Fast Distributed Asynchronous SGD with Variance Reduction

Ruiliang Zhang, Shuai Zheng and James T. Kwok

Abstract—With the recent proliferation of large-scale learning problems, there have been a lot of interest on distributed machine learning algorithms, particularly those that are based on stochastic gradient descent (SGD) and its variants. However, existing algorithms either suffer from slow convergence due to the inherent variance of stochastic gradients, or have a fast linear convergence rate but at the expense of poorer solution quality. In this paper, we combine their merits together by proposing a distributed asynchronous SGD-based algorithm with variance reduction. A constant learning rate can be used, and it is also guaranteed to converge linearly to the optimal solution. Experiments on the Google Cloud Computing Platform demonstrate that the proposed algorithm outperforms state-of-the-art distributed asynchronous algorithms in terms of both wall clock time and solution quality.

Index Terms—SGD, Variance reduction, SVRG, Dsitributed Optimization

I. INTRODUCTION

In recent years, there have been a rapid growth of data. A popular approach to address this challenge is by using stochastic gradient descent (SGD) and its variants [1]–[3]. However, it is often difficult to store and process a big data set on one single machine. Thus, there is now growing interest in distributed machine learning algorithms. The data set is partitioned into subsets, assigned to multiple machines, and the optimization problem solved in a distributed manner. In general, the distributed architecture may have shared memory [4] or distributed memory [2], [3], [5], [6]. In this paper, we will focus on the latter, which is more scalable. Usually, one of these machines is a server while the rest are workers. The workers store the data subsets, perform local computations and send their updates to the server, which then aggregates the local information and performs the actual update on the model parameter. Note that workers only need to communicate with the server but not among them. Such a distributed computing model has been commonly used in many recent large-scale machine learning implementations [2], [3], [5], [6].

Often, machines in these systems have to run synchronously [5], [7]. In each iteration, information from all workers need to be aggregated before the server can update. This can be expensive due to communication overhead and random network delay. It also suffers from the straggler problem [8], in which the system can move forward only at the pace of the slowest worker.

To alleviate these problems, asynchronicity is introduced [2], [6], [9]–[11]. The server is allowed to use only staled (delayed) information from the workers, and thus only needs to wait for a much smaller number of workers in each iteration. Promising theoretical/empirical results have been reported. One prominent example of asynchronous SGD is the downpour SGD [2]. Each worker independently reads the parameter from the server, computes the local gradient, and sends it back to the server. The server then immediately updates the parameter using the worker’s gradient information. Using an adaptive learning rate [12], downpour SGD achieves state-of-the-art performance.

However, in order for these algorithms to converge, the learning rate has to decrease not only with the number of iterations (as in standard single-machine SGD [1]), but also with the maximum delay τ (i.e., the duration between the time the gradient is computed by the worker and it is used by the server) among workers. Unlike [10], note that downpour SGD does not impose constraints on τ, and no convergence guarantee is provided.

In practice, a decreasing learning rate leads to slower convergence [1], [13]. Recently, Feyzmahdavian et al [14] proposed the delayed proximal gradient method in which the delayed gradient is used to update an analogously delayed model parameter (but not its current one). It is shown that even with a constant learning rate, the algorithm converges linearly to within ε of the optimal solution. However, to achieve a small ε, the learning rate needs to be small, which again means slow convergence.

Recently, there has been the flourish development of variance reduction techniques for SGD [13], [15]–[18]. The idea is to use past gradients to progressively reduce the stochastic gradient’s variance, so that a constant learning rate can again be used. However, these algorithms are designed for single machines and its properties in a distributed asynchronous environment remain unknown.

