Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)

Asynchronous Distributed Semi-Stochastic Gradient Optimization

Ruiliang Zhang, Shuai Zheng, James T. Kwok
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Hong Kong
{rzhangaf, szhengac, jamesk}@cse.ust.hk

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 by proposing a fast 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.

Introduction

With the recent proliferation of big data, learning the parameters in a large machine learning model is a challenging problem. A popular approach is to use stochastic gradient descent (SGD) and its variants (Bottou 2010; Dean et al. 2012; Gimpel, Das, and Smith 2010). However, it can still be difficult to store and process a big data set on one single machine. Thus, there is now growing interest in distributed machine learning algorithms. (Dean et al. 2012; Gimpel, Das, and Smith 2010; Niu et al. 2011; Shamir, Srebro, and Zhang 2014; Zhang and Kwok 2014). The data set is partitioned into subsets, assigned to multiple machines, and the optimization problem is solved in a distributed manner.

In general, distributed architectures can be categorized as shared-memory (Niu et al. 2011) or distributed-memory (Dean et al. 2012; Gimpel, Das, and Smith 2010; Shamir, Srebro, and Zhang 2014; Zhang and Kwok 2014). In this paper, we will focus on the latter, which is more scalable. Usually, one of the machines is the *server*, while the rest are *workers*. The workers store the data subsets, perform local computations and send their updates to the server. The server then aggregates the local information, performs the actual update on the model parameter, and sends it back to the workers. 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 (Dean et al. 2012; Gimpel, Das, and Smith 2010; Shamir, Srebro, and Zhang 2014; Zhang and Kwok 2014).

Often, machines in these systems have to run synchronously (Boyd et al. 2011; Shamir, Srebro, and Zhang 2014). In each iteration, information from all workers need to be ready before the server can aggregate the updates. This can be expensive due to communication overhead and random network delay. It also suffers from the straggler problem (Albrecht et al. 2006), in which the system can move forward only at the pace of the slowest worker.

To alleviate these problems, asynchronicity is introduced (Agarwal and Duchi 2011; Dean et al. 2012; Ho et al. 2013; Li et al. 2014; Zhang and Kwok 2014). 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* (Dean et al. 2012). 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 (Duchi, Hazan, and Singer 2011), 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 algorithms (Bottou 2010)), 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) (Ho et al. 2013). On the other hand, note that downpour SGD does not constrain τ, but no convergence guarantee is provided.

In practice, a decreasing learning rate leads to slower convergence (Bottou 2010; Johnson and Zhang 2013). Recently, Feyzmahdavian, Aytekin, and Johansson (2014) 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. Examples include *stochastic average gradient* (SAG) (Roux, Schmidt, and Bach 2012), *stochastic variance reduced gradient* (SVRG) (Johnson and Zhang 2013), *minimization by incremental surrogate optimization* (MISO) (Mairal 2013; 2015), SAGA (Defazio, Bach, and Lacoste-Julien 2014), *stochastic dual coordinate descent* (SDCA) (Shalev-Shwartz and Zhang 2013), and Proximal SVRG (Xiao and Zhang 2014). 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. When the optimization objective is strongly convex and Lipschitz-smooth, all these variance-reduced SGD algorithms converge linearly to the optimal solution. However, their space requirements are different. In particular, SVRG is advantageous in that it only needs to store the averaged sample gradient, while SAGA and SAG have to store all the samples' most recent gradients. Recently, Mania et al. (2015) and Reddi et al. (2015) extended SVRG to the parallel asynchronous setting. Their algorithms are designed for shared-memory multi-core systems, and assume that the data samples are sparse. However, in a distributed computing environment, the samples need to be mini-batched to reduce the communication overhead between workers and server. Even when the samples are sparse, the resultant mini-batch typically is not.

