Dynamic Backup Workers for Parallel Machine Learning

Chuan Xu, Giovanni Neglia, Nicola Sebastianelli

Dynamic Backup Workers for Parallel Machine Learning

Chuan Xu, Giovanni Neglia, Nicola Sebastianelli

Inria, Université Côte d'Azur, Sophia Antipolis, France

Abstract

The most popular framework for distributed training of machine learning models is the (synchronous) parameter server (PS). This paradigm consists of n workers, which iteratively compute updates of the model parameters, and a stateful PS, which waits and aggregates all updates to generate a new estimate of model parameters and sends it back to the workers for a new iteration. Transient computation slowdowns or transmission delays can intolerably lengthen the time of each iteration. An efficient way to mitigate this problem is to let the PS wait only for the fastest $n - b$ updates, before generating the new parameters. The slowest b workers are called *backup workers*. The correct choice of the number b of backup workers depends on the cluster configuration and workload, but also (as we show in this paper) on the hyper-parameters of the learning algorithm and the current stage of the training. We propose DBW, an algorithm that dynamically decides the number of backup workers during the training process to maximize the convergence speed at each iteration. Our experiments show that DBW 1) removes the necessity to tune b by preliminary time-consuming experiments, and 2) makes the training up to a factor 3 faster than the optimal static configuration.

Keywords: Machine learning, parameter server, gradient methods, distributed systems, stragglers.

1. Introduction

Already in 2014, state-of-the-art machine learning models counted hundreds of billions of parameters and required processing hundreds of terabytes through thousands of cores [1]. As models and datasets keep becoming larger, the need for efficient distributed solutions becomes even more urgent. These distributed systems are different from those used for traditional applications like transaction processing or data analytics, because of statistical and algorithmic characteristics unique to ML programs, like error tolerance, structural dependencies, and non-uniform convergence of parameters [2]. Currently, their operation requires a number of ad-hoc choices and time-consuming tuning through trial and error, e.g., to decide how to distribute ML programs over a cluster or how to bridge ML computation with inter-machine communication. For this reason, significant research effort (also from the networking community [3, 4, 5, 6, 7, 8, 9]) is devoted to design adaptive algorithms for a more effective use of computing resources for ML training.

For distributed ML training, there are two popular frameworks, the parameter server (PS) [10] and AllReduce (AR) [11, 12]. In PS, a stateful parameter server maintains the current version of the model parameters and broadcasts them to the workers (computing units e.g., GPUs). Every worker then computes "delta" updates of the parameters, e.g., through a gradient descent step. These updates are then aggregated by the PS in a synchronized way and combined with its current state to produce a new estimate of the optimal parameter vector. As the server may become a communication bottleneck, aggregation can be implemented in a distributed way through an AllReduce collective operation [13]. For example, in Ring-AllReduce [14] with n workers, $2(n-1)$ synchronized communications are required with $\mathcal{O}(1)$ data transmitted per worker. However, both the PS and AR are sensitive to *stragglers* [15, 16, 17, 18, 19], i.e., "*workers that are* ***randomly***

Email addresses: chuan.xu@inria.fr (Chuan Xu), giovanni.neglia@inria.fr (Giovanni Neglia), nicola.sebastianelli@inria.fr (Nicola Sebastianelli)

slowed down due to resource contention, background OS activities, garbage collection, and (for ML tasks) stopping criteria calculations" [3].

To mitigate the stragglers problem, coding techniques have been proposed both for PS [20, 21, 22, 23, 24, 25, 18] and AR [26, 19] frameworks. The main idea behind is that each worker performs some additional computation and codes its update in an opportune way, so that only a subset of the tasks is needed to recover the full information and to proceed to the next iteration. Hence, the system does not need to wait for the stragglers. Coding techniques are particularly helpful when data distribution across workers is heterogeneous [27] as it happens in federated learning [28]. In a cluster, all workers have access to the whole dataset or to a random sample of it, hence the advantage of coding is significantly reduced, and when computation time is larger than communication time, coding is even less beneficial [20]. In these settings, the additional overhead introduced by coding techniques may not be justified.

Alternative approaches to deal with stragglers are based on load-aware and interference-aware resource scheduling to monitor and avoid stragglers [29, 6]. These techniques are effective only if stragglers are *persistent*, i.e., the same workers are slow over a relatively long time period, but straggler effects often occur over short timescale.

Another possibility is to relax the full synchronization requirement avoiding to collect information from all workers before computing the new model parameters. One solution is to let the PS operate asynchronously, updating the parameter vector as soon as it receives the result of a single worker [30, 31]. While this approach increases system throughput (parameter updates per time unit), workers operate in general on stale versions of the parameter vector slowing and, in some cases, even preventing convergence to the optimal model [32]. Another solution is to apply decentralized learning methods, where there is no central server, but workers communicate only with their neighbours on an opportune communication graph [33, 34, 35, 36]. When the graph is sparse and the stragglers behave in a non-persistent way, such methods work well enjoying high system throughput and guaranteed convergence [37, 38, 39]. However, persistent stragglers can still slow down dramatically the throughput performance.

In the PS architecture, a simple solution to mitigate the effect of stragglers without jeopardizing convergence, is to rely on backup workers [40, 27]: instead of waiting for the updates from all workers (say it n), the PS waits for the fastest k out of n updates to proceed to the next iteration. The remaining $b \triangleq n - k$ workers are called backup workers.[1] Experiments on Google cluster with $n = 100$ workers show that a few backup workers (4–6) can reduce the training time by 30% in comparison to the synchronous PS and by 20% in comparison to the asynchronous PS [40].

The number of backup workers b has a double effect on the convergence speed. The larger b is, the faster each iteration is, because the PS needs to wait less inputs from the workers. At the same time, the PS aggregates less information, so the model update is noisier and more iterations are required to converge. Currently, the number of backup workers is configured manually through some experiments, before the actual training process starts. However, the optimal static setting is highly sensitive to the cluster configuration (e.g., GPU performances and their connectivity) as well as to its instantaneous workload. Both cluster configuration and workload may be unknown to the users (specially in a virtualized cloud setting) and may change as new jobs arrive/depart from the cluster. Moreover, in this paper we show that the choice of the number of backup workers 1) should depend also on hyper-parameters[2] like the batch size, and 2) should change during the training itself (!) as the loss function approaches a (local) minimum. Therefore, the static configuration of backup workers does not only require time-consuming experiments, but is particularly inefficient and fragile.

In this paper we propose the algorithm DBW (for Dynamic Backup Workers) that dynamically adapts the number of backup workers during the training process without prior knowledge about the cluster or

[1] We stick to the name used in the original paper [40], even if it is someway misleading, because backup workers do not replace other workers when needed. In fact all workers operate identically, and who are the backup workers change from one iteration to the other depending on their execution times at that specific iteration.

[2] An hyper-parameter is a parameter of the learning algorithm (and not of the model), but it can still influence the final model learned.

the optimization problem. Our algorithm identifies the sweet spot between the two contrasting effects of b (reducing the duration of an iteration and increasing the number of iterations for convergence), by maximizing at each iteration the decrease of the loss function *per time unit*.

This paper extends our conference submission [41] and is organized as follows. Sect. 2 provides relevant background and introduces the notation. Sect. 3 illustrates the different components of our algorithm DBW with their respective preliminary assessments. DBW is then evaluated on ML problems in Sect. 4. The results show that DBW is robust to different cluster environments and different hyper-parameters' settings. DBW does not only remove the necessity to configure an additional parameter (b) through costly experiments, but also reduce the training time by a factor as large as 3 in comparison to the best static configuration. Sect. 5 concludes the paper and discusses future research directions. The code of our implementation is available online [42].

2. Background and notation

Given a dataset $\mathbb{X} = \{x_l, l = 1, \dots S\}$, the training of ML models usually requires to find a parameter vector $\boldsymbol{w} \in \mathbb{R}^d$ minimizing a loss function:

$$\underset{\boldsymbol{w}\in\mathbb{R}^d}{\text{minimize}} \quad F(\boldsymbol{w}) \triangleq \frac{1}{S}\sum_{l=1}^{S} f(x_l, \boldsymbol{w}), \tag{1}$$

where $f(x_l, \boldsymbol{w})$ is the loss of the model $\boldsymbol{w}$ on the datapoint x_l. For example, in supervised learning, each point of the dataset is a pair $x_l = (\chi_l, y_l)$, consisting of an input object χ_l and a desired output value y_l. In the standard linear regression method $\chi_l \in \mathbb{R}^d$, $y_l \in \mathbb{R}$, the input-output function is a linear one ($\hat{y}_l = \chi_l^\intercal \boldsymbol{w}$) and the loss function is the mean squared error $(\chi_l^\intercal \boldsymbol{w} - y_l)^2$. More complex models like neural networks look for an input-output mapping in a much larger and more flexible family of functions, but they are trained solving an optimization problem like (1).

The standard way to solve Problem 1 is to use an iterative gradient method. Let n be the number of workers (e.g., GPUs) available. In a synchronous setting without backup workers, at each iteration t the PS sends the current estimate of the parameter vector $\boldsymbol{w}_t$ to all workers. Each worker computes then a stochastic gradient on a random mini-batch of size B ($\leq S$) drawn from its local dataset. We assume each worker has access to the complete dataset $\mathbb{X}$ as it is resonable in the cluster setting that we consider. Each worker sends the stochastic gradient back to the PS. We denote by $\boldsymbol{g}_{i,t}$ the i-th worker gradient received by the PS at iteration t, i.e.,

$$\boldsymbol{g}_{i,t} = \frac{1}{B}\sum_{x\in\mathbb{B}_i} \nabla f(x, \boldsymbol{w}_t), \tag{2}$$

and $\mathbb{B}_i \subseteq \mathbb{X}$ is the random minibatch of size B on which the gradient has been computed. Once n gradients are received, the PS computes the average gradient

$$\boldsymbol{g}_t = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{g}_{i,t},$$

and updates the parameter vector as follows:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \boldsymbol{g}_t, \tag{3}$$

where $\eta > 0$ is called the learning rate.

