STALENESS-AWARE ASYNC-SGD FOR DISTRIBUTED DEEP LEARNING

Wei Zhang, Suyog Gupta
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598, USA
{weiz,suyog}@us.ibm.com

Xiangru Lian & Ji Liu
Department of Computer Science
University of Rochester, NY 14627, USA
{lianxiangru,ji.liu.uwisc}@gmail.com

ABSTRACT

This paper investigates the effect of stale (delayed) gradient updates within the context of asynchronous stochastic gradient descent (Async-SGD) optimization for distributed training of deep neural networks. We demonstrate that our implementation of Async-SGD on a HPC cluster can achieve a tight bound on the gradient staleness while providing near-linear speedup. We propose a variant of the SGD algorithm in which the learning rate is modulated according to the gradient staleness and provide theoretical guarantees for convergence of this algorithm. Experimental verification is performed on commonly-used image classification benchmarks: CIFAR10 and Imagenet to demonstrate the effectiveness of the proposed approach. Additionally, our experiments show that there exists a fundamental tradeoff between model accuracy and runtime performance that places a limit on the maximum amount of parallelism that may be extracted from this workload under the constraints of preserving the model quality.

1 INTRODUCTION

Training of large-scale deep neural networks is often constrained by the available computational resources, motivating the development of computing infrastructure designed specifically for accelerating this workload. This includes distributing the training across several commodity CPUs (Dean et al. (2012), Chilimbi et al. (2014)), or using heterogeneous computing platforms containing multiple GPUs per computing node (Seide et al. (2014), Wu et al. (2015), Strom (2015)), or using a CPU-based HPC cluster (Gupta et al. (2015)).

A majority of these distributed implementations of neural nets employs the asynchronous version of stochastic gradient descent (Async-SGD) algorithm, pioneered by Bengio et al. (2003), in which the workload is partitioned amongst several workers (learners) that produce stochastic gradients on subsets of the training data. To achieve good scalability and runtime performance, learners are allowed to work independently of one another without any explicit synchronization. Permitting this asynchronous behavior inevitably adds "staleness" to the system wherein some of the learners compute gradients using model parameters that may be several gradient steps behind the most updated set of parameters. Prior theoretical work by Tsitsiklis et al. (1986) and Agarwal & Duchi (2011) and recent work by Lian et al. (2015) provide theoretical guarantees for convergence of stochastic optimization algorithms in the presence of stale gradient updates for convex optimization and nonconvex optimization, respectively. We find that adopting the approach of scale-out deep learning using Async-SGD gives rise to complex interdependencies between the training algorithm’s hyperparameters (such as learning rate, mini-batch size) and the distributed implementation’s design choices (such as synchronization protocol, number of learners), ultimately impacting the neural network’s accuracy and the overall system’s runtime performance. In practice, achieving good model accuracy through distributed training requires a careful selection of the training hyperparameters and much of the prior work cited above lacks enough useful insight to help guide this selection process.

The work presented in this paper intends to fill this void by undertaking a study of the interplay between the different design parameters encountered during distributed training of deep neural networks. In particular, we focus our attention on understanding the effect of stale gradient updates during distributed training and developing principled approaches for mitigating these effects. To
this end, we introduce a variant of the Async-SGD algorithm in which we keep track of the staleness associated with each gradient computation and adjust the learning rate on a per-gradient basis by simply dividing the learning rate by the staleness value. The implementation of this algorithm on a CPU-based HPC cluster with fast interconnect is shown to achieve a tight bound on the gradient staleness. We experimentally demonstrate the effectiveness of the proposed staleness-dependent learning rate scheme using commonly-used image classification benchmarks: CIFAR10 and ImageNet and show that this simple, yet effective technique is necessary for achieving good model accuracy during distributed training. Further, we build on the theoretical framework of Lian et al. (2015) and prove that the convergence rate of the staleness-aware Async-SGD algorithm is consistent with SGD: $O\left(\frac{1}{\sqrt{T}}\right)$ where $T$ is the number of gradient update steps.