In this paper, we propose a distributed asynchronous SGD-based algorithm with variance reduction, and the data samples can be sparse or dense. The algorithm is easy to implement, highly scalable, uses a constant learning rate, and converges linearly to the optimal solution. A prototype is implemented on the Google Cloud Computing Platform. Experiments on several big data sets from the Pascal Large Scale Learning Challenge and LibSVM archive demonstrate that it outperforms the state-of-the-art.

The rest of the paper is organized as follows. We first introduce related work. Next, we present the proposed distributed asynchronous algorithm. This is then followed by experimental results including comparisons with the state-of-the-art distributed asynchronous algorithms, and the last section gives concluding remarks.

Related Work

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, N is the number of training samples, and each $f_i : \mathbb{R}^d \to \mathbb{R}$ is the loss (possibly regularized) due to sample i. The following assumptions are commonly made.

Assumption 1 *Each f_i is L_i-smooth (Nesterov 2004), i.e., $f_i(x) \le f_i(y) + \langle \nabla f_i(y), x-y\rangle + \frac{L_i}{2}\|x-y\|^2 \ \forall x, y$.*

Assumption 2 *F is μ-strongly convex (Nesterov 2004), i.e., $F(x) \ge F(y) + \langle F(y), x-y\rangle + \frac{\mu}{2}\|x-y\|^2 \ \forall x, y$.*

Delayed Proximal Gradient (DPG)

At iteration t of the DPG (Feyzmahdavian, Aytekin, and Johansson 2014), a worker uses $w^{t-\tau_t}$, the copy of w delayed by τ_t iterations, to compute the stochastic gradient $g^{t-\tau_t} = \nabla f_i(w^{t-\tau_t})$ on a random sample i. The 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 η 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 $\hat{w}^{t-\tau_t}$:

$$w^{t+1} = (1-\theta)w^t + \theta\hat{w}^{t-\tau_t}, \ \ \theta \in (0, 1]. \tag{2}$$

It can be shown that the $\{w^t\}$ sequence converges linearly to the optimal solution w^*, but only within a tolerance of ϵ, i.e.,

$$\mathbb{E}\left[F(w^t) - F(w^*)\right] \le \rho^t(F(w^0) - F(w^*)) + \epsilon,$$

for some $\rho < 1$ and $\epsilon > 0$. The tolerance ϵ can be reduced by reducing η, though at the expense of increasing ρ and thus slowing down convergence. Moreover, though the learning rate of DPG is typically larger than that of SGD, the gradient of DPG (i.e., $\hat{w}^{t-\tau_t}$) is delayed and slows convergence.

Stochastic Variance Reduced Gradient

The SGD, though simple and scalable, has a slower convergence rate than batch gradient descent (Mairal 2013). As noted in (Johnson and Zhang 2013), 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 to reduce this variance and thus allows the use of a constant stepsize (Defazio, Bach, and Lacoste-Julien 2014; Johnson and Zhang 2013; Mairal 2013; Roux, Schmidt, and Bach 2012; Xiao and Zhang 2014).

In this paper, we focus on one of most popular techniques in this family, namely the *stochastic variance reduction gradient* (SVRG) (Johnson and Zhang 2013) (Algorithm 1). It is advantageous in 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

$$\hat{\nabla} f_i(w^t) = \nabla f_i(w^t) - \nabla f_i(\tilde{w}) + \nabla F(\tilde{w})$$

is used, where i is a sample randomly selected from $\{1, 2, \ldots, N\}$. Even with a constant learning rate η, the (expected) variance of $\hat{\nabla} f(w^t)$ goes to zero progressively, and the algorithm achieves linear convergence.

In contrast to DPG, SVRG can converge to the optimal solution. However, though SVRG has been extended to the parallel asynchronous setting on shared-memory multi-core systems (Reddi et al. 2015; Mania et al. 2015), its use and convergence properties in a distributed asynchronous learning setting remain unexplored.

Algorithm 1 Stochastic variance reduced gradient (SVRG) (Johnson and Zhang 2013).