Motivated by these, we propose in this paper a distributed asynchronous SGD-based algorithm with variance reduction. By incorporating the stochastic variance reduction gradient, a constant learning rate can be used. The algorithm is easy to implement and highly scalable. Moreover, it can be shown to converge linearly to the optimal solution. To the best of our knowledge, this is the first such result for distributed asynchronous SGD-based algorithms with constant learning rate. A prototype is implemented on the Google Cloud Computing Platform. Experiments on several big data sets from the Pascal Large Scale Learning Challenge demonstrate that it outperforms the state-of-the-art.
II. BACKGROUND

In this paper, we consider the following optimization problem

$$\min_w F(w) \equiv \frac{1}{N} \sum_{i=1}^{N} f_i(w). \tag{1}$$

In many machine learning applications, $w \in \mathbb{R}^d$ is the model parameter to be learned, $N$ is the number of training samples, and each $f_i : \mathbb{R}^d \to \mathbb{R}$ is a (possibly regularized) loss associated with sample $i$. The following assumptions are commonly made.

**Assumption 2.1:** Each $f_i$ is $L_i$-smooth, i.e., $f_i(x) \leq f_i(y) + \langle \nabla f_i(y), x - y \rangle + \frac{L_i}{2} \|x - y\|^2 \forall x, y$.

**Assumption 2.2:** $F$ is $\mu$-strongly convex, i.e., $F(x) \geq F(y) + \langle F'(y), x - y \rangle + \frac{\mu}{2} \|x - y\|^2 \forall x, y$.

A. Delayed Proximal Gradient (DPG) \cite{13}

At iteration $t$, a worker uses a copy of $w$, which has been delayed by $\tau_t$ iterations (denoted $w^{t-\tau_t}$), to compute the stochastic gradient $g^{t-\tau_t} \equiv \nabla f_i(w^{t-\tau_t})$. This delayed gradient is used to update the correspondingly delayed parameter copy $w^{t-\tau_t}$ to $\hat{w}^{t-\tau_t} = w^{t-\tau_t} - \eta g^{t-\tau_t}$, where $\eta$ is a constant learning rate. This $\hat{w}^{t-\tau_t}$ is then sent to the server, which obtains the new iterate $w^{t+1}$ as a convex combination of the current $w^t$ and $\hat{w}^{t-\tau_t}$:

$$w^{t+1} = (1 - \theta)w^t + \theta \hat{w}^{t-\tau_t}, \tag{2}$$

where $\theta \in (0, 1]$. It can be shown that the $\{w^t\}$ sequence converges linearly to the optimal solution $w^*$ within a tolerance of $\epsilon$, i.e., $\mathbb{E}[F(w^t) - F(w^*)] \leq \rho^t(F(w^0) - F(w^*)), \epsilon$ for some $\rho < 1$ and $\epsilon > 0$. The tolerance $\epsilon$ can be reduced by reducing $\eta$, though at the expense of increasing $\rho$ and thus slowing down convergence.

B. Stochastic Variance Reduced Gradient \cite{13}

The SGD, though simple and scalable, has a slower convergence rate than batch gradient descent \cite{19}. As noted in \cite{13}, the underlying reason is that the stepsize of SGD has to be decreasing so as to control the gradient’s variance. Recently, by observing that the training set is always finite in practice, a number of techniques have been developed that reduce this variance and thus allows the use of a constant stepsize \cite{13, 15, 16, 18, 19}.

In this paper, we focus on one of these techniques, namely the stochastic variance reduction gradient (SVRG) \cite{13} (Algorithm\cite{1}). It is advantageous is that no extra space is needed for the intermediate gradients or dual variables. The algorithm proceeds in stages. At the beginning of each stage, the gradient $\nabla F(\tilde{w}) = \frac{1}{N} \sum_{i=1}^{N} \nabla f_i(\tilde{w})$ is computed on the whole data set using a past parameter estimate $\tilde{w}$ (which is updated across stages). For each subsequent iteration $t$ in this stage, the approximate gradient $\tilde{\nabla} f(w^t) = \nabla f_i(w^t) - \nabla f_i(\tilde{w}) + \nabla F(\tilde{w})$ is used, where $i$ is a sample randomly selected from $\{1, 2, \ldots, N\}$. Even with a constant learning rate $\eta$, it can be shown that the (expected) variance of $\tilde{\nabla} f(w^t)$ goes to zero progressively, and the algorithm achieves linear convergence.