When b backup workers are used [40], the PS only waits for the first $k = n - b$ gradients and then evaluates the average gradient as

$$\boldsymbol{g}_t = \frac{1}{k}\sum_{i=1}^{k} \boldsymbol{g}_{i,t}. \tag{4}$$

In our dynamic algorithm (Sect. 3), the value of k is no longer static but changes in an adaptive manner from one iteration to the other, ensuring faster convergence speed. We denote by k_t the number of gradients of $\boldsymbol{w}_t$ the PS needs to wait for at iteration t, and by $T_{i,t}$ the time interval between the update of the parameter vector $\boldsymbol{w}_t$ at the PS and the reception of the i-th gradient $\boldsymbol{g}_{i,t}$.

The general backup-workers scheme can be implemented in different ways with quite different performance. When implementing the backup workers scheme, there are two general ways to synchronize the PS and the workers: either the PS *pushes* the updated parameter vector to workers or the workers *pull* the most updated parameter vector from the PS.

Pull (Pl). Whenever available to perform a new computation, a worker pulls the most updated parameter vector from the PS. Google's framework for distributed ML—TensorFlow 1.x [43]—implements Pl through a shared blocking FIFO queue of size n where the PS enqueues n copies of tokens indicating the corresponding iteration number. Whenever a worker becomes idle, it dequeues the token from the queue and retrieves the parameter vector directly from the PS.[3]

Push & Interrupt (PsI). After the PS updates the new parameter vector $\boldsymbol{w}$, it pushes $\boldsymbol{w}$ to all workers, which interrupt any ongoing computation to start computing a new gradient at $\boldsymbol{w}$. Interrupts can be implemented in different ways. For example, in [44, Algo. 2], the main thread at each worker creates a specific thread for each gradient computation and keeps listening for a new parameter vector. Once the worker receives the new one from PS, the computing thread is killed. However, the overhead of online creating/destroying threads is not negligible since it requires run-time memory allocation and de-allocation, which may even slow down the system [45]. In [46], the same thread performs the computation but periodically checks for new parameter vectors from the PS. When the worker receives a new parameter vector, it stops its ongoing computation. The performance of this interrupt mechanism depends on how often workers listen for messages from PS.

Push & Wait (PsW). The PS pushes the new parameter vector to each worker as in PsI, but the worker completes its current computation before dequeueing the most recent parameter vector from a local queue. PsW can be easily implemented using MPI non-blocking communication package [18] or the FIFO queue provided in TensorFlow [47].

Our algorithm works with any of the variants listed above, with minor adaptations. We have implemented and tested it both with PsI and PsW in the PyTorch framework [48]. Results are similar, therefore, in what follows, we refer only to PsW.

To the best of our knowledge, there are two other proposals to dynamically adapt the number of backup workers [44, 27]. Both consider a PsI approach. In [44] the PS uses a deep neural network to predict the time $T_{k,t}$ needed to collect $k = 1, 2, \dots n$ new gradients. It then greedily chooses k_t as the value that maximizes $k/T_{k,t}$. This neural network for time series forecasting needs itself to be trained in advance for each cluster and each ML model to be learned. No result is provided in [44] about the duration of this additional training phase or its sensitivity to changes in the cluster and/or ML models. Our algorithm DBW also selects k_t to maximize a similar ratio, but 1) replaces the numerator by the expected decrease of the loss function, 2) uses a simple estimator for $T_{k,t}$, that does not require any preliminary training. Moreover, results in [44] do not show a clear advantage of the proposed mechanism in comparison to the static setting suggested in [40] (see [44, Fig. 4]). Our experiments in Sect. 4 confirm that indeed considering a gain proportional to k as in [44] is too simplistic (and leads to worse results than DBW). The recent paper [27] proposes AdaSync that selects k_t to minimize the average expected squared norm of the gradients over a time horizon. AdaSync relies on an upper bound for the expected squared norm of the gradients and analytical formulas for $T_{k,t}$ for specific distributions of the computation times—they only develop the case for shifted exponential random variables. Finding the optimal k_t would require to know or estimate at run-time some quantities like the

[3] We describe what appears to be an inefficient implementation. The parameter vector retrieved by the worker may correspond to a more recent iteration than what indicated in the token. Nevertheless, the corresponding gradient is still associated to the old iteration and then will be discarded at the PS. The worker may start then a computation that is already known to be useless!

Lipschitz constant or noise variance. AdaSync instead determines k_t by solving an approximate quadratic equation that only depends on the current loss. On the contrary, DBW estimates the different quantities online without prior information about the distribution of the computation times, and it is then able to adapt to changes in the cluster, e.g., due to dynamic resource allocation (Sect. 4.3). When computation times are distributed according to a shifted exponential distribution, our experiments show that DBW trains faster than AdaSync when computation variability is small (Sect. 4.4).

Our approach to estimate the loss decrease as a function of k is inspired by the work [49] which evaluates the loss decrease as a function of the batch size. In fact, aggregating k gradients, each computed on a mini-batch of B samples, is almost equivalent to compute a single gradient on a mini-batch of kB samples.

While our algorithm adapts the number of backup workers b given an available pool of n workers, the authors of [4] proposes a reinforcement learning algorithm to adapt n in order to minimize the training time under a budget constraint. This algorithm and DBW are then complementary: once selected n with the approach in [4], DBW can be applied to tune the number of backup workers.

3. Dynamic backup workers

The rationale behind our algorithm DBW is to adaptively select k_t in order to maximize $\frac{F(\boldsymbol{w}_t)-F(\boldsymbol{w}_{t+1})}{T_{k,t}}$, i.e., to greedily maximize the decrease of the empirical loss per time unit. We decide k_t just after the update of $\boldsymbol{w}_t$.[4] In the following subsections, we detail how both numerator and denominator can be estimated, and how they depend on k. The notation is listed in Table 1.

t	iteration number
n	number of workers
$\boldsymbol{w}_t$	parameter vector at iteration t
F	(global) loss function to minimize
B	batch size
η	learning rate
L	Lipschitz smoothness constant of F
$\boldsymbol{g}_{i,t}$	i^{th} stochastic gradient PS receives at iteration t
$\mathbb{V}(\boldsymbol{g}_{i,t})$	variance of $\boldsymbol{g}_{i,t}$
k_t	number of stochastic gradients PS waits for at iteration t
$\boldsymbol{g}_t$	average gradient at iteration t
$\mathcal{G}_{k,t}$	gain (expected loss decrease) if PS receives k gradients
$T_{k,t}$	time between $\boldsymbol{w}_t$ update and $\boldsymbol{g}_{k,t}$ reception at PS
$\mathtt{t}_{h,i,t}$	time between $\boldsymbol{w}_t$ update and $\boldsymbol{g}_{i,t}$ reception at PS
	when PS has waited for h gradients at iteration $t-1$
$\mathcal{T}_{h,k}$	random variable from which $\mathtt{t}_{h,k,t}$ values are assumed to be sampled
$\mathbb{T}_{h,k,t}$	set of $\mathtt{t}_{h,k,t'}$ samples available up to iteration t

Table 1: Notation

3.1. Empirical Loss Decrease

We assume that the empirical loss function $F(\boldsymbol{w})$ is L-smooth, i.e., it exists a constant L such that

$$\|\nabla F(\boldsymbol{w}') - \nabla F(\boldsymbol{w}'')\| \leq L\|\boldsymbol{w}' - \boldsymbol{w}''\|, \forall \boldsymbol{w}', \boldsymbol{w}''. \tag{5}$$

Smoothness is a standard assumption in convergence results of gradient methods (see for example [50, 51]). In our experiments we show DBW reduces the convergence time also when the loss is not a smooth function. From (5) and (3) it follows (see [51, Sect. 4.1] for a proof):

[4] It is possible in principle to refine the choice of k_t upon the arrival of the first gradients of $\boldsymbol{w}_t$.

$$\Delta F_t \triangleq F(\boldsymbol{w}_t) - F(\boldsymbol{w}_{t+1}) \geq \eta \nabla F(\boldsymbol{w}_t)^\intercal \boldsymbol{g}_t - \frac{L\eta^2}{2}\|\boldsymbol{g}_t\|^2. \tag{6}$$

In order to select k_t, DBW uses this lower bound as a proxy for the loss decrease. We note, however, that $\boldsymbol{g}_t$ depends on the value of k_t (see (4)) and the random mini-batches drawn at the workers. So at the moment to decide for k_t, $\boldsymbol{g}_t$ is a random variable. We consider then the expected value (over the possible choices for the mini-batches) of the right-hand side of (6). We call it the *gain* and denote by $\mathcal{G}_{k,t}$, i.e.,:

$$\mathcal{G}_{k,t} \triangleq \mathbb{E}\left[\eta \nabla F(\boldsymbol{w}_t)^\intercal \boldsymbol{g}_t - \frac{L\eta^2}{2}\|\boldsymbol{g}_t\|^2\right]. \tag{7}$$

Each stochastic gradient is an unbiased estimator of the full gradient, then $\mathbb{E}[\boldsymbol{g}_t] = \nabla F(\boldsymbol{w}_t)$. Moreover, for any random variable X, it holds $\mathbb{E}[X^2] = \mathbb{E}[X]^2 + \mathrm{Var}(X)$. Applying this relation to each of the component of the vector $\boldsymbol{g}_t$, and then summing up, we obtain:

$$\mathbb{E}[\|\boldsymbol{g}_t\|^2] = \|\nabla F(\boldsymbol{w}_t)\|^2 + \mathbb{V}(\boldsymbol{g}_{i,t})/k, \tag{8}$$

where $\mathbb{V}(\boldsymbol{g}_{i,t})$ denotes the sum of the variances of the different components of $\boldsymbol{g}_{i,t}$, i.e., $\mathbb{V}(\boldsymbol{g}_{i,t}) \triangleq \sum_{l=1}^{d} \mathrm{Var}([\boldsymbol{g}_{i,t}]_l)$. Notice that $\mathbb{V}(\boldsymbol{g}_{i,t})$ does not depend on i, because each worker has access to the complete dataset. Then, combining (7) and (8), $\mathcal{G}_{k,t}$ can be rewritten as

$$\mathcal{G}_{k,t} = \left(\eta - \frac{L\eta^2}{2}\right)\|\nabla F(\boldsymbol{w}_t)\|^2 - \frac{L\eta^2}{2}\frac{\mathbb{V}(\boldsymbol{g}_{i,t})}{k}. \tag{9}$$

Equation (9) shows that the gain increases as k increases. This corresponds to the fact that the more gradients are aggregated at the PS, the closer the stochastic gradient $-\boldsymbol{g}_t$ is to its expected value $-\nabla F(\boldsymbol{w}_t)$, i.e., to the steepest descent direction for the loss function. We also remark that the gain sensitivity to k depends on the relative ratio of $\mathbb{V}(\boldsymbol{g}_{i,t})$ and $\|\nabla F(\boldsymbol{w}_t)\|^2$, that keeps changing during the training (see for example Fig. 1). Correspondingly, we can expect that the optimal value of k will vary during the training process, even when computation and communication times do not change in the cluster. Experiments in Sect. 4 confirm this point.