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

Proposed Algorithm

In this section, we consider the distributed asynchronous setting, and propose a hybrid of an improved DPG algorithm and SVRG that inherits the advantages of both. Similar to the two algorithms, it also uses a constant learning rate (which is typically larger than the one used by SGD), but with guaranteed linear convergence to the optimal solution.

Update using Delayed Gradient

We replace the SVRG update (line 8 in Algorithm 1) by

$$w^{t+1} = (1-\theta)v^t + \theta\bar{w}^{t-\tau_t}, \tag{3}$$

where

$$v^t = w^t - \eta\hat{\nabla} f_i(w^{t-\tau_t})$$

and

$$\bar{w}^{t-\tau_t} = w^{t-\tau_t} - \eta\hat{\nabla} f_i(w^{t-\tau_t}). \tag{4}$$

Obviously, when $\tau_t = 0$, (3) reduces to standard SVRG. Note that both the parameter and gradient in $\bar{w}^{t-\tau_t}$ are for the same iteration ($t - \tau_t$), while v^t is noisy as the gradient is delayed (by τ_t). This delayed gradient cannot be too old. Thus, similar to (Ho et al. 2013), we impose the *bounded delay* condition that $\tau_t \le \tau$ for some $\tau > 0$. This τ parameter determines the maximum duration between the time the gradient is computed and till it is used. A larger τ allows more asynchronicity, but also adds noise to the gradient and thus may slow convergence.

Update (3) is similar to (2) in DPG, but with two important differences. First, the gradient $\nabla f_i(w^{t-\tau_t})$ in DPG is replaced by its variance-reduced counterpart $\hat{\nabla} f_i(w^{t-\tau_t})$. As will be seen, this allows convergence to the optimal solution using a constant learning rate. The second difference is that the delayed gradient $\hat{\nabla} f_i(w^{t-\tau_t})$ is used not only on the past iterate $w^{t-\tau_t}$, but also on the current iterate w^t. This can potentially yield faster progress, as is most apparent when $\theta = 0$. In this special case, DPG reduces to $w^{t+1} = w^t$, and makes no progress; while (3) reduces to the asynchronous SVRG update in (Reddi et al. 2015).

Mini-Batch

In a distributed algorithm, communication overhead is incurred when a worker pulls parameters from the server or pushes update to it. In a distributed SGD-based algorithm, the communication cost is proportional to the number of gradient evaluations made by the workers. Similar to the other SGD-based distributed algorithms (Gimpel, Das, and Smith 2010; Dean et al. 2012; Ho et al. 2013), this cost can be reduced by the use of a mini-batch. Instead of pulling parameters from the server after every sample, the worker pulls only after processing each mini-batch of size B.

Distributed Implementation

There is a *scheduler*, a *server* and P *workers*. The server keeps a clock (denoted by an integer t), the most updated copy of parameter w, a past parameter estimate $\tilde{w}$ and the corresponding full gradient $\nabla F(\tilde{w})$ evaluated on the whole training set $\mathcal{D}$ (with N samples). We divide $\mathcal{D}$ into P disjoint subsets $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_P$, where $\mathcal{D}_p$ is owned by worker p. The number of samples in $\mathcal{D}_p$ is denoted n_p. Each worker p also keeps a local copy $\tilde{w}_p$ of $\tilde{w}$.

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 pull the parameter from the server by sending a *request*, which carries the type and timestamp of the task being run by the worker.

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. After spawning enough tasks, the server measures the progress by issuing an evaluation task to the server and all workers. As will be seen, the server ensures that evaluation is carried out only after all update tasks for the current stage have finished. If the stopping condition is met, the scheduler informs the server and all workers by issuing a STOP command; otherwise, it moves to the next stage and sends more update tasks.