Algorithm 1 Stochastic variance reduced gradient (SVRG) \cite{13}.

1: Initialize $\tilde{w}^0$;
2: for $s = 1, 2, \ldots$ do
3: \hspace{1em} $\tilde{w} = \tilde{w}^{s-1}$;
4: \hspace{1em} $\nabla F(\tilde{w}) = \frac{1}{N} \sum_{i=1}^{N} \nabla f_i(\tilde{w})$;
5: \hspace{1em} $w^0 = \tilde{w}$;
6: for $t = 1, 2, \ldots, m$ do
7: \hspace{2em} randomly pick $i \in \{1, \ldots, N\}$;
8: \hspace{2em} $w^{t+1} = w^t - \eta \tilde{\nabla} f(w^t)$;
9: \hspace{1em} end for
10: set $w^s = w^t$ for randomly chosen $t \in \{0, \ldots, m - 1\}$;
11: end for

III. PROPOSED ALGORITHM

Both DPG and SVRG allow the use of a constant, and thus, larger learning rate than typically used by SGD. However, while DPG is a distributed algorithm, it only converges to a neighborhood of the optimal solution. Better approximation quality comes at the expense of slower convergence. On the other hand, SVRG can converge to the optimal solution, but is only designed for use on single machines. Its use and convergence properties in the distributed asynchronous learning setting remain unexplored. In this section, we propose a hybrid of the two that combines the advantages of both. Similar to DPG and SVRG, it also uses a constant learning rate, but with a guaranteed linear convergence rate to the optimal solution in a distributed learning setting.

A. Update using Delayed Gradients

Recall that the server update is based on delayed gradients in distributed asynchronous learning. Here, we replace the update rule of SVRG (line 8 in algorithm\cite{1}) by

$$w^{t+1} = w^t - (\eta \tilde{\nabla} f(w^{t-\tau_t}) + \theta (w^t - w^{t-\tau_t})). \tag{3}$$

Obviously, when there is no delay (i.e., $\tau_t = 0$), \eqref{3} reduces to standard (serial) SVRG. The change to $w_t$ in \eqref{3} has two components. The first component $\eta \tilde{\nabla} f(w^{t-\tau_t})$ is a variance-reduced gradient as in SVRG. However, it is noisier as it is evaluated from the delayed parameter $w^{t-\tau_t}$. The second component $\theta (w^t - w^{t-\tau_t})$ then acts like a momentum term. With $\theta \in (0, 1)$, update \eqref{3} can also be rewritten as the following convex combination

$$w^{t+1} = (1 - \theta)w^t + \theta w^{t-\tau_t} - \beta \tilde{\nabla} f(w^{t-\tau_t}), \tag{4}$$

where $\beta = \eta / \theta$. This is similar to the update in \eqref{2}, but with the gradient $g^{t-\tau_t}$ replaced by its variance-reduced version $\tilde{\nabla} f(w^{t-\tau_t})$. Equation \eqref{3} can also be written as

$$w^{t+1} = (1 - \theta)w^t + \theta w^{t-\tau_t} - \eta (\nabla f_i(w^{t-\tau_t}) - \nabla f_i(\tilde{w})) - \eta \nabla F(\tilde{w}), \tag{4}$$

which will be useful in Section III-B2. Moreover, as in other SGD-based algorithms, we will use mini-batches to reduce the stochastic gradient’s variance \cite{20} and communication cost \cite{2, 6}. $\tilde{\nabla} f(w^{t-\tau_t})$ in \eqref{3} is then replaced by $\frac{1}{|B|} \sum_{i \in B^t} (\nabla f_i(w^{t-\tau_t}) - \nabla f_i(\tilde{w})) + \nabla F(\tilde{w})$, where $B^t$ is a mini-batch of size $B$ in the $t$th iteration.
B. Distributed Implementation

We have a scheduler, a server and P workers. The server keeps a clock (denoted by an integer value t), the most updated copy of parameter \( w \), a past parameter estimate \( w \) and the corresponding full gradient \( \nabla F(w) \) evaluated on the whole training set \( D \) (of \( N \) samples). We divide \( D \) into \( P \) disjoint subsets \( D_1, D_2, \ldots, D_P \), where \( D_p \) is owned by worker \( p \). The number of samples in \( D_p \) is denoted \( n_p \). Each worker \( p \) also keeps a local copy \( \tilde{w}_p \) of \( w \).

