>
</tr>
<tr>
<td>1bitSGD*</td>
<td>64</td>
<td>59.26%</td>
<td>59.92%</td>
<td>59.99%</td>
</tr>
<tr>
<td>1bitSGD**</td>
<td>512</td>
<td>58.44%</td>
<td>59.25%</td>
<td>59.29%</td>
</tr>
</tbody>
</table>

### ResNet50 ImageNet

<table>
<thead>
<tr>
<th>Setup</th>
<th>Precision</th>
<th>Bucket size</th>
<th>Accuracy</th>
<th>Samples per second (MPI)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>60</td>
<td>96</td>
<td>120</td>
</tr>
<tr>
<td>QSGD 16bit</td>
<td>8192</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>QSGD 8bit</td>
<td>512</td>
<td>70.66%</td>
<td>75.07%</td>
<td>75.19%</td>
</tr>
<tr>
<td>QSGD 4bit</td>
<td>512</td>
<td>70.66%</td>
<td>74.62%</td>
<td>74.76%</td>
</tr>
<tr>
<td>QSGD 2bit</td>
<td>128</td>
<td>70.66%</td>
<td>74.62%</td>
<td>74.76%</td>
</tr>
<tr>
<td>1bitSGD</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>1bitSGD*</td>
<td>64</td>
<td>70.78%</td>
<td>74.93%</td>
<td>75.02%</td>
</tr>
</tbody>
</table>

### ResNet110 CIFAR10

<table>
<thead>
<tr>
<th>Setup</th>
<th>Precision</th>
<th>Bucket size</th>
<th>Accuracy</th>
<th>Samples per second (MPI)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>80</td>
<td>96</td>
<td>160</td>
</tr>
<tr>
<td>QSGD 16bit</td>
<td>8192</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>QSGD 8bit</td>
<td>512</td>
<td>86.92%</td>
<td>93.97%</td>
<td>94.19%</td>
</tr>
<tr>
<td>QSGD 4bit</td>
<td>512</td>
<td>87.77%</td>
<td>93.52%</td>
<td>93.76%</td>
</tr>
<tr>
<td>QSGD 2bit</td>
<td>128</td>
<td>85.77%</td>
<td>92.92%</td>
<td>92.64%</td>
</tr>
<tr>
<td>1bitSGD</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>1bitSGD*</td>
<td>64</td>
<td>87.13%</td>
<td>94.02%</td>
<td>94.15%</td>
</tr>
</tbody>
</table>

### ResNet152 ImageNet

<table>
<thead>
<tr>
<th>Setup</th>
<th>Precision</th>
<th>Bucket size</th>
<th>Accuracy</th>
<th>Samples per second (MPI)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>60</td>
<td>96</td>
<td>160</td>
</tr>
<tr>
<td>QSGD 16bit</td>
<td>8192</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>QSGD 8bit</td>
<td>512</td>
<td>72.95%</td>
<td>76.66%</td>
<td>76.74%</td>
</tr>
<tr>
<td>QSGD 4bit</td>
<td>512</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>QSGD 2bit</td>
<td>128</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>1bitSGD</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>1bitSGD*</td>
<td>64</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
</tbody>
</table>

### VGG19 ImageNet

<table>
<thead>
<tr>
<th>Setup</th>
<th>Precision</th>
<th>Bucket size</th>
<th>Accuracy</th>
<th>Samples per second (MPI)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>40</td>
<td>64</td>
<td>80</td>
</tr>
<tr>
<td>QSGD 16bit</td>
<td>8192</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>QSGD 8bit</td>
<td>512</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>QSGD 4bit</td>
<td>512</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>QSGD 2bit</td>
<td>512</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>1bitSGD</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>1bitSGD*</td>
<td>64</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
</tbody>
</table>

### BN-Inception ImageNet

<table>
<thead>
<tr>
<th>Setup</th>
<th>Precision</th>
<th>Bucket size</th>
<th>Accuracy</th>
<th>Samples per second (MPI)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>150</td>
<td>250</td>
<td>300</td>
</tr>
<tr>
<td>QSGD 16bit</td>
<td>8192</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>QSGD 8bit</td>
<td>512</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>QSGD 4bit</td>
<td>512</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>QSGD 2bit</td>
<td>512</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>1bitSGD</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>1bitSGD*</td>
<td>64</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
</tbody>
</table>

Figure 3.5: Results of using MPI on Amazon EC2 instance. 