Algorithm 2 Scheduler.

for $s = 1, \ldots, S$ **do**
  for $k = 1, \ldots, m$ **do**
    pick worker p with probability $\frac{n_p}{N}$;
    issue an update task to the worker with timestamp $t = (s-1)m + k$;
  end for
  issue an evaluation task (with timestamp $t = sm+1$) to workers and server;
  wait and collect progress information from workers;
  if progress meets stopping condition **then**
    issue a STOP command to the workers and server;
  end if
end for

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 $t - \tau$.

Let $\hat{w}_{p,t}$ be the parameter value pulled. Worker p selects a mini-batch $\mathcal{B}^t \subset \mathcal{D}_p$ (of size B) randomly from its local data set. Analogous to $\bar{w}^{t-\tau_t}$ in (4), it computes

$$\bar{w}_{p,t} = \hat{w}_{p,t} - \eta \Delta w_{p,t}, \tag{5}$$

where $\Delta w_{p,t}$ is the mini-batch gradient evaluated at $\hat{w}_{p,t}$. An update task is then issued to push $\bar{w}_{p,t}$ and $\Delta w_{p,t}$ to the server.

When a worker receives an *evaluation* task, it again sends a parameter pull request to the server. As will be seen in the following section, the pulled $\hat{w}_{p,t}$ will always be the latest w kept by the server in the current stage. Hence, the $\hat{w}_{p,t}$'s pulled by all workers are the same. Worker p then updates $\tilde{w}_p$ as $\tilde{w}_p = \hat{w}_{p,t}$, computes and pushes the corresponding gradient

$$\nabla F_p(\tilde{w}_p) = \frac{1}{n_p} \sum_{i \in \mathcal{D}_p} \nabla f_i(\tilde{w}_p)$$

to the server. To inform the scheduler of its progress, worker p also computes its contribution to the optimization objective $\sum_{i \in \mathcal{D}_p} f_i(\tilde{w}_p)$ 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.

```
send a parameter pull request to the server;
wait for response from the server;
if task t is an update task then
    pick a mini-batch subset B^t randomly from the local data set;
    compute mini-batch gradient Δw_{p,t} and w̄_{p,t} using (5), and push them to the server as an update task;
else
    set w̃_p = ŵ_{p,t}; {task t is an evaluation task}
    push the local subset gradient ∇F_p(w̃_p) to the server as an update task;
    push the local objective value to the scheduler;
end if
```

Server There are two threads running on the server. One is a daemon thread that responds to parameter pull requests from workers (Algorithm 4); and the other is a computing thread for handling update tasks from workers and evaluation tasks from the scheduler (Algorithm 5).

When the daemon thread receives a parameter pull request, it reads the type and timestamp t within. 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 remains in the 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.

When the computing thread receives an update task (with timestamp t) from worker p, the $\bar{w}_{p,t}$ and $\Delta w_{p,t}$ contained inside are read. Analogous to (3), the server updates w as

$$w = (1-\theta)(w - \eta \Delta w_{p,t}) + \theta \bar{w}_{p,t}, \tag{6}$$

and marks this task as finished. During the update, the computing thread locks w so that the daemon thread cannot access until the update is finished.

When the server receives an evaluation task, it synchronizes all workers, and sets $\tilde{w} = w$. As all $\tilde{w}_p$'s are the same and equal to $\tilde{w}$, one can simply aggregate the local gradients to obtain $\nabla F(\tilde{w}) = \sum_{p=1}^{P} q_p \nabla F_p(\tilde{w}_p)$, where $q_p = \frac{n_p}{N}$. The server then broadcasts $\nabla F(\tilde{w})$ to all workers.

