TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning

Wei Wen
Duke University
wei.wen@duke.edu

Cong Xu
Hewlett Packard Labs
cong.xu@hpe.com

Feng Yan
University of Nevada, Reno
fyan@unr.edu

Chunpeng Wu
Duke University
chunpeng.wu@duke.edu

Yandan Wang
University of Pittsburgh
yaw46@pitt.edu

Yiran Chen
Duke University
yiran.chen@duke.edu

Hai Li
Duke University
hai.li@duke.edu

Abstract

High network communication cost for synchronizing gradients and parameters is the well-known bottleneck of distributed training. In this work, we propose TernGrad that uses ternary gradients to accelerate distributed deep learning in data parallelism. Our approach requires only three numerical levels \{-1, 0, 1\}, which can aggressively reduce the communication time. We mathematically prove the convergence of TernGrad under the assumption of a bound on gradients. Guided by the bound, we propose layer-wise ternarizing and gradient clipping to improve its convergence. Our experiments show that applying TernGrad on AlexNet doesn’t incur any accuracy loss and can even improve accuracy. The accuracy loss of GoogLeNet induced by TernGrad is less than 2% on average. Finally, a performance model is proposed to study the scalability of TernGrad. Experiments show significant speed gains for various deep neural networks.

1 Introduction

The remarkable advances in deep learning is driven by data explosion and increase of model size. The training of large-scale models with huge amounts of data are often carried on distributed systems \cite{1,2,3,4,5,6,7,8,9}, where data parallelism is adopted to exploit the compute capability empowered by multiple workers \cite{10}. Stochastic Gradient Descent (SGD) is usually selected as the optimization method because of its high computation efficiency. In realizing the data parallelism of SGD, model copies in computing workers are trained in parallel by applying different subsets of data. A centralized parameter server performs gradient synchronization by collecting all gradients and averaging them to update parameters. The updated parameters will be sent back to workers, that is, parameter synchronization. Increasing the number of workers helps reduce the computation time dramatically. However, as the scale of distributed systems grows up, the extensive gradient and parameter synchronizations prolong the communication time and even amortize the savings of computation time \cite{4,11,12}. A common approach to overcome such a network bottleneck is asynchronous SGD \cite{1,4,7,12,13,14}, which continues computation by using stale values without waiting for the completeness of synchronization. The inconsistency of parameters across computing workers, however, can degrade training accuracy and incur occasional divergence \cite{15,16}.

From the perspective of inference acceleration, sparse and quantized Deep Neural Networks (DNNs) have been widely studied, such as \cite{17,18,19,20,21,22}. However, these methods generally
aggravate the training effort. Researches such as sparse logistic regression and Lasso optimization problems \[\text{[24][25][26]}\] took advantage of the sparsity inherent in models and achieved remarkable speedup for distributed training. A more generic and important topic is how to accelerate the distributed training of dense models by utilizing sparsity and quantization techniques. For instance, Aji and Heafield \[\text{[24]}\] proposed to heuristically sparsify dense gradients by dropping off small values in order to reduce gradient communication. For the same purpose, quantizing gradients to low-precision values with smaller bit width has also been extensively studied \[\text{[22][25][26][27]}\].

Our work belongs to the category of gradient quantization, which is an orthogonal approach to sparsity methods. We propose \text{TernGrad} that quantizes gradients to ternary levels \[-1, 0, 1\] to reduce the overhead of gradient synchronization. Furthermore, we propose \text{scaler sharing and parameter localization}, which can replace parameter synchronization with a low-precision gradient pulling. Comparing with previous works, our major contributions include: (1) we mathematically prove the convergence of \text{TernGrad} in general by proposing a statistical bound on gradients; (2) we propose \text{layer-wise ternarizing and gradient clipping} to move this bound closer toward the bound of standard SGD. These simple techniques successfully improve the convergence; (3) we build a performance model to evaluate the speed of training methods with compressed gradients, like \text{TernGrad}. Our source code is available at \url{https://github.com/wenwei202/terngrad}

2 Related work

\textbf{Gradient sparsification.} Aji and Heafield \[\text{[24]}\] proposed a heuristic gradient sparsification method that truncated the smallest gradients and transmitted only the remaining large ones. The method greatly reduced the gradient communication and achieved 22\% speed gain on 4 GPUs for a neural machine translation, without impacting the translation quality. An earlier study by Garg et al. \[\text{[28]}\] adopted the similar approach, but targeted at sparsity recovery instead of training acceleration. Our proposed \text{TernGrad} is orthogonal to these sparsity-based methods.