Previously, Ho et al. (2013) presented a parameter server based distributed learning system where the staleness in parameter updates is bounded by forcing faster workers to wait for their slower counterparts. Perhaps the most closely related prior work is that of Chan & Lane (2014) which presented a multi-GPU system for distributed training of speech CNNs and acknowledge the need to modulate the learning rate in the presence of stale gradients. The authors proposed an exponential penalty for stale gradients and show results for up to 5 learners. However, in larger-scale distributed systems, the gradient staleness can assume values up to a few hundreds (Dean et al. (2012)) and the exponential penalty may reduce the learning rate to an arbitrarily small value, potentially slowing down the convergence.

2 SYSTEM ARCHITECTURE

2.1 ARCHITECTURE OVERVIEW

We implement a parameter server based distributed learning system, which is a superset of Downpour SGD in Dean et al. (2012), to evaluate the effectiveness of our proposed staleness-dependent learning rate modulation technique. Throughout the paper, we use the following definitions:

- $\lambda$: number of learners.
- $\mu$: mini-batch size used by each learner to produce stochastic gradients
- $\alpha$: learning rate
- Epoch: a pass through the entire training dataset.
- Timestamp: we use a scalar clock, proposed in Raynal & Singhal (1996), to represent weights timestamp $i$, starting from $i = 0$. Each weight update increments the timestamp by 1. The timestamp of a gradient is the same as the timestamp of the weight used to compute the gradient.
- $\tau_{i,l}$: staleness of the gradient from learner $l$. A learner $l$ pushes gradient with timestamp $j$ to the parameter server of timestamp $i$, where $i \geq j$. We calculate the staleness $\tau_{i,l}$ of this gradient as $i - j$. $\tau_{i,l} \geq 0$ for any $i$ and $l$.

The key functionalities of the learner and the parameter server are summarized below: Each learner performs the following sequence of steps:

1. getMinibatch: Randomly select a mini-batch of examples from the training data.
2. pullWeights: A learner pulls the current set of weights from the parameter server.
3. calcGradient: Compute stochastic gradients for the current mini-batch. We divide the gradients by the mini-batch size.
4. pushGradient: Send the computed gradients to the parameter server.

The parameter server group maintains a global view of the neural network weights and performs the functions listed below:

1. sumGradients: Receive and accumulate the gradients from the learners.
2. applyUpdate: Multiply the average of accumulated gradient by the learning rate (step length) and update the weights.
2.2 SYNCHRONIZATION PROTOCOLS

We implemented two synchronization protocols: hardsync protocol and $n$-softsync protocol.

- **Hardsync protocol**: To advance the weights’ timestamp $\theta$ from $i$ to $i+1$, each learner $l$ computes a gradient $\Delta \theta_l$ using a mini-batch size of $\mu$ and sends it to the parameter server. The parameter server averages the gradients over $\lambda$ learners and updates the weights according to equation (1), then broadcasts the new weights to all learners. The learners are forced to wait for the updated weights until the parameter server has received the gradient contribution from all the learners and finished updating the weights. This protocol guarantees that each learner computes gradients on the exactly same set of weights and ensures that the gradient staleness is 0. The hardsync protocol serves as the baseline since from the perspective of SGD optimization, and is equivalent to the synchronous SGD using batch size $\mu \lambda$.

$$g_i = \frac{1}{\lambda} \sum_{l=1}^{\lambda} \Delta \theta_l$$  \hspace{1cm} (1)

$$\theta_{i+1} = \theta_i - \alpha g_i, \hspace{1cm}$$

- **$n$-softsync protocol**: Each learner $l$ pulls the weights from the parameter server, calculates the gradients and pushes the gradients to the parameter server. The parameter server updates the weights after collecting at least $c = \lfloor (\lambda/n) \rfloor$ gradients from any of the $\lambda$ learners. Unlike hardsync, there are no explicit synchronization barriers imposed by the parameter server and the learners work asynchronously and independently. The splitting parameter $n$ can vary from 1 to $\lambda$. The $n$-softsync weight update rule is given by:

$$c = \lfloor (\lambda/n) \rfloor$$

$$g_i = \frac{1}{c} \sum_{l=1}^{c} \alpha(\tau_{i,l}) \Delta \theta_l, \hspace{0.5cm} l \in \{1, 2, \ldots, \lambda\}$$  \hspace{1cm} (2)

$$\theta_{i+1} = \theta_i - g_i,$$

where $\alpha(\tau_{i,l})$ is the gradient staleness-dependent learning rate.