Algorithm 4 Daemon thread of the server.

```
repeat
    if pull request buffer is not empty then
        for each request with timestamp t in the buffer do
            if request is triggered by an update task then
                if all update tasks before t − τ have finished then
                    push w to the requesting worker;
                    remove request from buffer;
                end if
            else
                if all update tasks before t have finished then
                    push w to the requesting worker;
                    remove request from buffer;
                end if
            end if
        end for
    else
        sleep for a while;
    end if
until STOP command is received.
```

Algorithm 5 Computing thread of the server.

```
repeat
    wait for tasks;
    if an update task received then
        update w using (6), and mark this task as finished;
    else
        wait for all update tasks to finish;
        set w̃ = w;
        collect local full gradients from workers and update ∇F(w̃);
        broadcast ∇F(w̃) to all workers;
    end if
until STOP command is received.
```

Discussion

Two state-of-the-art distributed asynchronous SGD algorithms are the downpour SGD (Dean et al. 2012) and Petuum SGD (Ho et al. 2013; Dai et al. 2013). Downpour SGD does not impose the bounded delay condition (essentially, $\tau = \infty$), while Petuum SGD does. Note that there is a subtle difference in the bounded delay condition of the proposed algorithm and that of Petuum SGD. In Petuum SGD, the amount of staleness is measured between workers, namely

that the slowest and fastest workers must be less than s timesteps apart (where s is the staleness parameter). Consequently, the delay in the gradient is always a multiple of P and is upper-bounded by sP. On the other hand, in the proposed algorithm, the bounded delay condition is imposed on the update tasks. It can be easily seen that τ is also the maximum delay in the gradient. Thus, τ can be any number which is not necessarily a multiple of P.

Convergence Analysis

For simplicity of analysis, we assume that the mini-batch size is one. Let $\tilde{w}^S$ be the w stored in the server at the end of stage S (step 7 in Algorithm 5). The following Theorem shows linear convergence of the proposed algorithm. It is the first such result for distributed asynchronous SGD-based algorithms with constant learning rate.

Theorem 3 *Suppose that problem (1) satisfies Assumption 1 and 2. Let* $L = \max\{L_i\}_{i=1}^N$, *and* $\gamma = \left(1 - 2\eta(\mu - \frac{\eta L^2}{\theta})\right)^{\frac{m}{1+\tau}} + \frac{\eta L^2}{\theta\mu - \eta L^2}$. *With* $\eta \in (0, \frac{\mu\theta}{2L^2})$ *and* m *sufficiently large such that* $\gamma < 1$. *Assume that the scheduler has been run for* S *stages, we obtain the following linear rate:*

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

As $L > \mu$, it is easy to see that $1 - 2\eta(\mu - \frac{\eta L^2}{\theta})$ and $\frac{\eta L^2}{\theta\mu - \eta L^2}$ are both smaller than 1. Thus, $\gamma < 1$ can be guaranteed for a sufficiently large m. Moreover, as F is strongly convex, the following Corollary shows that $\tilde{w}^S$ also converges to w^*. In contrast, DPG only converges to within a tolerance of ϵ.

Corollary 4 $\mathbb{E}\|\tilde{w}^S - w^*\|^2 \leq 2\gamma^S[F(\tilde{w}^0) - F(w^*)]/\mu$.

When $\tau < P$, the server can serve at most τ workers simultaneously. For maximum parallelism, τ should increase with P. However, γ also increases with τ. Thus, a larger m and/or S may be needed to achieve the same solution quality.

Similar to DPG, our learning rate does not depend on the delay τ and the number of workers P. This learning rate can be significantly larger than the one in Pentuum SGD (Ho et al. 2013), which has to be decayed and is proportional to $\mathcal{O}(1/\sqrt{P})$. Thus, the proposed algorithm can be much faster, as will be confirmed in the experiments. While our bound may be loose due to the use of worst-case analysis, linear convergence is always guaranteed for any $\theta \in (0, 1]$.

Experiments

In this section, we consider the K-class logistic regression problem:

$$\min_{\{w_k\}_{k=1}^K} \frac{1}{N} \sum_{i=1}^N \sum_{k=1}^K -I(y_i = k) \log\left(\frac{\exp(w_k^T x_i)}{\sum_{j=1}^K \exp(w_j^T x_i)}\right),$$

where $\{(x_i, y_i)\}_{i=1}^N$ are the training samples, w_k is the parameter vector of class k, and $I(\cdot)$ is the indicator function which returns 1 when the argument holds, and 0 otherwise. Experiments are performed on the *Mnist8m* and *DNA* data sets (Table 1) from the *LibSVM* archive[1] and Pascal Large Scale Learning Challenge[2].