In the following, a task refers to an event timestamped by the scheduler. It can be issued by the scheduler or a worker, and received by either the server or workers. Each worker can only process one task at a time. There are two types of tasks, update task and evaluation task, and will be discussed in more detail in the sequel. A worker may also pull the parameter from the server by sending a request, which carries the type and timestamp of the task being run by the worker.

1) Scheduler: The scheduler (Algorithm 2) runs in stages. In each stage, it first issues \( m \) update tasks to the workers, where \( m \) is usually a multiple of \( \lceil N/B \rceil \) as in SVRG [13]. After spawning enough tasks, the server measures the progress by issuing an evaluation task to the server and all workers. As will be seen in Section III-B3, the server ensures that evaluation is carried out only after all the update tasks for the current stage have finished. If the progress meets the stopping condition, the scheduler informs the server and all workers to stop (by issuing a STOP command); otherwise, it moves to the next stage and sends more update tasks.

Algorithm 2 Scheduler.

1: for \( s = 1, \ldots, S \) do
2: for \( k = 1, \ldots, m \) do
3: pick worker \( p \) with probability \( n_p/N \);
4: issue an update task to the worker with timestamp \( t = (s-1)m + k \);
5: end for
6: issue an evaluation task (with timestamp \( t = sm + 1 \)) to workers and server;
7: wait and collect progress information from workers;
8: if progress satisfies some stopping condition then
9: issue a STOP command to the workers and server;
10: end if
11: end for

2) Worker: At stage \( s \), when worker \( p \) receives an update task with timestamp \( t \), it sends a parameter pull request to the server. This request will not be responded by the server until it finishes all tasks with timestamps before \( t - \tau \). The parameter \( \tau \) determines the maximum duration between the time the gradient is computed by a worker and till it is used by the server. A larger \( \tau \) allows more asynchronicity, but also adds noise to the gradient and thus may slow down convergence.

Let \( \tilde{w}_{p,t} \) be the parameter value pulled. Worker \( p \) selects a mini-batch \( B^t \subset D_p \) (of size \( B \)) randomly from its local data set, and then computes

\[
\tilde{w}_{p,t} = \theta \tilde{w}_{p,t} - \frac{\eta}{B} \sum_{i \in B^t} (\nabla f_i(\tilde{w}_{p,t}) - \nabla f_i(\tilde{w}_p)). \tag{5}
\]

On comparison with \( \tilde{w}_{p,t} \) in \( \tilde{w}_{p,t} \), it can be seen that \( \tilde{w}_{p,t} \) corresponds to the term \( \theta w^{t-\tau} - \eta (\nabla f_i(w^{t-\tau}) - \nabla f_i(\tilde{w})) \). This \( \tilde{w}_{p,t} \) is then pushed to the server by issuing an update task.

When a worker receives an evaluation task, it again sends a parameter pull request to the server. As will be seen in Section III-B3, the pulled \( \tilde{w}_{p,t} \) will always be the latest \( w \) kept by the server in the current stage. Hence, the \( \tilde{w}_{p,t} \)'s pulled by all workers are indeed the same. Worker \( p \) then computes \( \tilde{w}_p \) as \( \tilde{w}_p = \tilde{w}_{p,t} \), and computes and pushes the corresponding gradient \( \nabla F_p(\tilde{w}_p) = \frac{1}{n_p} \sum_{i \in D_p} \nabla f_i(\tilde{w}_p) \) to the server. To inform the scheduler of its progress, worker \( p \) also computes its contribution to the optimization objective \( \frac{1}{n_p} \sum_{i \in D_p} f_i(\tilde{w}_p) \) and pushes it to the scheduler. The whole worker procedure is shown in Algorithm 3.