2.3 IMPLEMENTATION DETAILS

We use MPI as the communication mechanism between learners and parameter servers. Parameter servers are sharded. Each learner and parameter server are 4-way threaded. During the training process, a learner pulls weights from the parameter server, starts training when the weights arrive, and then calculates gradients. Finally it pushes the gradients back to the parameter server before it can pull the weights again. Note that our proposed $n$-softsync protocol is a superset of Downpour-SGD of Dean et al. (2012) (a commonly-used asynchronous-SGD implementation), in that when $n$ is set to be $\lambda$, our implementation is equivalent to Downpour-SGD. In addition, (1) we do not “accrue” gradients at the learner so that each gradient pushed to the parameter server is always calculated out of one mini-batch size. (2) The parameter server communicates with learners via MPI blocking-send calls (i.e., pullWeights and pushGradient), that is the computation on the learner is stalled until the corresponding blocking send call is finished. By doing so, the gradients’ staleness can
be effectively bounded, as we demonstrate in Section 2.4. Note that the computation in parameter servers and learners are however concurrent (except for the learner that is communicating with the server, if any). No synchronization is required between learners and no synchronization is required between parameter server shards.

We deploy our implementation on a P775 supercomputer. Each node of this system contains four eight-core 3.84 GHz IBM POWER7 processors, one optical connect controller chip and 128 GB of memory. A single node has a theoretical floating point peak performance of 982 Gflop/s, memory bandwidth of 512 GB/s and bi-directional interconnect bandwidth of 192 GB/s.

Since memory is abundant on each computing node, our implementation does not split the neural network model across multiple nodes (model parallelism). Rather, depending on the problem size, we pack either 4 or 6 learners on each node. Learners operate on homogeneous processors and run at similar speed. In addition, fast interconnect expedites pushing gradients and pulling weights. Both of these hardware aspects help bound gradients’ staleness.

Figure 1 shows the speedup measured on CIFAR10 and ImageNet, for up to 30 learners. Our implementation achieves 22x-28x speedup for different benchmarks and different batch sizes. The reason that it does not achieve linear speedup is mainly the constraint that the parameter server communicates with learners one at a time. This can be further improved by either using MPI non-blocking-send calls or adding additional communication threads in learners.

2.4 Staleness Analysis

In the hardsync protocol, the update of weights from $\theta_i$ to $\theta_{i+1}$ is computed by aggregating the gradients calculated using weights $\theta_i$. As a result, each of the gradients $g_i$ in the $i$th step carries with it a staleness $\tau_{i,l}$ equal to 0. Figure 2 shows the measured distribution of gradient staleness for different $n$-softsync protocols when using $\lambda = 30$ learners. For the 1-softsync, the parameter server updates the current set of weights when it has received a total of 30 gradients from (any of) the learners. In this case, the staleness $\tau_{i,l}$ for the gradients computed by the learner $l$ takes values 0, 1, or 2. Similarly, the 15-softsync protocol forces the parameter server to accumulate $\lambda/15 = 2$ gradient contributions from the learners before updating the weights. On the other hand, the parameter server updates the weights after receiving a gradient from any of the learners when the 30-softsync protocol is enforced. The average staleness $\langle \tau_i \rangle$ for the 15-softsync and 30-softsync protocols remains close to 15 and 30, respectively. Empirically, we have found that a large fraction of the gradients have staleness close to $n$, and only with a very low probability ($< 0.0001$) does $\tau$ exceed $2n$. These measurements show that, in general, $\tau_{i,l} \in \{0, 1, \ldots, 2n\}$ and $\langle \tau_i \rangle \approx n$ for the $n$-softsync protocol. Clearly, the $n$-softsync protocol provides an effective mechanism for controlling the staleness of the gradients in the system.

\[\text{Figure 2: Distribution of gradient staleness for 1, 15, and 30-softsync protocols. } \lambda = 30\]
In our implementation, the parameter server uses the staleness information to modulate the learning rate on a per-gradient basis. For an incoming gradient with staleness \( \tau_{i,l} \), the learning rate is set as:

\[
\alpha_{i,l} = \frac{\alpha_0}{\tau_{i,l}} \quad \text{if} \quad \tau_{i,l} > 0
\]  

(3)

where \( \alpha_0 \) is typically set as the ‘best-known’ learning rate when using synchronous SGD. In section 4 we show experimental results comparing this staleness-dependent learning rate scheme with the case where the learning rate is kept constant at \( \alpha_0 \).