\textbf{Gradient quantization.} \text{DoReFa-Net} \[\text{[22]}\] derived from \text{AlexNet} reduced the bit widths of weights, activations and gradients to 1, 2 and 6, respectively. However, \text{DoReFa-Net} showed 9.8\% accuracy loss as it targeted at acceleration on single worker. S. Gupta et al. \[\text{[27]}\] successfully trained neural networks on MNIST and CIFAR-10 datasets using 16-bit numerical precision for an energy-efficient hardware accelerator. Our work, instead, tends to speed up the distributed training by decreasing the communicated gradients to three numerical levels \{-1, 0, 1\}. F. Seide et al. \[\text{[25]}\] applied 1-bit SGD to accelerate distributed training and empirically verified its effectiveness in speech applications. As the gradient quantization is conducted by columns, a floating-point scaler \textit{per column} is required. So it cannot yield any speed benefit on convolutional neural networks \[\text{[26]}\]. Moreover, \textit{"cold start"} of the method \[\text{[25]}\] requires floating-point gradients to converge to a good initial point for the following 1-bit SGD. More importantly, it is unknown what conditions can guarantee its convergence. Comparably, our \text{TernGrad} can start the DNN training from scratch and we prove the conditions that promise the convergence of \text{TernGrad}.

Very recently, a preprint by D. Alistarh et al. \[\text{[26]}\] presented QSGD that explores the trade-off between accuracy and gradient precision. The effectiveness of gradient quantization was justified and the convergence of QSGD was provably guaranteed. Compared to QSGD developed simultaneously, our \text{TernGrad} shares the same concept but advances in the following three aspects: (1) we prove the convergence from the perspective of statistic bound on gradients. The bound also explains why multiple quantization buckets are necessary in QSGD; (2) the bound is used to guide practices and inspires techniques of \text{layer-wise ternarizing and gradient clipping}; (3) \text{TernGrad} using only 3-level gradients achieves 0.92\% top-1 accuracy improvement for \text{AlexNet}, while 1.73\% top-1 accuracy \textit{loss} is observed in QSGD with 4 levels. The accuracy loss in QSGD can be eliminated by paying the cost of increasing the precision to 4 bits (16 levels) and beyond.

3 Problem Formulation and Our Approach

3.1 Problem Formulation and \text{TernGrad}

Figure\[\text{[1]}\]formulates the distributed training problem of synchronous SGD using data parallelism. At iteration \(t\), a mini-batch of training samples are split and fed into multiple workers \((i \in \{1, ..., N\})\). Worker \(i\) computes the gradients \(g_{i}^{(t)}\) of parameters \textit{w.r.t.} its input samples \(x_{i}^{(t)}\). All gradients are first synchronized and averaged at \text{parameter server}, and then sent back to update workers.
Note that parameter server in most implementations \cite{1,12} are used to preserve shared parameters, while here we utilize it in a slightly different way to maintain shared gradients. In Figure 1, each worker keeps a copy of parameters locally. We name this technique as parameter localization. The parameter consistency among workers can be maintained by random initialization with an identical seed. Parameter localization changes the communication of parameters in floating-point form to the transfer of quantized gradients that require much lighter traffic. Note that our proposed TernGrad can be integrated with many settings like Asynchronous SGD \cite{1,4}, even though the scope of this paper only focuses on the distributed SGD in Figure 1.

Algorithm \ref{algo:TernGrad}formulates the $t$-th iteration of TernGrad algorithm according to Figure 1. Most steps of TernGrad remain the same as traditional distributed training, except that gradients shall be quantized into ternary precision before sending to parameter server. More specific, ternarize($\cdot$) aims to reduce the communication volume of gradients. It randomly quantizes gradient $g_t$ to a ternary vector with values $\in \{-1, 0, +1\}$. Formally, with a random binary vector $b_t$, $g_t$ is ternarized as

$$
\tilde{g}_t = \text{ternarize}(g_t) = s_t \cdot \text{sign}(g_t) \circ b_t,
$$

where $s_t \triangleq \max(\{\text{abs}(g_t)\})$ is a scaler that can shrink $\pm 1$ to a much smaller amplitude. $\circ$ is the Hadamard product. $\text{sign}(\cdot)$ and $\text{abs}(\cdot)$ respectively returns the sign and absolute value of each element. Giving a $g_t$, each element of $b_t$ independently follows the Bernoulli distribution

$$
P(b_{tk} = 1 \mid g_t) = |g_{tk}|/s_t \\
P(b_{tk} = 0 \mid g_t) = 1 - |g_{tk}|/s_t
$$