"/" means a number is missing for multiple reasons: (1) it might take more than 2 weeks to run to converge; (2) we already see preliminary result that certain run diverges or significantly worse than full precision. Because there are no data parallel SGD on 1 GPU therefore most columns about 1GPU is empty. We present the reason for each missing cell.
## 3.2. Low-Precision Support

<table>
<thead>
<tr>
<th>Precision</th>
<th>Bucket size</th>
<th>1 GPU</th>
<th>2 GPUS</th>
<th>4 GPUS</th>
<th>8 GPUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>32bit</td>
<td>/</td>
<td>240.80</td>
<td>458.20</td>
<td>625.00</td>
<td>1138.30</td>
</tr>
<tr>
<td>QSGD 16bit</td>
<td>8192</td>
<td>462.80</td>
<td>632.10</td>
<td>1157.60</td>
<td></td>
</tr>
<tr>
<td>QSGD 8bit</td>
<td>512</td>
<td>458.40</td>
<td>641.80</td>
<td>1214.80</td>
<td></td>
</tr>
<tr>
<td>QSGD 4bit</td>
<td>512</td>
<td>471.90</td>
<td>659.40</td>
<td>1247.70</td>
<td></td>
</tr>
<tr>
<td>QSGD 2bit</td>
<td>128</td>
<td>471.00</td>
<td>661.60</td>
<td>1229.70</td>
<td></td>
</tr>
</tbody>
</table>

### ResNet50 ImageNet

<table>
<thead>
<tr>
<th>Precision</th>
<th>Bucket size</th>
<th>1 GPU</th>
<th>2 GPUS</th>
<th>4 GPUS</th>
<th>8 GPUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>32bit</td>
<td>/</td>
<td>47.20</td>
<td>93.70</td>
<td>164.50</td>
<td>324.20</td>
</tr>
<tr>
<td>QSGD 16bit</td>
<td>8192</td>
<td>93.70</td>
<td>164.50</td>
<td>324.20</td>
<td></td>
</tr>
<tr>
<td>QSGD 8bit</td>
<td>512</td>
<td>94.00</td>
<td>165.80</td>
<td>297.40</td>
<td></td>
</tr>
<tr>
<td>QSGD 4bit</td>
<td>512</td>
<td>95.60</td>
<td>167.90</td>
<td>298.40</td>
<td></td>
</tr>
<tr>
<td>QSGD 2bit</td>
<td>128</td>
<td>95.50</td>
<td>168.20</td>
<td>304.10</td>
<td></td>
</tr>
</tbody>
</table>

### ResNet152 ImageNet

<table>
<thead>
<tr>
<th>Precision</th>
<th>Bucket size</th>
<th>1 GPU</th>
<th>2 GPUS</th>
<th>4 GPUS</th>
<th>8 GPUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>32bit</td>
<td>/</td>
<td>16.90</td>
<td>33.40</td>
<td>60.10</td>
<td>112.10</td>
</tr>
<tr>
<td>QSGD 16bit</td>
<td>8192</td>
<td>33.40</td>
<td>59.80</td>
<td>112.20</td>
<td></td>
</tr>
<tr>
<td>QSGD 8bit</td>
<td>512</td>
<td>33.70</td>
<td>60.80</td>
<td>115.10</td>
<td></td>
</tr>
<tr>
<td>QSGD 4bit</td>
<td>512</td>
<td>34.20</td>
<td>62.10</td>
<td>118.70</td>
<td></td>
</tr>
<tr>
<td>QSGD 2bit</td>
<td>128</td>
<td>34.30</td>
<td>62.20</td>
<td>119.90</td>
<td></td>
</tr>
</tbody>
</table>