3 Theoretical Analysis

This section provides theoretical analysis of the proposed asynchronous parallel SGD algorithm in Section 2. In particular, we will show the convergence rate and how the staleness affects the convergence.

In essence, we are solving the following generic optimization problem:

\[
\min_{\theta} \quad F(\theta) := \frac{1}{N} \sum_{i=1}^{N} f_i(\theta),
\]

where \( \theta \) is the parameter vector we are pursuing, \( N \) is the number of samples, and \( f_i(\theta) \) is the loss function for the \( i^{th} \) sample. If every learner computes \( \mu \) gradients at once and the parameter server updates the parameter when it receives \( c \) mini-batches from learners, from the perspective of the parameter server, the update procedure of parameter \( \theta \) mentioned can be written as

\[
g_{i} = \frac{1}{c} \sum_{i=1}^{c} \alpha_0 \frac{1}{\tau_{i,l}} \left( \frac{1}{\mu} \sum_{s=1}^{\mu} \nabla f_{\xi_{i,s,l}}(\theta_{i-s}) \right),
\]

(4)

\[
\begin{align*}
\theta_{i+1} &= \theta_i - g_{i}, \\
&\text{calculated in the parameter server}
\end{align*}
\]

where \( \tau_{i,l} \) is the staleness of the parameter used to calculate the gradients in the \( l^{th} \) learner in step \( i \) and \( \alpha_0 \) is a constant. \( \xi_{i,s,l} \) denotes the subscript of \( f \) used to calculate the \( s^{th} \) gradient in the \( l^{th} \) learner in the \( i^{th} \) step.

To simplify the analysis, we decompose every single step in (4) to \( c \) steps by updating only one batch of gradients in each step. Then the sequence \( \{\theta_{i}\}_{i \in \mathbb{N}} \) becomes \( \{\hat{\theta}_{i}\}_{i \in \mathbb{N}} \), and \( \hat{\theta}_{c,s} = \theta_i \). Formally, it will be

\[
\begin{align*}
\hat{g}_{t} &= \frac{1}{c\mu} \alpha_0 \frac{1}{\tau_{t}} \left( \sum_{s=1}^{\mu} \nabla f_{\hat{\xi}_{t,s}}(\hat{\theta}_{t-s}) \right), \\
\hat{\theta}_{t+1} &= \hat{\theta}_{t} - \hat{g}_{t}, \\
&\text{calculated in the parameter server}
\end{align*}
\]

(5)

where \( r_t \) is the remainder of \( t/c \). \( \hat{\xi}_{t,s} \) denotes the subscript of \( f \) used to calculate the \( s^{th} \) gradient in the \( t^{th} \) step in our new formulation. One can verify that here \( t \) increases \( c \) times faster than the \( i \) in (4).

Note that the \( \tilde{\tau}_t - r_t \) in (5) is always positive.\footnote{In (5) when a mini-batch is updated into the parameter, the counter \( (t) \) will increase by 1, while in (4) the counter \( (i) \) increases by 1 every \( c \) mini-batches updated into the parameter. For example if we have 3 mini-batches with staleness 1 pushed, in (4) all \( \tau_{i,l} \) will be 1. However, in (5), if one mini-batch is updated in iteration \( t \), the staleness \( \tilde{\tau}_{t+1} \) of the other two mini-batches becomes 2, so we need to subtract the redundant part of the staleness caused by the difference in counting. Because the staleness after subtraction is exactly the original staleness in (4), it is always positive.}

We use \( \{p_t\}_{t \in \mathbb{N}} \) to denote the difference \( p_t = \tilde{\tau}_t - r_t \). It immediately follows that \( p_t = \tilde{\tau}_{t/c} - r_t \). From the Theorem 1 in Lian et al. (2015) with some

Under review as a conference paper at ICLR 2016
modification, we have the following theorem, which indicates the convergence rate and the linear speedup property of our algorithm.  