where $b_{tk}$ and $g_{tk}$ is the $k$-th element of $b_t$ and $g_t$, respectively. This stochastic rounding, instead of deterministic one, is chosen by both our study and QSGD \cite{26}, as stochastic rounding has an unbiased expectation and has been successfully studied for low-precision processing \cite{20,27}. Optimally, ternary gradients reduces the worker-to-server traffic by a factor of $32/\log_2(3) = 20.18 \times$. Even using 2 bits to encode a ternary gradient, the reduction factor is still $16 \times$. As aforementioned, parameter localization reduces server-to-worker traffic by pulling quantized gradients from servers. However, summing up ternary values in $\sum_i \tilde{g}_t^{(i)}$ will produce more possible levels and thereby the final averaged gradient $\overline{g}$ is no longer ternary as shown in Figure 2(d). It emerges as a critical issue when workers use different scalers $s_t^{(i)}$. To minimize the number of levels, we propose a shared scaler $s_t = \max(\{s_t^{(i)}\} : i = 1...N)$ across all the workers. We name this technique as scaler sharing. The sharing process has a small overhead of transferring $2N$ floating scalars. By integrating parameter localization and scaler sharing, the maximum number of levels in $\overline{g}$ decreases to $2N + 1$. As a result, the server-to-worker communication reduces by a factor of $32/\log_2(1 + 2N)$, unless $N \geq 2^{30}$.

**Algorithm 1 TernGrad:** distributed SGD training using ternary gradients.

**Worker:** $i = 1, ..., N$
1. Input $z_t^{(i)}$, a part of a mini-batch of training samples $z_t$
2. Compute gradients $g_t^{(i)}$ under $z_t^{(i)}$
3. Ternarize gradients to $\tilde{g}_t^{(i)} = \text{ternarize}(g_t^{(i)})$
4. Push ternary $\tilde{g}_t^{(i)}$ to the server
5. Pull averaged gradients $\overline{g}$ from the server
6. Update parameters $w_{t+1} \leftarrow w_t - \eta \cdot \overline{g}$

**Server:**
7. Average ternary gradients $\overline{g} = \sum_i \tilde{g}_t^{(i)}/N$

![Figure 1: Distributed SGD with data parallelism.](image)

**3.2 Convergence Analysis and Gradient Bound**

We analyze the convergence of TernGrad in the framework of online learning systems. An online learning system adapts its parameter $w$ to a sequence of observations to maximize performance. Each observation $z$ is drawn from an unknown distribution, and a loss function $Q(z, w)$ is used to measure the performance of current system with parameter $w$ and input $z$. The minimization target then is the loss expectation

$$
C(w) \triangleq \mathbb{E}\{Q(z, w)\}.
$$

\footnote{Here, the superscript of $g_t$ is omitted for simplicity.}
In General Online Gradient Algorithm (GOGA) \cite{20}, parameter is updated at learning rate $\gamma_t$ as
\begin{equation}
w_{t+1} = w_t - \gamma_t g_t = w_t - \gamma_t \cdot \nabla_w Q(z_t, w_t),
\end{equation}
where $g \triangleq \nabla_w Q(z, w)$ and the subscript $t$ denotes observing step $t$. In GOGA, $E\{g\}$ is the gradient of the minimization target in Eq. (3).

According to Eq. (1), the parameter in TernGrad is updated, such as
\begin{equation}
w_{t+1} = w_t - \gamma_t (s_t \cdot \text{sign}(g_t) \circ b_t),
\end{equation}
where $s_t \triangleq \text{max(abs}(g_t))$ is a random variable depending on $z_t$ and $w_t$. As $g_t$ is known for given $z_t$ and $w_t$, Eq. (2) is equivalent to
\begin{equation}
\begin{cases}
P(b_{tk} = 1 \mid z_t, w_t) = \frac{|g_{tk}|}{s_t} \\
P(b_{tk} = 0 \mid z_t, w_t) = 1 - \frac{|g_{tk}|}{s_t}
\end{cases}.
\end{equation}

At any given $w_t$, the expectation of ternary gradient satisfies
\begin{equation}
E\{s_t \cdot \text{sign}(g_t) \circ b_t\} = E\{s_t \cdot \text{sign}(g_t) \circ E\{b_t|z_t\}\} = E\{g_t\} = \nabla_w C(w_t),
\end{equation}
which is an unbiased gradient of minimization target in Eq. (3). The convergence analysis of TernGrad is adapted from the convergence proof of GOGA presented in \cite{20}, which uses the following two assumptions. Without explicit mention, vectors indicate column vectors here.

**Assumption 1.** $C(w)$ has a single minimum $w^*$ and gradient $-\nabla_w C(w)$ always points to $w^*$, i.e.,
\begin{equation}
\forall \epsilon > 0, \inf_{||w-w^*||^2 \geq \epsilon} \langle w-w^* \rangle^T \nabla_w C(w) > 0.
\end{equation}

Convexity is a subset of Assumption 1 and we can easily find non-convex functions satisfying it.

**Assumption 2.** Learning rate $\gamma_t$ is positive and constrained as $\sum_{t=0}^{\infty} \gamma_t^2 < +\infty$ and $\sum_{t=0}^{\infty} \gamma_t = +\infty$, which ensures $\gamma_t$ decreases neither very fast nor very slow respectively.