Computing the exact value of $\mathcal{G}_{k,t}$ would require the workers to process the whole dataset, leading to much longer iterations. We want rather to evaluate $\mathcal{G}_{k,t}$ with limited overhead for the workers. In what follows, we discuss how to estimate $\|\nabla F(\boldsymbol{w}_t)\|^2$, $\mathbb{V}(\boldsymbol{g}_{i,t})$, and L to approximate $\mathcal{G}_{k,t}$ in (9). We first provide estimators that use information available *at the end* of iteration t, i.e., after k_t has been selected and the k_t fastest gradients have been received. Then, we build from these estimators new ones, that can be computed *at the beginning* of the iteration t and then can be used to select k_t. Given a quantity θ_t to be estimated at iteration t, we denote the first estimator as $\widehat{\theta}_t^{\,+}$ and the second one as $\widehat{\theta}_t$.

We start by estimating $\mathbb{V}(\boldsymbol{g}_{i,t})$ through the usual unbiased estimator for the variance:

$$\widehat{\mathbb{V}(\boldsymbol{g}_{i,t})}^{+} = \sum_{l=1}^{d} \frac{1}{k_t - 1} \sum_{j=1}^{k_t} \left([\boldsymbol{g}_{j,t} - \boldsymbol{g}_t]_l\right)^2. \tag{10}$$

It is possible to have more precise estimates (even when $k_t = 1$), if each worker can estimate $\mathbb{V}(\nabla f(x, \boldsymbol{w}_t))$ from its mini-batch. As GPUs' low-level APIs do not provide access to such information, we do not further develop the corresponding formulas here.

Next, we study the estimator of $\|\nabla F(\boldsymbol{w}_t)\|^2$. First, we can trivially use $\|\boldsymbol{g}_t\|^2$ to estimate $\mathbb{E}[\|\boldsymbol{g}_t\|^2]$, i.e., $\widehat{\mathbb{E}[\|\boldsymbol{g}_t\|^2]}^{+} = \|\boldsymbol{g}_t\|^2$. Since $\|\nabla F(\boldsymbol{w}_t)\|^2 = \mathbb{E}[\|\boldsymbol{g}_t\|^2] - \mathbb{V}(\boldsymbol{g}_{i,t})/k_t$ (from (8)), we can estimate $\|\nabla F(\boldsymbol{w}_t)\|^2$ as

follows

$$\widehat{\|\nabla F(\boldsymbol{w}_t)\|^2}^{+} = \max\left(\widehat{\mathbb{E}[\|\boldsymbol{g}_t\|^2]}^{+} - \frac{\widehat{\mathbb{V}(\boldsymbol{g}_{i,t})}^{+}}{k_t}, 0\right), \tag{11}$$

where the max operation guarantees non-negativity of the estimate.

To estimate L, we need also to estimate $\mathcal{G}_{k_{t-1},t-1}$. In most of the existing implementations of distributed gradient methods for ML (including PyTorch's one), each worker i can send to the PS the local average loss computed on its mini-batch. The PS can thus estimate the loss as

$$\widehat{F}_t = \frac{1}{k_t}\sum_{i=1}^{k_t}\frac{1}{B}\sum_{x\in\mathbb{B}_i} h(x, \boldsymbol{w}_t).$$

Thus, we have

$$\widehat{\mathcal{G}}^{+}_{k_{t-1},t-1} = \widehat{F}_{t-1} - \widehat{F}_t,$$

and substituting it to the left of (9), we get:

$$\widehat{L_t}^{+} = \frac{2\left(\eta\widehat{\|\nabla F(\boldsymbol{w}_{t-1})\|^2}^{+} - \widehat{\mathcal{G}}^{+}_{k_{t-1},t-1}\right)}{\eta^2\left(\widehat{\|\nabla F(\boldsymbol{w}_{t-1})\|^2}^{+} + \widehat{\mathbb{V}(\boldsymbol{g}_{i,t-1})}^{+}/k_{t-1}\right)} \tag{12}$$

Estimates in (10), (11) and (12) cannot be computed at the beginning of iteration t, but it is possible to compute them for earlier iterations, and use these past estimates to predict the future value. DBW simply averages the past D estimates (or the first $t-1$ if $t \le D$), i.e.,

$$\widehat{\mathbb{V}(\boldsymbol{g}_{i,t})} = \frac{1}{D}\sum_{v=1}^{D}\widehat{\mathbb{V}(\boldsymbol{g}_{i,t-v})}^{+}, \tag{13}$$

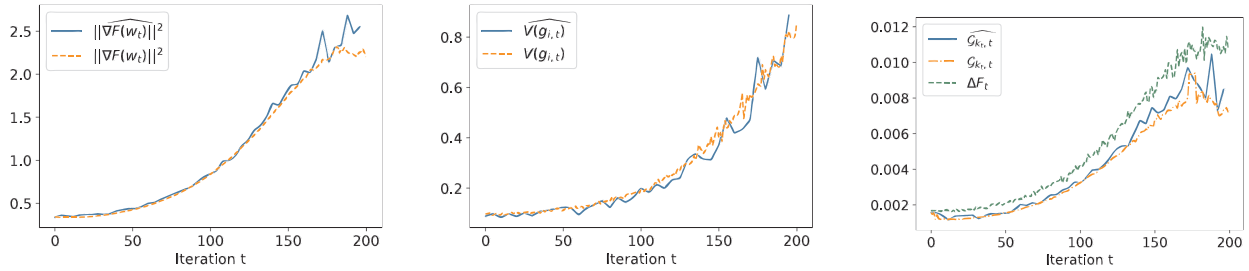
$$\widehat{\|\nabla F(\boldsymbol{w}_t)\|^2} = \frac{1}{D}\sum_{v=1}^{D}\widehat{\|\nabla F(\boldsymbol{w}_{t-v})\|^2}^{+}, \tag{14}$$

$$\widehat{L_t} = \frac{1}{D}\sum_{v=1}^{D}\widehat{L_{t-v}}^{+}. \tag{15}$$

Combining (9), (13), (14) and (15), the estimate of the gain is

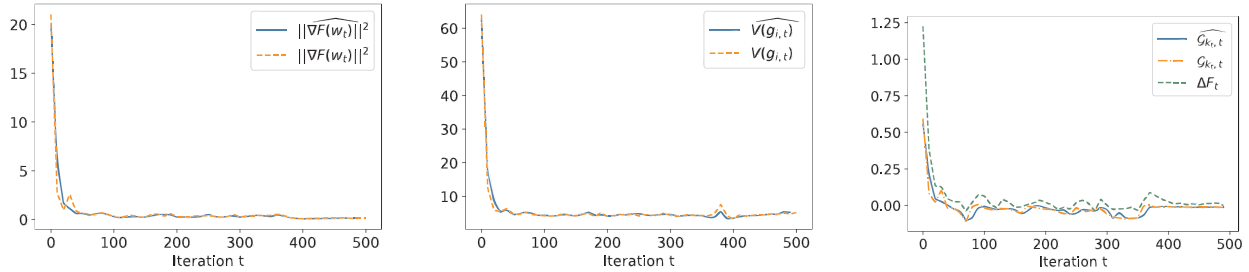
$$\widehat{\mathcal{G}_{k,t}} = \left(\eta - \frac{\widehat{L_t}\eta^2}{2}\right)\widehat{\|\nabla F(\boldsymbol{w}_t)\|^2} - \frac{\widehat{L_t}\eta^2}{2}\frac{\widehat{\mathbb{V}(\boldsymbol{g}_{i,t})}}{k}. \tag{16}$$

In Fig. 1 and Fig. 2, we show our estimates during one training process on the MNIST and CIFAR10 dataset respectively (details in Sect. 4), where our algorithm (described in Sect. 3.3) is applied to dynamically choose k. The solid lines are the estimates given by (13), (14), and (16). The dashed lines present the exact values (we have instrumented our code to compute them). We can see from Figures 1(a), 2(a), 1(b) and 2(b) that the proposed estimates $\widehat{\|\nabla F(\boldsymbol{w}_t)\|^2}$ and $\widehat{\mathbb{V}(\boldsymbol{g}_{i,t})}$ are close to the true ones. Figures 1(c) and 2(c) compare the loss decrease ΔF_t (observed a posteriori) and $\widehat{\mathcal{G}_{k_t,t}}$. As expected $\widehat{\mathcal{G}_{k_t,t}}$ is a lower bound for ΔF_t, but the two quantities are almost proportional. This is promising, because, if the lower bound $\widehat{\mathcal{G}_{k,t}}/T_{k,t}$ and the function $\Delta F_t/T_{k,t}$ were exactly proportional, their maximizers would coincide. Then, working on the lower bound, as we do, would not be an approximation. Note that, for CIFAR10 dataset, the stochastic gradients are so noisy that the gradient variance is much larger than the gradient norm (as observed also in [52]). Thus, the expected gain (9), which is the lower bound for the loss decrease, may become negative. In this case, DBW cautiously selects $k_t = n$ (see Sect. 3.3).



(a) Gradient norm (b) Gradient variance (c) Loss decrease

Figure 1: Estimation of the loss decrease. MNIST, $n = 16$ workers, batch size $B = 500$, learning rate $\eta = 0.01$, estimates computed over the last $D = 5$ iterations.



(a) Gradient norm (b) Gradient variance (c) Loss decrease

Figure 2: Estimation of the loss decrease. CIFAR10, $n = 16$ workers, batch size $B = 256$, learning rate $\eta = 0.05$, estimates computed over the last $D = 5$ iterations.