**Theorem 1.** Let $C_1, C_2, C_3, C_4$ be certain positive constants depending on the objective function $F(\theta)$. Under certain commonly used assumptions (please find in Theorem 1 in Lian et al. (2015)), we can achieve a convergence rate of

$$
\frac{1}{T} \sum_{t=1}^{T} \frac{1}{p_t} \mathbb{E}(\|\nabla F(\bar{\theta}_t)\|^2) \leq 2 \frac{\sqrt{2C_1C_2}}{\mu T} \sum_{t=1}^{T} \left( \frac{1}{p_t^2} \right),
$$

where $T$ is the total iteration number, if

$$\alpha_0 = \frac{C_1 C_2}{\mu} \left( \sum_{t=1}^{T} \frac{1}{p_t^2} \right),$$

under the prerequisite that

$$\alpha_0 \leq \frac{cC_2}{C_3 p_t \sum_{j=t-2n}^{t-1} \frac{1}{p_j}}, \quad \forall t,$$

and

$$C_3 \frac{\alpha_0}{\mu c p_t} + C_4 n \alpha_0^2 \sum_{k=1}^{2n} \frac{1}{p_{t+k}} \leq 1, \quad \forall t. \tag{9}$$

The proof is provied in the end of this section. First note that (8) and (9) can always be satisfied by selecting small enough $\alpha_0$ (or equivalently, large enough $T$). Thus, if the learning rate is appropriately chosen in our algorithm, the weighted average of the gradients (which is the LHS of (6)) is guaranteed to converge. Also note that that to achieve this convergence rate, the batch size $\mu$ cannot be too large, since from (7) we can see that a larger $\mu$ leads to a larger $\alpha_0$, which may not satisfy the prerequisites (8) and (9).

A clearer dependence between the staleness and the convergence rate can be found by taking a closer look at the RHS (6):

**Remark 1.** Note that the RHS of (6) is of the form $h(z_1, \cdots, z_T) = O\left( \sqrt{\frac{z_1^2 + z_2^2 + \cdots + z_T^2}{z_1 + z_2 + \cdots + z_T}} \right)$ by letting $z_t = \frac{1}{p_t}$. If the summation $z_1 + \cdots + z_T$ is fixed, one can verify that $h$ is minimized when $z_1 = 1 = \cdots = z_T$. Therefore our theorem suggests that a centralized distribution of staleness $p$ (or $\tau$ in (4)) leads to a better convergence rate.

Further, we have the following result by considering the ideal scenario of the staleness.  

**Remark 2.** Note that if we take $p_t$ as a constant $p$, we have

$$\frac{1}{T} \sum_{t=1}^{T} \mathbb{E}(\|\nabla F(\bar{\theta}_t)\|^2) \leq 2 \frac{\sqrt{2C_1C_2}}{\sqrt{T} \mu}. $$

Thus this convergence rate is roughly in the order of $O\left( \frac{1}{\sqrt{\mu T}} \right)$, where $T$ is the total iteration number and $\mu$ is the mini-batch size. Equivalently, a goal of $\frac{1}{T} \sum_{t=1}^{T} \mathbb{E}(\|\nabla F(\bar{\theta}_t)\|^2) \leq \epsilon$ can be achieved by having $\mu T = O(1/\epsilon^2)$. This is consistent with the convergence rate of SGD in Lian et al. (2015), which suggests a linear speedup can be achieved in our algorithm.

**Proof.** From (9) we have

$$C_3 \frac{\alpha_0}{\mu c p_t} + C_4 n \alpha_0^2 \sum_{k=1}^{2n} \frac{1}{p_{t+k}} \leq 1, \forall t.$$
With (8) we have

\[
\frac{\alpha_0^3}{\mu c^2 p^2 t^2} C_2 \geq \frac{\alpha_0^3}{c^3 p \mu} C_3 \sum_{j=t-2n}^{t-1} \frac{1}{p_j^2}, \forall t. 
\] (10)