We define the distance between current parameter $w_t$ and the minimum $w^*$ as $h_t \triangleq ||w_t - w^*||^2$, where $|| \cdot ||$ is $\ell_2$ norm. We also define the set of all random variables before step $t$ as $X_t \triangleq (z_{1\ldots t-1}, b_{1\ldots t-1})$. Under Assumption 1 and Assumption 2 using Lyapunov process and Quasi-Martingales convergence theorem, L. Bottou \cite{20} proved

**Lemma 1.** If $\exists A, B > 0$ s.t. $E\{(h_{t+1} - (1 + \gamma^2 B) h_t) \mid X_t\} \leq -2\gamma_t (w_t - w^*)^T \nabla_w C(w_t) + \gamma^2_t A$, then $C(z, w)$ converges almost surely toward minimum $w^*$, i.e., $P(\lim_{t \to +\infty} w_t = w^*) = 1$.

We assume the gradient is bounded as

**Assumption 3 (Gradient Bound).** The gradient $g$ is bounded as $E\{\text{max(abs}(g)) \cdot ||g||_1\} \leq A + B||w-w^*||^2$, where $A, B > 0$ and $|| \cdot ||_1$ is $\ell_1$ norm.

With Assumption 3 and Lemma 1 we prove (refer to Appendix A for details)

**Theorem 1.** When online learning systems update as $w_{t+1} = w_t - \gamma_t (s_t \cdot \text{sign}(g_t) \circ b_t)$ using stochastic ternary gradients, they converge almost surely toward minimum $w^*$, i.e., $P(\lim_{t \to +\infty} w_t = w^*) = 1$.

Comparing with the gradient bound of standard GOGA $E\{||g||^2\} \leq A + B||w-w^*||^2$ \cite{20}, the bound in Assumption 3 is stronger because $\text{max(abs}(g)) \cdot ||g||_1 \geq ||g||^2$. We propose layer-wise ternarizing and gradient clipping to make two bounds closer, which shall be explained in Section 3.3. A side benefit of our work is that, by following the similar proof procedure, we can prove the convergence of GOGA when Gaussian noise $\mathcal{N}(0, \sigma^2)$ is added to gradients \cite{20}, under the gradient bound of $E\{||g||^2\} \leq A + B||w-w^*||^2 - \sigma^2$. Although the bound is also stronger, Gaussian noise encourages active exploration of parameter space and improves accuracy as was empirically studied in \cite{20}. Similarly, the randomness of ternary gradients also encourages space exploration and improves accuracy for some models, as shall be presented in Section 4.
Figure 2: Histograms of (a) original floating gradients, (b) clipped gradients, (c) ternary gradients and (d) final averaged gradients. Visualization by TensorBoard. The DNN is AlexNet distributed on two workers, and vertical axis is the training iteration. Top row visualizes the third convolutional layer and bottom one visualizes the first fully-connected layer.

3.3 Feasibility Considerations

The gradient bound of TernGrad in Assumption 3 is stronger than the bound in standard GOGA. Pushing the two bounds closer can improve the convergence of TernGrad. In Assumption 3, \( \max (\|g\|) \) is the maximum absolute value of all the gradients in the DNN. So \( \max (\|g\|) \) could be relatively much larger than most gradients in a large DNN, implying that the bound in TernGrad becomes much stronger. Considering the situation, we propose layer-wise ternarizing and gradient clipping to reduce \( \max (\|g\|) \) and therefore shrink the gap between these two bounds.

Layer-wise ternarizing is proposed based on the observation that the range of gradients in each layer changes as gradients are back propagated. Instead of adopting a large global maximum scaler, we independently ternarize gradients in each layer using the layer-wise scalers. More specific, we separately ternarize the gradients of biases and weights by using Eq. (1), where \( g \) could be the gradients of biases or weights in each layer. To approach the standard bound more closely, we can split gradients to more buckets and ternarize each bucket independently. However, this will introduce more floating scalers and increase communication.

Layer-wise ternarizing can shrink the bound gap resulted from the dynamic ranges of the gradients across layers. However, the dynamic range within a layer still remains as a problem. We propose gradient clipping, which limits the magnitude of each gradient \( g_i \) in \( g \) as

\[
f(g_i) = \begin{cases} 
    g_i, & |g_i| \leq c\sigma \\
    \text{sign}(g_i) \cdot c\sigma, & |g_i| > c\sigma 
\end{cases}
\]

Where \( \sigma \) is the standard derivation of gradients in \( g \). In distributed training, gradient clipping is applied to every worker before ternarizing. \( c \) is a hyper-parameter to select, but we cross validate it only once and use the constant in all our experiments. Specifically, we used a CNN \cite{1} trained on CIFAR-10 by momentum SGD with staircase learning rate and obtained the optimal \( c = 2.5 \). Suppose the distribution of gradients is close to Gaussian distribution as shown in Figure 2(a), very few gradients can drop out of \([-2.5\sigma, 2.5\sigma]\]. Clipping these gradients in Figure 2(b) can significantly reduce the scaler but slightly changes the length and direction of original \( g \). Numerical analysis shows that gradient clipping with \( c = 2.5 \) only changes the length of \( g \) by 1.0% - 1.5% and its direction by 2° - 3°. In our experiments, \( c = 2.5 \) remains valid across multiple databases (MNIST, CIFAR-10 and ImageNet), various network structures (LeNet, CifarNet, AlexNet, GoogLeNet, etc) and training schemes (momentum, vanilla SGD, adam, etc).