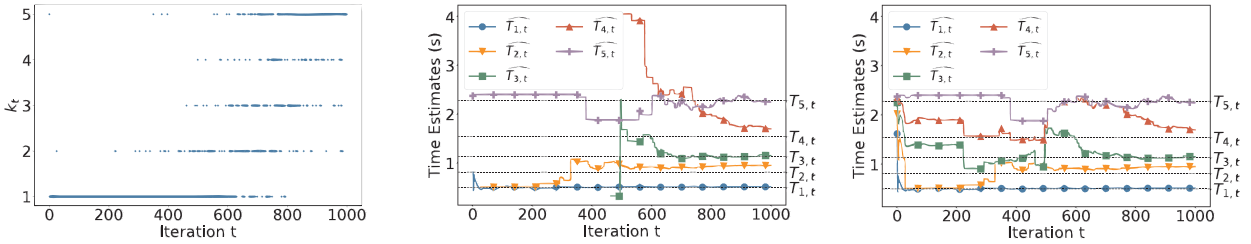
3.2. Iteration Duration

In this subsection, we discuss how to estimate the time $T_{k,t}$ the PS needs to receive k gradients of $\boldsymbol{w}_t$ after the update $\boldsymbol{w}_t$ at iteration t. As in [53], we call *round trip time* the total (random) time an idle worker needs to 1) retrieve the new parameter vector, 2) compute the corresponding gradient, and 3) send it back to the PS. Our estimators implicitly assume the cluster is stationary and homogeneous, in the sense that the distribution of round trip times does not change over time and from worker to worker. But in the experimental section, we show that they work also in dynamic and heterogeneous scenarios.

When the PS starts a new iteration t ($t > 0$), there are k_{t-1} workers ready to compute the new gradient while the other $n-k_{t-1}$ workers are still computing stale gradients, i.e., relative to past parameter vectors $\boldsymbol{w}_{t-\tau}$ with $\tau > 0$. $T_{k,t}$ depends not only on the value of k but also on the value of k_{t-1} and the $n - k_{t-1}$ residual round trip times (i.e., the remaining times for the $n - k_{t-1}$ busy workers to complete their tasks). We assume that most of such dependence is captured by the number k_{t-1}. This would be correct if round trip times were exponential random variables due to their memoryless properties. Let $\mathtt{t}_{h,i,t}$ denote the time the PS spends for receiving the i-th gradient of $\boldsymbol{w}_t$, provided that it has waited $k_{t-1} = h$ gradients at iteration $t-1$. Under our assumptions, for given values of h and i, the values $\{\mathtt{t}_{h,i,t}\}$ can be seen as samples of the same random variable that we denote by $\mathcal{T}_{h,i}$. For estimating $T_{k,t}$, we consider $\widehat{T_{k,t}} = \widehat{\mathbb{E}[\mathcal{T}_{k,k}]}$.[5]

Consider $k_{t-1} = h$ and $k_t = k$. The PS can collect the samples $\mathtt{t}_{h,i,t}$ for $i \leq k$ (it needs to wait k gradients before moving to the next iteration), but also for $i > k$ because late workers still complete the ongoing calculations. In fact, late workers may terminate the computation and send their (by now stale) gradients to

[5]It could seem more appropriate to consider $\widehat{T_{k,t}} = \mathbb{E}[\widehat{\mathcal{T}_{k_{t-1},k}}]$, but we want to select a value of k that leads to good performance on the long term, i.e., if constantly used. For this reason, we use $\widehat{\mathbb{E}[\mathcal{T}_{k,k}]}$, that corresponds to select k at each iteration.



(a) Values of k selected. (b) Empirical average. (c) Constraint-aware estimator.

Figure 3: Estimation of $T_{k,t}$. $n = 5$ workers.

the PS, before they receive the new parameter vector. Even if a new parameter vector is available at the local queue (and then they know their gradient is not needed), in DBW workers still notify the completion to the PS, providing useful information to estimate $T_{k,t}$ with limited communication overhead.

A first naive approach to estimate $\mathbb{E}[\mathcal{T}_{k,k}]$ is to average the samples obtained over the past history. But, actually, there is much more information that can be exploited to improve estimations if we jointly estimate the complete set of values $\mathbb{E}[\mathcal{T}_{h,k}]$, for $h, k = 1, \dots n$. In fact, the following pathwise relation holds for each h and i: $\mathtt{t}_{h,i,t} \leq \mathtt{t}_{h,i+1,t}$, because the index i denotes the order of arrivals of the gradients. As a consequence, $\mathbb{E}[\mathcal{T}_{h,i}] \leq \mathbb{E}[\mathcal{T}_{h,i+1}]$. Moreover, coupling arguments lead to conclude that $\mathbb{E}[\mathcal{T}_{h+1,i}] \leq \mathbb{E}[\mathcal{T}_{h,i}]$ and $\mathbb{E}[\mathcal{T}_{i,i}] \leq \mathbb{E}[\mathcal{T}_{i+1,i+1}]$. These two inequalities express the following intuitive facts: 1) if an iteration starts with more workers available to compute, the PS will collect i gradients faster (on average), 2) constantly waiting a smaller number of gradients leads to faster iterations. As $\mathbb{E}[\mathcal{T}_{i,i}] \leq \mathbb{E}[\mathcal{T}_{i+1,i+1}]$ may be less evident, we provide a proof in Appendix A. These inequalities allow us to couple the estimations of $\mathbb{E}[\mathcal{T}_{h,k}]$, for $h, k = 1, \dots n$. Samples for a given pair (h, k) can thus contribute not only to the estimation of $\mathbb{E}[\mathcal{T}_{h,k}]$ but also to the estimations of other pairs. This is useful because the number of samples for (h, k) is proportional to the number of times k_t has been selected equal to h. There can be many samples for a given pair and much less (even none) for another one.

Let $\mathbb{T}_{h,k,t}$ be the set of samples available up to iteration t for (h, k), i.e., $\mathbb{T}_{h,k,t} = \{\mathtt{t}_{h,k,t'}, \forall t' \leq t\}$. We propose to estimate $\{\mathbb{E}[\mathcal{T}_{h,k}], h, k = 1, \dots n\}$ by solving the following optimization problem:

$$\begin{aligned} \underset{x_{h,k}}{\text{minimize}} \quad & \sum_{h,k=1}^{n} \sum_{y \in \mathbb{T}_{h,k,t}} (y - x_{h,k})^2 \\ \text{subject to} \quad & x_{h,k} \leq x_{h,k+1}, \quad \text{for } k = 1, \dots n-1 \\ & x_{h+1,k} \leq x_{h,k}, \quad \text{for } h = 1, \dots n-1 \\ & x_{k,k} \leq x_{k+1,k+1}, \text{ for } k = 1, \dots n-1 \end{aligned} \tag{17}$$

Let $x^*_{h,k}$ be the solution of problem (17). Then, $\widehat{\mathbb{E}[\mathcal{T}_{h,k}]} = x^*_{h,k}$, $\forall h, k = 1, \dots, n$ and we have $\widehat{T_{k,t}} = x^*_{k,k}$. We observe that, without the constraints, the optimal value $x^*_{h,k}$ at iteration t is the empirical average of the corresponding set $\mathbb{T}_{h,k,t}$. Hence, Problem (17) is a natural way to extend the empirical average estimators, while accounting for the constraints. For our application, the quadratic optimization problem (17) can be solved fast through solvers like CVX [54, 55] for the typical values of n ($10 - 1000$).

In Fig. 3, we compare our estimator with the naive one (the empirical average). We observe that the naive method 1) cannot provide estimates for a given value h before it selects $k_t = h$, 2) leads often to estimates that are in the wrong relative order. By enforcing the inequality constraints, our estimator (17) is able to obtain more precise estimates, in particular for the values $k = 3$ and $k = 4$ that are tested less frequently in this experiment. Experiments similar to those in Sect. 4 (but not shown in this paper) confirm that naive estimators lead to longer training time.

3.3. Dynamic Choice of k_t

DBW rationale is to select the parameter k_t that maximizes the expected decrease of the loss function per time unit, i.e.,:

$$k_t = \arg\max_{1 \le k \le n} \frac{\widehat{\mathcal{G}_{k,t}}}{\widehat{T_{k,t}}}. \tag{18}$$

Note that (18) does not select values of k for which $\widehat{\mathcal{G}_{k,t}} < 0$, unless $\widehat{\mathcal{G}_{k,t}} < 0$ for all values k, in which case $k_t = n$.

This behaviour is correct. In fact, $\widehat{\mathcal{G}_{k,t}} < 0$ indicates the aggregate batch size kB may be too low to guarantee that the stochastic gradient $\boldsymbol{g}_t$ corresponds to a descent direction and then it is opportune to increase k (if possible). Our approach then recovers some behaviour of dynamic sample size methods (see [51, Sect. 5.2], [56]). At the same time, $\mathcal{G}_{k,t}$ is a lower bound for the loss decrease $\mathbb{E}[\Delta F_t]$ (see (6)). It may happen then that $\widehat{\mathcal{G}_{k,t}} < 0$, even if $\mathbb{E}[\Delta F_t] > 0$. In this situation, DBW's choice of k_t may not be optimal, as we observe in some settings in Sect. 4.4, but still DBW errs on the side of caution to prevent the loss function from increasing.

In addition, DBW exploits the local average loss $\widehat{F}_t$ to avoid decreasing k_t from one iteration to the other, when the loss appears to be increasing (and then we need more accurate gradient estimates, rather than noisier ones). We modify (18) to

$$k_t = \max\left(\arg\max_{1 \le k \le n} \frac{\widehat{\mathcal{G}_{k,t}}}{\widehat{T_{k,t}}},\ (k_{t-1}+1) \cdot \mathbb{1}_{\{\hat{F}_{t-1} > \beta \hat{F}_{t-2}\} \wedge \{k_{t-1} < n\}}\right), \tag{19}$$

where $\beta \ge 1$ (we select $\beta = 1.01$ in our experiments) and $\mathbb{1}_A$ denotes the indicator function (equal to 1 iff A is true). If the loss has become β times larger since the previous iteration, then (19) forces $k_t \ge k_{t-1} + 1$.