Note that the upperbound of the staleness is \(2n\) in our setting. Then it follows from Theorem 1 in Lian et al. (2015) that

\[
\frac{1}{\sum_{t=1}^{T} 1/p_t} \sum_{t=1}^{T} \mathbb{E}(\|\nabla F(\hat{\theta}_t)\|^2) 
\leq C_1 + \sum_{t=1}^{T} \left( \frac{\alpha_0^2}{\mu^2 c^2 p_t} C_2 + \frac{\alpha_0^3}{c^3 p \mu} C_3 \sum_{j=t-2n}^{t-1} \frac{1}{p_j^2} \right) 
\leq \frac{C_1 + \sum_{t=1}^{T} \frac{\alpha_0^2}{c^2 p_t} C_2}{\sum_{t=1}^{T} \frac{1}{p_t}} 
\leq 2 \sqrt{\frac{C_1 c}{\mu} \sum_{t=1}^{T} \left( \frac{2}{\mu c p_t} C_2 \right) \sum_{t=1}^{T} \frac{1}{p_t}} 
\leq 2 \sqrt{\frac{C_1 c}{\mu} \sum_{t=1}^{T} \left( \frac{1}{p_t} \right)} 
\leq \frac{2 \sqrt{C_1 c}}{\mu} \sum_{t=1}^{T} \frac{1}{p_t},
\]

completing the proof. \(\Box\)

4 Experimental Results

For experimental evaluation of the learning rate scheme proposed in Section 2.4 we employ two commonly-used image classification benchmark datasets: CIFAR10 [Krizhevsky & Hinton (2009)] and ImageNet [Russakovsky et al. (2015)]. We perform distributed training of the neural networks for each of these tasks using the \(n\)-softsync protocol for different values of \(n\). This allows us to systematically observe the effect of stale gradients on the convergence properties. For each of the benchmarks, we perform two sets of experiments: (a) setting learning rate fixed to the best-known learning rate for synchronous SGD, \(\alpha = \alpha_0\), and (b) tuning the learning rate on a per-gradient basis depending on the gradient staleness \(\tau\), \(\alpha = \alpha_0/\tau\). It is important to note that when \(\alpha = \alpha_0\) and \(n = \lambda\) (number of learners) in \(n\)-softsync protocol, our implementation is equivalent to the Downpour-SGD of Dean et al. (2012). Also, we train using the hardsync protocol since it guarantees zero gradient staleness and achieves the best model accuracy, albeit at the cost of poor runtime performance.

The CIFAR10 dataset comprises of a total of 60,000 RGB images of size 32 \(\times\) 32 pixels partitioned into the training set (50,000 images) and the test set (10,000 images). Each image belongs to one of the 10 classes, with 6000 images per class. For this dataset, we construct a deep convolutional neural network (CNN) with 3 convolutional layers each followed by a subsampling/pooling layer. The output of the 3\textsuperscript{rd} pooling layer connects, via a fully-connected layer, to a 10-way softmax output layer that generates a probability distribution over the 10 output classes. This neural network architecture closely mimics the CIFAR10 model (cifar10_full.prototxt) available as a part of the open-source Caffe deep learning package [Jia et al. (2014)]. The total number of trainable parameters in this network are \(\sim 90\) K, resulting in the model size of \(\sim 350\) kB. As a data preprocessing step, the per-pixel mean is computed over the entire training dataset and subtracted from the input to the neural network. The neural network is trained using momentum-accelerated mini-batch SGD with a batch size of 128 and momentum set to 0.9.

When using a single learner, the mini-batch size is set to 128 and training for 140 epochs using momentum accelerated SGD (momentum = 0.9) results in a model that achieves \(\sim 18\%\) misclassification error rate on the test dataset. The base learning rate \(\alpha_0\) is set to 0.001 and reduced by
Figure 3: Results on CIFAR10 benchmark: Training error, test error for different $n$-softsync protocols and learning rate set as $\alpha_0$ (top) or staleness-dependent learning rate modulation scheme of equation 3 (bottom). Hardsync, Downpour-SGD shown as baseline for comparison. $\lambda = 30$, $\mu = 4$. Note that when $\alpha = \alpha_0$ and $n = \lambda$, $n$-softsync protocol is equivalent to the Downpour-SGD.