The effectiveness of layer-wise ternarizing and gradient clipping can also be explained as follows. When the scalar \( s_i \) in Eq. (1) and Eq. (2) is very large, most gradients have a high possibility to be ternarized to zeros, leaving only a few gradients to large-magnitude values. The scenario raises a severe parameter update pattern: most parameters keep unchanged while others likely overshoot. This will introduce large training variance. Our experiments on AlexNet show that by applying both layer-wise ternarizing and gradient clipping techniques, TernGrad can converge to the same accuracy as standard SGD. Removing any of the two techniques can result in accuracy degradation, e.g., 3% top-1 accuracy loss without applying gradient clipping as we shall show in Table 2.
4 Experiments

We first investigate the convergence of TernGrad under various training schemes on relatively small databases and show the results in Section 4.1. Then the scalability of TernGrad to large-scale distributed deep learning is explored and discussed in Section 4.2. The experiments are performed by TensorFlow. We maintain the exponential moving average of parameters by employing an exponential decay of $0.9999$. The accuracy is evaluated by the final averaged parameters. This gives slightly better accuracy in our experiments. For fair comparison, in each pair of comparative experiments using either floating or ternary gradients, all the other training hyper-parameters are same unless differences are explicitly pointed out. In experiments, when SGD with momentum is adopted, momentum value of $0.9$ is used. When polynomial decay is applied to decay the learning rate (LR), the power of $0.15$ is used to decay LR from the base LR to zero.

4.1 Integrating with Various Training Schemes

We study the convergence of TernGrad using LeNet on MNIST and a ConvNet on CIFAR-10. LeNet is trained without data augmentation. While training CifarNet, images are randomly cropped to $24 \times 24$ images and mirrored. Brightness and contrast are also randomly adjusted. During the testing of CifarNet, only center crop is used. Our experiments cover the scope of SGD optimizers over vanilla SGD, SGD with momentum and Adam. Figure 3 shows the results of CifarNet, where all trainings terminate after the same epochs. Adam SGD is used for training. Instead of keeping total mini-batch size unchanged, we maintain the mini-batch size per worker. Therefore, the total mini-batch size linearly increases as the number of workers grows. Though the base learning rate of 0.0005 seems small, it can achieve better accuracy than larger ones like 0.001. In each pair of experiments, TernGrad can converge to the accuracy level with less than 1% degradation. The accuracy degrades under a large mini-batch size in both baseline and TernGrad. This is because parameters are updated less frequently and large-batch training tends to converge to poorer sharp minima. However, the noise inherent in TernGrad can help converge to better flat minimizers, which could explain the smaller accuracy gap between the baseline and TernGrad when the mini-batch size is 2048. In our experiments of AlexNet in Section 4.2, TernGrad even improves the accuracy in the large-batch scenario. This attribute is beneficial for distributed training as a large mini-batch size is usually required.
4.2 Scaling to Large-scale Deep Learning

We also evaluate TernGrad by AlexNet and GoogLeNet trained on ImageNet. It is more challenging to apply TernGrad to large-scale DNNs. It may result in some accuracy loss when simply replacing the floating gradients with ternary gradients while keeping other hyper-parameters unchanged. However, we are able to train large-scale DNNs by TernGrad successfully after making some or all of the following changes: (1) decreasing dropout ratio to keep more neurons; (2) using smaller weight decay; and (3) disabling ternarizing in the last classification layer. Dropout can regularize DNNs by adding randomness, while TernGrad also introduces randomness. Thus, dropping fewer neurons helps avoid over-randomness. Similarly, as the randomness of TernGrad introduces regularization, smaller weight decay may be adopted. We suggest not to apply ternarizing to the last layer, considering that the one-hot encoding of labels generates a skew distribution of gradients and the symmetric ternary encoding \{-1, 0, 1\} is not optimal for such a skew distribution. Though asymmetric ternary levels could be an option, we decide to stick to floating gradients in the last layer for simplicity. The overhead of communicating these floating gradients is small, as the last layer occupies only a small percentage of total parameters, like 6.7% in AlexNet and 3.99% in ResNet-152 [35].