### VGG19 ImageNet

<table>
<thead>
<tr>
<th>Precision</th>
<th>Bucket size</th>
<th>1 GPU</th>
<th>2 GPUS</th>
<th>4 GPUS</th>
<th>8 GPUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>32bit</td>
<td>/</td>
<td>12.40</td>
<td>24.90</td>
<td>48.70</td>
<td>163.10</td>
</tr>
<tr>
<td>QSGD 16bit</td>
<td>8192</td>
<td>24.90</td>
<td>49.10</td>
<td>168.00</td>
<td></td>
</tr>
<tr>
<td>QSGD 8bit</td>
<td>512</td>
<td>25.50</td>
<td>50.50</td>
<td>175.20</td>
<td></td>
</tr>
<tr>
<td>QSGD 4bit</td>
<td>512</td>
<td>25.60</td>
<td>51.00</td>
<td>179.50</td>
<td></td>
</tr>
<tr>
<td>QSGD 2bit</td>
<td>128</td>
<td>25.60</td>
<td>51.10</td>
<td>177.80</td>
<td></td>
</tr>
</tbody>
</table>

### BN-Inception ImageNet

<table>
<thead>
<tr>
<th>Precision</th>
<th>Bucket size</th>
<th>1 GPU</th>
<th>2 GPUS</th>
<th>4 GPUS</th>
<th>8 GPUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>32bit</td>
<td>/</td>
<td>88.30</td>
<td>175.30</td>
<td>342.00</td>
<td>486.70</td>
</tr>
<tr>
<td>QSGD 16bit</td>
<td>8192</td>
<td>174.30</td>
<td>342.70</td>
<td>497.10</td>
<td></td>
</tr>
<tr>
<td>QSGD 8bit</td>
<td>512</td>
<td>174.50</td>
<td>345.30</td>
<td>510.10</td>
<td></td>
</tr>
<tr>
<td>QSGD 4bit</td>
<td>512</td>
<td>178.60</td>
<td>349.00</td>
<td>598.90</td>
<td></td>
</tr>
<tr>
<td>QSGD 2bit</td>
<td>128</td>
<td>177.20</td>
<td>349.00</td>
<td>608.20</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.6: Results of using NCCL on Amazon EC2 instance. For 1GPU there is no parallel data SGD and therefore it is not applicable. For 32bit it does not have bucket and therefore the corresponding number is not applicable.
3. System Description

(a) Test accuracy per epoch, for AlexNet on 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 3.7: Accuracy results for various networks and datasets. Cyan = 32bit.
4.1 DataSets

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

The ImageNet Dataset. The ILSRVC 2015 dataset [42], also known as 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.

The CIFAR-10 Dataset. We also consider the small-scale CIFAR-10 object classification dataset [30]. The original training set consists of 50,000 32 × 32 color 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.

The AN4 Dataset. 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
4. Experimental Setup

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 instances of ResNet20 and 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 3.2.

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, which provides a reproducible environment for each experiment. 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 protocol to set the hyperparameters.

**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 ResNet because the whole batch does not fit in memory. In this case, we decrease the batch size until CNTK runs. Figure 3.4 shows the batch sizes used by each experiment.

**Learning Rate.** For learning rate, we always use the default setting, 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
4.4. Tuning and Experimental Protocol

and final quality given a fixed value of the learning rate and momentum. This has been observed previously, in e.g. \[25\].

**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. As shown in Figure [3.5], 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 to the full-precision NCCL version.
Chapter 5

Analysis

Figures 3.5 and Figure 3.6 contain all of our experiment results. Entries that are “/”, are explained in the caption.

5.1 Accuracy

Does low-precision always hurt accuracy?

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 for each epoch. Figure 3.7 illustrates the result for AlexNet, ResNet50 and ResNet152 on ImageNet, ResNet110 on CIFAR, and an LSTM network on AN4.

1We also conduct similar experiments for training accuracy. We leave the result to the full version as it does not change any claims.
Figure 5.1: Performance: Amazon EC2 Instance with MPI. QN = QSGD with \( N \) bits. 1b = 1bitSGD, 1b* = 1bitSGD with reshaping.