For further experimental validation, we consider the natural images dataset used as a part of the 2012 edition of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC 2012). The training set is a subset of the hand-labeled ImageNet database and contains 1.2 million images. The validation dataset has 50,000 images. Each image maps to one of the 1000 non-overlapping object categories. The images are converted to a fixed resolution of $256 \times 256$ to be used as input to a deep convolutional neural network. For this dataset, we consider the neural network architecture introduced in Krizhevsky et al. (2012) consisting of 5 convolutional layers and 3 fully-connected layers. The last layer outputs the probability distribution over the 1000 object categories. In all, the neural network has $\sim 72$ million trainable parameters and the total model size is 289 MB. Similar to the CIFAR10 benchmark, per-pixel mean computed over the entire training dataset is subtracted from the input image feeding into the neural network. To prevent overfitting, a weight regularization penalty of 0.0005 is applied to all the layers in the network and a dropout of 50% is applied to the 1st and 2nd fully-connected layers. The initial learning rate $\alpha_0$ is set equal to 0.01 and reduced by a factor of 5 after the 20th and again after the 30th epoch.

With a single learner, training using a mini-batch size of 256, momentum 0.9 results in a top-1 error of 42.56% and top-3 error of 19.18% on the validation set at the end of 35 epochs. Next,
Figure 4: Results on ImageNet benchmark: Training error, top-1 validation error for different $n$-softsync protocols and learning rate set as $\alpha_0$ (top) or staleness-dependent learning rate modulation scheme of equation (bottom). Hardsync, Downpour-SGD shown as baseline for comparison. $\lambda = 18$, $\mu = 16$. Note that when $\alpha = \alpha_0$ and $n = \lambda$, $n$-softsync protocol is equivalent to the Downpour-SGD.

Table 1: Test (top-1 validation) error on CIFAR10 (ImageNet) on a model trained for 140 (35) epochs. NC = no convergence. $n$-softsync with $n = \lambda$ used in these experiments when $\lambda > 1$. Note that the column $\alpha = \alpha_0$ is equivalent to the Downpour-SGD.

---

For this particular benchmark, we found it necessary to use an adaptive learning rate method (AdaGrad [Duchi et al. (2011); Dean et al. (2012)]) to improve the stability during distributed training. Also, to improve convergence we adopt the strategy of warmstarting (Senior et al. (2013)) the training procedure by initializing the network’s weights from a model that has been trained using hardsync for 1 epoch.
Table 1 lists the time/epoch and test error achieved on CIFAR10 and ImageNet benchmarks for different configurations of $\mu$ and $\lambda$. In all cases, modulating the learning rate proves to be necessary in achieving convergence during distributed training of these models. As observed previously in Figure 1, reducing $\mu$, the mini-batch size per learner, results in an increase in time/epoch. This is primarily due to the combined effect of degraded computational efficiency at each learner and more frequent communication between parameter servers and learners. Also note that in spite of using the staleness-dependent learning rate scheme, for a fixed number of learners $\lambda$, the test error increases monotonically with the mini-batch size $\mu$. At the same time, distributed training using Async-SGD and $\lambda$ learners can achieve model quality similar to synchronous SGD (and within the same number of training epochs) so long as $\mu \lambda$ product is approximately the same as the mini-batch size used for synchronous SGD.

These results help define a principled approach for distributed training of neural networks: the mini-batch size per learner should be reduced as more learners are added to the system in way that keeps $\mu \lambda$ product constant. In addition, the learning rate should be modulated to account for stale gradients. These results reveal the scalability limits under the constraints of preserving the model accuracy. Since the smallest possible mini-batch size is 1, the maximum number of learners is bounded. This upper bound on the maximum number of learners can be relaxed if an inferior model is acceptable. Alternatively, it may be possible to reduce the test error for higher $\mu \lambda$ by training for a greater number of epochs. However, in such a scenario adding more learners to the system gives only a diminishing improvement in the overall training time. From machine learning and optimization perspective, this points to an interesting research direction that seeks to design optimization algorithm and learning strategies that perform well with large mini-batch sizes.