All DNNs are trained by momentum SGD with Batch Normalization [36] on convolutional layers. AlexNet is trained by the hyper-parameters and data augmentation depicted in Caffe. GoogLeNet is trained by polynomial LR decay and data augmentation in [37]. Our implementation of GoogLeNet does not utilize any auxiliary classifiers, that is, the loss from the last softmax layer is the total loss. More training hyper-parameters are reported in corresponding tables and published source code. Validation accuracy is evaluated using only the central crops of images. The results of AlexNet are shown in Table 2. Mini-batch size per worker is fixed to 128. For fast development, all DNNs are trained through the same epochs of images. In this setting, when there are more workers, the number of iterations becomes smaller and parameters are less frequently updated. To overcome this problem, we increase the learning rate for large-batch scenario [10]. Using this scheme, SGD with floating gradients successfully trains AlexNet to similar accuracy, for mini-batch size of 256 and 512. However, when mini-batch size is 1024, the top-1 accuracy drops 0.71% for the same reason as we point out in Section 4.1.

TernGrad converges to approximate accuracy levels regardless of mini-batch size. Notably, it improves the top-1 accuracy by 0.92% when mini-batch size is 1024, because its inherent randomness encourages to escape from poorer sharp minima [30] [34]. Figure 4 plots training details vs. iteration when mini-batch size is 512. Figure 4(a) shows that the convergence curve of TernGrad matches well with the baseline’s, demonstrating the effectiveness of TernGrad. The training efficiency can be further improved by reducing communication time as shall be discussed in Section 5. The training data loss in Figure 4(b) shows that TernGrad converges to a slightly lower level, which further proves

![Figure 4: AlexNet trained on 4 workers with mini-batch size 512: (a) top-1 validation accuracy, (b) training data loss and (c) sparsity of gradients in first fully-connected layer (fc6) vs. iteration.](image)

Table 2: Accuracy comparison for AlexNet.

| base LR | mini-batch size | workers | iterations | gradients | weight decay | DR$^*$ | top-1 | top-5 |
|---------|-----------------|---------|------------|-----------|--------------|-------|-------|-------|
| 0.01    | 256             | 2       | 370K       | floating  | 0.0005       | 0.5   | 57.33%| 80.56%|
|         |                 |         |            | TernGrad  | 0.0005       | 0.2   | 57.61%| 80.47%|
|         |                 |         |            | TernGrad-nclip$^*$ | 0.0005 | 0.2   | 54.63%| 78.16%|
| 0.02    | 512             | 4       | 185K       | floating  | 0.0005       | 0.5   | 57.32%| 80.73%|
|         |                 |         |            | TernGrad  | 0.0005       | 0.2   | 57.28%| 80.23%|
| 0.04    | 1024            | 8       | 92.5K      | floating  | 0.0005       | 0.5   | 56.62%| 80.28%|
|         |                 |         |            | TernGrad  | 0.0005       | 0.2   | 57.54%| 80.25%|

$^*$ DR: dropout ratio, the ratio of dropped neurons. $^*$ TernGrad without gradient clipping.
Table 3: Accuracy comparison for GoogLeNet.

| base LR | mini-batch size | workers | iterations | gradients | weight decay | DR   | top-5  |
|---------|-----------------|---------|------------|-----------|--------------|------|-------|
| 0.04    | 128             | 2       | 600K       | floating  | 4e-5         | 0.2  | 88.30%|
|         |                 |         |            | TernGrad  | 1e-5         |      |       |
| 0.08    | 256             | 4       | 300K       | floating  | 4e-5         | 0.2  | 87.82%|
|         |                 |         |            | TernGrad  | 1e-5         |      |       |
| 0.10    | 512             | 8       | 300K       | floating  | 4e-5         | 0.2  | 89.00%|
|         |                 |         |            | TernGrad  | 2e-5         |      |       |

the capability of TernGrad to minimize the target function even with ternary gradients. A smaller dropout ratio in TernGrad can be another reason of the lower loss. Figure 4(c) simply illustrate that on average 71.32% gradients of a fully-connected layer (fc6) are ternarized to zeros.

Finally, we summarize the results of GoogLeNet in Table 3. On average, the accuracy loss is less than 2%. In TernGrad, we adopted all that hyper-parameters (except dropout ratio and weight decay) that are well tuned for the baseline [38]. Tuning these hyper-parameters could further optimize TernGrad and obtain higher accuracy.