Algorithm 3 Worker \( p \) receiving an update/evaluation task \( t \) at stage \( s \).

1: sends a parameter pull request to the server;
2: waits for response from server;
3: if task \( t \) is an update task then
4: pick a mini-batch subset \( B^t \) randomly from local data set;
5: compute \( \tilde{w}_{p,t} \) using \( \tilde{w}_p \), and push to server as an update task;
6: else
7: set \( \tilde{w}_p = \hat{w}_p; \)
8: evaluate \( \nabla F_p(\tilde{w}_p) \) on the local subset, and push it to the server as an update task;
9: computes and pushes the local objective value to the scheduler;
10: end if

3) Server: There are two threads running on the server. One is a daemon thread that responds to parameter pull requests from workers; and the other is a computing thread for handling update tasks from workers and evaluation tasks from the scheduler.

When the daemon thread receives a parameter pull request, it reads the type and timestamp \( t \) it contains. If the request is from a worker running an update task, it checks whether all update tasks before \( t - \tau \) have finished. If not, the request is pushed to a buffer; otherwise, it pushes its \( w \) value to the requesting worker. Thus, this controls the allowed asynchronicity. On the other hand, if the request is from a worker executing an evaluation task, the daemon thread does not push \( w \) to the workers until all update tasks before \( t \) have finished. This ensures that the \( w \) pulled by the worker is the most up-to-date for the current stage. Procedure for the daemon thread is shown in Algorithm 4.

When the computing thread receives an update task (with timestamp \( t \)) from worker \( p \), the \( \tilde{w}_{p,t} \) contained inside is read. Recall that \( \tilde{w}_{p,t} \) corresponds to the term \( \theta w^{t-\tau} - \eta (\nabla f_i(w^{t-\tau}) - \nabla f_i(\tilde{w})) \) in \( \tilde{w}_{p,t} \). The server thus finishes the remaining update as

\[
w \leftarrow (1-\theta)w + \tilde{w}_{p,t} - \eta \nabla F(w), \tag{6}
\]

and marks this task as finished. During the update, the computing thread locks \( w \) so that the daemon thread cannot access
The following asynchronous SGD-based algorithms are run for 10,000 iterations within a tolerance of $\epsilon$ and equal to $\gamma \leq 1$, and thus $\gamma < 1$ can be guaranteed.

The sequence $\{\tilde{w}^S\}$ output from the algorithm converges to $w^*$ linearly:

$$\mathbb{E}[F(\tilde{w}^S) - F(w^*)] \leq \gamma^S [F(\tilde{w}^0) - F(w^*)].$$

When $\tau < P$, the server can serve at most $\tau$ workers. Thus, for maximum parallelism, $\tau$ should increase with $P$. However, $\gamma$ also increases with $\tau$, and a larger $m$ and/or $S$ is needed to achieve the same solution quality. This compromise will be empirically studied in Section IV-C.

### IV. Experiments

In this section, we consider the $\ell_2$-regularized logistic regression problem:

$$\min_w \frac{1}{N} \sum_{i=1}^{N} \log(1 + \exp(-y_i x_i^T w)) + \frac{\lambda}{2} \|w\|^2,$$

where $\{(x_i, y_i)\}_{i=1}^{N}$ are the training samples, and $\lambda$ is a regularization parameter (fixed to 0.01). Experiments are performed on the Epsilon, OCR and DNA data sets (Table I) from the Pascal Large Scale Learning Challenge. The mini-batch size $B$ is set to 128 for Epsilon, 500 for OCR and 50,000 for DNA.

| Data Set | # of Samples | # of Features | Size (GB) |
|----------|--------------|---------------|-----------|
| Epsilon  | 0.5M         | 2000          | 12        |
| OCR      | 3.5M         | 1156          | 53        |
| DNA      | 50M          | 800           | 74        |

We perform experiments using the Google Cloud Computing Platform. Each computing node is a google cloud n1-highmem-2 instance with dual-core and 13GB memory. Each worker/scheduler/server is an instance. The system is implemented in C++, with the ZeroMC package for communication.