5 CONCLUSION

In this paper, we study how to effectively counter gradient staleness in a distributed implementation of the asynchronous SGD algorithm. In summary, the key contributions of this work are:

- We prove that by using our proposed staleness-dependent learning rate scheme, asynchronous-SGD can converge at the same rate as synchronous-SGD.
- We quantify the distribution of gradient staleness in our framework and demonstrate the effectiveness of the learning rate strategy on standard benchmarks (CIFAR10 and ImageNet). The experimental results show that our implementation achieves close to linear speedup for up to 30 learners while maintaining the same convergence rate in spite of the varying degree of staleness in the system and across vastly different data and model sizes.
- We find that to achieve a given model accuracy during distributed training, it is necessary to reduce the mini-batch size as the number of learners is increased. These results suggests an upper limit exits for the level of parallelism that can be extracted from this workload, and consequently a need for algorithms that permit training over larger mini-batch sizes.

REFERENCES

Agarwal, Alekh and Duchi, John C. Distributed delayed stochastic optimization. In Advances in Neural Information Processing Systems, pp. 873–881, 2011.

Bengio, Yoshua, Ducharme, Réjean, Vincent, Pascal, and Jauvin, Christian. A neural probabilistic language model. 3:1137–1155, 2003.

Chan, William and Lane, Ian. Distributed asynchronous optimization of convolutional neural networks. In Fifteenth Annual Conference of the International Speech Communication Association, 2014.

Chilimbi, Trishul, Suzue, Yutaka, Apacible, Johnson, and Kalyanaraman, Karthik. Project Adam: Building an efficient and scalable deep learning training system. OSDI’14, pp. 571–582. USENIX Association, 2014. ISBN 978-1-931971-16-4.

Dean, Jeffrey, Corrado, Greg S., Monga, Rajat, Chen, Kai, Devin, Matthieu, Le, Quoc V., Mao, Mark Z., Ranzato, Marc Aurelio, Senior, Andrew, Tucker, Paul, Yang, Ke, and Ng, Andrew Y. Large scale distributed deep networks. In NIPS, 2012.
Duchi, John, Hazan, Elad, and Singer, Yoram. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

Gupta, S., Zhang, W., and Milthorpe, J. Model Accuracy and Runtime Tradeoff in Distributed Deep Learning. *ArXiv e-prints*, September 2015.

Ho, Qirong, Cipar, James, Cui, Henggang, Lee, Seunghak, Kim, Jin Kyu, Gibbons, Phillip B, Gibson, Garth A, Ganger, Greg, and Xing, Eric P. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in Neural Information Processing Systems*, pp. 1223–1231, 2013.

Jia, Yangqing, Shelhamer, Evan, Donahue, Jeff, Karayev, Sergey, Long, Jonathan, Girshick, Ross, Guadarrama, Sergio, and Darrell, Trevor. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

Krizhevsky, Alex and Hinton, Geoffrey. Learning multiple layers of features from tiny images. *Computer Science Department, University of Toronto, Tech. Rep*, 1(4):7, 2009.

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Lian, Xiangru, Huang, Yijun, Li, Yuncheng, and Liu, Ji. Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization. In *NIPS*, 2015.

Raynal, Michel and Singhal, Mukesh. Logical time: Capturing causality in distributed systems. *Computer*, 29(2):49–56, February 1996. ISSN 0018-9162. doi: 10.1109/2.485846.

Russakovsky, Olga, Deng, Jia, Su, Hao, Krause, Jonathan, Satheesh, Sanjeev, Ma, Sean, Huang, Zhiheng, Karpathy, Andrej, Khosla, Aditya, Bernstein, Michael, Berg, Alexander C., and Fei-Fei, Li. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, pp. 1–42, April 2015. doi: 10.1007/s11263-015-0816-y.

Seide, Frank, Fu, Hao, Droppo, Jasha, Li, Gang, and Yu, Dong. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

Senior, Alan, Heigold, Georg, Ranzato, Marc’Aurelio, and Yang, Ke. An empirical study of learning rates in deep neural networks for speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 6724–6728. IEEE, 2013.

Strom, Nikko. Scalable distributed dnn training using commodity gpu cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

Tsitsiklis, John N, Bertsekas, Dimitri P, Athans, Michael, et al. Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE transactions on automatic control*, 31(9):803–812, 1986.

Wu, Ren, Yan, Shengen, Shan, Yi, Dang, Qingqing, and Sun, Gang. Deep image: Scaling up image recognition. *CoRR*, abs/1501.02876, 2015. URL http://arxiv.org/abs/1501.02876