	#samples	#features	#classes
Mnist8m	8,100,000	784	10
DNA	50,000,000	800	2

Table 1: Summary of the data sets used.

Using the Google Cloud Computing Platform[3], we set up a cluster with 18 computing nodes. Each node is a google cloud n1-highmem-8 instance with eight cores and 52GB memory. Each scheduler/server takes one instance, while each worker takes a core. Thus, we have a maximum of 128 workers. The system is implemented in C++, with the ZeroMQ package for communication.

Comparison with the State-of-the-Art

In this section, the following distributed asynchronous algorithms are compared:

- Downpour SGD ("*downpour-sgd*") (Dean et al. 2012), with the adaptive learning rate in Adagrad (Duchi, Hazan, and Singer 2011);
- Petuum SGD (Dai et al. 2013) ("*petuum-sgd*"), the state-of-the-art implementation of asynchronous SGD. The learning rate is reduced by a fixed factor 0.95 at the end of each epoch. The staleness s is set to 2, and so the delay in the gradient is bounded by $2P$.
- DPG (Feyzmahdavian, Aytekin, and Johansson 2014) ("*dpg*");
- A variant of DPG ("*vr-dpg*"), in which the gradient in update (2) is replaced by its variance-reduced version;
- The proposed "distributed variance-reduced stochastic gradient decent" (*distr-vr-sgd*) algorithm.
- A special case of *distr-vr-sgd*, with $\theta = 0$ (denoted "*distr-svrg*"). This reduces to the asynchronous SVRG algorithm in (Reddi et al. 2015).

We use 128 workers. To maximize parallelism, we fix τ to 128. The Petuum SGD code is downloaded from http://petuum.github.io/, while the other asynchronous algorithms are implemented in C++ by reusing most of our system's codes. Preliminary studies show that synchronous SVRG is much slower and so is not included for comparison. For *distr-vr-sgd* and *distr-svrg*, the number of stages is $S = 50$, and the number of iterations in each stage is $m = \lceil N/B \rceil$, where B is about 10% of each worker's local data set size. For fair comparison, the other algorithms are run for mS iterations. All other parameters are tuned by a validation set, which is 1% of the data set.

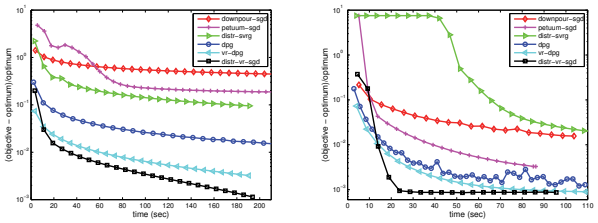
Figure 1 shows convergence of the objective w.r.t. wall clock time. As can be seen, *distr-vr-sgd* outperforms all the other algorithms. Moreover, unlike *dpg*, it can converge to the optimal solution and attains a much smaller objective value. Note that *distr-svrg* is slow. Since $\tau = 128$,

[1] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/

[2] http://argescale.ml.tu-berlin.de/

[3] http://cloud.google.com

the delayed gradient can be noisy, and the learning rate used by *distr-svrg* (as determined by the validation set) is small (10^{-6} vs 10^{-3} in *distr-vr-sgd*). On the *DNA* data set, *distr-svrg* is even slower than *petuum-sgd* and *downpour-sgd* (which use adaptive/decaying learning rates). The *vr-dpg*, which uses variance-reduced gradient, is always faster than *dpg*. Moreover, *distr-vr-sgd* is faster than *vr-sgd*, showing that replacing w^t in (2) by v^t in (3) is useful.