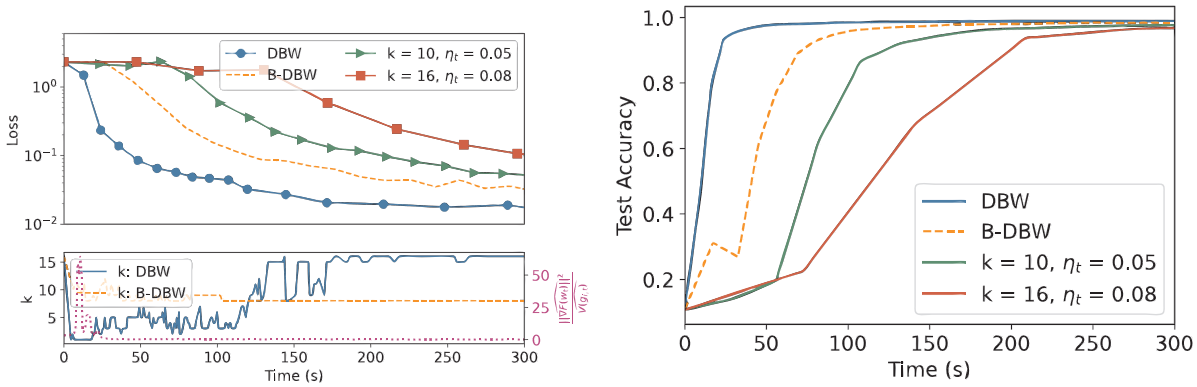
4. Experiments

We have implemented DBW in PyTorch [48], using the MPI backend for distributed communications. The experiments have been run on a real CPU/GPU cluster platform, with different GPUs available (e.g., GeForce GTX 1080 Ti, GeForce GTX Titan X, and Nvidia Tesla V100). In order to have a fine control over the round trip times, our code can generate computation and communication times according to different distributions (uniform, exponential, Pareto, etc.) or read them from a trace provided as input file. The system operates at the maximum speed guaranteed by the underlying cluster, but it maintains a virtual clock to keep track of when events would have happened. Note that the virtual time is not a simple relabeling of the time axis: for example virtual time instants at which gradients are received by the PS determine which of them are actually used to update the parameter vector. So the virtual time has an effect on the optimization dynamics. Our code is available online [42].

In what follows, we show that the number of backup workers should vary, not only with the round trip time distribution, but also with the hyper-parameters of the optimization algorithm like the batch size B. Moreover, the optimal setting depends as well on the stage of the training process, and then changes over time, even when the cluster is stationary (round trip times do not change during the training period).

In all experiments, DBW achieves nearly optimal performance in terms of convergence time, and sometimes it even outperforms the optimal static setting, that is found through an exhaustive offline search over all values $k \in \{1, \dots, n\}$. We also compare DBW with a variant where the gain $\mathcal{G}_{k,t}$ is not estimated as in (16), but it equals the number of aggregated gradients k, as proposed in [44]. We call this variant blind DBW (B-DBW), because it is oblivious to the current state of the training. We find that this approach is too simplistic: ignoring the current stage of the optimization problem leads to worse performance than DBW.

We evaluated DBW, B-DBW, and different static settings for k on two classification problems 1) MNIST [57], a dataset with 70000 28×28 images portraying handwritten digits from 0 to 9 and 2) CIFAR10 [58], a



(a) Loss versus Time

(b) Accuracy versus Time

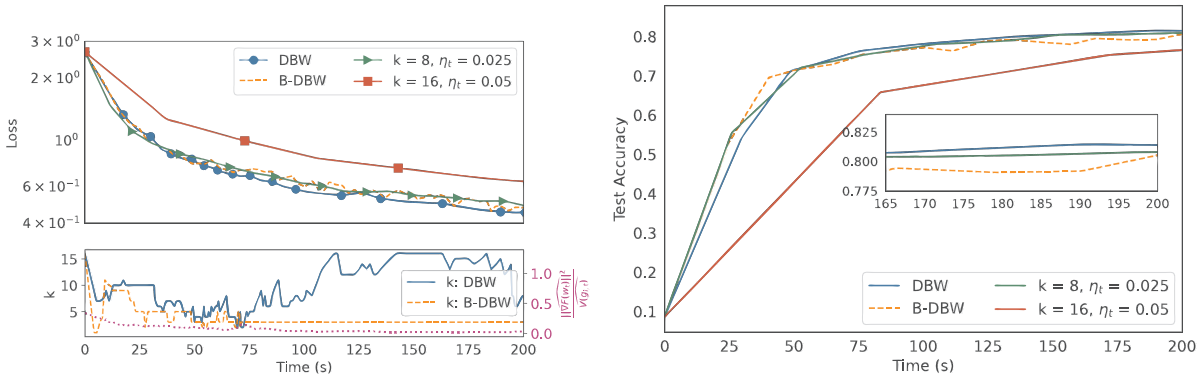
Figure 4: Training on MNIST, batch size $B = 500$, $n = 16$ workers, estimates computed over the last $D = 5$ iterations, proportional rule with $\eta(k) = 0.005k$, round trip times follow shifted exponential distribution $0.3 + 0.7\text{Exp}(1)$.

dataset with 60000 32×32 colour images in 10 classes.[6] We trained a neural network with two convolutional layers with 5×5 filters and two fully connected layers for MNIST and we trained a ResNet18 [59] network for CIFAR10. The loss function was the cross-entropy one. For MNIST, every worker had access to the entire dataset. For CIFAR10, the data set was split uniformly at random among workers.

The learning rate is probably the most critical hyper-parameter in ML optimization problems. Ideally, it should be set to that largest value that still guarantees convergence. It is important to note that different static settings for the number of backup workers require different values for the learning rate. In fact, the smaller is k, the noisier is the aggregate gradient $\boldsymbol{g}_t$, so that the smaller should be the learning rate. The rule of thumb proposed in the seminal paper [40] is to set the learning rate proportional to k, i.e., $\eta(k) \propto k$. This corresponds to the standard recommendation to have the learning rate proportional to the (aggregate) batch size [60, 61]. In static settings, aggregating k gradients is equivalent to use a batch size equal to kB, so that the learning rate should scale accordingly. An alternative approach is to tune the learning rate independently for each static value of k according to the empirical rule in [62], that requires to run a number of experiments and determine the inflection points of a specific curve. This rule leads as well to learning rates increasing with k. We call the two settings respectively the *proportional* and the *knee* rule. The maximum learning rate for the proportional rule is set equal to the value determined for $k_t = n$ by the knee rule. The same value is also used as learning rate for DBW and B-DBW, independently from the specific value they select for k_t. In fact, DBW and B-DBW can safely operate with a large learning rate because they dynamically increase k_t up to n, when they detect that the loss is increasing.

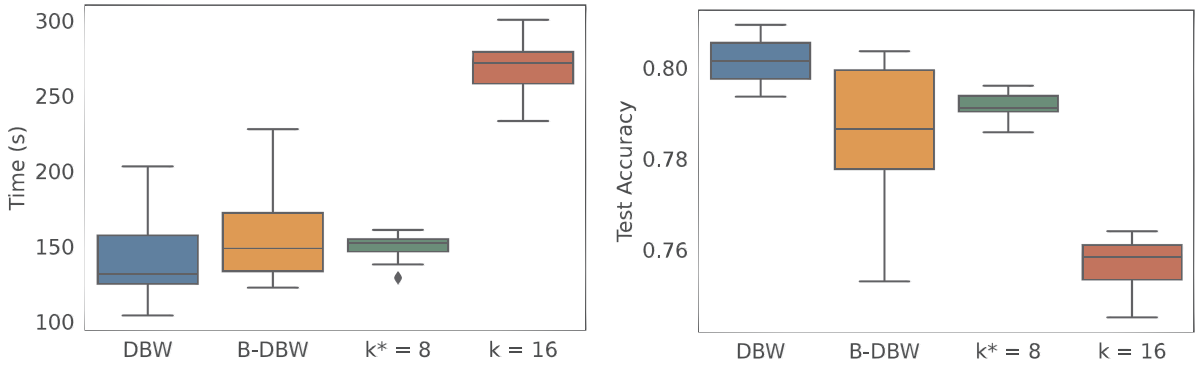
Figures 4(a) and 5(a) show, for a single run of the training process, the evolution of the loss over time and the corresponding choices of k_t for the two dynamic algorithms. For static settings, the learning rate follows the proportional rule and the optimal static settings are $k^* = 10$ for MNIST and $k^* = 8$ for CIFAR10. We can see that DBW achieves the fastest convergence across all other tested configurations of k, by using a different value of k in different stages of the training process. In fact, as we have discussed after introducing (9), the effect of k on the gain depends on the module of the gradient and on the variability of the local gradients. In the bottom subplot, the dotted line shows how their ratio varies during the training process. For MNIST, up to iteration 38, $\mathbb{V}(\boldsymbol{g}_{i,t})$ is negligible in comparison to $\|\nabla F(\boldsymbol{w}_t)\|^2$. DBW then selects small values for k_t loosing a bit in terms of the gain, but significantly speeding up the duration of each iteration by only waiting for the fastest workers. As the parameter vector approaches a local minimum, $\|\nabla F(\boldsymbol{w}_t)\|^2$ approaches zero, and the gain becomes more and more sensitive to k, so that DBW progressively increases k_t up to reach $k_t = n = 16$ as shown by the solid line. On the contrary B-DBW (the dashed line) selects most of the

[6] Both dataset include 10000 test images.



(a) Loss versus Time

(b) Accuracy versus Time



(c) Time to reach 80% test accuracy

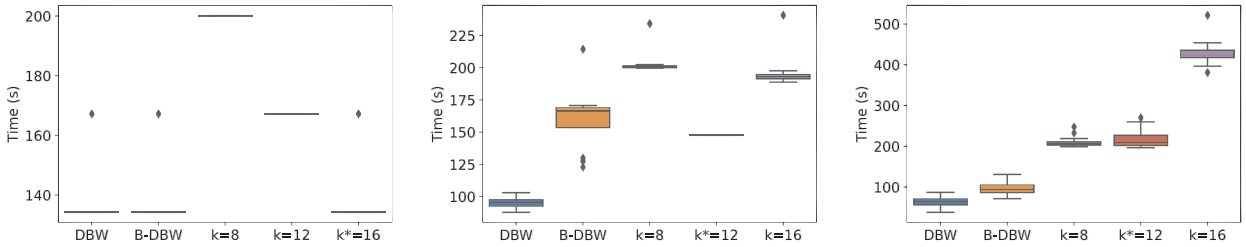
(d) Test accuracy after 200 seconds

Figure 5: Training on CIFAR10, batch size $B = 256$, $n = 16$ workers, estimates computed over the last $D = 5$ iterations, proportional rule with $\eta(k) = \frac{0.05k}{16}$, round trip times follow exponential distribution Exp(1). Box plots are bases on 20 independent runs.

time $k_t = 9$ with some variability to the randomness of the estimates $\widehat{T_{k,t}}$. For CIFAR10, as the stochastic gradients are more noisy, the ratio values $\|\nabla F(\boldsymbol{w}_t)\|^2/\mathbb{V}(\boldsymbol{g}_{i,t})$ are smaller than in MNIST, DBW selects higher values for k_t (around 10) in the beginning of the training. After iteration 130, the gain becomes more sensitive to k and thus DBW progressively increases k_t as observed in MNIST dataset. Note that DBW performs less advantageous in CIFAR10, although it is still the best one. As discussed in Sect. 3.1, the gain (9) can be negative when the stochastic gradients are very noisy, which is the case for CIFAR10 dataset. This results in DBW cautiously selecting $k_t = n$ according to (18), while the optimal k_t at the iteration t may be smaller. Note that working with significantly larger batch sizes would reduce the variability of the stochastic gradients.