Figure 5.2: Performance: Amazon EC2 Instance with NCCL. QN = QSGD with \( N \) bits. 1b = 1bitSGD, 1b* = 1bitSGD with reshaping.

Figure 5.3: Performance: NVIDIA DGX-1 with MPI.

Figure 5.4: Performance: NVIDIA DGX-1 with NCCL.

Figure 5.5: Scalability: Amazon EC2 Instance with MPI.

Figure 5.6: Scalability: Amazon EC2 Instance with NCCL.

Figure 5.7: Scalability: NVIDIA DGX-1 with MPI.

Figure 5.8: Scalability: NVIDIA DGX-1 with NCCL.
5.1. Accuracy

**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 [40]. Additional experiments show that, for this dataset, quantization also improves training loss (see Figure 3.7e), which is perhaps surprising. It is not clear what is the definitive reason of this behavior.

**Accuracy: Negatives.** At the same time, it is important to note that quantizing too aggressively 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 [4, 58]: 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 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.

**Impact of Layer Types.** We also find that convolutional layers as being 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). During the tuning process, 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 $>1\%$ inferior to the full-precision variant. Adjusting the bucket size to 512 allows it to get accuracy which is 0.3% away from 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 normalization). 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 [45].

### 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 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 5.1 and Figure 5.2 show the result on the 16GPU Amazon instance, and Figure 5.3 and Figure 5.4 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 5.1 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 net-
work, 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 components.

**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 observe 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 occurring when
lowering precision 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 also 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 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 commu-
nication 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 5.2
shows the result of using NCCL on Amazon instances.

**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 Figure 5.2 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 5.1) and NCCL (Figure 5.2). It is clear that the MPI implementation is slower than the NCCL implementation — 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 5.2 on communication-dominated networks such as VGG, we get up to $1.5\times$ speedup with 4-bit or 8-bit precision NCCL. This end-to-end speedup is 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 $5\times$ (VGG). However, with NCCL, the achievable speedup is again limited significantly. In fact, with NCCL, on VGG, we get up only $1.6\times$ 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 the scalability of the system: 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 during a training epoch. Figure 5.5-5.8 shows the result.
Figure 5.9: 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.

**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 $6 \times$ with MPI and $7 \times$ with NCCL on 8 GPUs. However, for communication-dominated networks, 32-bit suffers. For example, for AlexNet, 32-bit full precision with MPI only achieves $2 \times$ 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 communication as we discussed earlier.

**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 $< 3 \times$ to $8 \times$ by using 1bitSGD. Networks such as ResNet152 scale almost linearly once quantization is applied even with MPI: with quantization, scalability at 16GPUs is $2 \times$, and $5 \times$ without.

When using NCCL, the difference between quantized communication and full precision decreases significantly. As shown in Figure 5.6 and Figure 5.8, 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 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 the original 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 5.9 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. As future work, 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.
Chapter 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.

It is then natural to ask 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 5.9, 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 $4 \times$.

Another trend we can consider 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.
Chapter 7

TopK

In this chapter we present new quantization technique TopK.

7.1 Gradient vector distribution

After plotting the gradients using various networks, we noticed that distribution of coordinates in gradients is similar to Gaussian distribution.

As could be seen in Figure 7.1, mentioned property holds with different bucket sizes, different networks and different types of layers. That property holds stronger with larger bucket sizes and convolutional layers.

Figure 7.1: Gradient vector distribution, x-axis - vector’s coordinates values, y-axis - number of vector’s coordinates, xb = bucket size equal x
7. TopK

7.2 Algorithm

Based on gradient distribution observations we proposed new quantization technique TopK. For each vector only \( k \) coordinates with highest absolute value are stored and sent, and all other coordinates are set to 0. In order to compensate for quantization loss error correction vector is stored, as described in 1bitSGD.

This way advantage can be taken of sparsity and only \( k \) floating point numbers with their positions in the vector will be sent.