#### A. Comparison with Other Distributed Asynchronous SGD-based Algorithms

In this experiment, 16 workers are used, and we also set $\tau = 16$. The following asynchronous SGD-based algorithms are compared: (i) downpour SGD \[denoted \textit{“downpour”}\] with the adaptive learning rate in Adagrad \[\cite{12}\], (ii) a variant of Downpour SGD (denoted “downpour-$\tau$”), in which the staled stochastic gradients have a maximum delay of $\tau$. As shown in \[\cite{10}, \cite{11}\], this leads to better performance in practice; (iii) delayed proximal gradient \[\cite{14}\]; (iv) the proposed algorithm, which will be called “distributed variance-reduced stochastic gradient decent” (distr-VRSGD). The number of stages $S$ is set to 50, and the number of iterations $m$ in each stage is $\left\lceil N/B \right\rceil$. For fair comparison, the other algorithms are run for $mS$ iterations. We also include the (synchronous) SVRG, by setting $\tau = 0$. Parameter tuning for all algorithms is based on a validation set.

Figures 4(a) and 4(c) show convergence of the objective w.r.t. number of stages. As can be seen, SVRG is the fastest as its...
gradients are not delayed and thus have the highest quality. However, in terms of wall clock time (Figures 1(d)-1(f)), SVRG is the slowest because of its synchronized nature and thus high communication cost. Among the distributed async-

\[ \text{(objective − optimum)}/\text{optimum} \]
chronous algorithms, the proposed algorithm has the fastest convergence w.r.t. both the number of iterations and running time for all three data sets. Figure 2 shows a breakdown of the total time into computation time and network communication time. As can be seen, the proposed algorithm reduces the communication time the most, as it requires the fewest number of iterations.

B. Different Number of Workers

In this experiment, we vary the number of workers from 1 to 16. With fewer workers, each worker needs to store and process a larger data subset. As each computing node in our configuration has 13GB memory, we only perform experiments on the smallest Epsilon data set.

Figure 3(a) shows that both computation time and communication time decrease with the number of workers. Note that the most expensive step in the algorithm is on gradient evaluations. In each stage, there are \( m \) iterations, each of which involves a mini-batch of size \( B \). Hence, this requires a total of \( O(B) \) gradient evaluations. At the end of each stage, an additional \( O(N) \) gradient evaluations are required for performance evaluation. Hence, each worker spends \( O((mB + N)/P) \) time on computation. As for the communication time, having more workers implies that more parameter pull requests, tasks and data can be sent simultaneously between the server and workers. Hence, communication time also decreases. As a result, the speedup with the number of workers is close to linear.

However, recall that at the end of each stage, the server needs to aggregate local gradients \( \nabla F_p(\tilde{w}_p)'s \) from all workers. This requires a synchronous barrier across all workers. With more and more workers, the server may have to wait for a larger number of workers (Figure 3(c)). This increases communication time, and slows down speedup.

C. Effect of \( \tau \)

In this experiment, \( \tau \) is varied from 0 to 32. We fix \( m \), and run the algorithm for a sufficient number of \( S \) stages until a target objective value is attained. Timing results on the three data sets are shown in Figure 4. As \( \tau \) increases, more asynchronicity is allowed and there is a significant reduction in communication time. On the other hand, recall that in the convergence analysis, the convergence factor in each stage is given by \( \gamma \) in Theorem 3.1. Increasing \( \tau \) means more stages are needed, and this leads to higher computation and communication time. Hence, when \( \tau \) becomes very large, the computation and communication time increase again. Moreover, as can be seen, the choice of \( \tau = P \), which has been used in Sections [IV-A] and [IV-B] is a good compromise.

REFERENCES

[1] L. Bottou, “Large-scale machine learning with stochastic gradient descent,” in Proceedings of the International Conference on Computational Statistics, 2010, pp. 177–186.