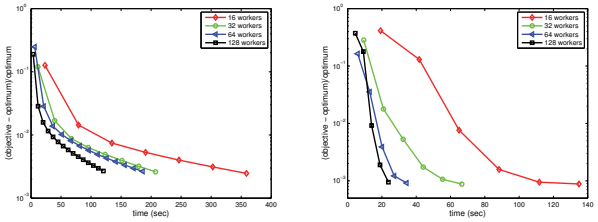


(a) *Mnist8m*. (b) *DNA*.

Figure 1: Objective vs time (in sec).

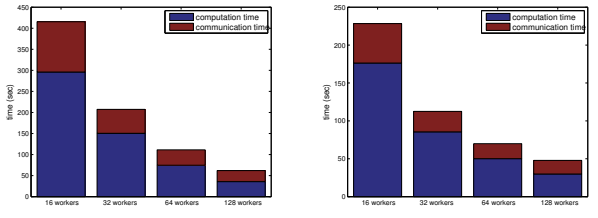
Varying the Number of Workers

In this experiment, we run *distr-vr-sgd* with varying number of workers (16, 32, 64 and 128) until a target objective value is met. Figure 2 shows convergence of the objective with time. On the *Mnist8m* data set, using 128 workers is about 3 times faster than using 16 workers. On the *DNA* data set, the speedup is about 6 times.



(a) *Mnist8m*. (b) *DNA*.

Figure 2: Objective vs time (in sec), with different numbers of workers.



(a) *Mnist8m*. (b) *DNA*.

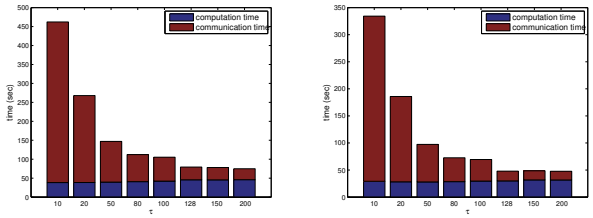
Figure 3: Breakdown into computation time and communication time, with different numbers of workers.

Note that the most expensive step in the algorithm is on gradient evaluations (the scheduler and server operations are simple). Recall that each stage has m iterations, and each iteration involvs $O(B)$ gradient evaluations. At the end of each stage, an additional $O(N)$ gradients are evaluated to obtain the full gradient and monitor the progress. Hence, each worker spends $\mathcal{O}((mB + N)/P)$ time on computation. The computation time thus decreases linearly with P, as can be seen from the breakdown of total wall clock time into computation time and communication time in Figure 3.

Moreover, having more workers means more tasks/data can be sent among the server and workers simultaneously, reducing the communication time. On the other hand, as synchronization is required among all workers at the end of each stage, having more workers increases the communication overhead. Hence, as can be seen from Figure 3, the communication time first decreases with the number of workers, but then increases as the communication cost in the synchronization step starts to dominate.

Effect of τ

In this experiment, we use 128 workers. Figure 4 shows the time for *distr-vr-sgd* to finish mS tasks (where $S = 50$ and $m = \lceil N/B \rceil$) when τ is varied from 10 to 200. As can be seen, with increasing τ, higher asynchronicity is allowed, and the communication cost is reduced significantly.



(a) *Mnist8m*. (b) *DNA*.

Figure 4: Breakdown of the total time into computation time and communication time, with different τ's.

Conclusion

Existing distributed asynchronous SGD algorithms often rely on a decaying learning rate, and thus suffer from a sublinear convergence rate. On the other hand, the recent delayed proximal gradient algorithm uses a constant learning rate and has linear convergence rate, but can only converge to within a neighborhood of the optimal solution. In this paper, we proposed a novel distributed asynchronous SGD algorithm by integrating the merits of the stochastic variance reduced gradient algorithm and delayed proximal gradient algorithm. Using a constant learning rate, it still guarantees convergence to the optimal solution at a fast linear rate. A prototype system is implemented and run on the Google cloud platform. Experimental results show that the proposed algorithm can reduce the communication cost significantly with the use of asynchronicity. Moreover, it converges much

faster and yields more accurate solutions than the state-of-the-art distributed asynchronous SGD algorithms.