Figures 4(b) and 5(b) show, for a single run of the training process, the evolution of the test accuracy over time. We can see that DBW converges to a better model faster than the other methods for MNIST. The advantages of DBW on CIFAR10 are less evident on this specific run, but Figs. 5(c) and 5(d) show the distribution of the time to reach 80% test accuracy and the distribution of the test accuracy after 200 seconds using box plots.[7] On average DBW performs better than B-DBW or the optimal static setting.

[7]The box shows the quartiles of the dataset while the whiskers extend to show the rest of the distribution. The middle bar gives the median value.



(a) $\alpha = 0$ (b) $\alpha = 0.2$ (c) $\alpha = 1$

Figure 6: Effect of round trip time distribution. MNIST, $n = 16$ workers, batch size $B = 500$, estimates computed over the last $D = 5$ iterations, proportional rule for $\eta(k)$ in static settings where $\eta(k) = 0.005k$.

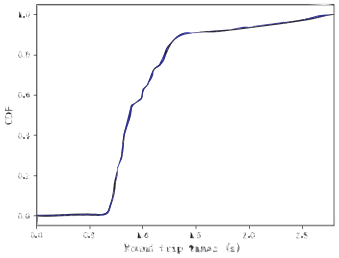


Figure 7: Empirical distribution of round trip times on a Spark cluster

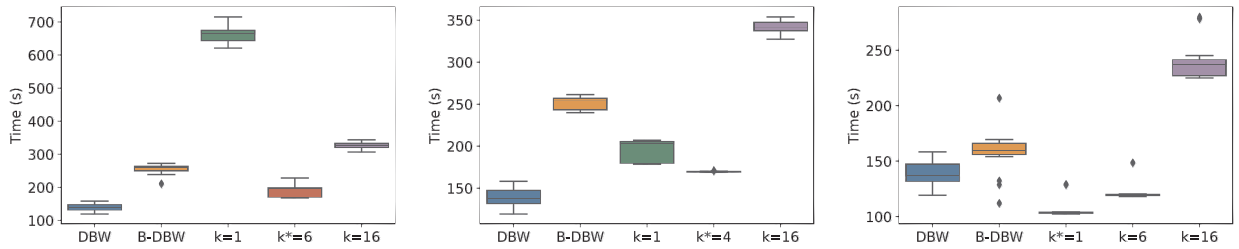
4.1. Round trip time effect

In this subsection we consider round trip times (see Sect. 3.2) are i.i.d. according to a shifted exponential random variable $1 - \alpha + \alpha \times \text{Exp}(1)$, where $0 \leq \alpha \leq 1$. We consider later realistic time distributions. This choice, common to [53, 63], allows us to easily tune the variability of the round trip times by changing α. When $\alpha = 0$, all gradients arrive at the same time at the PS, so that the PS should always aggregate all of them. As α changes from 0 to 1, the variance of the round trip times increases, and waiting for $k < n$ gradients becomes advantageous.

Figure 6 compares the time needed to reach a training loss smaller than 0.2 for the two dynamic algorithms and the static settings $k = 16$, $k = 12$, and $k = 8$, that are optimal respectively for $\alpha = 0$, $\alpha = 0.2$, and $\alpha = 1$. For each of them, we carried out 20 independent runs with different seeds. We find that our dynamic algorithm achieves the fastest convergence in all three scenarios, it is even 1.2x faster and 3x faster than the optimal static settings for $\alpha = 0.2$ and $\alpha = 1$. There are two factors that determine this observation. First, as discussed for Fig. 4, there is no unique optimal value of k to be used across the whole training process, and DBW manages to select the most indicated value in different stages of the training process. Second, DBW takes advantage of a larger learning rate. Both factors play a role. For example if we focus on Fig. 6(c), the learning rate for DBW is twice faster than that for $k = 8$, but DBW is on average 3x faster. Then, adapting k achieves an additional 1.5x improvement. The importance of capturing the dynamics of the optimization process is again also evident by comparing DBW with B-DBW. While B-DBW takes advantage of a higher learning rate as well, it performs worse than our solution DBW.

4.2. Batch size effect

The batch size B is another important hyper-parameter. It is often limited by the memory available at each worker, but can also be determined by generalization performance of the final model [64]. In this subsection we highlight how B also affects the optimal setting for k. These findings confirm that configuring the number of backup workers is indeed a difficult task, and knowing the characteristics of the underlying cluster is not sufficient.



(a) $B = 16$, $\eta \in \{0.01, 0.045, 0.05\}$ (b) $B = 128$, $\eta \in \{0.04, 0.044, 0.05\}$ (c) $B = 500$, $\eta \in \{0.07, 0.07, 0.08\}$

Figure 8: Effect of batch size B. MNIST, $n = 16$ workers, estimates computed over the last $D = 5$ iterations, knee rule for η in static settings with values shown above for each k.

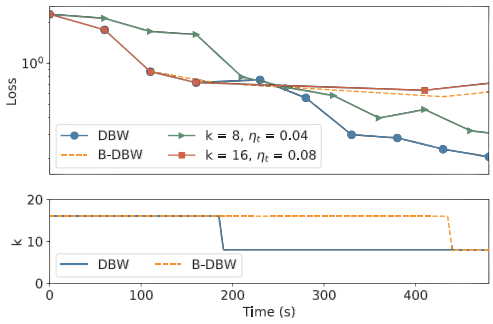


Figure 9: Robustness to slowdowns of the system. MNIST, $n = 16$ workers, batch size $B = 500$, estimates computed over the last $D = 5$ iterations, proportional rule for $\eta(k)$ in static settings where $\eta(k) = 0.005k$.

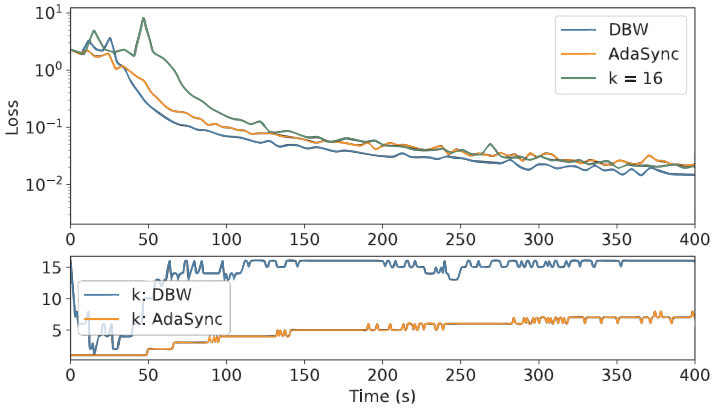
The experiments differ in two additional aspects from those in Fig. 6. First, the distribution of the round trip times (shown in Fig. 7) is taken from a training a ML model through stochastic gradient descent on a production Spark cluster with sixteen servers, each with two 8-core Intel E5-2630 CPUs running at 2.40GHz. The cluster was managed using Zoe Analytics [65]. Second, learning rates are configured according to the knee rule. We observe that the knee rule leads to a weaker variability of the learning rate in comparison to the proportional rule: for example, for $B = 16$, η increases by less than a factor 5 when k changes from $k = 1$ to $k = 16$, and it increases much less for larger B.

Figure 8 shows the results for $B = 16, 128, 500$, comparing the dynamic methods with a few static settings, including the optimal static one that decreases from $k^* = 6$ for $B = 16$ to $k^* = 1$ for $B = 500$. Again, Equation (9) helps to understand this change of the optimal static setting with different batch size: as the batch size increases, the variability of gradients decreases, so that the numerator depends less on k. The advantage of reducing $T_{k,t}$ by selecting a small k can compensate the corresponding decrease of the gain $\mathcal{G}_{k,t}$.

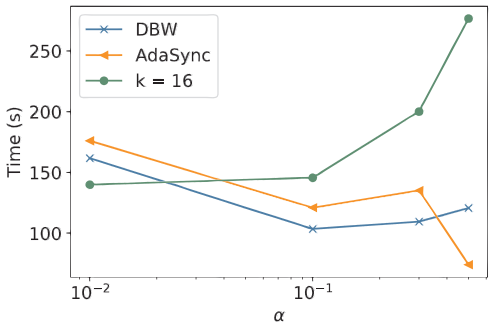
Since learning rates chosen by the knee rule for the static settings are now close to dynamic ones, DBW does not outperform the optimal static setting, but its performance are quite close, and significantly better than B-DBW for $B = 128, 500$. It is worthy to stress that, when running a given ML problem on a specific cluster environment, the user cannot predict the optimal static setting k^* without running preliminary short training experiments for every k. DBW does not need them.

4.3. Robustness to slowdowns

Until now, we have considered a stationary setting where the distribution of round trip times does not change during the training. Figure 9 shows an experiment in which half of the workers experience a sudden slowdown during the training process. Initially, round trip times are all equal and deterministic, so that the optimal setting is $k_t = n = 16$. Suddenly, at time $t = 160$s, half of the workers in the clusters slow down by a



(a) Loss versus Time ($\alpha = 0.1$)



(b) Average convergence time versus different α

Figure 10: Training on MNIST, batch size $B = 500$, $n = 16$ workers, estimates computed over the last $D = 5$ iterations. $\eta = 0.08$. Round trip times follow shifted exponential distribution $1 - \alpha + \alpha\text{Exp}(1)$

factor 5 and the optimal static configuration is now to select $k_t = n/2 = 8$. We can see that DBW detects the slowdowns in the system and then correctly selects $k_t = 8$.