[2] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le et al., “Large scale distributed deep networks,” in Advances in Neural Information Processing Systems, 2012, pp. 1223–1231.

[3] K. Gimpel, D. Das, and N. A. Smith, “Distributed asynchronous online learning for natural language processing,” in Proceedings of the Fourteenth Conference on Computational Natural Language Learning, 2010, pp. 213–222.

[4] F. Niu, B. Recht, C. Ré, and S. Wright, “Hogwild!: A lock-free approach to parallelizing stochastic gradient descent,” in Advances in Neural Information Processing Systems 24, 2011.

[5] O. Shamir, N. Srebro, and T. Zhang, “Communication-efficient distributed optimization using an approximate Newton-type method,” in Proceedings of the 31st International Conference on Machine Learning, 2014, pp. 1000–1008.

[6] R. Zhang and J. Kwok, “Asynchronous distributed ADMM for consensus optimization,” in Proceedings of the 31st International Conference on Machine Learning, 2014, pp. 1701–1709.

[7] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, “Distributed optimization and statistical learning via the alternating direction method of multipliers,” Foundations and Trends in Machine Learning, vol. 3, no. 1, pp. 1–122, 2011.

[8] J. Albrecht, C. Tuttle, A. Snoeren, and A. Vahdat, “Loosely synchronization for large-scale networked systems,” in Proceedings of the USENIX Annual Technical Conference, 2006, pp. 301–314.

[9] A. Agarwal and J. Duchi, “Distributed delayed stochastic optimization,” in Advances in Neural Information Processing Systems 24, 2011.

[10] Q. Ho, J. Cipar, H. Cui, S. Lee, J. Kim, P. Gibbons, G. Gibson, G. Ganger, and E. Xing, “More effective distributed ML via a stale synchronous parallel parameter server,” in Advances in Neural Information Processing Systems 26, 2013, pp. 1223–1231.

[11] M. Li, D. G. Andersen, A. J. Smola, and K. Yu, “Communication efficient distributed machine learning with the parameter server,” in Advances in Neural Information Processing Systems, 2014, pp. 19–27.

[12] J. Duchi, E. Hazan, and Y. Singer, “Adaptive subgradient methods for online learning and stochastic optimization,” Journal of Machine Learning Research, vol. 12, pp. 2121–2159, 2011.

[13] R. Johnson and T. Zhang, “Accelerating stochastic gradient descent using predictive variance reduction,” in Advances in Neural Information Processing Systems, 2013, pp. 315–323.

[14] H. R. Feyzmahdavian, A. Aytekin, and M. Johansson, “A delayed proximal gradient method with linear convergence rate,” in Proceedings of the International Workshop on Machine Learning for Signal Processing, 2014, pp. 1–6.

[15] A. Defazio, F. Bach, and S. Lacoste-Julien, “SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives,” in Advances in Neural Information Processing Systems, 2014, pp. 1646–1654.

[16] N. L. Roux, M. Schmidt, and F. R. Bach, “A stochastic gradient method with exponential convergence rate for finite training sets,” in Advances in Neural Information Processing Systems, 2012, pp. 2663–2671.

[17] S. Shalev-Shwartz and T. Zhang, “Stochastic dual coordinate ascent methods for regularized loss,” Journal of Machine Learning Research, vol. 14, no. 1, pp. 567–599, 2013.

[18] L. Xiao and T. Zhang, “A proximal stochastic gradient method with progressive variance reduction,” SIAM Journal on Optimization, vol. 24, no. 4, pp. 2057–2075, 2014.

[19] J. Mairal, “Optimization with first-order surrogates,” in Proceedings of the 30th International Conference on Machine Learning, 2013.

[20] R. H. Byrd, G. M. Chin, J. Nocedal, and Y. Wu, “Sample size selection in optimization methods for machine learning,” Mathematical programming, vol. 134, no. 1, pp. 127–155, 2012.

The scheduler and server operations are simple and their computational loads are negligible.