Acknowledgments

This research was supported in part by the Research Grants Council of the Hong Kong Special Administrative Region (Grant 614012).

References

Agarwal, A., and Duchi, J. 2011. Distributed delayed stochastic optimization. In *Advances in Neural Information Processing Systems 24*.

Albrecht, J.; Tuttle, C.; Snoeren, A.; and Vahdat, A. 2006. Loose synchronization for large-scale networked systems. In *Proceedings of the USENIX Annual Technical Conference*, 301–314.

Bottou, L. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the International Conference on Computational Statistics*. 177–186.

Boyd, S.; Parikh, N.; Chu, E.; Peleato, B.; and Eckstein, J. 2011. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning* 3(1):1–122.

Dai, W.; Wei, J.; Zheng, X.; Kim, J. K.; Lee, S.; Yin, J.; Ho, Q.; and Xing, E. P. 2013. Petuum: A framework for iterative-convergent distributed ML. Technical Report arXiv:1312.7651.

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

Defazio, A.; Bach, F.; and Lacoste-Julien, S. 2014. SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives. In *Advances in Neural Information Processing Systems*, 1646–1654.

Duchi, J.; Hazan, E.; and Singer, Y. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* 12:2121–2159.

Feyzmahdavian, H. R.; Aytekin, A.; and Johansson, M. 2014. A delayed proximal gradient method with linear convergence rate. In *Proceedings of the International Workshop on Machine Learning for Signal Processing*, 1–6.

Gimpel, K.; Das, D.; and Smith, N. A. 2010. Distributed asynchronous online learning for natural language processing. In *Proceedings of the 14th Conference on Computational Natural Language Learning*, 213–222.

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

Johnson, R., and Zhang, T. 2013. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in Neural Information Processing Systems*, 315–323.

Li, M.; Andersen, D. G.; Smola, A. J.; and Yu, K. 2014. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems*, 19–27.

Mairal, J. 2013. Optimization with first-order surrogate functions. In *Proceedings of the 30th International Conference on Machine Learning*.

Mairal, J. 2015. Incremental majorization-minimization optimization with application to large-scale machine learning. Technical Report arXiv:1402.4419.

Mania, H.; Pan, X.; Papailiopoulos, D.; Recht, B.; Ramchandran, K.; and Jordan, M. I. 2015. Perturbed iterate analysis for asynchronous stochastic optimization. Technical Report arXiv:1507.06970.

Nesterov, Y. 2004. *Introductory Lectures on Convex Optimization*, volume 87. Springer Science & Business Media.

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

Reddi, S. J.; Hefny, A.; Sra, S.; Póczos, B.; and Smola, A. 2015. On variance reduction in stochastic gradient descent and its asynchronous variants. Technical Report arXiv:1506.06840.

Roux, N. L.; Schmidt, M.; and Bach, F. R. 2012. A stochastic gradient method with an exponential convergence rate for finite training sets. In *Advances in Neural Information Processing Systems*, 2663–2671.

Shalev-Shwartz, S., and Zhang, T. 2013. Stochastic dual coordinate ascent methods for regularized loss. *Journal of Machine Learning Research* 14(1):567–599.

Shamir, O.; Srebro, N.; and Zhang, T. 2014. Communication-efficient distributed optimization using an approximate Newton-type method. In *Proceedings of the 31st International Conference on Machine Learning*, 1000–1008.

Xiao, L., and Zhang, T. 2014. A proximal stochastic gradient method with progressive variance reduction. *SIAM Journal on Optimization* 24(4):2057–2075.

Zhang, R., and Kwok, J. 2014. Asynchronous distributed ADMM for consensus optimization. In *Proceedings of the 31st International Conference on Machine Learning*, 1701–1709.