The resulting algorithm works as Algorithm 3, which can be used as the \texttt{Encode} function of standard SGD (Algorithm 1).

\begin{verbatim}
Data: Gradient vector \( v \) to be quantized
Data: Error \( \epsilon \) from the previous round
1 \( v \leftarrow v + \epsilon \) //add error from previous round ;
2 \( \epsilon = v \) // store new values as errors for all coordinates except for top \( k \);
3 // find \( k \) coordinates in \( v \) with highest absolute value and store their position;
4 for idx in (indices of top \( k \) coordinates sorted by \(|v_i|\)) do
5     q.append((v[\text{idx}], idx));
6     \( \epsilon_{\text{idx}} \leftarrow 0 \);
7 end
8 return q
\end{verbatim}

Algorithm 3: TopK procedure.

7.3 Accuracy results

In this set of experiments we study the impact of different parameters on accuracy when using TopK. We run all experiments on a network ResNet20 that uses CIFAR-10 dataset.

Parameter \( k \). We can control loss and sparsity during quantization by changing parameters \( k \) and having bucket size set as a constant. In Figure 7.2 we show top-1 test accuracy with different values for parameter \( k \) with constant bucket size set to 1024. From the mentioned figure we can see that sending only top 8 values from a bucket is not enough in order to maintain accuracy and that it underperforms 32bit version by 0.3%. Having \( k \geq 16 \) with bucket size set to 1024 we were able to achieve same accuracy as 32bit version, even outperform 32bit in some cases by 0.25%.

Parameter bucket size. Having \( k \) as a constant parameter, we can change bucket size and control sparsity which has significant impact on a communication speed. As shown in Figure 7.3 having large bucket size can impact convergence. For instance, when \( k = 2 \) and bucket size equals 32 TopK has accuracy loss of 0.55\%, while with bucket size 16 test accuracy decrease is 0.3\%. Similar observations could be made for \( k = 8 \) where by decreasing bucket size we also get accuracy improvement.
7.4 Discussion

We presented new quantization technique and showed that it can obtain same accuracy as SGD without quantization. Advantage of TopK is that we can benefit from using sparsity, especially in setups where we do not have all computational power on same machine. One example of such setup is a supercomputer where we train a model on hundreds of GPUs where each GPU is on a separate machine.
Chapter 8

Related Work

Changing the precision of data representation is machine learning systems has received tremendous interest recently [59, 58, 53, 46, 41, 38, 34, 32, 23, 18, 16, 15, 13, 12, 4, 20]. 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 all this prior arts 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 [18, 3, 44, 23, 4]. Most algorithms use quantizing different data channels to lower the precision of the data representation. We considered two state-of-the-art strategies and study their system implications.

One of the first references to recognize the performance impact of gradient communication when training large speech models was 1BigSGD [44], in which Seide et al. proposed the 1BitSGD algorithm. This algorithm is inspired by delta-sigma modulation [43] 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 [49]. The 1BitSGD algorithm is available by default in the Microsoft Cognitive Toolkit (CNTK) [3]. Another algorithm we consider is by Alistarh et al. [4], who developed the Quantized SGD (QSGD) algorithm, 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.

Distributed Machine Learning There is certainly no shortage of distributed systems for machine learning, and deep learning in particular, such as TensorFlow [1], CNTK [3], Theano [51], Torch [11], Caffe [26], and MXNet [8]. Database community has also been contributing to this list intensively, and examples include MLlib [36] built upon Spark [56], KeystoneML [48], SystemML [7], and GraphLab [35]. 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 in the future.

Other Related Work. Another line of research concerns inference and model compression of a pre-trained model [53, 15, 20, 33, 28, 29, 55] and use the resulting model for inference. In this paper, we focus on training and leave the study of inference to future work.

Quantization has also been studied by the DSP community, and there has been research on linear regression models in the presence of quantization error or other types of noise. For example, [16] studied compressive sensing with binary quantized measurement. Precision of data representation is a also key design decision for configurable hardwares such as FPGA. There have been attempts to lower the precision when training on such hardware [27].
Bibliography


[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient


<table>
<thead>
<tr>
<th>Citation</th>
<th>Reference</th>
</tr>
</thead>
</table>


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Communication-Scalable Machine Learning

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):

GRUBIC

First name(s):

DEMJAN

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Zürich, April 19, 2017

Signature(s)