5 Performance Model and Discussion

Our proposed TernGrad requires only three numerical levels \{-1, 0, 1\}, which can aggressively reduce the communication time. Moreover, our experiments in Section 4 demonstrate that within the same iterations, TernGrad can converge to approximately the same accuracy as its corresponding baseline. Consequently, a dramatical throughput improvement on the distributed DNN training is expected. Due to the resource and time constraint, unfortunately, we aren’t able to perform the training of more DNN models like VggNet-A [39] and distributed training beyond 8 workers. We plan to continue the experiments in our future work. We opt for using a performance model to conduct the scalability analysis of DNN models when utilizing up to 512 GPUs, with and without applying TernGrad. Three neural network models—AlexNet, GoogLeNet and VggNet-A—are investigated. In discussions of performance model, performance refers to training speed. Here, we extend the performance model that was initially developed for CPU-based deep learning systems [40] to estimate the performance of distributed GPUs/machines. The key idea is combining the lightweight profiling on single machine with analytical modeling for accurate performance estimation. In the interest of space, please refer to Appendix B for details of the performance model.
Figure 5 presents the training throughput on two different GPUs clusters. Our results show that TernGrad effectively increases the training throughput for the three DNNs. The speedup depends on the communication-to-computation ratio of the DNN, the number of GPUs, and the communication bandwidth. DNNs with larger communication-to-computation ratios (e.g. AlexNet and VggNet-A) can benefit more from TernGrad than those with smaller ratios (e.g., GoogLeNet). Even on a very high-end HPC system with InfiniBand and NVLink, TernGrad is still able to double the training speed of VggNet-A on 128 nodes as shown in Figure 5(b). Moreover, the TernGrad becomes more efficient when the bandwidth becomes smaller, such as 1Gbps Ethernet and PCI switch in Figure 5(a) where TernGrad can have $3.04 \times$ training speedup for AlexNet on 8 GPUs.

References

[1] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In Advances in Neural Information Processing Systems, pages 1223–1231. 2012.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint:1603.04467, 2016.

[3] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In International Conference on Machine Learning, pages 1337–1345, 2013.

[4] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In Advances in Neural Information Processing Systems, pages 693–701, 2011.

[5] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In OSDI, volume 14, pages 571–582, 2014.

[6] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. IEEE Transactions on Big Data, 1(2):49–67, 2015.

[7] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I Jordan. Sparknet: Training deep networks in spark. arXiv preprint:1511.06051, 2015.

[8] 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 machine learning library for heterogeneous distributed systems. arXiv preprint:1512.01274, 2015.

[9] Sixin Zhang, Anna E Choromanska, and Yann LeCun. Deep learning with elastic averaging sgd. In Advances in Neural Information Processing Systems, pages 685–693, 2015.

[10] Mu Li. Scaling Distributed Machine Learning with System and Algorithm Co-design. PhD thesis, Carnegie Mellon University, 2017.

[11] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifoski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In OSDI, volume 14, pages 583–598, 2014.

[12] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In Advances in Neural Information Processing Systems, pages 19–27, 2014.

[13] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In Advances in neural information processing systems, pages 1223–1231, 2013.

[14] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In Advances in neural information processing systems, pages 2595–2603, 2010.

[15] Xinghao Pan, Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. arXiv preprint:1702.05800, 2017.

[16] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. Staleness-aware async-sgd for distributed deep learning. In Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI’16, pages 2350–2356. AAAI Press, 2016. ISBN 978-1-57735-770-4. URL http://dl.acm.org/citation.cfm?id=3060832.3060950

[17] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00419, 2015.

[18] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In Advances in Neural Information Processing Systems, pages 2074–2082, 2016.
[19] J Park, S Li, W Wen, PTP Tang, H Li, Y Chen, and P Dubey. Faster cnns with direct sparse convolutions and guided pruning. In *International Conference on Learning Representations (ICLR)*, 2017.

[20] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in Neural Information Processing Systems*, pages 4107–4115, 2016.

[21] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[22] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[23] Joseph K Bradley, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. Parallel coordinate descent for l1-regularized loss minimization. *arXiv preprint arXiv:1105.5379*, 2011.

[24] Alham Fikri Aji and Kenneth Heafield. Sparse communication for distributed gradient descent. *arXiv preprint:1704.05021*, 2017.

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

[26] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-optimal stochastic gradient descent, with applications to training neural networks. *arXiv preprint:1610.02132*, 2017.

[27] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Prithish Narayanan. Deep learning with limited numerical precision. In *ICML*, pages 1737–1746, 2015.

[28] Rahul Garg and Rohit Khandekar. Gradient descent with sparsification: an iterative algorithm for sparse recovery with restricted isometry property. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 337–344. ACM, 2009.

[29] Léon Bottou. Online learning and stochastic approximations. *On-line learning in neural networks*, 17(9):142, 1998.

[30] Arvind Neelakantan, Luke Vilnis, Quoc V Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks. *arXiv preprint:1511.06907*, 2015.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105. 2012.

[32] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[33] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint:1412.6980*, 2014.

[34] Nitish Shirish Keskar, Dheevatsa Madigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. In *International Conference on Learning Representations*, 2017.

[35] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[36] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint:1502.03167*, 2015.

[37] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[38] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *arXiv preprint:1409.4842*, 2015.

[39] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint:1409.1556*, 2014.