4.4. Comparison with AdaSync

AdaSync [27] is a dynamic backup scheme designed for the Push and Interrupt (PsI) case, under the assumption that the round trip times follow shifted exponential distribution. For the comparison, we consider then this setting. For AdaSync, the quadratic formulation in [27, Appendix D.1] is used to derive the number of backup workers. AdaSync updates k at the end of a time-window. We consider this time-window small enough for AdaSync evaluating the possibility to update k_t at each iteration, as DBW does.

Figure 10(a) shows, for a single run of the training process, the evolution of the loss over time and the corresponding choices of k_t for DBW and AdaSync, when $\alpha = 0.1$, i.e., round trip times follow distribution $0.9 + 0.1\text{Exp}(1)$. DBW quickly reaches a large value of k_t close to n. For small α the variance of round trip time is small, so choosing large k_t does not lead large iteration times $\mathbb{E}[T_{k,t}]$ but benefits the gain in (9). The approximated formula used by AdaSync, even if derived under the assumption of shifted exponential distributions, does not depend on α, and AdaSync fails to increase fast the value of k_t.

Fig. 10(b) shows the average convergence time[8] computed over 10 independent runs under different α. The larger α, the larger the variance of round trip times. We can see that when α is smaller than 0.3, DBW performs better than AdaSync. While, AdaSync works better for larger α, which suggests DBW may be too conservative on the number of backup workers in the late phase of the training.

Remember that the estimated gain $\widehat{\mathcal{G}_{k,t}}$ used in (18) for choosing k_t, is a lower bound for the true loss decrease. In the late training phase, when the gradient norm becomes smaller, small values of k may lead to estimate a negative (see (16)). In this case, DBW conservatively chooses a larger k for which the gain is estimated to be positive. On the other hand, AdaSync requires prior knowledge on the round trip time distribution. This distribution may be hard to estimate and may change during the training period, that is often very long for state-of-the-art machine learning models (e.g., weeks). Notice that DBW does not require any prior knowledge on the system.

5. Conclusions

In this paper, we have shown that the number of backup workers needs to be adapted at run-time and the correct choice is inextricably bounded, not only to the cluster's configuration and workload, but also to

[8]The convergence time noted here is the time when the training loss reaches 0.07.

the hyper-parameters of the learning algorithm and the stage of the training. We have proposed a simple algorithm DBW that, without prior knowledge about the cluster or the problem, achieves good performance across a variety of scenarios, and even outperforms in some cases the optimal static setting.

As a future research direction, we want to extend the scope of DBW to dynamic resource allocation, e.g., by automatically releasing computing resources if $k_t < n$ and the fastest k_t gradients are always coming from the same set of workers. In general, we believe that distributed systems for ML are in need of adaptive algorithms in the same spirit of the utility-based congestion control schemes developed in our community starting from the seminal paper [66]. As our work points out, it is important to define new utility functions that take into account the learning process. Adaptive algorithms are even more needed in the federated learning scenario [67], where ML training is no more relegated to the cloud, but it occurs in the wild over the whole internet. Our paper shows that even simple algorithms can provide significant improvements.

6. Acknowledgements

This work has been carried out in the framework of a common lab agreement between Inria and Nokia Bell Labs (ADR 'Rethinking the Network'). We thank Alain Jean-Marie for having suggested the estimation technique in Sect. 3.2 and Pietro Michiardi for many helpful discussions.

Appendix A. Proof of $\mathbb{E}[\mathcal{T}_{i,i}] \leq \mathbb{E}[\mathcal{T}_{i+1,i+1}]$

Remember that we assume that $T_{k,t}$ depends on the past only through the number of workers k_{t-1} selected at the previous iteration. This approximation is correct when round trip times are exponentially distributed. We start proving the inequality under the assumption that round trip times are exponentially distributed. We move then to the general case.

Consider the beginning of a new iteration t when the PS systematically waits for $i+1$ nodes. Without loss of generality, let us assume that the workers who finished the computation are labeled $1, 2, \ldots, i+1$. Worker $j \leq i+1$ needs an exponentially distributed round trip time ω_j to complete the new computation. Worker $j > i+1$ needs to complete iteration $t-1$, with residual time ω'_j, and possibly start a new one with the updated parameter vector, with corresponding residual time ω_j; both ω_j and ω'_j are exponentially distributed.

Let $\mu(l, A)$ denote the l-th smallest element of the multiset A. The duration of the new iteration is then $T_{i+1,t} = \mu(i+1, \{\omega_1, \ldots, \omega_i, \omega_{i+1}, \omega'_{i+2} + \omega_{i+2}, \ldots, \omega'_n + \omega_n\})$.

Now consider the case when the PS only waits for the i workers. Again we assume the the first workers who finished the iteration are labeled $1, 2, \ldots, i$. We also couple all the round trip times so that ω_j for $j = 1, \ldots, n$ and ω'_j for $j = i+2, \ldots, n$ denote the same quantities and have the same values. In this case also worker $i+1$ needs to terminate the previous computation; this will require a time ω'_{i+1}, but its specific value is irrelevant. The duration of the new iteration is $T_{i,t} = \mu(i, \{\omega_1, \ldots, \omega_i, \omega'_{i+1} + \omega_{i+1}, \omega'_{i+2} + \omega_{i+2}, \ldots, \omega'_n + \omega_n\})$.

$$
\begin{aligned}
T_{i+1,t} &= \mu\big(i+1, \{\omega_1, \ldots, \omega_i, \omega_{i+1}, \omega'_{i+2} + \omega_{i+2}, \ldots, \omega'_n + \omega_n\}\big) \\
&\geq \mu\big(i+1, \{\omega_1, \ldots, \omega_i, 0, \omega'_{i+2} + \omega_{i+2}, \ldots, \omega'_n + \omega_n\}\big) \\
&= \mu\big(i, \{\omega_1, \ldots, \omega_i, \omega'_{i+2} + \omega_{i+2}, \ldots, \omega'_n + \omega_n\}\big) \\
&\geq \mu\big(i, \{\omega_1, \ldots, \omega_i, \omega'_{i+1} + \omega_{i+1}, \omega'_{i+2} + \omega_{i+2}, \ldots, \omega'_n + \omega_n\}\big) \\
&= T_{i,t},
\end{aligned}
$$

where the first inequality follows from the fact that replacing an element in the set with a smaller one can only decrease the $(i+1)$-th smallest element of the multiset, the second equality from the fact that 0 is necessarily the smallest value in the multiset, and the last inequality from the fact that enlarging a multiset cannot increase its i-th smallest element.

In the general case, we show that the time at which the t-th iteration will start is not larger when the PS waits for i workers than when it waits for $i+1$ workers. We will couple the round trip times so that in both cases the duration of the m-th round trip time for worker j is the same in both systems.

Let $\chi_{i,t}$ denote the time at which the t-th system iteration starts when then PS waits for i workers. We also consider a *lazy* system, where the PS does not need to start the new iteration as soon as i new updates are available, but it can start after an arbitrary delay. We say that a sequence $(\chi^{(l)}_{i,t})_{t\in\mathbb{N}}$ is feasible for the lazy system, if it corresponds to a valid sequence of starting times. We observe that for any feasible sequence $\chi^{(l)}_{i,t} \geq \chi_{i,t}$ for each t as the lazy system can only introduce slack times. Finally, we note that $(\chi_{i+1,t})_{t\in\mathbb{N}}$ is a feasible sequence for the lazy system, as at each time $\chi_{i+1,t}$, the system has available i new updates (it has $i+1$) and can then start a new iteration. It follows that $\chi_{i+1,t} \geq \chi_{i,t}$.

References

[1] K. Canini, T. Chandra, E. Ie, J. McFadden, K. Goldman, M. Gunter, J. Harmsen, K. LeFevre, D. Lepikhin, T. Lloret Llinares, I. Mukherjee, F. Pereira, J. Redstone, T. Shaked, Y. Singer, Sibyl: A system for large scale supervised machine learning, Tech. talk (2014).

[2] E. P. Xing, Q. Ho, P. Xie, D. Wei, Strategies and principles of distributed machine learning on big data, Engineering 2 (2) (2016) 179 – 195.

[3] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, E. P. Xing, Addressing the straggler problem for iterative convergent parallel ML, in: Proceedings of the Seventh ACM Symposium on Cloud Computing, SoCC '16, ACM, New York, NY, USA, 2016, pp. 98–111.

[4] H. Wang, D. Niu, B. Li, Distributed machine learning with a serverless architecture, in: IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, 2019, pp. 1288–1296.

[5] Z. Shi, A. Eryilmaz, A flexible distributed optimization framework for service of concurrent tasks in processing networks, in: IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, 2019, pp. 1072–1080.

[6] Y. Bao, Y. Peng, C. Wu, Deep learning-based job placement in distributed machine learning clusters, in: IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, 2019, pp. 505–513.

[7] C. Chen, W. Wang, B. Li, Round-robin synchronization: Mitigating communication bottlenecks in parameter servers, in: IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, 2019, pp. 532–540.

[8] S. Shi, X. Chu, B. Li, Mg-wfbp: Efficient data communication for distributed synchronous SGD algorithms, in: IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, 2019, pp. 172–180.

[9] X. Zhang, J. Wang, G. Joshi, C. Joe-Wong, Machine learning on volatile instances, in: IEEE INFOCOM 2020 - IEEE Conference on Computer Communications, 2020, pp. 139–148. doi:10.1109/INFOCOM41043.2020.9155448.

[10] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, B.-Y. Su, Scaling distributed machine learning with the parameter server, in: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, 2014, pp. 583–598.

[11] P. Patarasuk, X. Yuan, Bandwidth optimal all-reduce algorithms for clusters of workstations, J. Parallel Distributed Comput. 69 (2) (2009) 117–124. doi:10.1016/j.jpdc.2008.09.002.
URL https://doi.org/10.1016/j.jpdc.2008.09.002

[12] S. Jeaugey, Massively scale your deep learningtraining with nccl 2.4. (2019).
URL https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4/

[13] MPI (https://www.mpich.org/static/docs/v3.1/www3/MPI_Allreduce.html).

[14] A. Gibiansky, Bringing hpc techniques to deep learning (2017).

[15] J. Dean, L. A. Barroso, The tail at scale, Communications of the ACM 56 (2) (2013) 74–80.

[16] G. Ananthanarayanan, A. Ghodsi, S. Shenker, I. Stoica, Effective straggler mitigation: Attack of the clones, in: Proc. of the 10th USENIX Conf. NSDI, 2013, pp. 185–198.

[17] C. Karakus, Y. Sun, S. Diggavi, W. Yin, Straggler mitigation in distributed optimization through data encoding, in: Proc. of NIPS, 2017, pp. 5434–5442.

[18] S. Li, S. M. M. Kalan, A. S. Avestimehr, M. Soltanolkotabi, Near-optimal straggler mitigation for distributed gradient methods, in: IEEE International Parallel and Distributed Processing Symposium Workshops, 2018, pp. 857–866.

[19] A. Reisizadeh, S. Prakash, R. Pedarsani, A. S. Avestimehr, Codedreduce: A fast and robust framework for gradient aggregation in distributed learning, CoRR abs/1902.01981 (2019). arXiv:1902.01981.
URL http://arxiv.org/abs/1902.01981

[20] R. Tandon, Q. Lei, A. G. Dimakis, N. Karampatziakis, Gradient coding: Avoiding stragglers in distributed learning, Vol. 70 of Proceedings of Machine Learning Research, PMLR, International Convention Centre, Sydney, Australia, 2017, pp. 3368–3376.
URL http://proceedings.mlr.press/v70/tandon17a.html

[21] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, K. Ramchandran, Speeding up distributed machine learning using codes, IEEE Transactions on Information Theory 64 (3) (2017) 1514–1529.

[22] W. Halbawi, N. A. Ruhi, F. Salehi, B. Hassibi, Improving distributed gradient descent using reed-solomon codes, in: 2018 IEEE International Symposium on Information Theory, ISIT 2018, Vail, CO, USA, June 17-22, 2018, IEEE, 2018, pp. 2027–2031. doi:10.1109/ISIT.2018.8437467.
URL https://doi.org/10.1109/ISIT.2018.8437467

[23] Q. Yu, S. Li, N. Raviv, S. M. M. Kalan, M. Soltanolkotabi, S. A. Avestimehr, Lagrange coded computing: Optimal design for resiliency, security, and privacy, Vol. 89 of Proceedings of Machine Learning Research, PMLR, 2019, pp. 1215–1225.
URL http://proceedings.mlr.press/v89/yu19b.html

[24] A. Reisizadeh, S. Prakash, R. Pedarsani, A. S. Avestimehr, Coded computation over heterogeneous clusters, IEEE Transactions on Information Theory 65 (7) (2019) 4227–4242.

[25] E. Ozfatura, S. Ulukus, D. Gündüz, Straggler-aware distributed learning: Communication–computation latency trade-off, Entropy 22 (5) (2020) 544.

[26] A. Reisizadeh, S. Prakash, R. Pedarsani, A. S. Avestimehr, Tree gradient coding, in: 2019 IEEE International Symposium on Information Theory (ISIT), IEEE, 2019, pp. 2808–2812.

[27] S. Dutta, J. Wang, G. Joshi, Slow and stale gradients can win the race (2020). arXiv:arXiv:2003.10579.

[28] B. McMahan, E. Moore, D. Ramage, S. Hampson, B. A. y Arcas, Communication-efficient learning of deep networks from decentralized data, in: Artificial Intelligence and Statistics, PMLR, 2017, pp. 1273–1282.

[29] Y. Peng, Y. Bao, Y. Chen, C. Wu, C. Guo, Optimus: an efficient dynamic resource scheduler for deep learning clusters, in: Proceedings of the Thirteenth EuroSys Conference, 2018, pp. 1–14.

[30] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, E. Xing, Solving the straggler problem with bounded staleness, in: Presented as part of the 14th Workshop on Hot Topics in Operating Systems, 2013.

[31] S. Dutta, G. Joshi, S. Ghosh, P. Dube, P. Nagpurkar, Slow and stale gradients can win the race: Error-runtime trade-offs in distributed SGD, AISTATS (2018).

[32] W. Dai, Y. Zhou, N. Dong, H. Zhang, E. P. Xing, Toward understanding the impact of staleness in distributed machine learning, in: 7th International Conference on Learning Representations, ICLR 2019, 2019.

[33] A. Nedic, A. Ozdaglar, Distributed subgradient methods for multi-agent optimization, IEEE Transactions on Automatic Control 54 (1) (2009) 48–61.

[34] J. C. Duchi, A. Agarwal, M. J. Wainwright, Dual averaging for distributed optimization: Convergence analysis and network scaling, IEEE Trans. on Automatic Control 57 (3) (2012) 592–606.

[35] X. Lian, C. Zhang, H. Zhang, C. Hsieh, W. Zhang, J. Liu, Can decentralized algorithms outperform centralized algorithms? A case study for decentralized parallel stochastic gradient descent, in: Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 2017, pp. 5330–5340.

[36] A. Elgabli, J. Park, A. S. Bedi, M. Bennis, V. Aggarwal, Gadmm: Fast and communication efficient framework for distributed machine learning., Journal of Machine Learning Research 21 (76) (2020) 1–39.

[37] G. Neglia, G. Calbi, D. Towsley, G. Vardoyan, The role of network topology for distributed machine learning, in: IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, 2019, pp. 2350–2358.

[38] G. Neglia, C. Xu, D. Towsley, G. Calbi, Decentralized gradient methods: does topology matter?, Vol. 108 of Proceedings of Machine Learning Research, PMLR, 2020, pp. 2348–2358.

[39] O. Marfoq, C. Xu, G. Neglia, R. Vidal, Throughput-optimal topology design for cross-silo federated learning, in: Advances in Neural Information Processing Systems (NeurIPS), 2020.

[40] J. Chen, R. Monga, S. Bengio, R. Jozefowicz, Revisiting distributed synchronous SGD, in: International Conference on Learning Representations Workshop Track, 2016.

[41] C. Xu, G. Neglia, N. Sebastianelli, Dynamic backup workers for parallel machine learning, in: 2020 IFIP Networking Conference, Networking 2020, Paris, France, June 22-26, 2020, IEEE, 2020, pp. 574–578. URL `https://ieeexplore.ieee.org/document/9142724`

[42] DBW, `https://gitlab.inria.fr/chxu/dbw`.

[43] TensorFlow (https://www.tensorflow.org/).

[44] M. Teng, F. Wood, Bayesian distributed stochastic gradient descent, in: Advances in Neural Information Processing Systems 31, 2018, pp. 6378–6388.

[45] Y. Ling, T. Mullen, X. Lin, Analysis of optimal thread pool size, SIGOPS Oper. Syst. Rev. 34 (2) (2000) 42–55.

[46] M. M. Amiri, D. Gündüz, Computation scheduling for distributed machine learning with straggling workers, in: IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2019, 2019, pp. 8177–8181.

[47] Q. Luo, J. Lin, Y. Zhuo, X. Qian, Hop: Heterogeneity-aware decentralized training, in: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 893–907.

[48] PyTorch (https://pytorch.org/).

[49] L. Balles, J. Romero, P. Hennig, Coupling adaptive batch sizes with learning rates, in: Proc. of the Thirty-Third Conference on Uncertainty in Artificial Intelligence (UAI), 2017.

[50] S. Bubeck, Convex optimization: Algorithms and complexity, Found. Trends Mach. Learn. 8 (3-4) (2015) 231–357.

[51] L. Bottou, F. E. Curtis, J. Nocedal, Optimization methods for large-scale machine learning, Siam Review 60 (2) (2018) 223–311.

[52] G. Neglia, C. Xu, D. Towsley, G. Calbi, Decentralized gradient methods: does topology matter?, in: S. Chiappa, R. Calandra (Eds.), The 23rd International Conference on Artificial Intelligence and Statistics, AISTATS 2020, 26-28 August 2020, Online [Palermo, Sicily, Italy], Vol. 108 of Proceedings of Machine Learning Research, PMLR, 2020, pp. 2348–2358.
URL http://proceedings.mlr.press/v108/neglia20a.html

[53] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, K. Ramchandran, Speeding up distributed machine learning using codes, IEEE Transactions on Information Theory 64 (3) (2018) 1514–1529.

[54] M. Grant, S. Boyd, CVX: Matlab software for disciplined convex programming, version 2.1 (Mar. http://cvxr.com/cvx, 2014).

[55] M. Grant, S. Boyd, Graph implementations for nonsmooth convex programs, in: Recent Advances in Learning and Control, Lecture Notes in Control and Information Sciences, Springer-Verlag Limited, 2008, pp. 95–110.

[56] S. De, A. Yadav, D. Jacobs, T. Goldstein, Automated inference with adaptive batches, in: Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS), 2017, pp. 1504–1513.

[57] MNIST database (http://yann.lecun.com/exdb/mnist/).

[58] A. Krizhevsky, G. Hinton, et al., Learning multiple layers of features from tiny images (2009).

[59] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

[60] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, K. He, Accurate, large minibatch SGD: training imagenet in 1 hour, CoRR abs/1706.02677 (2017). arXiv:1706.02677.

[61] S. L. Smith, P.-J. Kindermans, Q. V. Le, Don't decay the learning rate, increase the batch size, in: International Conference on Learning Representations, 2018.

[62] L. N. Smith, Cyclical learning rates for training neural networks, in: Applications of Computer Vision (WACV), 2017 IEEE Winter Conference on, IEEE, 2017, pp. 464–472.

[63] S. Dutta, G. Joshi, S. Ghosh, P. Dube, P. Nagpurkar, Slow and stale gradients can win the race: Error-runtime trade-offs in distributed SGD, in: International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 2018, pp. 803–812.

[64] E. Hoffer, I. Hubara, D. Soudry, Train longer, generalize better: closing the generalization gap in large batch training of neural networks, in: Advances in Neural Information Processing Systems 30, Curran Associates, Inc., 2017, pp. 1731–1741.

[65] F. Pace, D. Venzano, D. Carra, P. Michiardi, Flexible scheduling of distributed analytic applications, CCGrid '17, IEEE Press, 2017, pp. 100–109.

[66] F. P. Kelly, A. K. Maulloo, D. K. Tan, Rate control for communication networks: shadow prices, proportional fairness and stability, Journal of the Operational Research society 49 (3) (1998) 237–252.

[67] J. Konecný, B. McMahan, D. Ramage, Federated optimization: Distributed optimization beyond the datacenter, in: Neural Information Processing Systems (workshop), 2015.