[40] Feng Yan, Olatunji Ruwase, Yuxiong He, and Trishul M. Chilimbi. Performance modeling and scalability optimization of distributed deep learning systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Sydney, NSW, Australia, August 10-13, 2015, pages 1355–1364, 2015. doi: 10.1145/2783258.2783270. URL [http://doi.acm.org/10.1145/2783258](http://doi.acm.org/10.1145/2783258).
We start with strong scaling to illustrate our performance model. According to the definition of strong scaling, as mentioned in the main context of our paper, the performance model was developed based on the one initially extended to support heterogeneous system architecture. accumulated together. Here we assume all-reduce communication model, that is, each GPU communicates to their different communication patterns. Next, we illustrate how to accurately model the impacts in 2 GPUs on all machines, i.e., $N = i \times j$. We need to distinguish workers within a machine and across machines due to their different communication patterns. Next, we illustrate how to accurately model the impacts in communication and computation to capture both the benefits and overheads. 2

Communication. For GPUs within a machine, first, the gradient $g$ computed at each GPU needs to be accumulated together. Here we assume all-reduce communication model, that is, each GPU communicates
with its neighbor until all gradient $g$ is accumulated into a single GPU. The communication complexity for $i$ GPUs is $\log_2 i$. The GPU with accumulated gradient then sends the accumulated gradient to CPU for further processing. Note for each communication (either GPU-to-GPU or GPU-to-CPU), the communication data size is the same, i.e., $|g|$. Assume that within a machine, the communication bandwidth between GPUs is $C_{gwd}$ and the communication bandwidth between CPU and GPU is $C_{cwd}$, then the communication overhead within a machine can be computed as $\frac{|g|}{C_{gwd}} \log_2 i + \frac{|g|}{C_{cwd}}$. We successfully used NCCL benchmark to validate our model. For communication between machines, we also assume all-reduce communication model, so the communication time between machines are: $(C_{ncost} + \frac{|g|}{C_{gwd}}) \log_2 j$, where $C_{ncost}$ is the network latency and $C_{cwd}$ is the network bandwidth. So the total communication time is $T_{comm}(i, j, K, |g|) = \frac{|g|}{C_{gwd}} \log_2 i + \frac{|g|}{C_{cwd}} + (C_{ncost} + \frac{|g|}{C_{nwd}}) \log_2 j$. We successfully used OSU Allreduce benchmark to validate this model.

**Computation.** To estimate computation time, we rely on profiling the time for training a mini-batch of totally $K$ images on a machine with a single CPU and a single GPU. We define this profiled time as $T(1, 1, K, |g|)$. In strong scaling, each work only trains $\frac{K}{N}$ samples, so the total computation time is $T_{comp}(i, j, K, |g|) = (T(1, 1, K, |g|) - \frac{|g|}{C_{cwd}}) * \frac{1}{N}$, where $\frac{|g|}{C_{cwd}}$ is the communication time (between GPU and CPU) included in when we profile $T(1, 1, K, |g|)$.

Therefore, the time to train a mini-batch of $K$ samples is:

$$T_{strong}(i, j, K, |g|) = T_{comp}(i, j, K, |g|) + T_{comm}(i, j, K, |g|)$$

$$= (T(1, 1, K, |g|) - \frac{|g|}{C_{cwd}}) * \frac{1}{N} + \frac{|g|}{C_{gwd}} \log_2 i + \frac{|g|}{C_{cwd}} + (C_{ncost} + \frac{|g|}{C_{nwd}}) \log_2 j. \quad (15)$$

The throughput of strong scaling is:

$$T_{put, strong}(i, j, K, |g|) = \frac{K}{T_{strong}(i, j, K, |g|)}. \quad (16)$$

For weak scaling, the difference is that each worker always trains $K$ samples. So the mini-batch size becomes $N * K$. In the interest of space, we do not present the detailed reasoning here. Basically, it follows the same logic for developing the performance model of strong scaling. We can compute the time to train a mini-batch of $N * K$ samples as follows:

$$T_{weak}(i, j, K, |g|) = T_{comp}(i, j, K, |g|) + T_{comm}(i, j, K, |g|)$$

$$= T(1, 1, K, |g|) - \frac{|g|}{C_{gwd}} + \frac{|g|}{C_{gwd}} \log_2 i + \frac{|g|}{C_{cwd}} + (C_{ncost} + \frac{|g|}{C_{nwd}}) \log_2 j$$

$$= T(1, 1, K, |g|) + \frac{|g|}{C_{gwd}} \log_2 i + (C_{ncost} + \frac{|g|}{C_{nwd}}) \log_2 j. \quad (17)$$

So the throughput of weak scaling is:

$$T_{put, weak}(i, j, K, |g|) = \frac{N * K}{T_{weak}(i, j, K, |g|)}. \quad (18)$$

For ease of the discussion, we assume GPU-to-GPU communication has Dedicated Bandwidth.